Politécnico
Castelo Branco
Escola Superior
de Tecnologia

Framework para gamificação de atividades educativas - jogos - “Supernova QuEST”.

Projeto II

Luís Miguel da Encarnação Salvado, 62006157

Orientador

Fernando Sérgio Rodrigues de Brito da Mota Barbosa

Trabalho de Projeto apresentado à Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco para cumprimento dos requisitos necessários à obtenção do grau de Licenciado em Informática e Multimédia, realizada sob a orientação científica do Doutor Fernando Sérgio Rodrigues de Brito da Mota Barbosa, do Instituto Politécnico de Castelo Branco.

Junho de 2025

Composição do júri

Presidente do júri

Doutor, Carlos Manuel de Oliveira Alves

Professor Adjunto, Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco.

Vogais

Doutor, Fernando Sérgio Rodrigues de Brito da Mota Barbosa

Professor Adjunto, Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco.

Mestre, Paulo Alexandre Correia da Silva Neves

Professor Adjunto, Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco.

Agradecimentos

Aproveito este espaço para agradecer ao orientador deste projeto, o professor Fernando Sérgio Rodrigues de Brito da Mota Barbosa, que sempre mostrou disponibilidade, para marcações de sessões de esclarecimento online, rapidez de resposta nos e-mails e grande profissionalismo ao apoiar-me na construção deste relatório. Sem ele, com certeza que teria sido muito mais difícil.

Outro agradecimento que gostaria de fazer é dirigido à minha companheira Maria João Cerieiro Mendes, que, ao longo dos últimos meses, tem sido a primeira pessoa a fazer a revisão deste documento. Agradeço assim a sua disponibilidade e paciência, tem sido um apoio essencial para que este projeto siga o rumo desejado.

Por fim, gostaria de agradecer a minha família e a todos os colegas de trabalho, que sempre me incentivaram nesta minha aventura de conclusão do grau de licenciado.

Resumo

É cada vez mais comum, a tentativa de uso de gamificação de conteúdos educacionais por parte dos docentes. Contudo, algumas das estratégias adotadas para fazer essa metodologia resultar, podem não estar a obter os resultados pretendidos.

A pensar nesse problema, foi criada a *framework* "GamEducation" que promete ajudar os docentes a ultrapassar essa dificuldade.

Este projeto, tem como propósito a criação de um videojogo intitulado de "Supernova QuEST". Este videojogo promete captar o interesse e a atenção dos estudantes de forma natural, para que os mesmos possam assimilar conteúdos educacionais sem se aperceberem disso. Para manter a motivação ao longo do estudo, os jogadores serão desafiados em momentos chave do jogo, de forma a promover uma progressão continua durante a sua experiência.

O desenvolvimento deste projeto teve como ponto de partida o *GDD – Game Desing Document* escrito em Projeto I. O nível de detalhe colocado no mesmo, foi responsável por uma implementação quase perfeita do videojogo. Quase, porque o jogo não foi completamente concluído como era desejado como também ficou em falta a sua ligação com a "GamEducation".

Em suma, a versão atual do jogo apresenta-se estável com uma arquitetura modular que garante escalabilidade e permite a integração de novos conteúdos de forma eficiente.

Palavras-chave

Supernova QuEST, Videojogo, Unity, Gamificação

Abstract

It is becoming increasingly common for teachers to try to gamify educational content. However, some of the strategies adopted to make this methodology work may not be achieving the desired results.

With this problem in mind, the "GamEducation" framework was created, which promises to help teachers overcome this difficulty.

The purpose of this project is to create a video game called "Supernova QuEST". This video game promises to capture the interest and attention of students in a natural way, so that they can assimilate educational content without realizing it. To maintain motivation throughout the study, players will be challenged at key moments in the game, in order to promote continuous progression during their experience.

The development of this project took as its starting point the GDD - Game Design Document written in Project I. The level of detail put into it was responsible for an almost perfect implementation of the video game. Almost, because the game was not completely finished as desired and its connection with "GamEducation" was also missing.

In short, the current version of the game is stable with a modular architecture that guarantees scalability and allows new content to be integrated efficiently.

Keywords

Supernova QuEST, Video game, Unity, Gamification

Índice geral

Índice de figuras

Lista de abreviaturas, siglas e acrónimos

API - Application Programming Interface

APK - Android Package Kit

FPS - Frames per Second

GDD - Game Design Document

HUD - Heads-up display

UI - User Interface

1. Introdução

Após a conclusão da primeira etapa do desenvolvimento do videojogo "Supernova QuEST", na cadeira de Projeto I, iniciou-se imediatamente o desenvolvimento de Projeto II, tendo em conta algumas melhorias sugeridas pelo júri e utilizando como ponto de partida o protótipo construído para a demonstração do trabalho feito na primeira fase, bem como, o estudo efetuado para o desenvolvimento do *GDD*.

Este videojogo tem como intuito ser integrado com a *framework* educacional "GamEducation", realizada pelos ex-alunos João Tiago Alves Dias e Luís Miguel Farinha Mateus. O objetivo desta *framework* é a inclusão de elementos didáticos nos videojogos, de forma a motivar os jovens a estudar.

O objetivo principal consiste em criar um jogo que se encaixe no universo mobile, mais concretamente no sistema operativo Android, utilizado pela *framework*. Tendo isso em mente, foi escolhido o motor de desenvolvimento de jogos *Unity* com esse propósito.

Ao longo deste documento, serão descritos todos os elementos necessários para o desenvolvimento do videojogo, desde os alienígenas, os "*power-ups*", as moedas, as animações, toda a UI, bem como a música e os efeitos sonoros. Como também será demonstrada a criação da *APK*, que futuramente será usada para a integração com a *framework* "GamEducation" e para a realização de testes do videojogo no ambiente de Android.

O trabalho realizado desde a conclusão de Projeto I, visou a criação de um videojogo com bases fortes e escalável, de forma a poder incluir o mesmo na *framework* e para que esta possa ser usada em contexto real.

1.1 Objetivos

Os objetivos delineados para este projeto, centram-se na implementação de todas as fases de desenvolvimento do videojogo, conforme documentado no *GDD*.

Parte-se da validação do que foi desenvolvido para o protótipo do "Supernova QuEST", realizado na unidade curricular de Projeto I. Este será a base para o desenvolvimento completo do videojogo.

O primeiro objetivo, consiste na implementação da interface, as cenas correspondentes a todos os capítulos, as animações, os objetos de jogo necessários, bem como os scripts correspondentes responsáveis pelas suas funcionalidades e pela jogabilidade geral do jogo.

O segundo objetivo, é a criação de um executável do jogo para o sistema operativo Android, de forma a ser possível testar o jogo neste ambiente móvel usado pela *framework* educacional.

O terceiro e último objetivo, é a integração entre o "Supernova QuEST" e a "GamEducation", de forma a poder unir ambas as plataformas e permitir a comunicação entre elas, para que possa a vir a ser usada por docentes e alunos.

1.2 Estrutura do relatório

O presente documento encontra-se dividido em sete capítulos. No primeiro capítulo, onde é feito um enquadramento geral sobre o projeto, quais são os seus objetivos e a estrutura do documento; o segundo capítulo, destina-se a uma síntese geral sobre o que foi feito na primeira fase do projeto, desde os "*prefabs*" criados, as "*scenes*", os "*sprites*", entre outros; no terceiro capítulo, é mostrado em detalhe todo o desenvolvimento realizado para a construção do jogo, desde a implementação da *UI - User Interface*, aos "*game objects*" necessários, às animações, música e efeitos sonoros; o quarto capítulo, é referente aos testes realizados no ambiente de Android e quais os resultados obtidos; o quinto capítulo, refere-se às conclusões que podem ser retiradas da construção do projeto e a partir destas serão elaborados planos de possível trabalho futuro; o sexto capítulo, diz respeito às referências usadas para a construção do documento; o sétimo e último capítulo, diz respeito aos anexos onde é mostrado e explicado em detalhe quase por completo o código implementado no projeto.

2. Ponto de partida

O objetivo deste capítulo é resumir o trabalho realizado na primeira etapa do projeto, especificamente no que diz respeito ao protótipo do jogo "Supernova QuEST". Este capítulo torna evidente o progresso do projeto, fundamentado nas implementações previstas no "*Game Design Document*" do Projeto I.

Posteriormente, serão exibidos os "*prefabs*" criados, as "*scenes*" que correspondem aos capítulos do jogo, os scripts implementados para tornar o protótipo jogável e demonstrável, os "*sprites*" empregados, os "*game objects*" requeridos e, para finalizar, um resumo da "*gameplay*" apresentada na demonstração do Projeto I.

2.1 Prefabs

Em Projeto I, foram necessários nove "*prefabs*" na criação do protótipo.

No *Unity*, "*prefabs*" [1] são "*game objects*" pré-configurados que podem ser utilizados em várias cenas do projeto, permitindo a criação de instâncias desse mesmo elemento.

A utilização de "*prefabs*" oferece benefícios, como a prevenção da duplicação de código e recursos, a simplificação da aplicação de elementos compartilhados, bem como a aceleração na correção de erros e na implementação de alterações em larga escala.

Todos os componentes contidos na pasta de "*prefabs*" são mostrados na **Figura 1**. Isso inclui "Asteroid_1", "Asteroid_2", "Insetroid", "Green Shoot", "Yellow Shoot", Míssil Azul, Míssil Branco, Míssil Laranja e Míssil Vermelho.

Figura 1 - "*Prefabs*" do Protótipo

2.1.1 Asteroides

Foram criados dois "*prefabs*" de forma a representar o elemento asteroide no jogo: o "Asteroid_1" e o "Asteroide_2". A **Figura 2,** mostra a configuração final de ambos estes "*prefabs*".

Para além das diferenças ao nível do aspeto visual, estes elementos distinguem-se por outra caraterística que é o dano provocado. O "Asteroid_2", por possuir maiores dimensões será mais difícil de destruir, assim a sua vitalidade irá ter o valor de 20 unidades, ao passo que o "Asteroid_1" terá o valor de 10 unidades. Tendo ainda em conta as suas dimensões, o dano causado caso estes embatam com a nave do jogador também será diferente, o "Asteroid_2" causa um dano de 10 unidades enquanto o "Asteroid_1", apenas de 5 unidades. O componente "*PolygonCollider2D*", também teve de ser ajustado de forma diferente, o "Asteroid_1" tendo uma "*sprite*" mais circular tem outros pontos de colisão diferentes do "Asteroid_2", que apresenta um "*sprite*" com uma forma mais cilíndrica. As velocidades de queda são geradas de forma aleatória, tendo como base um intervalo de valores definidos no próprio "*prefab*". O mesmo acontece para a rotação dos asteroides, existe um valor configurável no qual se atribui um valor de modo que o "Asteroid_1" de dimensões menores, possa rodar de forma mais rápida que o "Asteroid_2". O script usado por ambos os "*prefabs*" é o "Asteroid", que tem como superclasse o "Enemy".

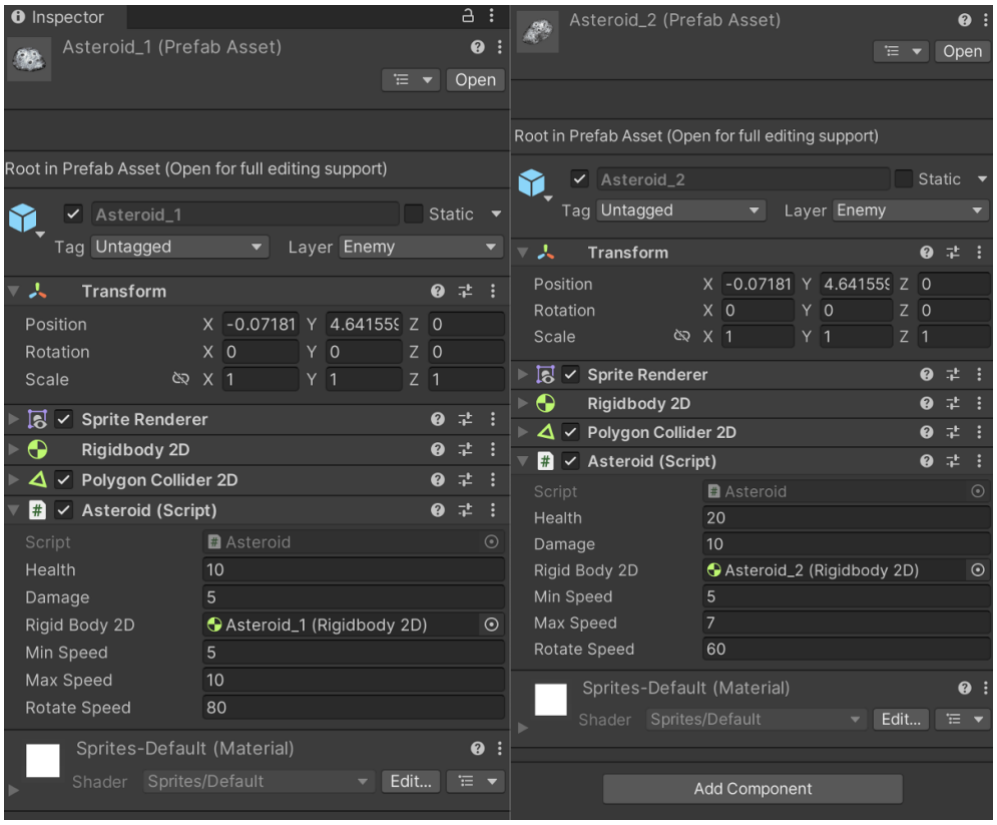

Figura 2 - "*Prefabs*" dos asteroides

2.1.2 Tiro do Inimigo

Para os ataques realizados pelos inimigos, foram criados dois “*prefabs*”: o “Green Shoot” e o “Yellow Shoot”. Os atributos do “*prefab*” “Green Shoot” são mostrados na **Figura 3**.

A única distinção entre os dois reside no valor do dano causado. Por um lado, o “Yellow Shoot” retira da nave do jogador o valor de 20 unidades de “*power*”, por outro lado, o "GreenShoot" apenas 10 unidades de “*power*”.

No protótipo, apenas foi utilizado o "*prefab*" "GreenShoot", pois é este que faz parte do único inimigo presente, o "Insetroid_2".

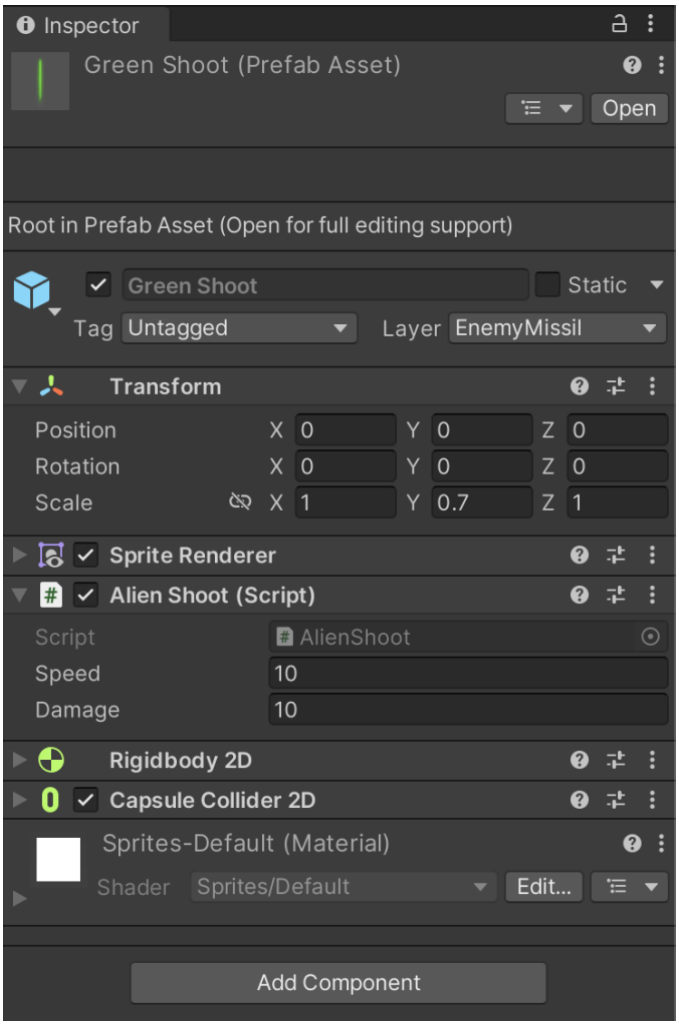

Figura 3 - "*Prefab*" do "Green Shoot"

2.1.3 Inimigo

Como já foi referido no subcapítulo anterior, o único inimigo implementado no protótipo foi o "Insetroid_02", cujo "*prefab*" está representado na **Figura 4**. Este "*prefab*", tem como base a utilização do script "Insetroid". Neste, são definidos vários parâmetros que indicam o comportamento do inimigo durante o jogo. Todos os parâmetros foram definidos, tendo como base os valores descritos no *GDD*.

Para o caso do "Insetroid_02", a "health" foi colocada com 20 valores e o "damage" com 10 valores. Caso este colida com a nave será esse o valor a ser retirado da mesma. O "rigidbody2D" foi utilizado de forma a aplicar física ao objeto, a "speed" com 4 valores, que define este objeto como sendo o inimigo mais lento do jogo, o "shootTime", que define a cadência de tiros disparados pelo "Insetroid", o "shootPoint", no qual é atribuído um ponto por onde este inimigo irá efetuar os seus disparos e por fim, o "shoot", ao qual é atribuído o "*prefab*" do "Green Shoot".

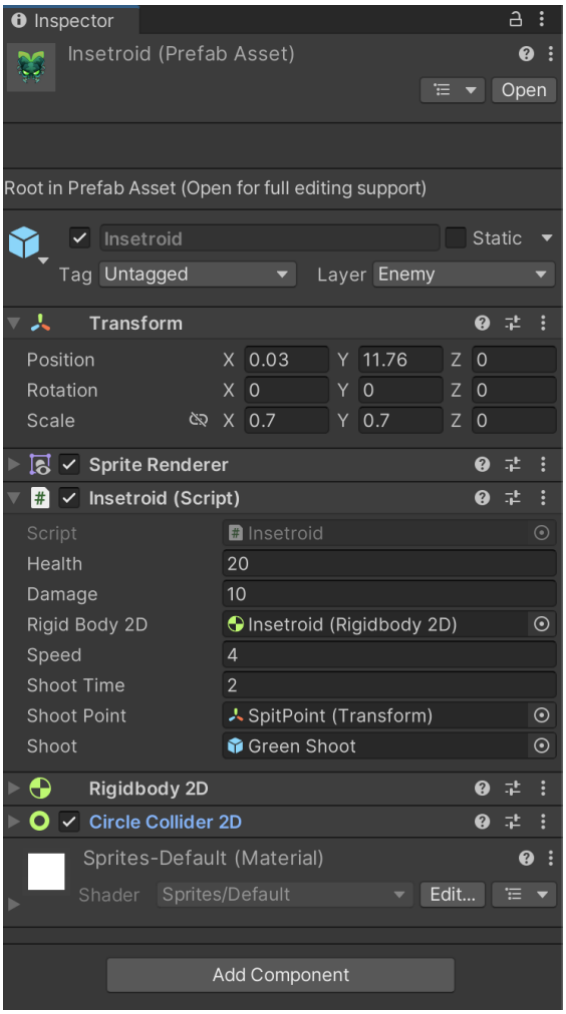

Figura 4 - "*Prefab*" do "Insetroid"

2.1.4 Mísseis das naves

Os últimos "*prefabs*" criados para o protótipo, foram os mísseis utilizados pelas naves controladas pelo jogador. A **Figura 5**, mostra a configuração do “*prefab*” "Missil Branco", associado as naves "Azure Horizon" e "Celestial Blue".

Os restantes mísseis, serão criados tendo como base o que foi estabelecido na Tabela 3 do relatório de Projeto 1.

O “*prefab*” "Missil Branco" possui o script "Missil", no qual foram definidos os comportamentos que este objeto deve ter durante o jogo. Para isso, são usados vários parâmetros de forma a poder controlar estes comportamentos. Nomeadamente, a "speed", o "damage" e o "rigidBody2D". Todos eles receberam os valores de acordo com o que foi estipulado no *GDD*.

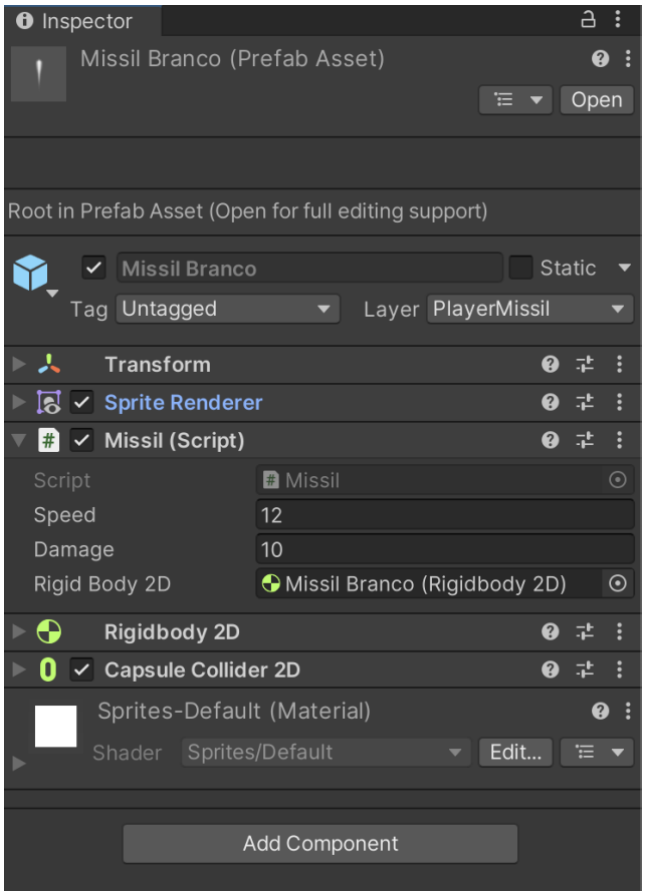

Figura 5 - "*Prefab*" do Míssil Branco

2.2 Scenes

O protótipo desenvolvido contém apenas uma única “*scene*”, à qual foi atribuído o nome de “Level1”. As “*scenes*” [2], funcionam de forma geral, como espaços destinados à construção e organização de todos os objetos e elementos visuais que compõem o jogo. Neste caso específico, o capítulo 1, corresponde ao planeta “Rosara”.

Na **Figura 6**, temos a ilustração da forma como a “*scene*” "Level1" está organizada. É possível identificar vários “Game Objects”, tais como: “MainCamera”, “VirtualCamera”, “Player”, “Background”, entre outros. O “RightPoint” e o “LetfPoint”, servem como pontos de referência, usados pelo “TargetGroup”, este associado a “VirtualCamera”. Por último, é possível identificar os objetos de "*spawn*", o "AsteroidSpawner" e o "AlienSpawner", são os responsáveis pela geração dos inimigos no jogo.

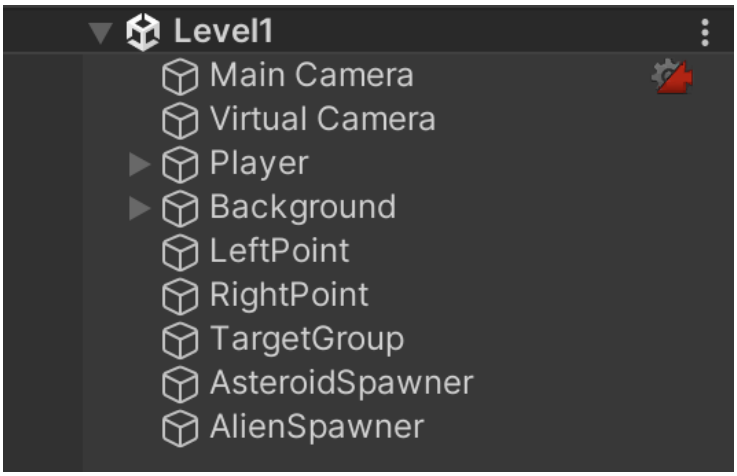

Figura 6 - "*Scene*" "Level 1"

2.3 Scripts

No que diz respeito aos scripts, foram desenvolvidos no total onze para a construção do protótipo. Neste capítulo, faz -se uma breve descrição de cada um, com ênfase nas suas principais funcionalidades e propósitos no contexto do jogo.

2.3.1 AlienShoot

O script "AlienShoot", é responsável por controlar os disparos feitos pelos inimigos. Este gere a movimentação, a deteção de colisões com o jogador e a sua destruição automática no momento que este sai do campo de visão do jogador. A implementação detalhada desta classe encontra-se ilustrada na **Figura 94**, no capítulo de Anexo I – Detalhes de programação.

2.3.2 AlienSpawner

O script "AlienSpawner", é o responsável pela criação de inimigos alienígenas no videojogo. Estes são criados de forma aleatória, fora do campo de visão do jogador, mais concretamente na parte superior. Estes vão descendo até que possam ser eliminados pelo jogador.

A estrutura do código desta classe encontra-se ilustrada na **Figura 95**, presente nos Anexos deste relatório.

2.3.3 Asteroid

A classe "Asteroid" herda da classe abstrata "Enemy", esta é a responsável por gerir o comportamento dos asteroides no jogo. A implementação pode ser consultada na **Figura 96**, presente no Anexo I – Detalhes de programação.

2.3.4 AsteroidSpawner

A classe "AsteroidSpawner", é responsável pela geração dinâmica dos asteroides no jogo. O seu funcionamento é bastante semelhante ao outro "*spawner*" presente no jogo, o "AlienSpawner", tem pequenas nuances, como o uso de um "*array*" para permitir a criação de vários tipos de asteroides. Ao contrário do "AlienSpawner", que no contexto do protótipo apenas gera um tipo de inimigo.

O código correspondente à implementação desta classe encontra-se ilustrado na **Figura 97**, no Anexo I – Detalhes de Programação.

2.3.5 Enemy

A classe "Enemy", ilustrada na **Figura 98** do Anexo I, é a superclasse usada para a criação dos vários inimigos presentes no jogo. A partir desta é possível criar subclasses de inimigos mais específicos que iram conter atributos e métodos mais específicos.

2.3.6 Insetroid

Esta classe denominada de "Insetroid", é uma subclasse de "Enemy" que define os atributos e os comportamentos de um inimigo específico do jogo.

A implementação detalhada desta classe encontra-se representada na **Figura 99** do Anexo I – Detalhes de Programação.

2.3.7 Missil

A classe "Missil", representa os mísseis disparados pela nave comandada pelo jogador. Esta gere a sua movimentação, como os danos causados em caso de colisão com os inimigos, sejam eles quais forem.

O código da implementação desta classe encontra-se apresentado na **Figura 100**, presente no Anexo I – Detalhes de Programação.

2.3.8 ParallaxEffect

O script “ParallaxEffect”, apresentado na **Figura 101** do Anexo I tem como principal função, movimentar os fundos presentes no videojogo. Deste pretende-se, que crie um efeito de profundidade e dinamismo.

2.3.9 PlayerController

A classe “PlayerController”, é a responsável por movimentar a nave controlada pelo jogador. A sua implementação encontra-se presente na **Figura 102,** do Anexo I – Detalhes de Programação.

2.3.10 PlayerShooting

O script “PlayerShooting**”**, representado na **Figura 103** do Anexo I, é responsável por gerir o sistema de disparo da nave controlada pelo jogador.

2.3.11 PlayerStats

A classe “PlayerStats”, que a implementação está representada na **Figura 104** do Anexo I – Detalhes de Programação, é responsável por gerir o “power” da nave do jogador.

2.4 Sprites

Os "*sprites*" utilizados no protótipo foram organizados em pastas individuais. Existe um total de nove pastas, com as seguintes identificações: "Aliens", "Asteroids", "Backgrounds", "Liders", "Missiles", "Planets", "PowerUp", "Shield" e "Ships". A **Figura 7**, ilustra essa estrutura.

Os "*assets*" utilizados no *GDD*, são parte de dois pacotes gratuitos disponíveis na "*Asset Store*" do *Unity*. O primeiro, de Brett Gregory, chama-se "*2D Space Kit"* [3], enquanto o segundo, de Larzes, é intitulado "*Free Stylized 2D Space Shooter Pack"* [4].

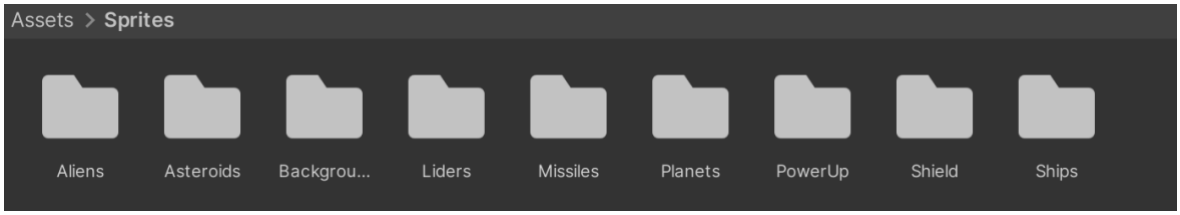

Figura 7 - Estrutura de pastas dos "*sprites*"

2.4.1 Aliens

No que diz respeito aos "*sprites*" utilizados para representar os "*aliens*", foram criados os seguintes: "Insetroid_02", "Insetroid_04", "Oculon" e "Omnivision", visíveis na **Figura 8**, abaixo.

No entanto, apenas o "Insetroid_02" foi utilizado no protótipo.

Durante a apresentação do mesmo, foi feito um comentário pelo professor Paulo Neves relativamente à aparência muito semelhante entre os dois "Insetroids", observação essa que será considerada em futuras iterações do projeto.

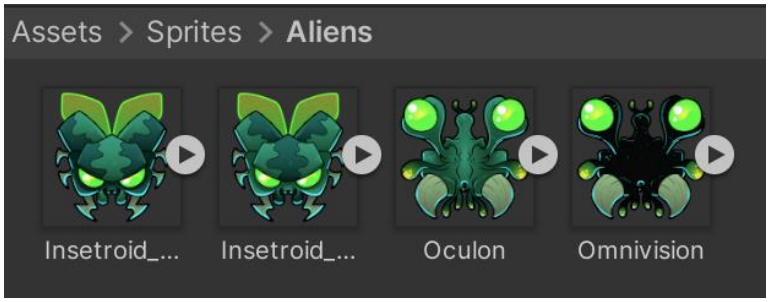

Figura 8 - "*Sprites*" dos "*Aliens*"

2.4.2 Asteroids

Estão presentes no protótipo do jogo, dois "*sprites*" distintos utilizados na representação dos asteroides: "Asteroid_1" e "Asteroid_2", como mostra a **Figura 9**.

Estes "*sprites",* diferem entre si tanto a nível visual como em termos de tamanho. O objetivo foi criar algum dinamismo neste tipo de inimigo.

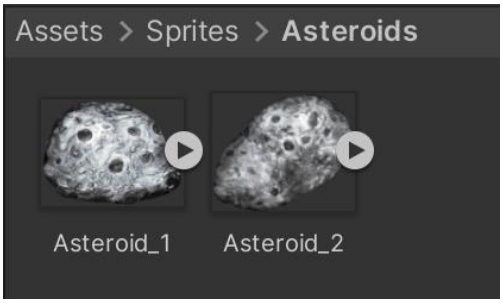

Figura 9 - "*Sprites*" dos Asteroides

2.4.3 Backgrounds

Na pasta de "*sprites"* de "*backgrounds"* encontram-se, para além dos fundos principais de cada um dos cinco capítulos: "Level 1", "Level 2", "Level 3", "Level 4" e "Level 5", os fundos auxiliares utilizados na criação do efeito de "*parallax"* implementado no protótipo do jogo. Estes elementos, denominados por "Clouds_1", "Clouds_2" e "Stars_1", são responsáveis pela formação das três camadas necessárias para alcançar o efeito visual pretendido. A **Figura 10** demonstra todos estes componentes.

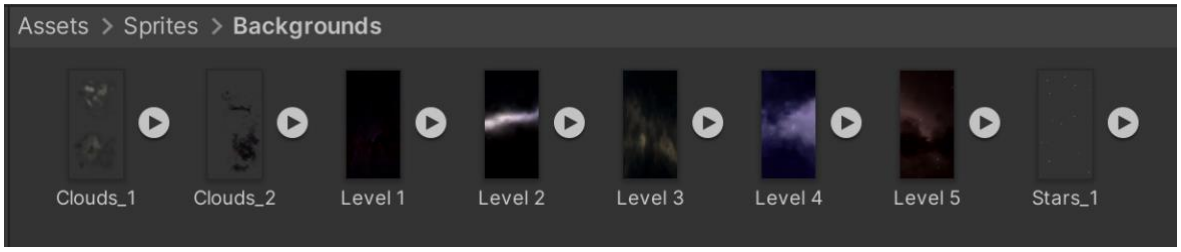

Figura 10 - "*Sprites*" dos "*backgrounds*"

2.4.4 Liders

Existem cinco tipos de "*sprites"* utilizados para representar os líderes: "*Glacithron"*, "*Krakthorn"*, "*Redhorn"*, "*Ruckhorn"* e "*Toxicaris"*, tal como ilustrado na **Figura 11**. Estes inimigos estarão presentes no final de cada capítulo do jogo. No entanto, no protótipo desenvolvido, nenhum deles foi incluído, uma vez que, foi apenas construída uma pequena demonstração da jogabilidade, sem a implementação de um final de capítulo.

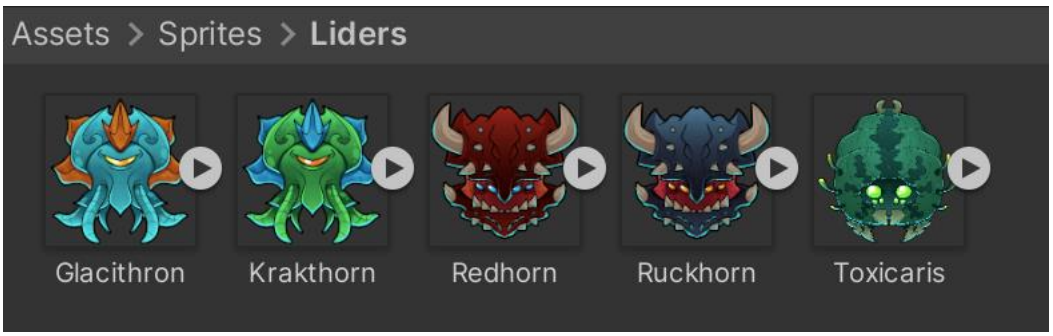

Figura 11 - "*Sprites*" dos "*Liders*"

2.4.5 Missiles

No protótipo, foram utilizados apenas dois tipos de "*sprites"* no que diz respeito aos mísseis: o "Big Projectile", utilizado nos disparos efetuados pelos "Insetroids", e o "Projectile Sharp", que serviu de base para os disparos realizados pela nave do jogador. Ambos podem ser observados na **Figura 12**.

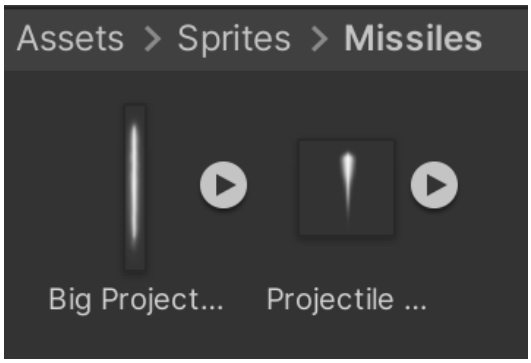

Figura 12 - "*Sprites*" dos mísseis

2.4.6 Planets

Foram recolhidos cinco "*sprites*" para representar os capítulos do videojogo, sendo que cada um corresponde à representação de um planeta, ao qual estará associado um capítulo da história do jogo. Os planetas são: "Aurelion", "Bronzara", "Celestia", "Rosara" e "Violurn", e podem ser visualizados na **Figura 13**.

Estes "*sprites*" não foram utilizados no protótipo, uma vez que, farão parte do ecrã de seleção de capítulos, conforme delineado no GDD.

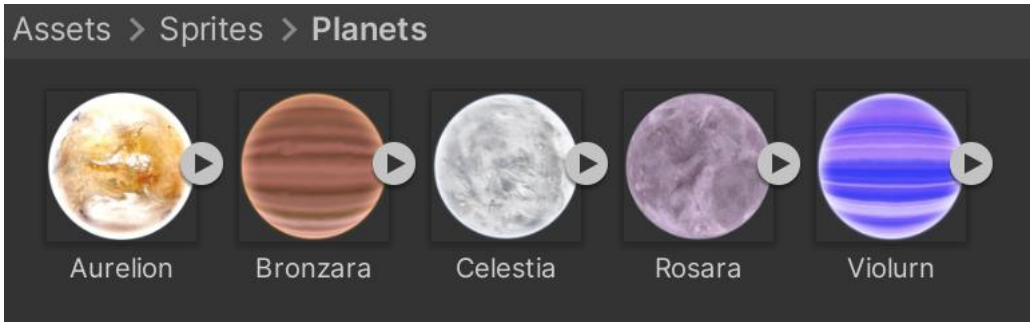

Figura 13 - "Sprites" dos planetas

2.4.7 PowerUps

Os "*sprites*" utilizados para os *power-ups* estão representados na **Figura 14**. São três no total, e cada um será associado ao respetivo "*power-up*": "Power Plus", "Shield Protection" e "Ultra Shot".

Nenhum destes foi implementado no protótipo, uma vez que este, se limitou a demonstrar uma jogabilidade simples, sem incluir esta componente do jogo.

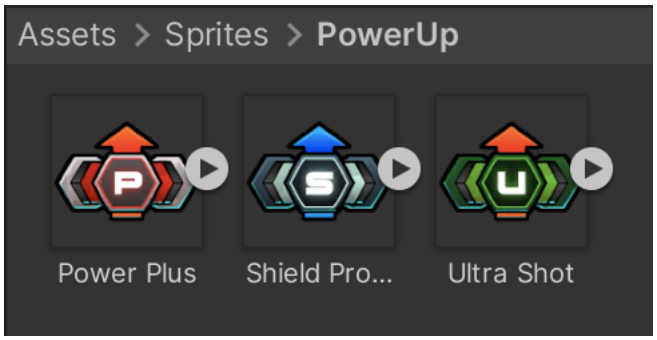

Figura 14 - "*Sprites*" dos *power-ups*

2.4.8 Shield

O “*sprite*” “Shield” apresentado na **Figura 15**, representa a proteção da nave do jogador contra qualquer tipo de impacto externo.

Esta proteção, é ativada através da recolha do “*power-up*” “Shield Protection”. Foi realizado um teste com este “*sprite*”, para garantir que se adapta corretamente a qualquer um dos “*sprites*” das naves incluídas no projeto.

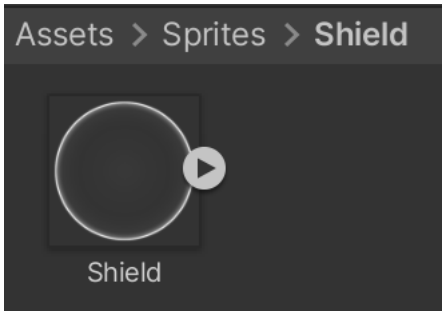

Figura 15 - "*Sprite*" da "*Shield*"

2.4.9 Ships

Existem oito “*sprites*” que representam as naves jogáveis: “Azure Horizon”, “Celestial Blue”, “Ember Leaf”, “Inferno Falcon”, “Midnight Voyager”, “Phoenix's Talon”, “Shadow Star” e “Verdant Flame”, conforme ilustrado na **Figura 16**.

No protótipo, foi utilizado o “*sprite*” da nave “Azure Horizon”, uma vez que, conforme definido no *GDD*, esta seria a nave inicialmente desbloqueada no jogo.

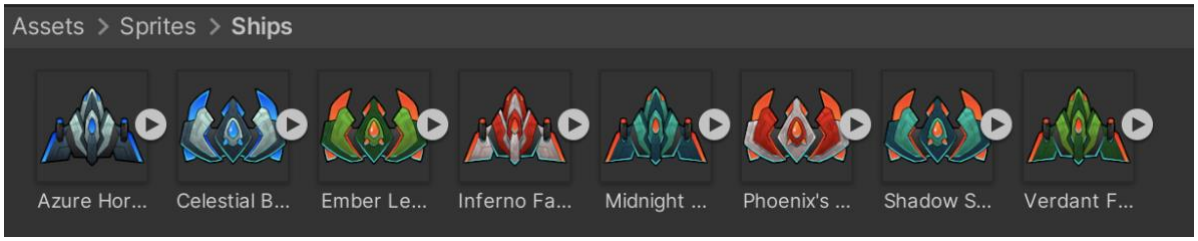

Figura 16 - "*Sprites*" das naves

2.5 GameObjects

No *Unity*, um "*GameObject*" [5] é uma parte fundamental de qualquer cena. Representa qualquer entidade presente no jogo, desde personagens, "*spawners*", itens colecionáveis, câmaras, entre outros, sejam estes visíveis ou não, como será possível verificar nos tópicos seguintes. Todos os "*GameObjects*" necessitam de componentes para poderem interagir com o jogo, sejam eles responsáveis pela gravidade, renderização, comportamento, entre outros.

2.5.1 Player

A **Figura 17** mostra o "GameObject" que representa a nave do jogador. Os seus principais componentes são três scripts: "Player Controller", "Player Shooting" e "Player Stats". Nestes scripts, é possível definir o "*missil*", o "shooting point", o "shooting timer" e o "powerMax". Todos estes atributos receberam os valores respetivos a nave "Azure Horizon".

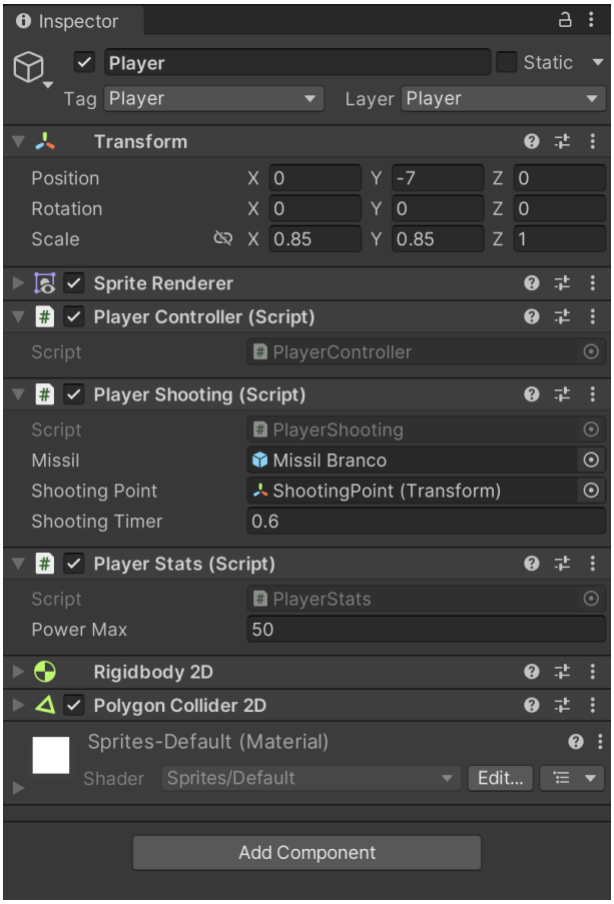

Figura 17 - "*GameObject*" "*Player*"

2.5.2 Asteroid Spawner

O "AsteroidSpawner" está representado na **Figura 18**, é um dos casos em que o objeto não aparece no ecrã de jogo. Contudo, o seu comportamento é bastante importante pois é responsável por gerar asteroides durante o jogo.

O único componente presente neste "*GameObject",* é o script "AsteroidSpawner", o qual recebe os vários "*prefabs*" dos asteroides que se pretende incluir no nível. Outro parâmetro, definido no script é o "spawner time", que determina o intervalo de tempo desejado para a criação de novos asteroides.

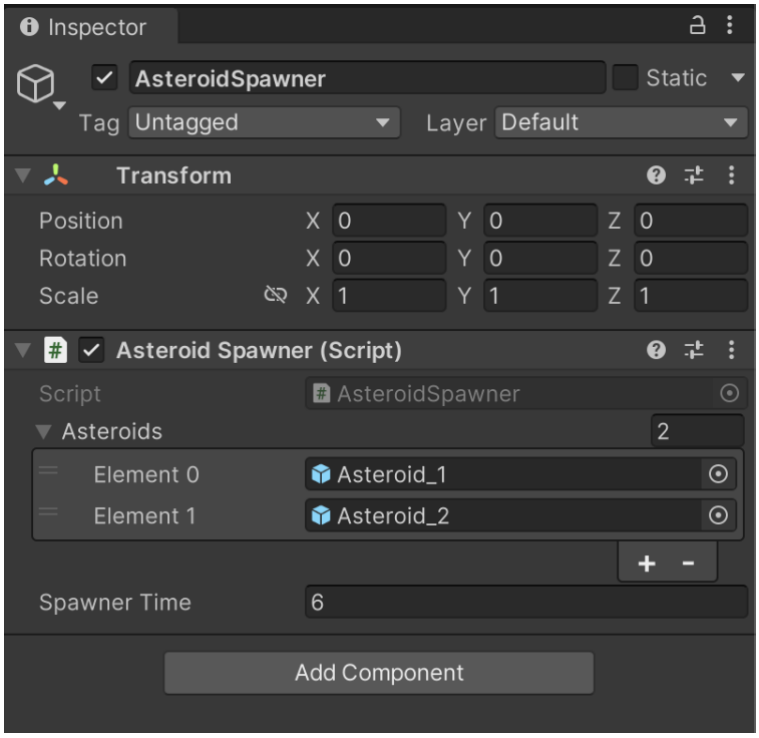

Figura 18 - "*GameObject*" "AsteroidSpawner"

2.5.3 Alien Spawner

Tal como o "AsteroidSpawner", este objeto não é visível na cena de jogo, e o seu comportamento é semelhante ao anterior. O único componente presente é também um script, denominado "AlienSpawner". Contudo, no contexto do protótipo, apenas foi feito "*spawn*" de um tipo de alienígena, o "Insetroid, assim, o parâmetro requerido pelo script não envolve um *array*, mas apenas um único "*GameObject*". Esta configuração do "*GameObject*" "AlienSpawner" pode ser verificada abaixo na **Figura 19**.

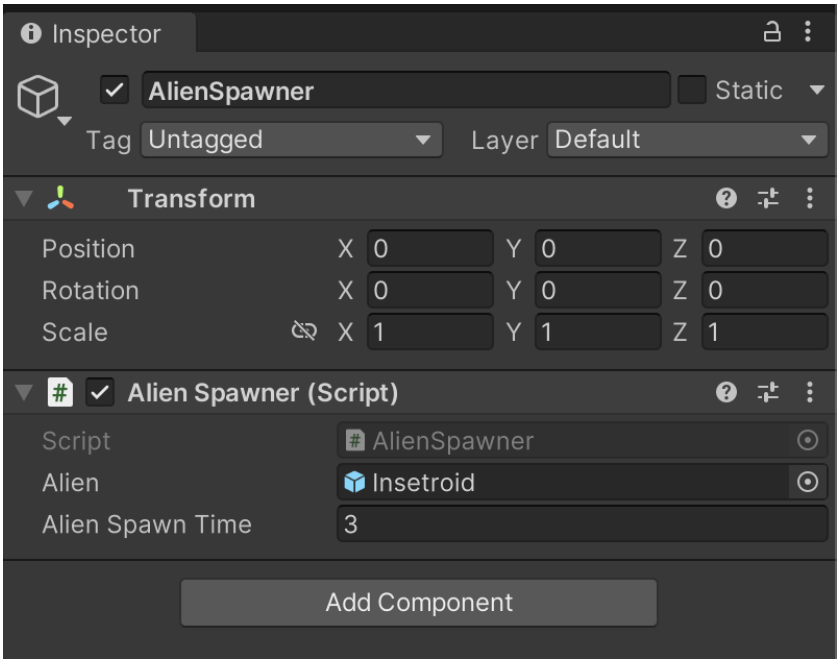

Figura 19 - "*GameObject*" "AlienSpawner"

2.5.4 Background

O "GameObject" denominado de "Background" é constituído por três "GameObjects" filhos: "Background_Base", "Clouds_1" e "Stars_1". Cada um destes tem como único componente o script "Parallax Effect", no qual apenas é necessário definir a "speed".

Para que estes "GameObjects" apresentem o comportamento esperado, é necessário atribuir velocidades diferentes a cada um deles, o que permite simular o efeito de "parallax". Esta implementação pode ser observada na **Figura 20**.

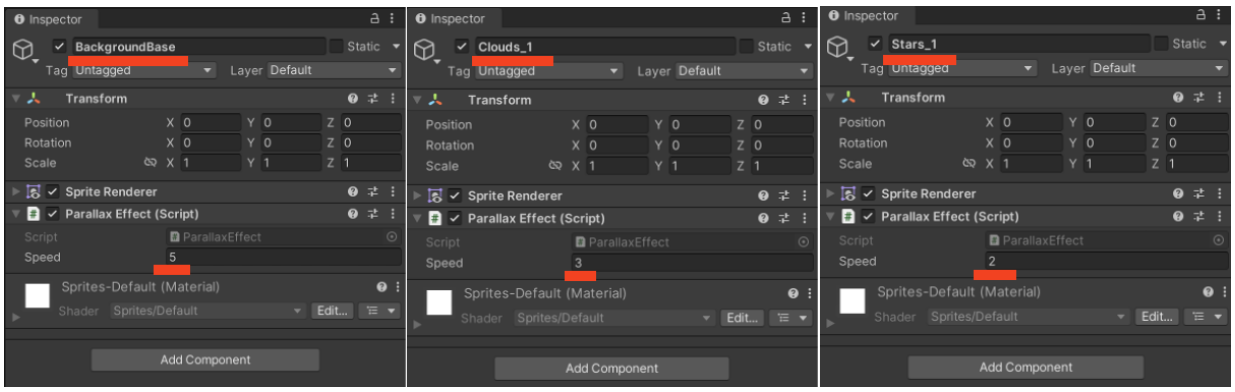

Figura 20 - "*GameObject's*" filhos do "*Background*"

2.6 Jogabilidade

O protótipo, apresenta uma jogabilidade simples, sendo já percetível a visão geral do jogo e o rumo do que se poderá tornar na sua versão final. Mesmo simples, foi possível validar através da demonstração do mesmo, várias funcionalidades importantes do videojogo, como por exemplo, a criação de asteroides e inimigos alienígenas, os disparos da nave, a própria movimentação da nave que ao ser tocada no ecrã de jogo por parte do jogador poderia ser movida em qualquer direção dentro dos limites e os disparos efetuados pelos inimigos, que em caso de colisão com a nave podem eliminá-la.

Para além destas funcionalidades, foi possível verificar que o protótipo já está preparado para se ajustar a qualquer tipo de dispositivo móvel, assegurando a mesma experiência de jogo a todo o tipo de jogador.

A **Figura 21**, apresenta duas capturas de ecrã feitas ao protótipo. É possível identificar em ambas que tal como mencionado anteriormente, os "*spawns*" já se encontram em funcionamento, pois é possível identificar dois tipos de inimigo na figura, assim como, o sistema de disparos tanto por parte da nave como do inimigo “Insetroid_02”.

Figura 21 - Jogabilidade do protótipo

3. Desenvolvimento

Neste capítulo são apresentados os principais aspetos do desenvolvimento do jogo “Supernova QuEST”, a partir da base construída no protótipo desenvolvido na unidade curricular de Projeto I.

Entre os elementos abordados incluem-se a construção do *HUD* – *Heads-up dispaly* do jogo, a implementação de animações, o desenvolvimento dos inimigos, tanto os alienígenas como os seus líderes, a criação dos capítulos do jogo, a implementação de um manager e a condição de vitória necessária para a conclusão de cada capítulo do jogo, são ainda descritas todas as componentes da interface planeadas no *GDD*, a criação dos *power-ups*, as moedas que funcionam como recompensas no jogo e, por fim, a integração da música e dos efeitos sonoros, que conferem o toque final à experiência do jogador.

3.1 HUD

O *HUD*, é a interface onde são apresentadas as informações relevantes do jogo, neste caso o nível de “power” da nave e o número de moedas coletadas pelo jogador.

O subcapítulo seguinte, descreve a implementação da barra de “*power*” da nave do jogador. Quanto a implementação do contador de moedas, esta será abordada no capítulo dedicado apenas a esse tema.

3.1.1 Barra de “*power*” da nave do jogador

A barra de “*power*”, permite ao jogador verificar rapidamente o nível de “power” que a nave possui atualmente. Como é óbvio, caso esta barra fique sem qualquer preenchimento, significa que a nave ficou sem “power”, sinal de que o jogo terminou.

A ideia definida no *GDD* está ilustrada na **Figura 22**. É possível verificar que a barra apresenta um formato retangular, preenchido por uma cor que indica o nível de “*power*” atual da nave. À esquerda da barra encontra-se o ícone de um trovão, que a identifica como sendo a barra de “*power*” da nave.

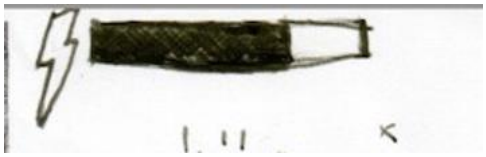

Figura 22 - Esboço da barra de "*power*" da nave

A implementação da barra de "*power*", iniciou-se com a colocação de um novo elemento de "*Image*" à "*scene*" "Level1". Com a criação desta imagem foram automaticamente criados outros dois "GameObjects": o "*Canvas*" [6] e o "*EventSystem*".

O "*Canvas*", é o objeto onde serão mostrados todos os elementos pertencentes à *UI*, funciona como uma "*layer*" onde estes são desenhados e organizados. É neste, que são inseridos elementos como o: *HUD*, botões, *"pop-ups"*, entre outros.

Por sua vez, o "*EventSystem*", é responsável pela interação do jogador com os elementos da interface, como por exemplo, o registo de cliques, toques ou outras formas de input. Neste projeto, faz-se uso do novo "*Input System"*, o que faz com que seja necessário substituir o módulo padrão chamado de "*Standalone Input Module*" pelo módulo do novo sistema, o "*InputSystemUIInputModule*".

Esta substituição, pode ser realizada através do "*EventSystem*", selecionando a opção "*Replace with InputSystemUIInputModule*", como ilustrado na **Figura 23**.

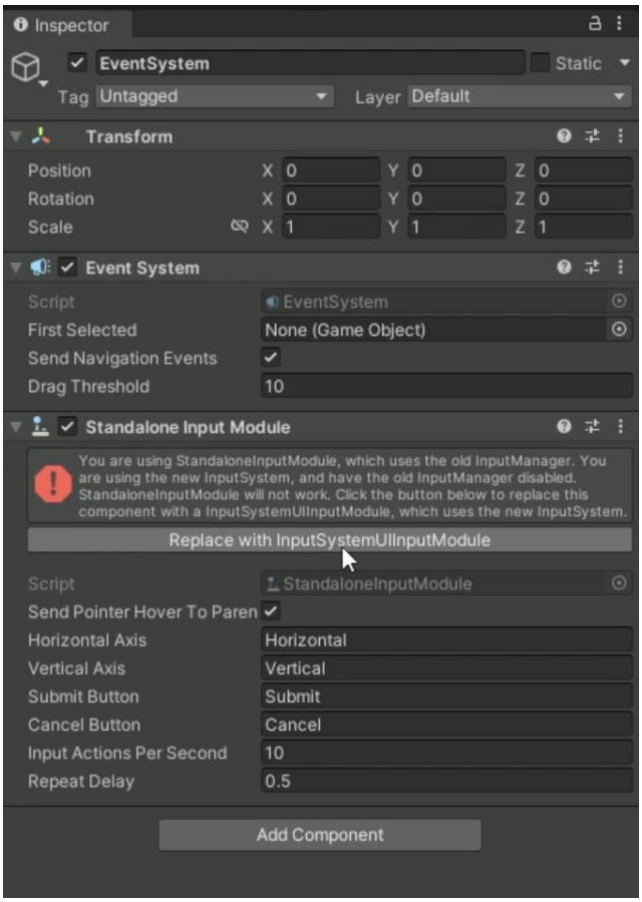

Figura 23 - Configuração do "*Event System*"

Outro aspeto a ter em consideração, é de alguma forma garantir que o "Canvas" e todos os objetos pertencentes a este, se consigam adaptar de forma correta a qualquer tipo de resolução de ecrã. De forma a resolver este problema, foi necessário selecionar o objeto "Canvas" e neste, na opção "Canvas Scaler", alterar o parâmetro "*UI Scale Mode* " para "*Scale with Screen Size*".

Dado que o "Supernova QuEST", será jogado em modo "*Portrait*", foi definida como resolução base a “1080x1920” no parâmetro de “*Reference Resolution*”. Por último, ao parâmetro de "Match" foi atribuído o valor de 0.5, de forma a assegurar que a adaptação da *UI* seja proporcional, tanto em largura como em altura.

Esta foi a configuração necessária, para que os elementos que estejam presentes na *UI* se possam ajustar as diferentes resoluções e tamanhos de ecrã dos dispositivos móveis, assegurando a visibilidade em todos eles. Esta configuração está presente na **Figura 24**.

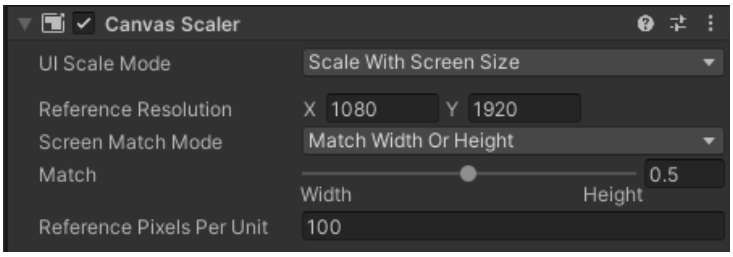

Figura 24 - Configuração "*Canvas Scaler*"

Após a conclusão da configuração do "Canvas" e do "EventSystem", passou-se para a criação e edição do objeto "*Image*" adicionado a “*scene*” do capítulo, será este objeto que irá aportar a barra de "*power*".

Neste momento, a imagem já se adapta a qualquer resolução de ecrã. Contudo, ao posicioná-la no canto superior esquerdo como previsto no *GDD*, verificou-se um pequeno problema, que em simuladores com ecrãs mais pequenos, esta imagem poderia aparecer cortada, que em comparação com os simuladores com ecrãs de maior dimensão, ela se aproximaria do centro do ecrã, ao invés de se manter na posição como era pretendido.

A solução para este problema, passou pela configuração correta da propriedade "*Anchor Presets*" [7] do objeto "*Image*". Esta por predefinição encontra-se centralizada, como se pode verificar na **Figura 25**.

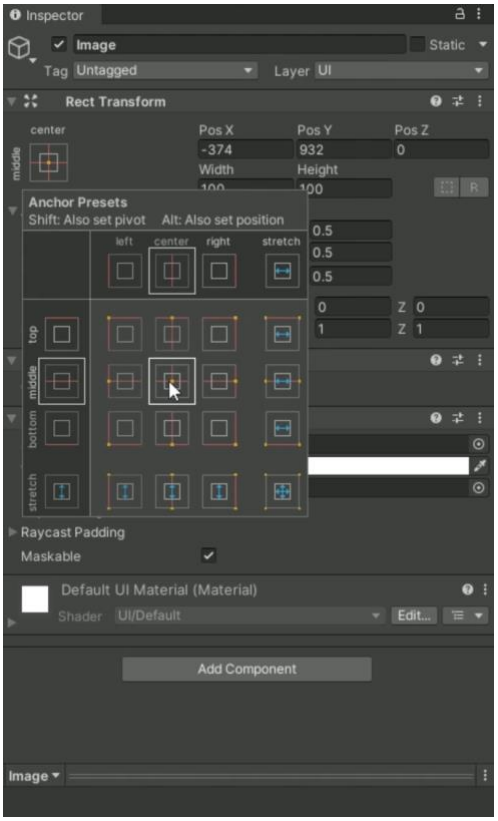

Figura 25 - Configuração do "*Anchor Presets*" por predefenição

A **Figura 26**, mostra a solução aplicada a imagem, na qual se fez o ajuste para que o pivô da âncora se posicionasse no canto superior esquerdo do ecrã.

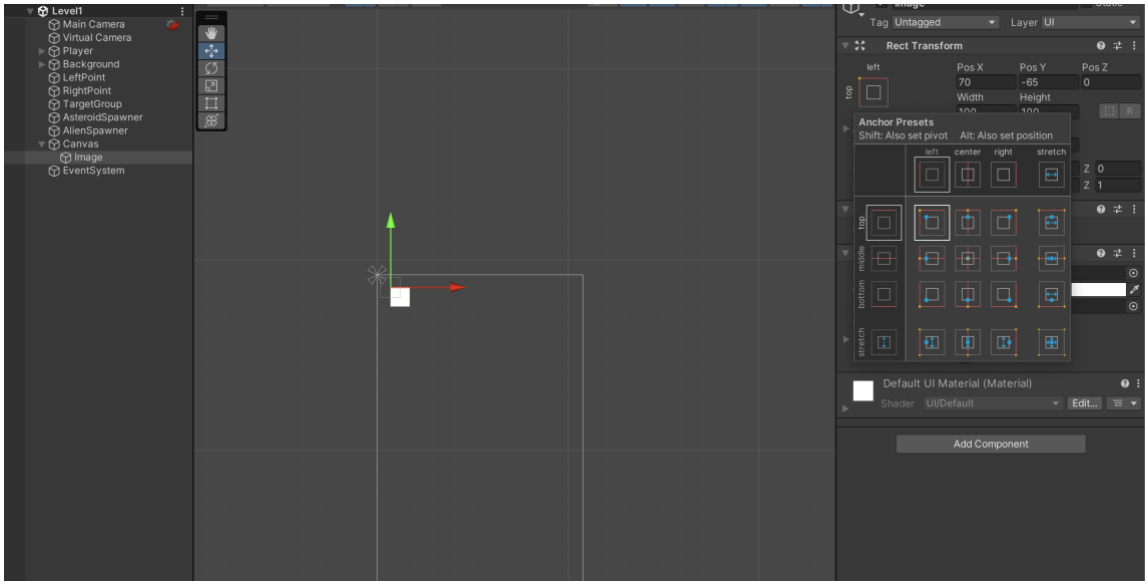

Figura 26 - Configuração "*Anchor Presets*" final

Agora que o “*game object*” “*Image*”, já se encontra na posição correta sem sofrer qualquer tipo de alterações, dá-se início à implementação da barra dentro deste mesmo objeto.

No atributo “*Source Image*” é atribuído o “asset” “PowerBarImage”. Contudo, foi possível verificar, que esta barra ficava deformada quando aplicada em resoluções de ecrã diferentes.

Para resolver este problema, foi aplicada a técnica "*nine-slice*" [8]. Esta técnica, consiste em fazer a divisão de uma imagem em nove pedaços, permitindo assim, que os pedaços referentes aos cantos não sofram qualquer tipo de distorção, sendo apenas os pedaços centrais e laterais a sofrer possíveis adaptações quanto ao seu tamanho. A aplicação desta técnica, pode ser observada na **Figura 27**.

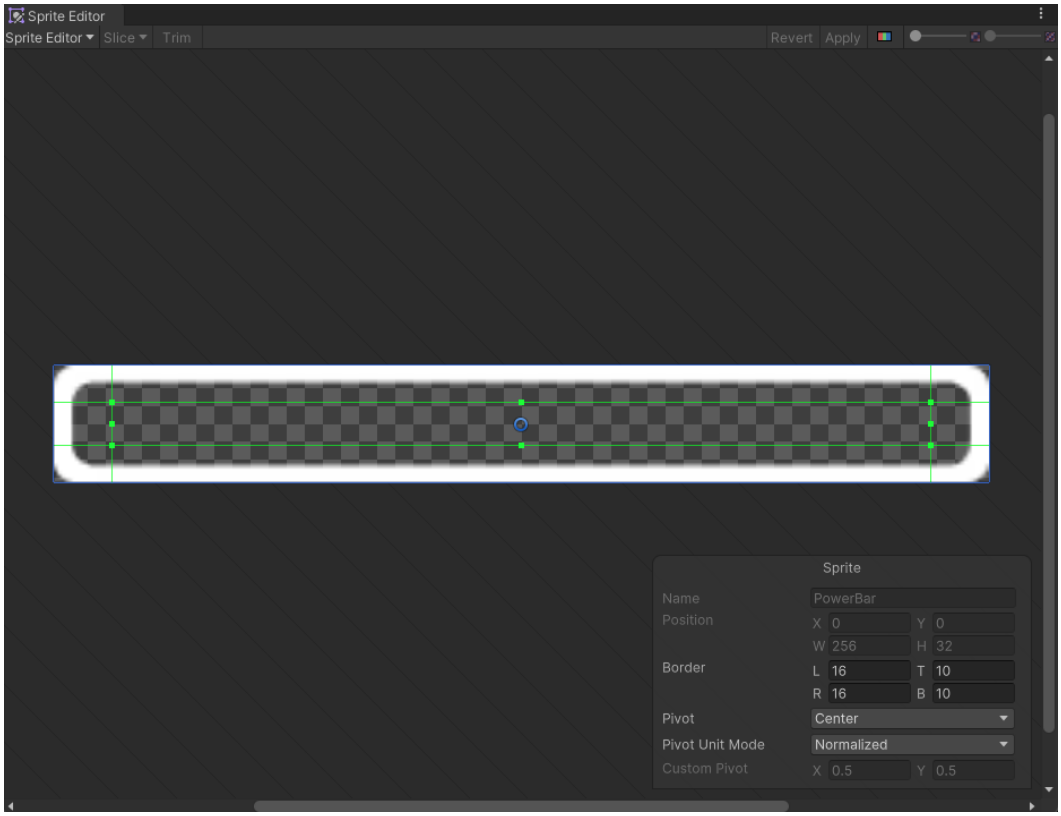

Figura 27 - Técnica "*nine-slice*"

Aplicada a técnica de “nine-slice”, é apenas necessário fazer a alteração do atributo "*Image Type*" para "*Sliced*".

Com esta configuração, a barra de "power" poderá ser usada em qualquer resolução de ecrã, fazendo com que não exista mais nenhum tipo de distorção.

A **Figura 28**, mostra a configuração final da aplicação desta técnica.

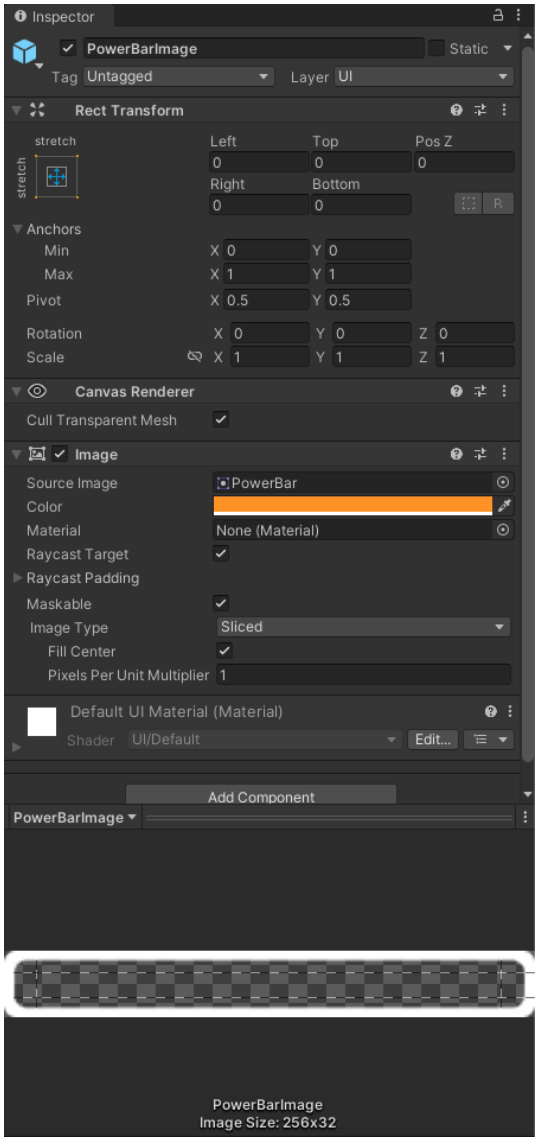

Figura 28 - Configuração final da "*PowerBarImage*"

Com o invólucro da barra de "power" já criado, passa-se para a criação do elemento responsável pelo seu preenchimento.

Foi adicionado um "*game object*" do tipo "*Image*" com o nome "PowerBackgroundImage". Este, será o elemento pai de outro objeto do mesmo tipo chamado de "PowerFill".

Para que este "PowerFill" funcione como uma imagem de preenchimento, foi necessário alterar a propriedade "Image Type", para "Filled", desta forma ela irá obter o comportamento desejado.

Toda esta configuração pode ser vista na **Figura 29**.

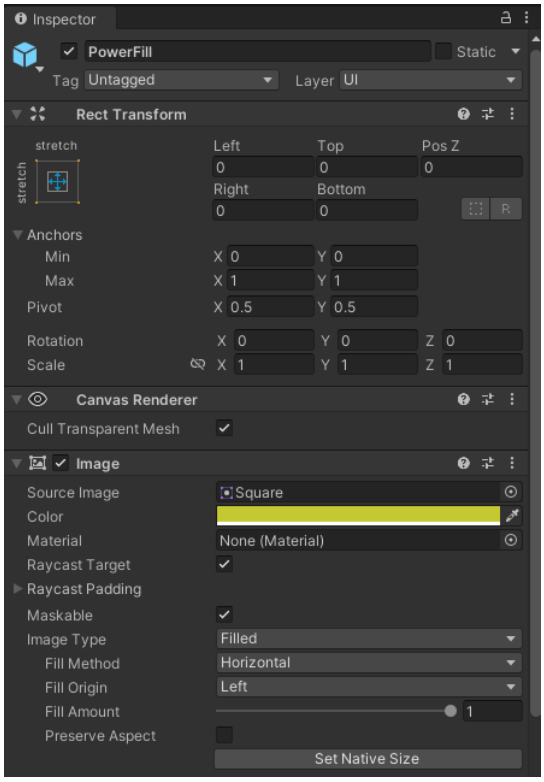

Figura 29 - Configuração do "*PowerFill*"

Foi feita a inclusão de um novo atributo denominado de "powerFill", na classe "PlayerStats" do tipo "*GameObject*". Será através deste que será possível fazer alterações ao atributo "fillAmount", no qual será espelhado o valor de “power” atual da nave. A implementação detalhada deste processo, encontra-se descrita no Anexo I – Detalhes de Programação na **Figura 105**.

Para que a barra de "power" se assemelhasse o mais possível ao proposto no *GDD*, foram realizados mais uns pequenos ajustes, sendo um deles a colocação do ícone do trovão no seu lado esquerdo, como apresentado na **Figura 30**.

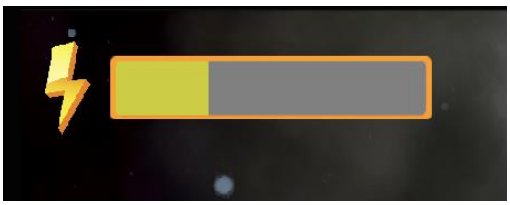

Figura 30 - Representação final da barra de "*power*"

3.2 Animações

Neste capítulo, será abordada a implementação de animações no jogo.

Foram criados dois tipos de animações: as de explosões, que são despoletados quando os inimigos são destruídos; e uma animação que identifica quando algo é atingido, como acontece com a nave do jogador, os alienígenas e os líderes.

3.2.1 Explosões

Para criar as animações de explosão no jogo, foi feita uma pesquisa por "*assets"* que pudessem servir este propósito. O package que melhor se enquadrou foi o "*explosions-1*" [9], encontrado no site "*opengameart.com"* e da autoria de Graul98. Este package inclui três "*assets"* de explosões com cores diferentes, apresentados na **Figura 31**, que serão atribuídos a diferentes objetos no jogo: a explosão verde será associada aos alienígenas, a laranja aos asteroides e a roxa aos líderes.

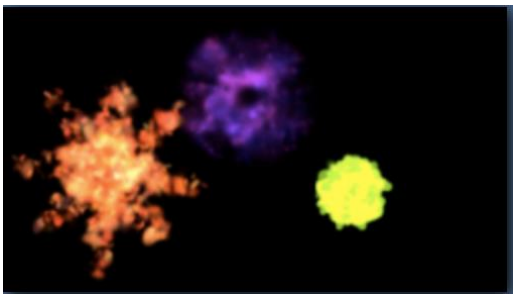

Figura 31 - Conteúdo do package explosions-1

Feito o download do package, foi feita a adição do mesmo a pasta de "Assets" do projeto. Na **Figura 32,** podemos verificar um exemplo de uma das explosões, no caso a verde.

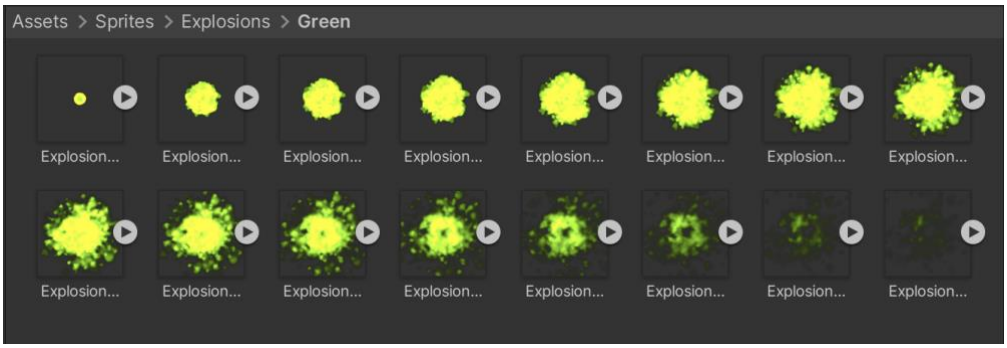

Figura 32 - Explosão verde

Com as “*sprites*” presentes no projeto, foi criado um “*GameObject*” denominado de “Explosion_Green”.

Selecionando este “*GameObject*”, pode-se dar início a criação da animação, acedendo ao menu: “*Window* > *Animation* > *Animation*”, conforme apresenta a **Figura 33**.

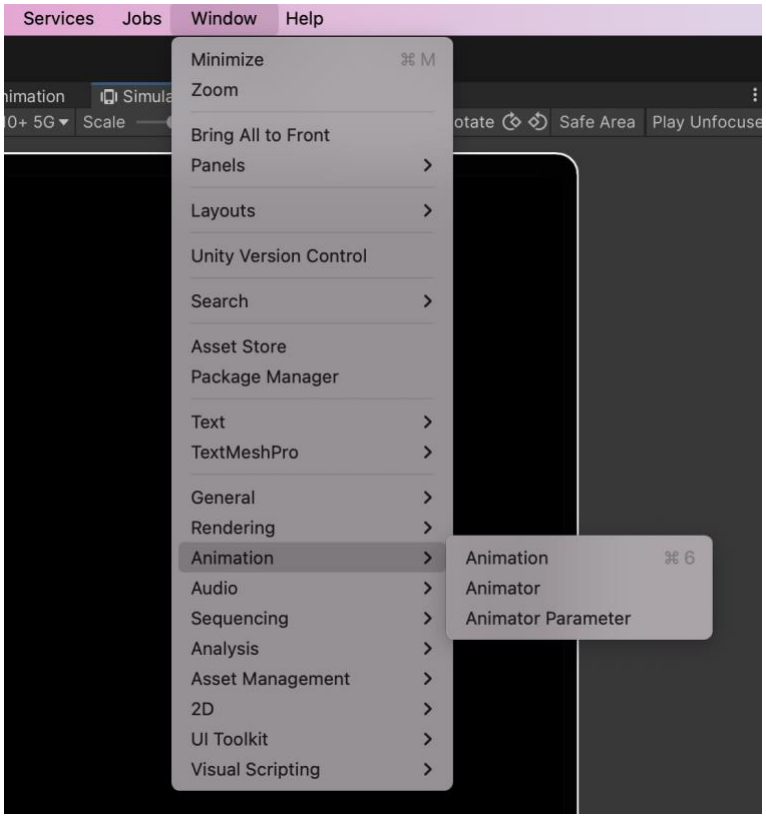

Figura 33 - Menu "Animation"

Clicando no botão de "*Create*", são criados dois componentes: “*Explosion.anim*”, correspondente à animação e o “*Explosion.controller*”, que tal como a sua extensão indica, funciona como controlador dessa mesma animação. Estes estão presentes na **Figura 34**.

Figura 34 - Elementos de animação

Na janela "Animation", deve-se selecionar as "*sprites*" que compõem a animação e arrastá-las para a "*timeline*", definindo assim a sequência de imagens que irá dar origem à animação. Para uma melhor compreensão, esta ação está representada na **Figura 35**.

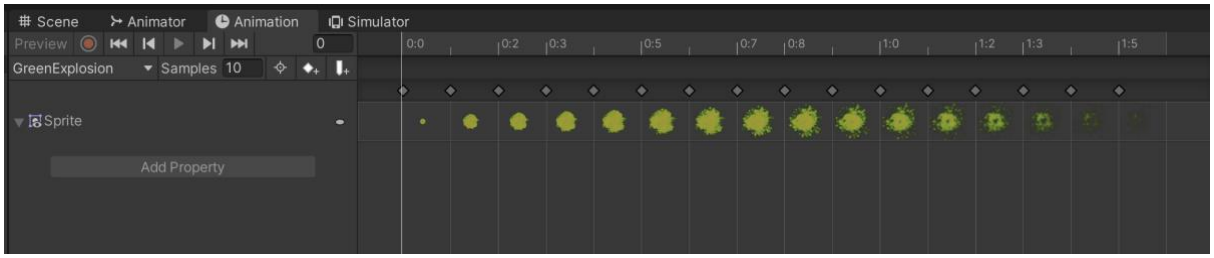

Figura 35 - Criação da animação de explosão verde

Feitos alguns testes, verificou-se que a animação estava demasiado rápida, não sendo quase percetível o processo da explosão. De forma a resolver este problema, foi feito o ajuste de 60 para 10 *FPS – Frames per Second*, no atributo de "*Samples*" da animação.

Verificou-se também, que a animação se repetia de forma contínua, pois o atributo "*Loop Time*" estava ativo. A configuração final está representada na **Figura 36**.

Figura 36 - Inspetor "Green Explosion"

Para que a animação pudesse ser destruída quando a sua reprodução terminasse, foi necessária a criação de um novo script chamado de "Explosion". No qual, é definido um "*delay*" de dois segundos, permitindo assim que este processo seja possível de realizar com sucesso.

A partir do "*GameObject*" "Explosion", foi criado o respetivo "*prefab*" *no caso "Explosion_Green".* Foram associados o componente "*Animator*" e o script "Explosion", conforme é possível observar na **Figura 37**.

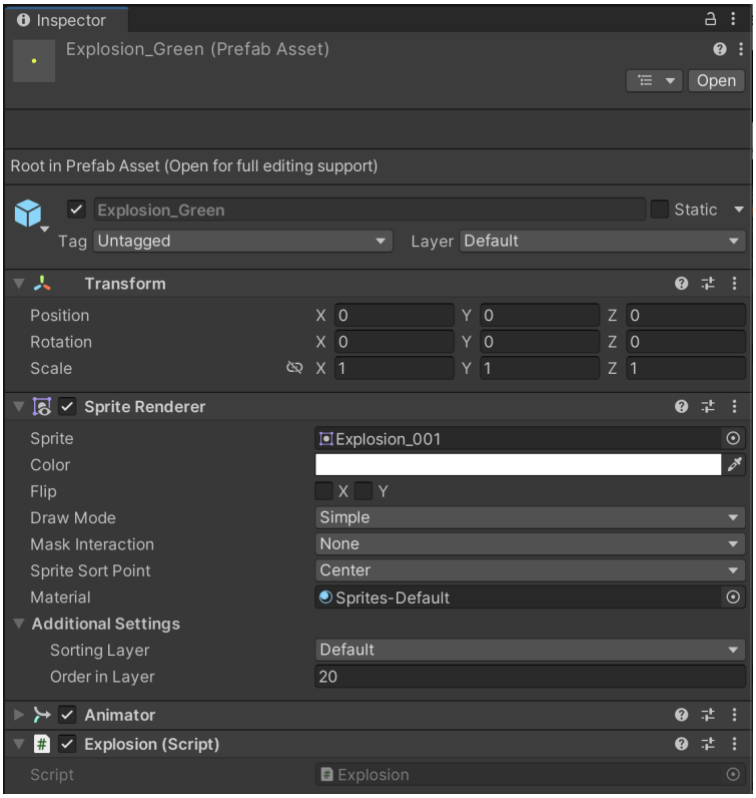

Figura 37 - "Prefab" "Explosion_Green"

Com o "*prefab*" criado, fez-se a ligação à classe "Enemy", através da criação de uma variável do tipo "*GameObject*", chamada de "explosionPrefab". Desta maneira, todos os inimigos poderão agora receber a sua respetiva animação de explosão.

Como exemplo, na classe "Insetroid", nos métodos "OnTriggerEnter2D()" e "Destroyed()", é instanciado o "*GameObject*" correspondente à respetiva animação, neste caso, o "Explosion_Green". Esta implementação encontra-se documentada no Anexo I – Detalhes de Programação, na **Figura 106**.

O resultado da implementação desta animação está presente na **Figura 38**, onde se pode observar que um dos "Insetroid_02" colidiu com a nave do jogador. Essa colisão, despoletou a animação "*Explosion_Green*" associada ao alienígena previamente implementada.

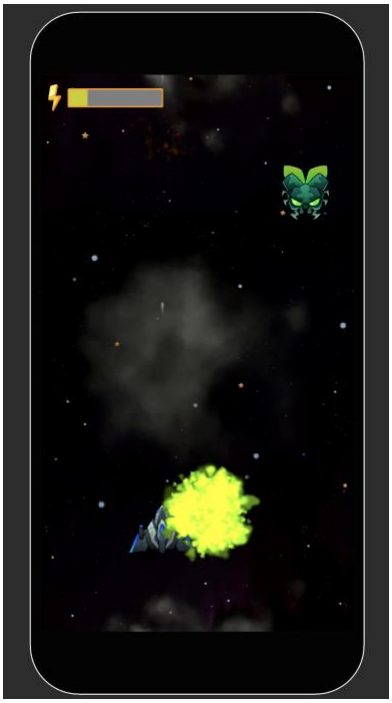

Figura 38 - Animação da explosão verde

Na **Figura 39**, apresenta-se a mesma implementação, mas neste caso aplicada à explosão de um dos asteroides presentes no jogo eliminado pela nave do jogador. A explosão utilizada é de cor laranja e recorre a "*sprites*" distintos dos usados na explosão dos "Insetroids". Contudo, todo o processo de criação foi idêntico.

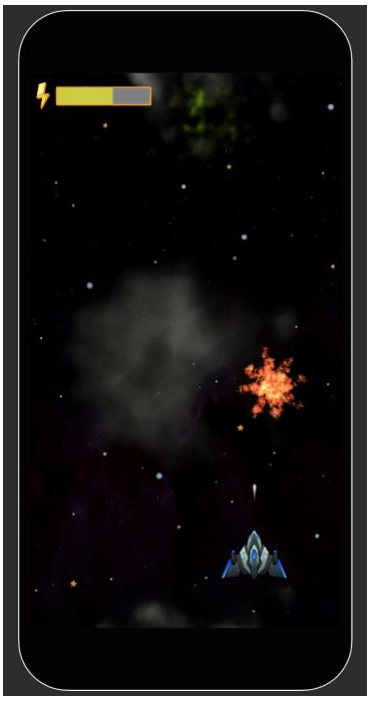

Figura 39 - Animação da explosão de um asteroide

3.2.2 Dano

Para além da animação de explosão, foi também desenvolvida uma animação com o objetivo de fornecer feedback visual sempre que um dos objetos do jogo sofre dano, por exemplo, quando um dos alienígenas é atingido ou quando a nave do jogador é danificada.

O primeiro passo, foi criar os "*assets*" referentes a esta animação. Para tal, fez se uso das imagens originais das naves, que foram modificadas para que a sua única cor fosse um vermelho escuro, com um nível de transparência razoável, para que desse o efeito pretendido a animação. A **Figura 40** ilustra essa edição feita aos "*assets*" das naves.

Figura 40 - Assets de "Hitted"

Ao selecionar o "Player", na "*scene*", seguiu-se o mesmo processo para criar a animação de "*explosion*" para a de "*hitted*".

Contudo, esta animação foi dividida em dois estados: o estado "*idle*", que representa a inatividade do jogador, e o estado "*hitted*", que é ativado quando a nave é atingida.

A animação "*idle*" apresenta a nave no seu estado normal e permanece estática. Esta animação permite a transição entre o estado normal da nave e o estado de "*hitted*", e vice-versa.

Na criação da animação "*idle*", foi colocado o mesmo "*sprite*" na "*timeline*" no início e no fim, como é possível observar na **Figura 41**.

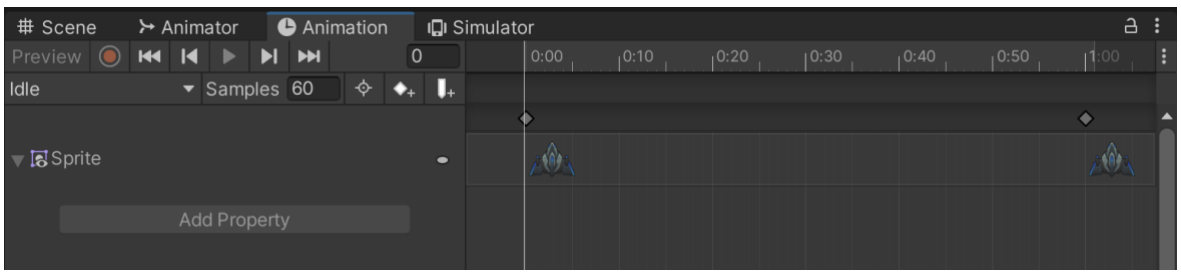

Figura 41 - Animação de "Idle"

O passo seguinte será criar a animação "*hitted*", nesta serão incluídos os "*sprites*" criados para este fim, distribuídos num intervalo de 30 *FPS*. Esta redução na taxa de “*frame rate*, tem como objetivo prolongar visualmente o efeito da animação. Importa relembrar que a opção “*Loop Time*” deve permanecer desativada. A **Figura 42**, mostra a implementação desta animação.

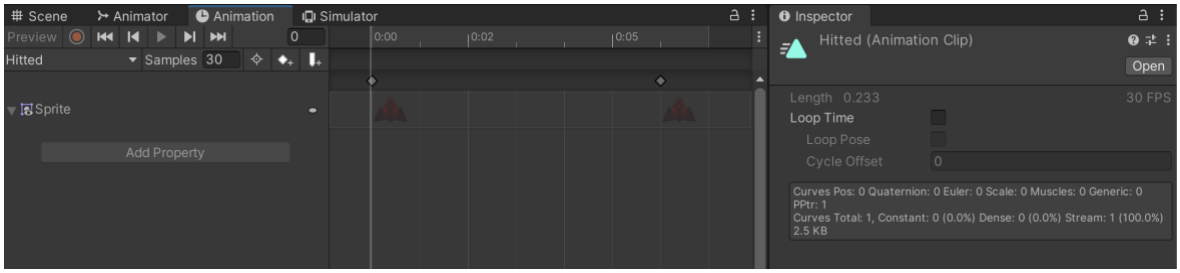

Figura 42 - Animação de "Hitted"

Segue-se agora, o passo que difere da implementação desta animação em relação à anterior, a “Explosion”.

Na aba “*Animator*”, encontram-se as duas animações criadas: "Idle" e "Hitted". É nesta janela que serão definidas as transições necessárias entre ambas, de forma a permitir a mudança de estado consoante o comportamento desejado.

A transição de "Idle" para "Hitted" será acionada quando a nave do jogador for atingida por algum inimigo. Para isso acontecer, é necessário criar um parâmetro do tipo “Trigger”, este é o responsável por ativar a passagem entre os dois.

Uma vez que a transição é controlada por um “*trigger*” e não por tempo, é importante desativar a opção “*Has Exit Time*” que vem ativa por predefinição nas definições da transição. Adicionalmente, o “*Transition Duration*” deve ser definido como zero, de modo a garantir uma transição imediata assim que o “*trigger*” for acionado. Toda esta implementação, está presente na **Figura 43**.

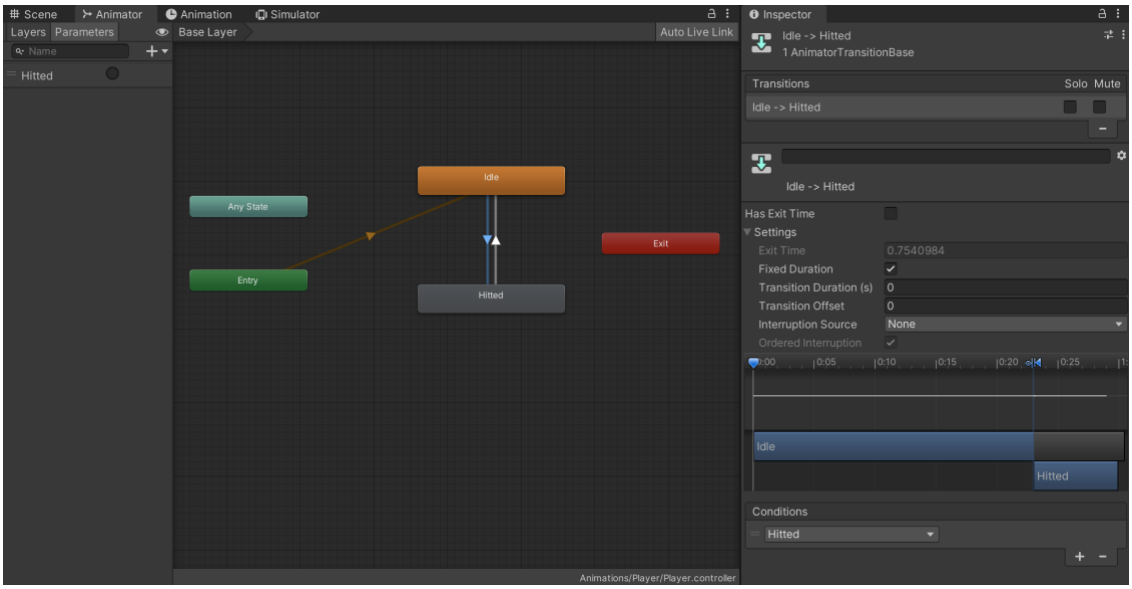

Figura 43 - Transição entre "Idle" e "Hitted"

Posteriormente, é necessário efetuar a transição entre o estado “Hitted” para o estado “Idle”. Para que isto aconteça, recorre-se à opção “*Has Exit Time*”, em vez da utilização de um parâmetro do tipo “*Trigger*”, como foi feito na transição anterior, configurando-se o “*Exit Time*” com o valor 1 e a “*Transition Duration*” para 0 segundos, de forma a assegurar uma transição imediata.

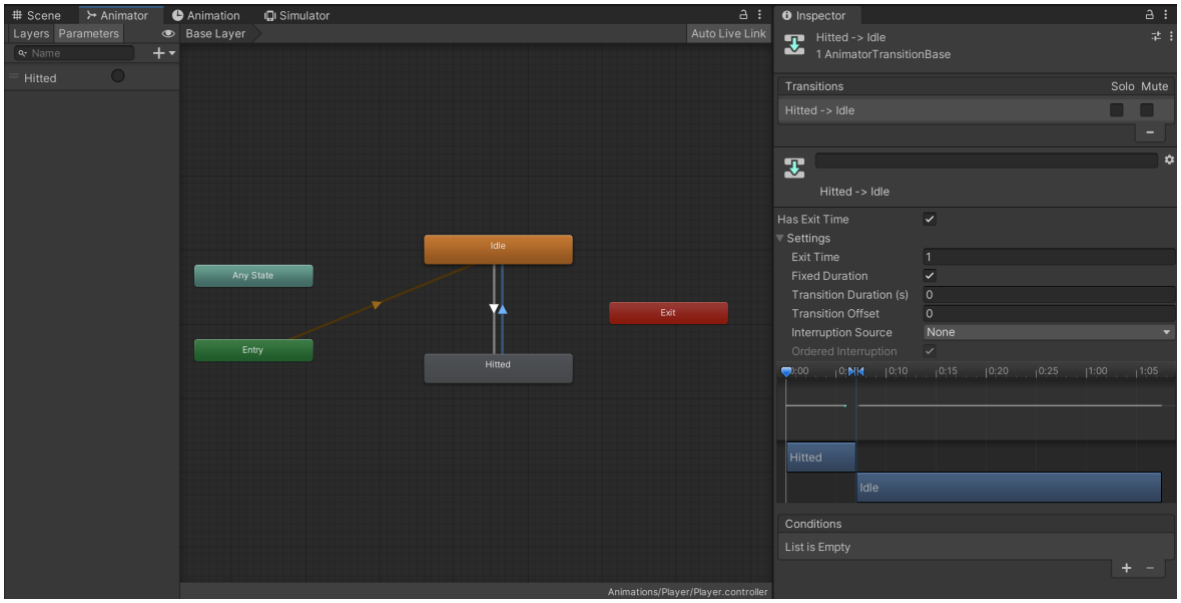

Figura 44 - Transição entre "Hitted" e "Idle"

Neste ponto, é necessário ser feito uma atualização a classe "PlayerStats". Nesta será declarada uma variável do tipo "*Animator*", onde será passada a referência à animação criada anteriormente. Já no método “takeDamage”, será ativado o parâmetro do tipo “*Trigger*”, correspondente à transição pretendida, no caso “Hitted”. É importante ressaltar que este nome deve ser exatamente igual ao definido na animação, de forma a funcionar corretamente.

Durante alguns testes, verificou-se alguns comportamentos estranhos por parte da nave quando esta é atingida várias vezes num curto espaço de tempo. A forma de resolver este problema, foi criar uma "*flag*" denominada de "canPlayAnimation" que fará o controlo se a animação pode ser reproduzida ou não, dentro de uma “*coroutine*” "AntiSpamAnimation".

Na “AntiSpamAnimation”, a execução da animação é temporariamente desativada, após uma pausa de 0,15 segundos a sua execução volta a ser permitida. Este procedimento garante que a animação não seja reproduzida repetidamente, caso a nave sofra impactos sucessivos num curto intervalo de tempo. Esta implementação está ilustrada no Anexo I, na **Figura 107**.

Foram criadas animações equivalentes para os alienígenas, recorrendo a “*sprites*” visualmente distintos, com tonalidades predominantes de verde e cinzento e um efeito ligeiro de transparência. O estilo gráfico adotado, segue a mesma linha dos “*sprites*” “Hitted” desenvolvidos para a nave do jogador. Estas representações podem ser observadas na **Figura 45**.

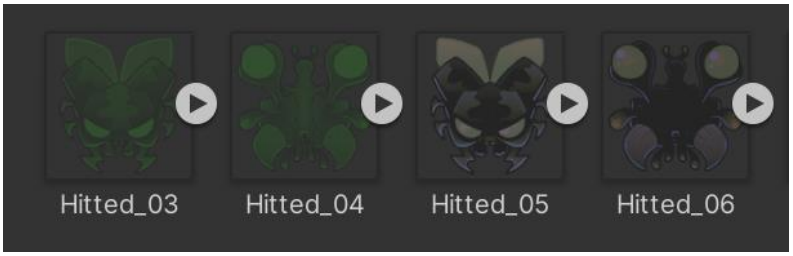

Figura 45 - Sprites de "Hitted" dos "aliens"

As **Figuras 46** e **47,** mostram o resultado da animação "Hitted" implementada. A **Figura 46** mostra que a nave foi atingida por um asteroide, que explodiu no instante do impacto, acionando a animação "Hitted" e aplicando os tons vermelhos e a transparência desejadas à nave do jogador.

Figura 46 - Animação de "Hitted" da nave

A **Figura 47**, mostra a nave do jogador a disparar sobre um “Insetroid”, que se encontra com a animação “Hitted” ativa. Este efeito é percetível pela coloração do inimigo, que apresenta um tom totalmente esverdeado e uma transparência similar a criada para a nave do jogador.

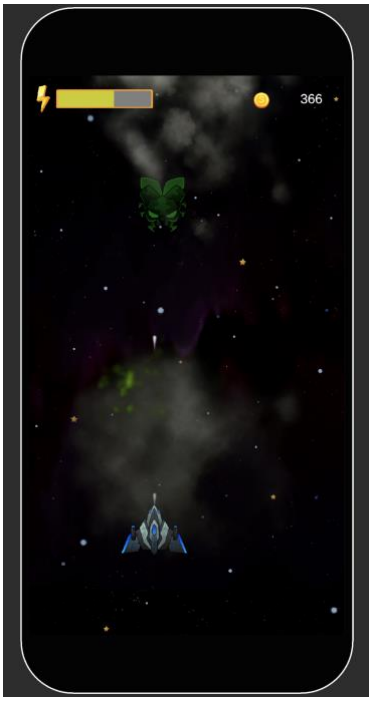

Figura 47 - Animação de "Hitted" no "Insetroid"

3.3 Aliens

A criação dos restantes inimigos do tipo “Alien”, o “Insetroid_04”, o “Oculon” e o “Omnivision”, baseou-se no “*prefab*” previamente desenvolvido para o “Insetroid_02”, construído na fase de prototipagem do jogo. A **Figura 48,** mostra estes “*prefabs*” criados para este tipo de alienígenas.

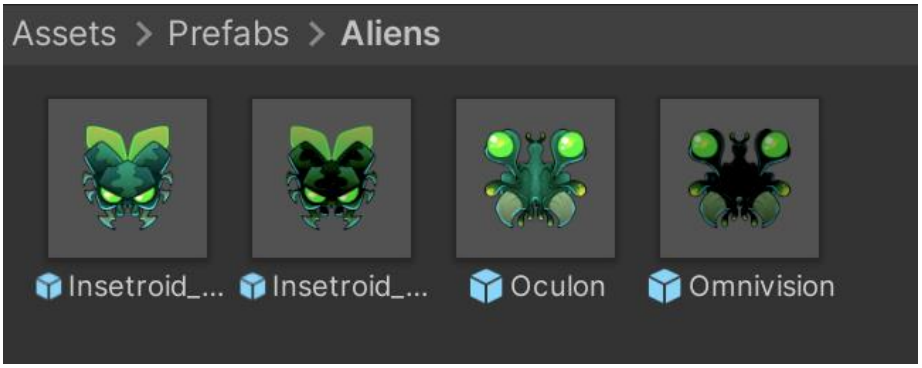

Figura 48 - "*Prefabs*" dos "Aliens"

No entanto, alguns dos alienígenas exigiram algumas adaptações. Uma das adaptações foi o número de pontos de disparo associados ao mesmo. Um caso de exemplo, é o do “*prefab*” do “Omnivision”, o qual passou a dispor de dois pontos de disparo, em contraste com o único existente no “Insetroid_02”.

Para fazer uso desta alteração, foi necessário atualizar o script “Insetroid”, substituindo assim o único ponto de disparo por um “*array*” de pontos, de forma a poder passar a referência de cada um destes pontos para cada um dos “*prefabs*” dos alienígenas.

A **Figura 49** mostra o resultado desta implementação, nesta é possível observar um “Omnivision” a fazer disparos por dois pontos diferentes, conforme planeado no *GDD*.

Para além desta funcionalidade, existem outros parâmetros que requerem atualização consoante o tipo de inimigo, como os níveis de dano causado, a quantidade de vida, entre outros.

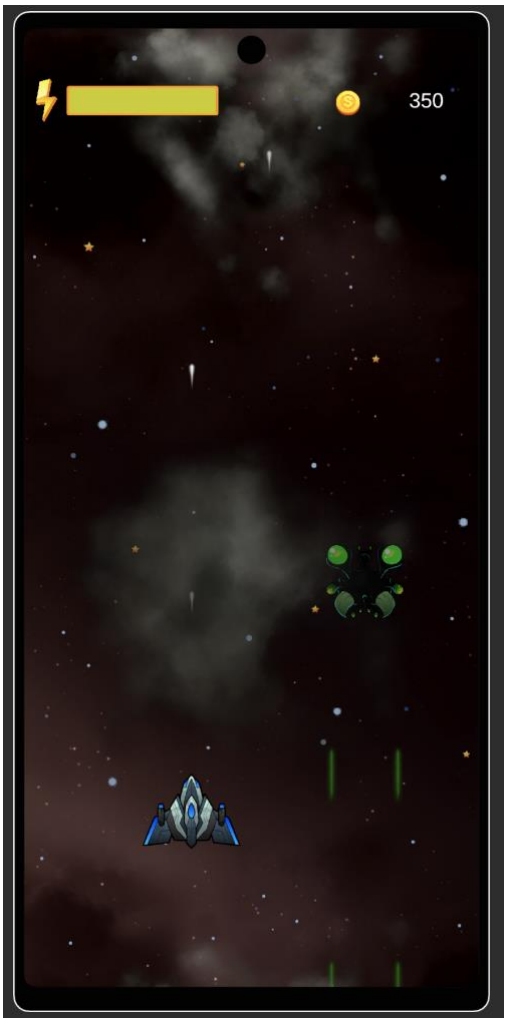

Figura 49 - Disparos do inimigo "Omnivision"

3.4 Lógica do jogo

Este capítulo, serve para mostrar aspetos relacionados com a lógica do jogo, como a condição de vitória necessária para a conclusão tanto dos capítulos como do próprio jogo.

Após a explicação deste mecanismo, será apresentada a implementação dos "*pop-ups*" de vitória ou de derrota, os quais são mostrados consoante o retorno dado pela condição de vitória.

Por fim, será descrito o papel do "*GameManager*". Este é o elemento central de ligação entre a condição de vitória e o painel onde os "*pop-ups*" são apresentados. A sua implementação segue princípios de arquitetura limpa e escalabilidade, tendo sido aplicado o "*design pattern*" "*Singleton*", de forma a garantir uma única instância global acessível deste "*manager*" ao longo de todo o jogo.

3.4.1 Condição de vitória

O critério que avalia que um capítulo tenha sido concluído com sucesso, é se o jogador conseguiu chegar ou não ao fim do tempo definido para esse capítulo. A nave do jogador deve sobreviver a todos os ataques alienígenas, caso isso aconteça ele irá enfrentar o último obstáculo do capítulo que é o líder, que ocupou o planeta em questão.

Desta forma, será criada uma condição de vitória, responsável pela gestão do tempo do capítulo. No futuro, com a implementação do líder no jogo, será feita a verificação se este foi ou não derrotado, de forma a validar o fim do capítulo jogado com sucesso ou insucesso.

Para este fim, foi criado um "*GameObject"* denominado "WinCondition", ao qual está associado um *script* com o mesmo nome. Este componente é responsável pela verificação do tempo decorrido do capítulo e pela desativação da criação de inimigos assim que o tempo pré-definido for atingido. Este "*GameObject*" pode ser observado na **Figura 50**.

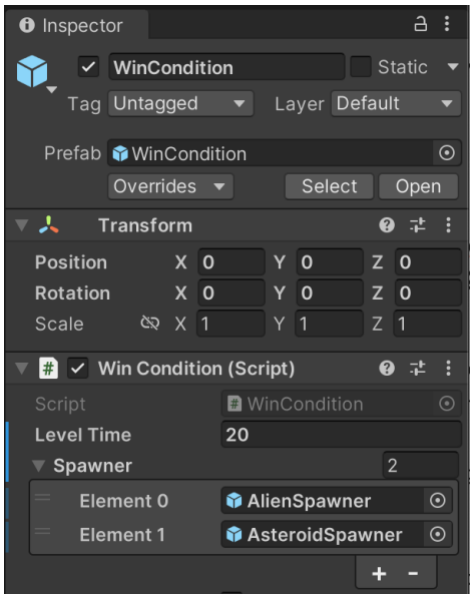

Figura 50 - "GameObject" "WinCondition"

A classe "WinCondition" contém dois atributos principais: o "levelTime", do tipo "*float*", onde é definido o tempo de duração do capítulo; o segundo é um "*array*" de "*GameObjects*", ao qual são associados os "*spawners*" que deverão ser desativados assim que o tempo definido em "levelTime" for atingido, dando assim inicio a entrada do líder no jogo. Esta implementação pode ser consultada no Anexo I – Detalhes de Programação, na **Figura 108**.

3.4.2 Pop-ups de vitória ou derrota

Foram realizados alguns testes e verificou-se de facto, que os geradores de inimigos deixavam de funcionar assim que era atingido o tempo estipulado para o capítulo. No entanto, para que esta condição de vitória pudesse ser detetada e apresentada ao jogador, procedeu-se à criação dos "*pop-ups"* de vitória e/ou derrota, conforme definido no *GDD*.

Assim fez-se uso do "*GameObject"* "Canvas" já disponível no projeto, onde foi criado o *HUD* do jogo. É neste elemento que será feita a implementação dos referidos "*pop-ups"*.

Dentro do "Canvas", foi criado um novo "*GameObject"* do tipo "*Image"*, designado por "Panel". A este foi associada a imagem padrão "square", que define as dimensões do painel, para tal foi utilizado o modo "*Stretch*" no componente "*Rect Transform*", tal como aplicado anteriormente ao "*GameObject"* "SafeArea". A cor do painel foi ajustada para preto com transparência, de forma a funcionar como fundo dos "*pop-ups"* que serão apresentados no ecrã em caso de vitória ou derrota.

Com o fundo criado, serão então adicionados dois novos objetos, o "WinPopUp" e o "LosePopUp", do tipo "*Image*" e utilizam como base o "*sprite*" denominado "LargePanel", anteriormente aplicado na barra de "*power*" presente no *HUD*. Cada um destes "*pop-ups*", possui dimensões de setecentos por setecentos e contém no seu interior elementos adicionais. Um desses elementos é uma caixa de texto que apresenta a mensagem adequada a cada situação, indicando sucesso no caso do "WinPopUp" e insucesso no caso do "LosePopUp". Estes "*pop-ups",* incluem também os botões correspondentes conforme estabelecido *GDD*. Quando se conclui o capítulo com sucesso, é apresentado apenas um botão que permite ao jogador prosseguir para o menu de seleção de capítulos, já com o novo capítulo desbloqueado. Em situações de derrota, surgem dois botões, um direciona o jogador para o ecrã de seleção de capítulos e o outro destinado a reiniciar o mesmo capítulo jogado por parte do jogador, permitindo assim, uma nova tentativa de finalizar o capítulo. A **Figura 51** apresenta a organização de todos estes novos componentes no projeto.

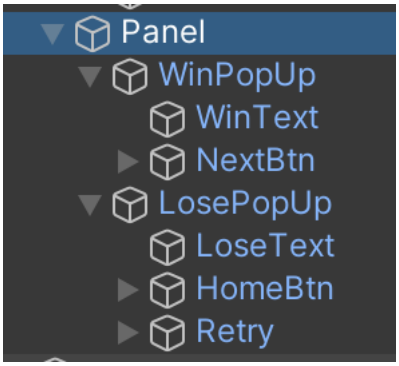

Figura 51 - Organização dos *"Pop-Ups"*

Foram feitas pesquisas sobre a implementação de "*pop-ups"* em *Unity.* Numa destas pesquisas, descobriu-se o componente "*Canvas Group*" [10]. Este componente permite controlar, a visibilidade e a interatividade de um conjunto de elementos de *UI* que pertencem a um objeto principal.

No caso dos "*pop-ups*", será explorada principalmente a funcionalidade de controlo de visibilidade. Para isso, foi adicionado o componente "*Canvas Group*" ao objeto "Panel", o qual contém os objetos "WinPopUp" e "LosePopUp". É através da variável "*Alpha*", que é possível definir se os elementos do grupo estão visíveis ou não. Quando o valor de "*Alpha*" é um, os elementos estão totalmente visíveis; quando é de zero, tornam-se invisíveis. A **Figura 52** apresenta um exemplo com o "*Alpha*" definido em zero virgula cinco, o meio termo, onde é visível o efeito de transparência aplicado ao "*pop-up*".

É muito importante manter o atributo **"***Interactable*" ativo, dado que os "*pop-ups"* incluem botões com os quais o jogador irá interagir.

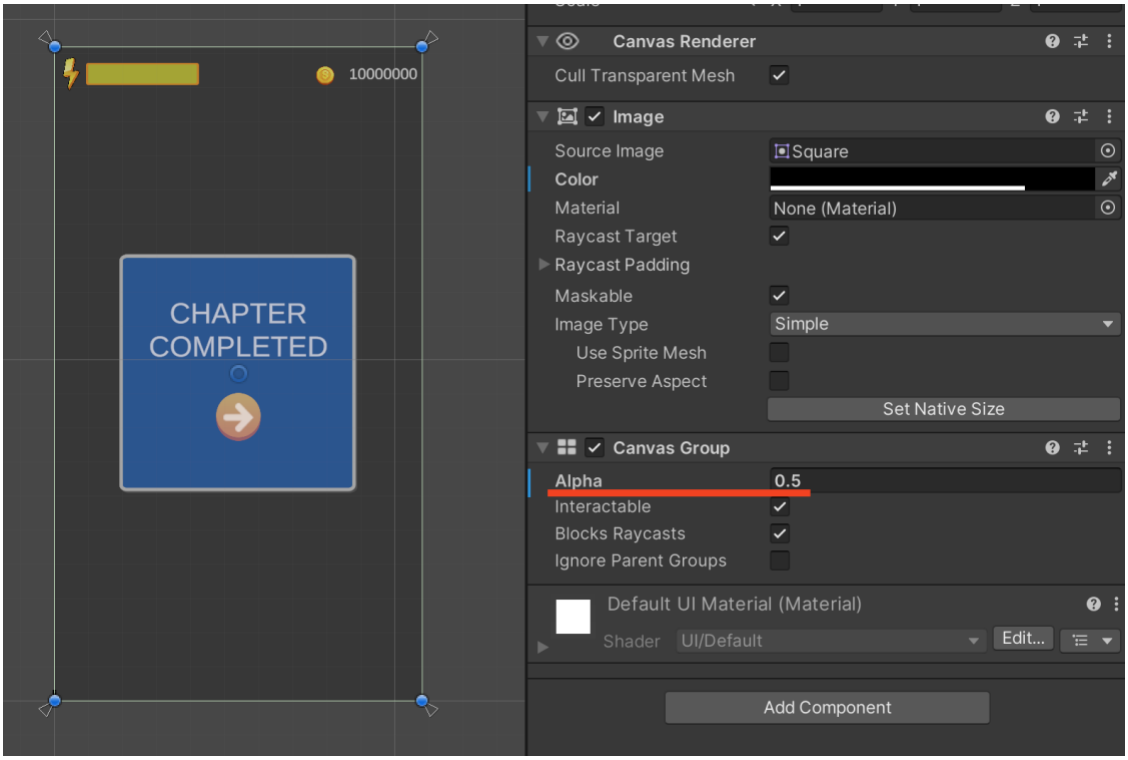

Figura 52 - Configuração do "*Canvas Group*"

O passo seguinte, passou por criar as funcionalidades para quando os botões seriam premidos pelo jogador. Para esse efeito, foi criado o "*GameObject*" chamado de "ButtonController", ao qual foi associado um script com o mesmo nome.

No caso do "*pop-up*" de vitória, existe apenas um botão que, nesta fase inicial, redireciona para uma nova "*scene*" de teste, correspondente a um capítulo diferente. No entanto, a intenção final é que este botão conduza à "*scene*" de seleção de capítulos, permitindo ao jogador escolher livremente o capítulo que deseja jogar, desde que este esteja desbloqueado. O "*pop-up*" de derrota apresenta dois botões, um que faz com que o jogador possa voltar ao ecrã de seleção de capítulos e outro que permite ao jogador jogar o mesmo capítulo novamente de forma que o consiga desta vez concluir com sucesso.

O script criado para fazer este controlo, contém apenas dois métodos: o "LoadLevel" e o "Restart". Em ambos é usado o "*SceneManager*", uma classe do *Unity* responsável pelas transições entre cenas. Esta implementação pode ser observada na **Figura 53**.

```
public class ButtonController : MonoBehaviour
{

    0 references
    public void LoadLevel(int level)
    {
        SceneManager.LoadScene(level);

    }

    0 references
    public void Restart()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);

    }

}
```

Figura 53 - Implementação "ButtonController"

Por último, foi criado um "*GameObject"* com o nome de "PanelController", ao qual foi associado um script com o mesmo nome. Este tem como função principal fazer o controlo da apresentação dos "*pop-ups"* de vitória ou derrota. A implementação detalhada deste script pode ser consultada na **Figura 109**, presente no Anexo I – Detalhes de Programação.

3.4.3 GameManager

Usualmente na criação de videojogos, faz-se uso de um padrão de desenvolvimento ao qual se chama de "*GameManager*". A principal função é centralizar vários aspetos de gestão do jogo, como por exemplo, transcrições entre cenas. Neste projeto, este padrão será usado como elo entre a classe "WinCondition" e "PanelController".

Desta forma, foi criado o “GameObject” denominado de “EndGameManager”, a sua função principal será controlar o fim dos capítulos. Este objeto possui um script com o mesmo nome, que é responsável por fazer a gestão de todos os processos e funcionalidades relativas a esse controlo.

Como esta classe será usada por outras, o primeiro passo é criar uma variável estática do tipo da classe, tornando-a acessível a nível global dentro do projeto. A esta variável criada, será feita a atribuição do "*this*", dentro do método de "*Awake*", assim todos os elementos pertencentes à classe podem ser acedidos através da mesma.

Na classe "EndGameManager", são adicionados dois métodos: o "WinGame" e o "LoseGame", que são os responsáveis pelas chamadas dos respetivos *“pop-ups”*. Passando para a explicação entre as classes, a **Figura 54** é exemplificativa da mesma.

Figura 54 - Esquema de implementação do "EndGameManager"

Na classe "EndGameManager", é criada uma variável do tipo "PanelController". Será por meio desta, que será feito o registo da referência externa que fará a ligação com o "manager". Para tal, foi criado o método "RegisterPanelController" que recebe como atributo uma variável do tipo "PanelController", neste é feita a atribuição desta variável externa à interna da classe. Esta implementação está representada na **Figura 55**.

Figura 55 - Implementação do registo do "PanelController"

Seguindo para a classe "PanelController", mais especificamente no método de "*Start*()", é feito o registo desta classe através da chamada ao método "RegisterPanelController", passando o "this". Desta forma, as duas classes ficam ligadas mal a execução do jogo inicie. A **Figura 56**, mostra como é feito esse registo.

Figura 56 - Registo do "PanelController"

Na classe "EndGameManager", foi implementado o método com o nome "ResolveGame". Este, é o responsável por avaliar o estado do jogo tendo como base o uso de uma variável booleana chamada "gameOver". Caso o valor desta seja verdadeiro é invocado o método "LoseGame", caso contrário o método "WinGame".

Este método “ResolveGame”, será acionado na classe "WinCondition" logo após o fim do processo de criação de alienígenas. A **Figura 57** ilustra esta implementação.

```
void Update()
{
    timer += Time.deltaTime;

    // verify if the timer achieved the level time
    if(timer >= levelTime)
    {

        // desactivate all the game objects spawned
        for (int i = 0; i < spawner.Length; i++)
        {
            spawner[i].SetActive(false);
        }

        EndGameManager.endGameManager.ResolveGame();
        gameObject.SetActive(false);
    }
}
```

Figura 57 - Chamada do "ResolveGame" na "WinCondition"

Feitos alguns testes, verificou-se que mesmo em caso de vitória por parte do jogador, poderiam ainda existir elementos inimigos no ecrã de jogo, que poderiam causar dano na nave, destruindo-a.

De forma a resolver este problema, foi criada uma "*coroutine*" com o nome de "ResolveSequence", esta coloca um “delay” de dois segundos, suficientes para que todos os inimigos já tenham desaparecido, antes da exibição do *“pop-up”* de vitória.

Esta “*coroutine*” é chamada a partir da função “StartResolveSequence()”. Para evitar que a sequência seja executada várias vezes por chamadas repetidas de diferentes classes, o método começa por interromper qualquer instância anterior da desta através do “StopCoroutine()” e antes de a iniciar novamente com o método “StartCoroutine()”, ilustrado na **Figura 58**.

```
public void StartResolveSequence()
{
    // to avoid multiple calls from different scripts
    StopCoroutine(ResolveSequence());
    StartCoroutine(ResolveSequence());
}

2 references
private IEnumerator ResolveSequence()
{
    yield return new WaitForSeconds(2);
    ResolveGame();
}
```

Figura 58 - Implementação do "ResolveSequence" e "StartResolveSequence"

A classe "WinCondition" precisa agora de ser atualizada e para isso deve ser feita a troca do método "ResolveGame" pelo "StartResolveSequence". A partir desta mudança, foram feitos mais alguns testes, em que o jogador concluía o capítulo com sucesso e o resultado está ilustrado na **Figura 59**, onde é possível verificar a ativação do *"pop-up"* de vitória com sucesso.

Figura 59 - *"Pop-up"* de vitória

Fechada a lógica para a parte da vitória, passou-se para a lógica em caso de derrota.

Para que isso aconteça, na classe "PlayerStats" onde é verificada a condição em que a nave fica sem "power", deverá ser invocado o *"pop-up"* de derrota.

Recorre-se ao "EndGameManager", o qual possibilita a chamada do método "StartResolveSequence", responsável por apresentar o respetivo *"pop-up"*. A **Figura 60**, mostra esta implementação.

```
if (power <= 0)
{
    EndGameManager.endGameManager.gameOver = true;
    EndGameManager.endGameManager.StartResolveSequence();
    Instantiate(explosionPrefab, transform.position, transform.rotation);
    Destroy(gameObject);
}
```

Figura 60 - Implementação para a invocação do "*pop-up*" de derrota

Seguidamente, foram realizados mais testes de forma a comprovar que ficou tudo como pretendido. A **Figura 61**, mostra o *"pop-up"* de derrota após a nave do jogador ter ficado sem "power", conforme indica o *HUD*.

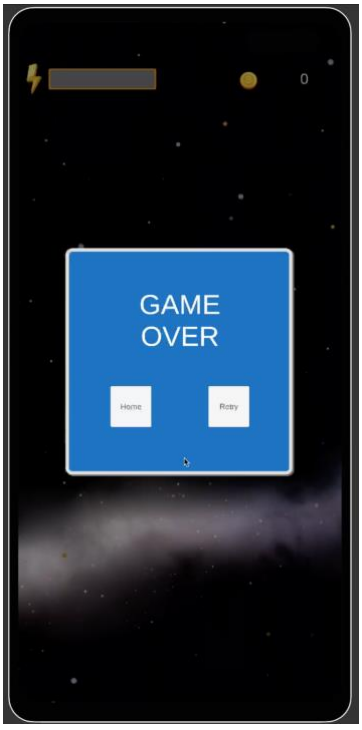

Figura 61 - "*Pop-up*" de derrota

O “EndGameManager” foi atualizado para usar o padrão “*Singleton*” [11], garantindo assim que só exista uma instância deste durante todo o jogo.

Como o jogo tem vários capítulos, cada um sendo uma “*scene*” diferente, é importante que alguns objetos, como o *“manager”*, não sejam destruídos.

O comportamento base do *Unity*, é que ao mudar de “*scene*”, todos os objetos da anterior são eliminados para dar lugar aos novos. No entanto, queremos que o “EndGameManager” continue ativo desde o momento em que o jogador começa um capítulo do jogo, mantendo-se disponível nas “*scenes*” seguintes.

Para isso, foram feitas algumas mudanças no método “*Awake*()” da classe “EndGameManager”. É feita a verificação se a instância ainda não foi atribuída através da comparação se é igual a “*null*”. Caso isto se verifique, é então feita a atribuição da própria classe a variável, de seguida faz-se a chamada ao método "*DontDestroyOnLoad*", de forma a garantir que este não seja destruído.

Com esta abordagem, o objeto “EndGameManager” estará sempre presente em qualquer “*scene*” para a qual o jogador transite.

Esta implementação do “*Singleton*”, está ilustrada na **Figura 62**.

```
void Awake()
{

    // singleton pattern - to only exist on object of the same type on whole game
    if (endGameManager == null)
    {
        endGameManager = this;
        // perserve the manager from the level where the player started by default;
        DontDestroyOnLoad(gameObject);
    }
    else
    {
        Destroy(gameObject);
    }

}
```

Figura 62 - Implementação padrão "Singleton"

A **Figura 63,** mostra em “*runtime*”, como a destruição do objeto “EndGameManager” é evitada graças à utilização do método “DontDestroyOnLoad()”.

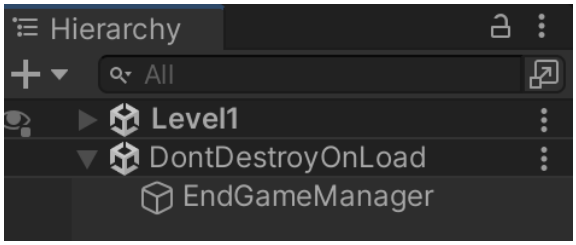

Figura 63 - Utilização do "DontDestroyOnLoad"

O resultado desta implementação, ficou muito fiel ao que havia sido idealizado no *Game Design Document*. A **Figura 63** e **64**, mostra a comparação entre os "*pop-ups*" do jogo com o esboço original dos mesmos. É possível observar, que em ambos já são aplicados os "*assets"* dos botões do pacote "*Simple Button Set 01*" [12].

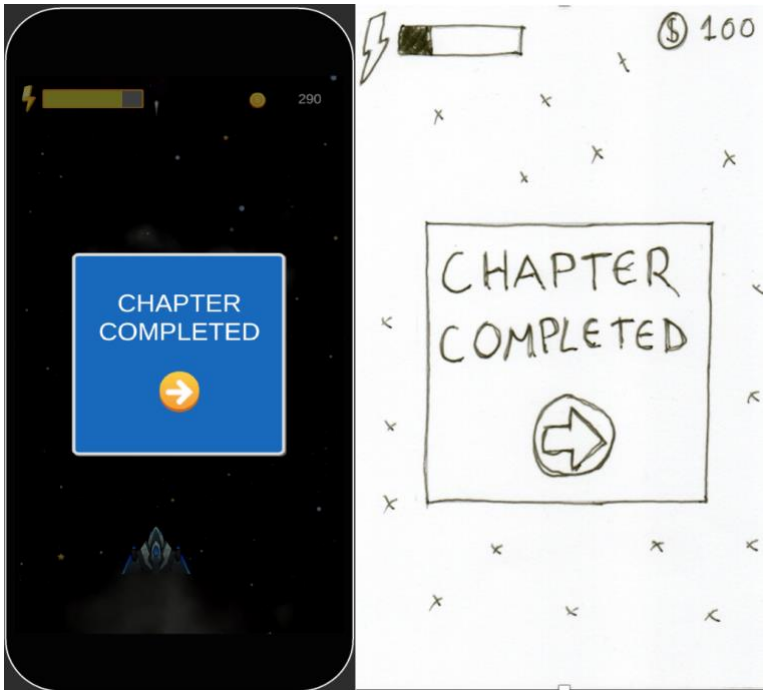

Figura 64 - Comparação final *"pop-up"* de vitória

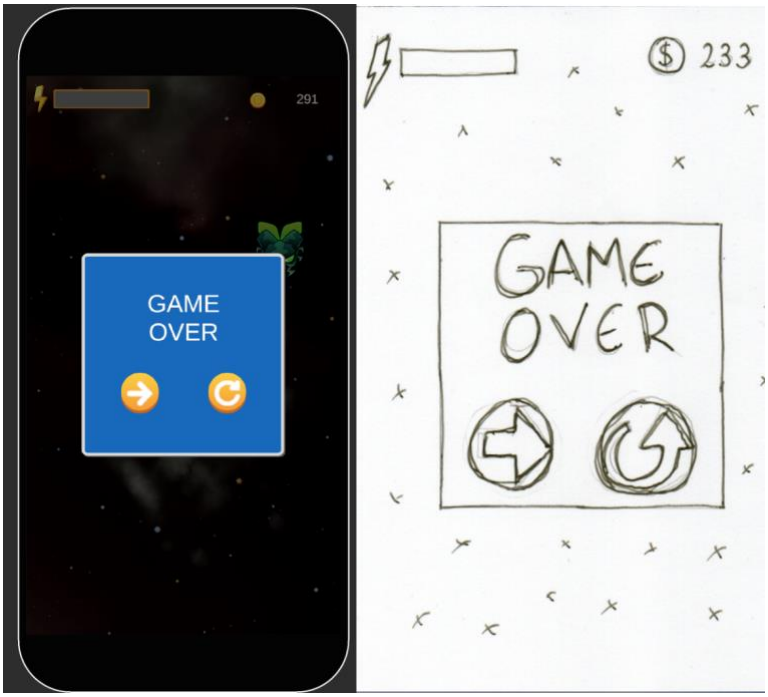

Figura 65 - Comparação final *"pop-up"* de derrota

3.5 User Interface

Neste subcapítulo, serão abordadas todas as implementações de *UI* criada no jogo. Desde o desenvolvimento do "Fader", que será usado entre transições de "*scenes*", como o ecrã de "Loading", a ser mostrado entre carregamentos de ecrãs. Em termos de outros ecrãs, será também abordada a implementação do ecrã de seleção de capítulos, a partir do qual é possível ao jogador escolher o capítulo que pretende jogar e por fim, o ecrã inicial do jogo, a partir do qual é possível dar início à experiência.

Todos estes elementos, foram previamente planeados no "*Game Design Document*" na primeira fase deste projeto.

3.5.1 Fader

Durante o desenvolvimento dos botões dos *"pop-ups"* e as suas funcionalidades, surgiu a ideia da implementação de um "Fader".

Foram feitas várias pesquisas acerca deste tema, até que foi encontrado o artigo "*Unity-Fader*", com a autoria de Benjamim Calvin [13]. Este foi um apoio essencial para a implementação que se segue.

Começou-se por criar o objeto "Fader" do tipo "*Canvas*". Onde foram necessárias algumas alterações nas variáveis de "*Scale Mode*" e "*Reference Resolution*". A primeira foi alterada para "*Scale With Screen Size*", já a segunda recebeu o valor de 1080x1920.

Foi adicionado ao "Fader", um objeto filho do tipo "*Image*" denominado de "Background", neste foi colocada a imagem auxiliar presente nos “*assets*” do projeto chamada de "Square", a qual foi esticada de forma a ocupar o ecrã. De seguida, foi feita a alteração da cor para preto, de modo a servir como um fundo escuro.

Para permitir o controlo do desvanecimento da cor, foi adicionado o componente “CanvasGroup”, uma vez que, interessa controlar a variável “Alpha” contida neste componente através de um script, tal como foi feito no caso dos “*pop-ups*”.

Esse script foi denominado como “FadeEffect”, de forma resumida, é nele que será implementada toda a lógica responsável pela transição suave entre ecrãs. Será aplicado novamente, nesta classe, o padrão “*Singleton”*, tal como foi feito no “*manager”*, com o objetivo de garantir que o objeto não seja destruído durante as transições entre “*scenes”*. A classe contará com dois métodos implementados como “*coroutines”*, responsáveis pela transição visual do ecrã entre os estados de escuro para transparente e vice-versa, estes métodos foram chamados de “FadeIn” e “FadeOut”. A implementação completa do script “FadeEffect” pode ser consultada no Anexo I – Detalhes de Programação, na **Figura 110**.

De forma a garantir que o elemento “Fader” apareça acima de todos os outros elementos da *UI* do jogo, foi alterado o valor da propriedade “*Sort Order”* para cinco valores.

A **Figura 66**, mostra que quando o botão do *“pop-up”* é pressionado, inicia-se o processo de "fade out", onde é possível identificar a transição entre o estado de transparência para o preto total.

Figura 66 - Exemplo de “*Fade Out*”

A **Figura 67,** mostra o caso do “*fade in*” que acontece durante a transição para a entrada de um novo capítulo.

Figura 67 - Exemplo de “*Fade In*”

3.5.2 Loading

O ecrã de "Loading", foi construído dentro do "Canvas" posteriormente criado.

Para tal inclui-se um “*game object*” do tipo “*Image*” denominado de "Loading", este foi esticado de forma a poder ocupar todo o espaço do ecrã e foi atribuída a cor de fundo preta. Dentro deste novo objeto, criou-se a "LoadingBar", objeto clonado a partir da "PowerBar", dado que ambos tem comportamentos bastantes semelhantes, fez uso desta, adaptando-a a nível de dimensões e cores.

Outro elemento presente neste ecrã, é o texto identificativo do ecrã onde estamos presentes, dessa forma foi colocado um objeto que pudesse introduzir o texto de "Loading" no ecrã.

Quanto a comportamentos, foi necessário fazer um ajuste ao script "FadeEffect" de modo a incluir este novo ecrã.

Fez-se uso da classe "*AsyncOperation*", de forma a usar o ecrã de "Loading" sem que ocorra nenhum tipo de bloqueio durante a execução do mesmo.

Esta implementação, encontra-se de forma detalhada no Anexo I – Detalhes de Programação, na **Figura 111**.

A **Figura 68**, mostra o ecrã de "Loading" no seu estado inicial, quando a barra de carregamento se encontra vazia.

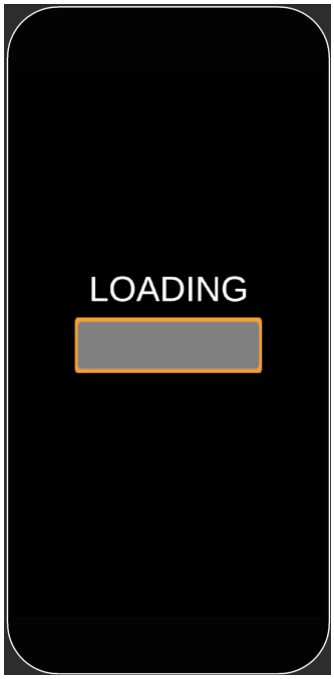

Figura 68 - Ecrã de "*loading*" a iniciar

A **Figura 69**, mostra o estado final do processo de "Loading" no qual a barra se encontra completamente preenchida, mostrando que o processo ficou concluído.

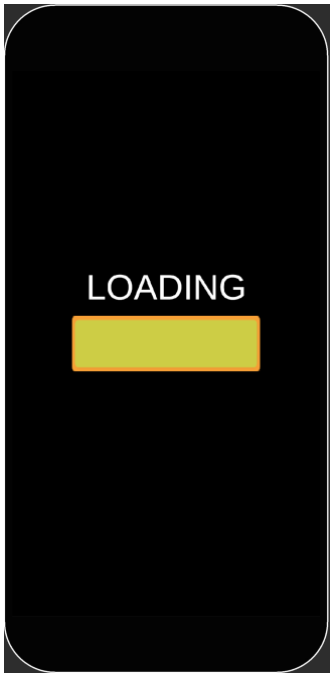

Figura 69 - Ecrã de "*loading*" com barra preenchida

Como é possível observar, o resultado apresentado é bastante semelhante ao definido no *GDD*, conforme ilustrado na **Figura 70**.

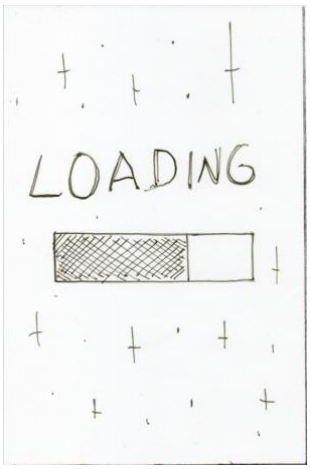

Figura 70 - Esboço do ecrã de "*loading*"

Importa salientar que a progressão da barra de carregamento apenas pode ser visualizada com clareza quando a aplicação é executada num dispositivo móvel. No ambiente de simulação, devido ao desempenho superior do computador, o processo de carregamento decorre de forma demasiado rápida, o que dificulta a perceção visual da progressão da barra.

3.5.3 Seleção de Capítulos

Para a criação do ecrã de seleção de capítulos, fez se uso da cena "Level1", de forma a aproveitar todos os componentes já disponíveis na mesma. Desta modo, fez-se o clone da cena e renomeou-se para "LevelSelection", sendo apenas necessário fazer a remoção dos objetos que não seriam necessários para este contexto. O resultado desta adaptação pode ser visto na **Figura 71**.

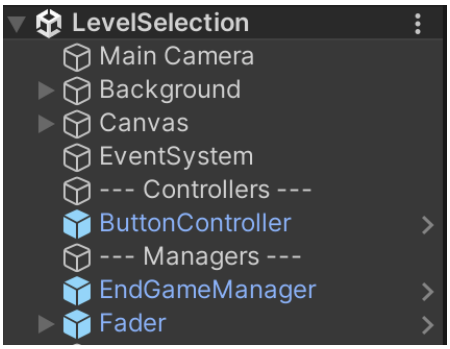

Figura 71 - Estrutura do ecrã "Level Selection"

De forma a construir a interface deste ecrã, fez-se uso do elemento "Canvas" dentro deste foi colocado um novo elemento do tipo "*Text*", denominado "GameTitle" onde é colocada a frase "Chapter Selection".

Abaixo deste, foi criado um outro objeto chamado de "LevelContainer", será este que vai conter os vários botões. Foram criados então cinco botões, aos quais foram associadas as imagens correspondentes aos planetas que representam. Para além da imagem, foi associado o evento "*OnClick*()", configurado de forma a invocar o método "LoadLevel", pertencente a classe "ButtonController".

Estes botões, são colocados como elementos filhos do objeto "LevelContainer", que vai ter associado a função de mostrar os capítulos desbloqueados ou bloqueados. Caso o planeta tenha as suas cores originais, significa que está desbloqueado, caso contrário é apresentada a imagem do planeta a negro.

A nível funcional, foi feita uma alteração à classe "EndGameManager", mais concretamente no método "WinGame", pois após o jogador concluir um capítulo com sucesso, deve-se verificar qual o capítulo seguinte a ser desbloqueado.

Essa verificação, é realizada através do valor guardado no "*PlayerPrefs*"[13] usado para armazenar este valor entre sessões do jogo.

Recorre-se ao método "*GetInt*()", de forma a consultar o valor guardado na chave "levelUnlock" criada para o efeito e de seguida é feito o "*SetInt*()" de forma a guardar o novo valor do nível desbloqueado.

Esta implementação pode ser observada na **Figura 72**.

```
// unlock next level
int nextLevel = SceneManager.GetActiveScene().buildIndex + 1;
if (nextLevel > PlayerPrefs.GetInt(levelUnlock, 2))
{
    PlayerPrefs.SetInt(levelUnlock, nextLevel);
}
```

Figura 72 - Atualização no "EndGameManager"

Foi associado o script "ButtonIcon" ao objeto "LevelContainer", este será o responsável por gerir os capítulos disponíveis ou não dentro do mesmo.

A sua implementação, pode ser encontrada de forma detalhada no Anexo I – Detalhes de Programação, na **Figura 112**.

A **Figura 73**, mostra a implementação em funcionamento, no caso apenas o capítulo de “*Rosara*” está disponível para o jogador.

Figura 73 - Ecrã "Chapter Selection"

Após a realização de alguns testes, verificou-se que o comportamento esperado foi feito com sucesso.

Na **Figura 74**, temos o caso em que o jogador conseguiu ultrapassar o primeiro capítulo com sucesso tendo agora disponível o segundo capítulo do jogo.

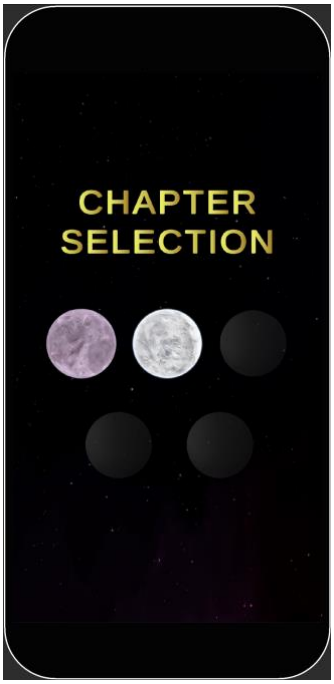

Figura 74 - "Chapter Selection" após concluir o primeiro capítulo

O resultado obtido, está de acordo com o que foi pensado no "*Game Design Document*".

A **Figura 75**, mostra o esboço deste ecrã onde se pode verificar que o que foi implementado está de acordo com o que foi planeado. Foi feita apenas uma pequena alteração que foi remover os nomes dos capítulos do ecrã, de forma a tornar este mais limpo.

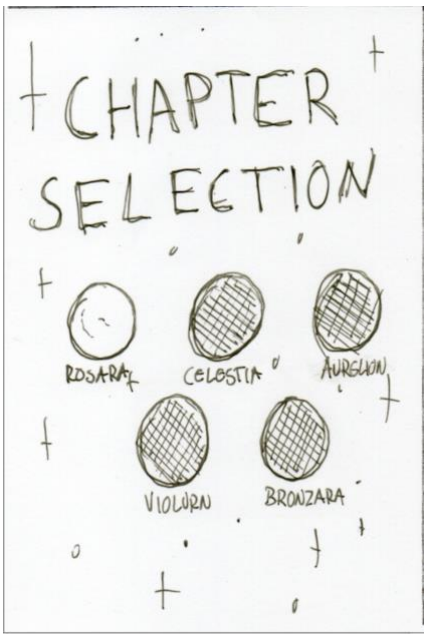

Figura 75 - Esboço do ecrã de seleção de capítulos

3.5.4 Ecrã inicial

O método usado para a criação do ecrã inicial, seguiu um procedimento parecido ao que foi usado para a criação do ecrã de seleção de capítulos. Foi usado este último, de forma a alterá-lo apenas removendo os objetos que não eram necessários a este ecrã inicial, renomeando-o para "Start".

A partir do elemento “Canvas”, já presente na cena, mais concretamente no elemento filho “SafeArea”, foi inserido um objeto do tipo “*Image*”, denominado “*Planet*”.

Este objeto, recebeu como “*sprite*” a imagem do planeta Saturno [16], encontrado de forma aleatória, mas que se revelou visualmente adequada ao estilo pretendido para o ecrã inicial.

De seguida, foram adicionados os "*game objects*" de texto e do botão presentes no esboço do ecrã, o primeiro recebeu o título do jogo "Supernova QuEST" e no segundo foi atribuído o evento "*onClick*()".

Será através deste, que o jogador irá conseguir passar do ecrã inicial para o ecrã de seleção de capítulos.

A **Figura 76**, apresenta a implementação descrita anteriormente ao nível dos objetos presentes na cena.

Figura 76 - Implementação do ecrã inicial

Como é possível observar na **Figura 77**, apresenta a comparação entre o resultado do ecrã inicial implementado e o esboço definido no "*Game Design Document*". Os únicos elementos em falta, são o botão da loja e o botão para desativar o som do jogo, tirando esses dois elementos, o resultado apresenta-se bastante fiel ao conceito inicialmente proposto.

Figura 77 - Comparação entre ecrã final e esboço inicial

3.6 Power-ups

Neste capítulo, é descrita a implementação dos três "*power-ups*" presentes no jogo, o "PowerPlus", o "ShieldProtection" e o "UltraShoot". Como base para a construção destes objetos, foi feita uma pesquisa sobre "*Scriptable Objects*" [17].

3.6.1 Scriptable Objects

Considerando que o jogo precisava de um sistema de criação de coletáveis, tais como os “*power-ups*” e as moedas, foi feita alguma pesquisa sobre o assunto o que conduziu ao tema de "*Scriptable Objects*".

Foi criado o script "SO_Spawner", no qual o primeiro passo é substituir a herança "*default*" desta classe para "*ScriptableObject*" ao invés do "*MonoBehaviour*" e o segundo passo, é colocar a anotação "*CreateAssetMenu*", acima da declaração da classe, na qual se indica o nome do ficheiro que será gerado como também o caminho para o mesmo.

Dentro do "SO_Spawner", foi implementado o método "Spawn()" este recebe como parâmetro uma posição, de forma a poder criar objetos na posição passada. A implementação completa encontra-se detalhada no Anexo I, **Figura 113**.

Agora é possível criar um “*ScriptableObject*”, clicando com o botão direito do rato sobre a pasta de “*assets*” onde se pretende criar o objeto, selecionando a opção "*Create* > *ScriptableObject* > *Spawner*".

Após criado o “*ScriptableObject*”, basta incluir os “*prefabs*” dos objetos que se pretendem gerar, bem como definir a percentagem de frequência com que devem ser instanciados, conforme ilustrado na **Figura 78**.

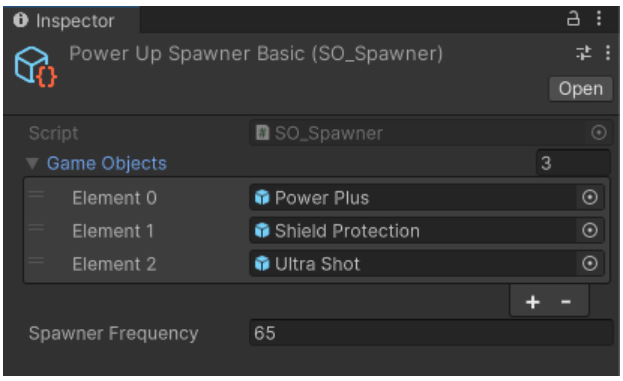

Figura 78 - "*ScriptableObject*" "*PowerUpSpwaner*"

Deste modo, basta incluir este objeto no script “Insetroid” e no momento em que este é destruído, é feita a chamada ao método “Spawn()”.

3.6.2 Power Plus

A implementação do "*power-up*" "Power Plus", tem o propósito de restituir o nível de "power" da nave do jogador. Este processo iniciou-se, com a criação do "*game object*" denominado de "Power Plus", no qual foi atribuído o componente "*Rigidbody2D*" e "*Box Collider2D*", de forma a ser possível verificar a colisão quando este embate com a nave. Por fim, foi criado o script "PowerPlus" o qual vai tratar das funcionalidades deste objeto. Concluída esta configuração, foi colocado na pasta de "*prefabs*" de maneira a criar um objeto deste tipo, conforme mostra **Figura 79**.

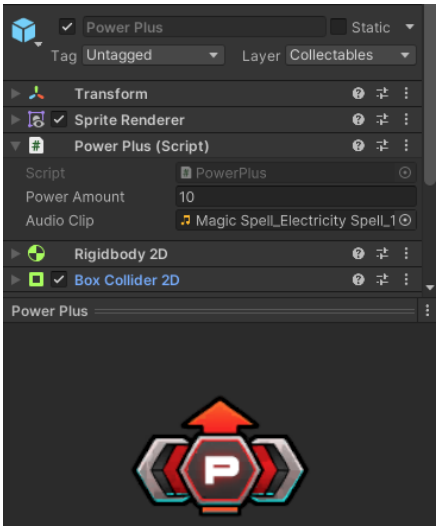

Figura 79 - "*Prefab*" "*Power Plus*"

Outra observação relevante a incluir neste capítulo é a criação da "*layer*" "Collectables", na qual serão incluídos todos os "*prefabs*" dos "power-ups".

Na "*Colission Matrix*", foi feita a atualização para que esta nova "*layer*" seja reconhecida pela "*layer*" do "Player".

No script "PowerPlus", é feita a verificação de quando este objeto sofre alguma colisão com a nave do jogador, caso esta colisão aconteça é chamado o método "AddPower", no qual se passa o valor de "*power*" a ser restaurado. A implementação do script "PowerPlus" pode ser consultada no Anexo I – Detalhes de Programação, na **Figura 114**.

No "*Game Design Document*", está indicado que este "power-up" deveria restituir o "*power*" da nave na totalidade. Contudo, isso não pareceu o mais correto, pelo que passou a restituir apenas uma percentagem do que foi retirado, tornando-se necessária uma emenda ao que fora escrito na altura.

A **Figura 80**, mostra a captura de dois ecrãs do jogo. No ecrã da esquerda é possível visualizar que depois de se eliminar um "Insetroid_02", esta morte gerou um “*power-up*” “Power Plus”. A captura do lado direito, mostra que a nave do jogador já colidiu com este “*power-up*”, isto é, verificável pelo *HUD* do nível de “power” da nave, que foi restituído em 10% do seu preenchimento.

Figura 80 - "Power Plus" em jogo

3.6.3 Shield Protection

A configuração deste "*power-up*", teve início com o ajuste do "*sprite*" referente à "Shield", de forma a alinhá-lo e dimensioná-lo em relação ao "*sprite*" da nave de jogador.

À semelhança dos outros "*power-ups*", também foi criada uma "*layer*" de colisão para este novo objeto, a qual se chamou de "Shield". Após esta criação foi feito o ajuste na "*Colission Matrix*", de forma que a "*layer*" de "Shield" só tenha interações com as "*layers*" "Enemy" e "EnemyMissile".

A este "*game object*", foi associado um script de forma a fazer a gestão do mesmo, com o nome de "Shield".

Quando o "*power-up*" é coletado pela nave do jogador, o "shield" é ativo e cria uma camada de proteção ao redor da nave, que aguenta até três colisões. Caso, este "*power-up*" seja recolhido e a "shield" se encontre ativa, este é restaurado na sua totalidade.

Em caso de colisões com um inimigo, este é automaticamente "destruído", sendo retirada uma percentagem de 33% a barra da "shield", mas, no entanto, existe uma exceção para o inimigo líder, este não é automaticamente destruído, em vez disso o escudo é imediatamente retirado ao jogador.

A implementação deste script encontra-se representada na **Figura 115**, no Anexo I – Detalhes de Programação.

O passo seguinte, é incluir uma variável do tipo "Shield" na classe "PlayerStats". Para que no método "takeDamage" seja verificado o valor de "isProtected", sendo este verdadeiro, o método deve ser interrompido de ser executado, não retirando qualquer valor de "power" à nave.

Falta criar um script que faça o controlo da ativação da "shield", ao qual se deu o nome de "PlayerShieldActivator", caso a "shield" se encontre ativa é feita a execução do método de reparo, caso contrário procede-se a ativação da "shield".

Será agora necessário criar o "*prefab"* que representará o "*power-up"* "Shield Protection.

Criado o "*prefab*", foi implementado o script de controlo de colisão, denominado "*ShieldProtection*". A implementação deste script encontra-se no Anexo I – Detalhes de Programação, na **Figura 116**.

O resultado obtido foi o esperado e planeado no *GDD* anteriormente. A **Figura 81,** mostra dois ecrãs do videojogo: o da esquerda apresenta o jogador quando recolhe o “*power-up*” “ShieldProtection”, o da direita indica que o “*power-up*” já foi recolhido. É possível verificar que a nave do jogador tem, à sua volta, o “*sprite*” da “shield”, a barra de duração do mesmo já se encontra no *HUD* e a nave fica protegida momentaneamente.

Figura 81 - "Shield Protection" em jogo

3.6.4 Ultra Shot

O primeiro passo para a criação do “*power-up*” “UltaShoot”, foi adicionar mais pontos de tiro à nave usada pelo jogador. A nave por predefinição usada pelo jogador é a "Azure Horizon" que na sua configuração por predefinição apenas possui um ponto de tiro, sendo necessário adicionar mais quatro pontos de tiro de acordo com o que foi escrito no *GDD*.

A **Figura 82**, ilustra esta criação dos novos pontos de tiro no "*game object*" "Player".

Figura 82 - Novos pontos de tiro na nave

Desta forma, foi necessária a atualização da classe “PlayerShooting”, pois esta apenas dispunha de um único ponto de tiro, tendo de ser substituído por um “*array*” de forma a comportar os novos pontos de tiro atribuídos a nave.

A gestão do número de tiros ativos na nave, é feita através dos métodos "IncreaseShooting" e "DecreaseShooting". Uma vez que, tal como foi definido no *GDD*, caso o jogador colete este “*power-up*” o número de pontos de tiros deve aumentar progressivamente, como também, caso sofra qualquer tipo de dano, este número de tiros deve decrementar da mesma forma.

Esta implementação encontra-se descrita na **Figura 117**, no Anexo I – Detalhes de Programação.

Tal como os restantes “*power-ups*”, foi criado o “prefab” "UltraShoot" ao qual foi associado o script com o mesmo nome. Este, avalia se existiu alguma colisão entre o “power-up” e a nave do jogador, caso isso tenha acontecido, é chamado o método de "IncreaseShooting".

O resultado obtido, corresponde exatamente ao que foi planeado no “*Game Design Document*”. Para comprovar que a implementação foi bem-sucedida, pode-se observar a **Figura 83**, que se encontra dividida em dois ecrãs. No lado esquerdo, é visível a nave do jogador após eliminar um inimigo, o que originou o “*power-up*” “Ultra Shoot”. Já no lado direito da figura, verifica-se que, após a recolha do “*power-up*” pelo jogador a nave passa a dispor de três pontos de tiro, claramente identificáveis através dos três mísseis visíveis na imagem.

Figura 83 - "Ultra Shoot" em jogo

3.7 Liders

Este capítulo diz respeito a implementação do inimigo "Lider", este é o adversário mais temido no "Supernova QuEST".

Neste será focada a implementação da "*state machine*", e a maneira de como é feito o controlo dos estados pelo que o líder pode passar. São quatro os estados: entrada, disparo, ataque especial e morte.

3.7.1 Padrão de projeto "State Machine"

A ideia para a implementação do comportamento do "Lider" no jogo, consiste em seguir um padrão de transições entre estados.

Para tal, foi adotado o padrão de desenvolvimento "*State Machine*" [18], este método de desenvolvimento é usado para controlar mudanças de estados em objetos, enquadrando-se assim na perfeição com o requisitado.

Foi implementada uma superclasse designada de "LiderBaseState", a partir da qual foram criadas subclasses para cada um dos estados que o líder possa ter, nesta classe foram definidos dois métodos, o "RunState" e o "StopState", ambos serão "*override*" pelas classes filhas.

Adicionalmente, foi criada a classe "LiderController", na qual esta presente o método "ChangeState" este será o responsável por mudar o estado em que o líder se encontra de momento.

Voltando à classe "LiderBaseState", foi necessário incluir as variáveis de limitação de movimento, como já tinha sido feito na classe "PlayerController", de forma que o líder não saia da sua zona de atuação.

A implementação desta classe encontra-se representada na **Figura 118**, incluída no Anexo I – Detalhes de Programação.

De forma auxiliar, foi criado o "*enum*" "LiderState" o qual contém os quatro estados em que o jogador pode estar.

Este "*enum*" será usado na classe "LiderController", mais concretamente no método "ChangeState", em que é usado um ciclo condicional "*switch*", o qual avalia qual o estado em que o líder se encontra e para o qual vai transitar, acionado depois o respetivo método "RunState".

Desta maneira, garante-se uma separação clara de responsabilidades em conformidade com o padrão "*state machine*".

O script "LiderController", encontra-se representado na **Figura 119**, incluída no Anexo I – Detalhes de Programação.

3.7.2 Estado de entrada

O primeiro estado implementado é o estado de entrada. Este tem como objetivo colocar o líder no ecrã de jogo, através de um movimento descendente até que se posicione numa posição previamente definida no ecrã de jogo.

Foi criada a primeira classe que herda de "LiderBaseState", com o nome de "LiderEnter".

A "LiderEnter", tem apenas dois atributos necessários para o comportamento do inimigo são eles, a “speed” e o “enterPoint”.

A movimentação, que é feita dentro de uma “coroutine” chamada de “RunEnterState”, esta verifica se a posição atual do líder é ou não igual a posição definida, caso seja verdade, o inimigo fica imóvel, caso contrário o inimigo continua a movimentar-se.

Quando o líder atinge o ponto definido, a “coroutine” termina e é feita a mudança de estado.

A implementação detalhada desta classe pode ser encontrada no Anexo I – Detalhes de Programação, **Figura 120**.

Na **Figura 84**, é ilustrado o estado de entrada do líder. É possível observar a sua movimentação, dado que a captura do lado esquerdo foi feita no início do movimento descendente do inimigo e a captura do lado direito mostra que este chegou ao ponto previamente definido.

Figura 84 - Estado de entrada do "Lider"

3.7.3 Estado de disparo

O segundo estado implementado é o estado de disparo, correspondente ao momento em que o líder irá fazer os seus disparos de forma a eliminar o jogador.

Foi necessário definir os pontos de disparo de cada um dos líderes existentes, de acordo com o definido no *GDD*.

Da mesma forma que foi criada a classe "LiderEnter", foi criada a "LiderFire", também subclasse de "LiderBaseState".

Para além dos disparos, será atribuída ao líder uma movimentação randómica, com uma determinada velocidade associada, que lhe permitirá deslocar-se em ziguezague no topo do ecrã de jogo. Entre estas movimentações, o líder contará ainda com um atributo responsável por definir o intervalo entre disparos, bem como o tipo de míssil a ser utilizado.

De forma a introduzir mais dinâmica neste confronto, o tempo de duração do mesmo é feito de forma randómica. Este tempo sendo atingido, é feita uma decisão se o líder permanece no estado de disparo ou transita para o estado de ataque especial.

A implementação desta classe “LiderFire” pode ser encontrada no Anexo I – Detalhes de Programação, na **Figura 121**.

A **Figura 85**, mostra o resultado desta implementação. É possível observar, que o líder já se deslocou para uma posição aleatória e iniciou os disparos sobre a nave do jogador. Podem ser vistos quatro mísseis disparados por este líder o que significa que o mesmo possui quatro pontos de disparo.

Figura 85 - Estado de disparo do “Lider”

3.7.4 Estado de ataque especial

A implementação do estado de ataque especial segue as mesmas diretrizes dos estados anteriores, para tal foi criada a classe com o nome de "LiderSpecial".

Quando este estado é ativado, o líder deve dirigir-se para uma posição previamente definida numa zona central na parte superior do ecrã de jogo, sendo necessário recorrer ao método de “*Start*” para que tal aconteça.

O líder atingindo este ponto, irá dar começo ao seu ataque, para que este aconteça é instanciado o “*prefab*” criado para o efeito com o nome "SpecialShooting", este tem associada a classe "SpecialAttack", responsável pelo comportamento deste disparo.

Este disparo, movimenta-se de forma descendente e durante um espaço de tempo aleatório entre 0.5 e 1.5 segundos. Terminado este tempo, este objeto é destruído e dá-se um efeito de explosão deste disparo convertendo-se em vários disparos mais pequenos, estes dirigem-se em várias direções de forma a atingir a nave do jogador seja qual for a sua posição no ecrã.

Após a conclusão deste ataque especial, o líder muda de estado de forma automática retornando para o estado de disparo normal.

A implementação da classe "LiderSpecial" e "SpecialAttack", podem ser encontradas no Anexo I – Detalhes de Programação, nas **Figura 122** e **123**, respetivamente.

A **Figura 86**, mostra duas capturas de ecrã que ilustram a implementação do estado de ataque especial. No lado esquerdo, é possível verificar que o líder já se deslocou para o ponto definido e disparou o primeiro projétil respetivo ao ataque especial. Na captura do lado direito, é mostrado quando este disparo se multiplica em vários disparos mais pequenos e cada um deles segue uma direção distinta de forma a poder acertar na nave do jogador e tornar o ataque mais eficiente.

Figura 86 - Estado de ataque especial do "Lider"

3.7.5 Estado de morte

Para a implementação deste último estado, foi necessário criar previamente uma classe denominada “LiderStats”, responsável por definir o comportamento do líder no que diz respeito ao dano sofrido e a sua morte.

Esta classe, herda da superclasse "Enemy" permitindo dessa forma dar "*override*" dos seus métodos. No método "Hitted", será feita a chamada da animação referente ao dano do líder, enquanto no método "Destroyed", será realizada a mudança para o estado de morte e é instanciada a animação de explosão do líder.

A implementação da classe "LiderStats", esta presente na **Figura 124**, no Anexo I – Detalhes de Programação.

O passo seguinte é criar a classe "LiderDeath", que representa o estado final em que o líder se pode encontrar.

Nesta classe, apenas foi feito o “*override*” do método "RunState", de forma a chamar o "StartResolveSequence" responsável por finalizar o capítulo com sucesso.

A **Figura 87**, mostra esta implementação.

```
public class LiderDeath : LiderBaseState
{
    2 references
    public override void RunState()
    {
        EndGameManager.endGameManager.StartResolveSequence();
        gameObject.SetActive(false);
    }

}
```

Figura 87 - Classe "LiderDeath"

O resultado obtido após esta implementação pode ser observado na **Figura 88**, onde se verifica uma grande explosão, indicando a morte do “Lider” no jogo.

Figura 88 - Estado de morte do "Lider"

3.7.6 Spawn

O "spawn" do líder, deve apenas ocorrer quando o tempo de duração do capítulo é atingido.

Este comportamento, está implementado na classe "WinCondition". Nesta é declarada uma variável booleana chamada de "canSpawnLider", indicando quando o líder pode aparecer no ecrã de jogo. Caso esta variável seja verdadeira, o "Lider" é então criado através da classe "AlienSpawner", nesta é associado o prefab do “Lider” correspondente ao capítulo, de acordo com o que foi definido anteriormente no “*Game Design Document”*.

É desta forma simples, mas eficaz que ocorre a criação do líder no fim de cada capítulo do jogo.

3.8 Moedas

Neste subcapítulo, será explicada a implementação do objeto moedas no videojogo.

Dessa forma é feita a divisão desta implementação em duas partes, a primeira será a descrição de como foi criado o *HUD*, que mostra a contagem das moedas obtidas no decorrer do jogo e a segunda parte, aborda a construção do objeto e como foram atribuídos os comportamentos e funcionalidades ao mesmo.

3.8.1 HUD

Para construir o sistema de contagem de moedas, o primeiro passo foi procurar um "*asset*" que se assemelhasse ao definido no *GDD,* tendo sido encontrado no site "pngtree" [19]. Este "*asset*" será também utilizado para representar o objeto moeda dentro do jogo.

O passo seguinte, foi criar o objeto "CoinsCounter". Este irá conter dois objetos filhos, o "CoinsText", onde será apresentado o valor das moedas que o jogador possui e o "Coin", ao qual é atribuída a imagem escolhida anteriormente.

A **Figura 89**, mostra esta configuração, onde também é possível observar que estes objetos são parte do objeto "Canvas".

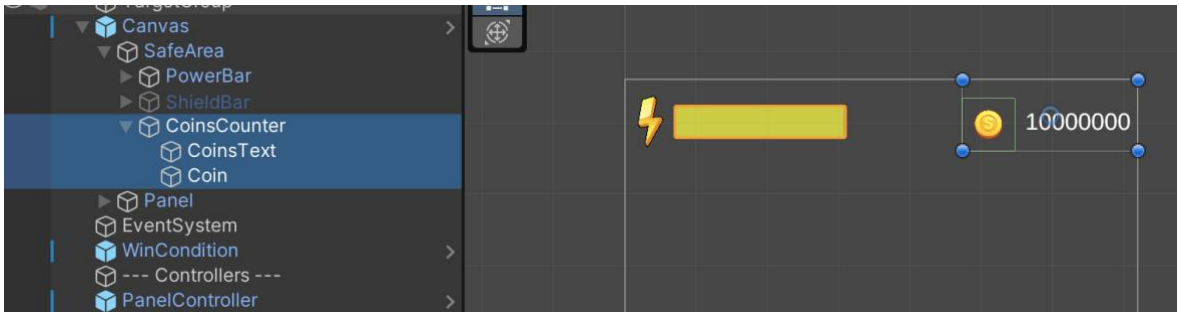

Figura 89 - Implementação do HUD das moedas

3.8.2 Criação e gestão de moedas

Criado o *HUD*, foi necessário efetuar algumas alterações na classe "EndGameManager". Uma destas alterações, foi atribuir uma nova variável do tipo "*TextMeshProGui*", com o objetivo de passar a referência do objeto de texto criado anteriormente. Outra alteração necessária a esta classe, foi a criação do método de registo desta referência externa tal como foi feito para os "*pop-ups*".

O passo seguinte, foi criar a classe "CoinsRegistration" que obtém o objeto do tipo "*TextMeshProGui*" e faz o registo do mesmo através da chamada ao método criado anteriormente. Por último, é feita a atualização do texto do objeto, recorrendo ao "*PlayerPrefs*" onde é feito o armazenamento do valor das moedas através da chave "Coins".

Esta implementação, pode ser observada na **Figura 90**.

```
public class CoinsRegistration : MonoBehaviour
{

    0 references
    void Start()
    {
        TextMeshProUGUI textForRegistration = GetComponent<TextMeshProUGUI>();
        EndGameManager.endGameManager.RegisterCoinsText(textForRegistration);
        textForRegistration.text = PlayerPrefs.GetInt("Coins").ToString();

    }

}
```

Figura 90 - Implementação da "CoinsRegistration"

Será necessário fazer mais atualizações à classe "EndGameManager", onde foi criada a variável "coins" e o método "UpdateCoins" que será o responsável por atualizar o valor apresentado no texto.

O método “UpdateCoins”, será chamado duas vezes na própria classe, uma quando o jogador consegue concluir um capítulo com sucesso, este irá receber o bónus de quinze moedas e o outro momento é quando a nave do jogador colide com o objeto moeda.

Desta forma, foi criada a classe "Coins" que será associada ao “*prefab*” "Coin" na qual será usado o método de “*OnTriggerEnter2D*”, responsável por verificar as colisões do objeto com a nave do jogador.

De acordo com o *GDD*, as moedas deveriam ser criadas sempre que um inimigo fosse eliminado. Ao fim de alguns testes, este comportamento foi ajustado de forma que apenas os asteroides possam dar "*spawn*", de moedas, já que os inimigos quando mortos já dão o "*spawn*" de “*power-ups*”, de forma a equilibrar a criação destes objetos.

Para tal, foi usado o "*ScriptableObject*" denominado de "SO_Spawner" tal como foi feito para o caso dos "*power-ups*", contudo agora foi atribuído ao objeto asteroide.

Na **Figura 91**, é apresentado o resultado da implementação das moedas no jogo. É possível verificar que após a destruição de um dos asteroides, foi gerada uma moeda que pode ser recolhida pelo jogador. Ainda na figura, é possível identificar o contador de moedas no canto superior direito, com o número de moedas atuais pertencentes ao jogador.

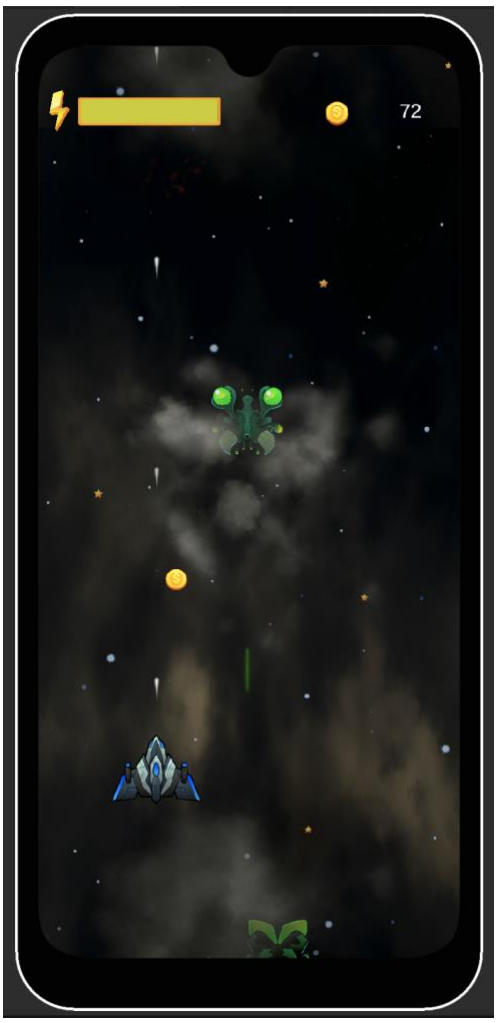

Figura 91 - Implementação das moedas no jogo

3.9 Música e efeitos sonoros

Neste subcapítulo, será descrita a implementação dos sons e efeitos sonoros no videojogo, desde sons que serão atribuídos aos capítulos do jogo até aos efeitos sonoros para quando as naves disparam ou quando existem explosões.

O primeiro passo, consistiu numa pesquisa de sons de fundo para os capítulos presentes no jogo, para propósito, foi usada a "*Unity Asset Store*" como principal fonte de recursos.

O primeiro pacote identificado como ideal para músicas de ambiente dos cinco capítulos do jogo foi o *"Free – Sci-Fi and Cyberpunk Music Pack – SLD Audio"* [20]. Para completar a componente sonora de fundo, foram encontrados os sons adequados para os ecrãs de início e de seleção de capítulos no pacote *"Universe Sounds Free Pack – Marcin Klosowski"* [21].

Relativamente aos efeitos sonoros, o primeiro som procurado foi o do disparo da nave do jogador. O efeito considerado mais adequado foi retirado do pacote "*Free Laser Weapons"* [22]. Com o som do disparo definido, procuraram-se efeitos para as explosões dos asteroides e para os momentos de morte dos alienígenas e dos seus líderes. Estes, bem como, os sons reproduzidos quando o jogador recolhe um ***"****power-up"*, cada um com um som distinto, foram obtidos a partir do ***"****Free Sound Effects Pack"* [23].

Em seguida, foi necessário encontrar um efeito sonoro para a recolha de moedas. A intenção era que este som remetesse para os jogos clássicos, como ***"****Super Mario"*. Após alguma pesquisa, encontrou-se um som adequado no site "*pixabay*", denominado de *"coin"* [24].

Por fim, foram também adicionadas vozes para os ecrãs de final de capítulo, uma voz de vitória, reproduzida quando o jogador é bem-sucedido [25], e a clássica expressão *"Game Over",* no caso de derrota [26].

A implementação dos sons de fundo, foi bastante simples, apenas foi necessário criar um novo "*game object*" ao qual se chamou de "BackGroundSound", no qual foi adicionado o componente "*Audiosource*". Neste é inserido o som desejado no atributo "*audioClip*" e foram ativas as opções de "*Play On Awake*" e "Loop", desta forma, a música será reproduzida automaticamente e de forma contínua.

Relativamente aos efeitos sonoros a abordagem foi diferente, uma vez que estes sons são concebidos para serem reproduzidos em momentos específicos, como por exemplo o som emitido quando a nave dispara.

Na classe "PlayerShooting", foram feitas algumas atualizações. Foi adicionada a variável "audioSource", de forma a ser possível chamar o método "Play" dentro do "Shoot", desta forma, será possível tocar o som desejado quando o tiro é realizado pela nave.

No caso dos colecionáveis presentes no jogo, a abordagem de implementação consistiu na adição de uma variável do tipo "*AudioClip*", que irá ser usada dentro do método "*OnTriggerEnter2D*" de forma a fazer-se uso do "*PlayClipAtPoint*" quando a colisão acontece. Esta abordagem, é uma maneira eficiente de reproduzir sons curtos, como é o caso.

No caso das explosões, utilizaram-se os "*prefabs*", de forma a incorporar um componente "*AudioSource*". Neste componente, é passado o clip de áudio e apenas se mantém selecionada a opção "*Play On Awake*", o que garante que o som é reproduzido assim que este objeto se torna ativo na cena.

Por último, no que diz respeito aos sons de vitória e derrota estes são reproduzidos quando os respetivos "*pop-ups*" aparecem no jogo. É utilizada a classe "EndGameManager", na qual são associadas duas variáveis do tipo "*AudioClip*", uma para o som de vitória e outra para o de derrota. O som de vitória será reproduzido no método "WinGame", enquanto o de derrota será reproduzido no método "LoseGame".

Em suma, a implementação das músicas de fundo e dos efeitos sonoros no videojogo decorreu com sucesso, estando tudo em conformidade com o que se encontrava planeado no "*Game Design Document*".

4. Testes

Neste capítulo de testes, será descrito todo o processo relativo a criação do arquivo *APK – Android Package Kit*, do jogo. Este ficheiro, é o executável necessário para que o jogo possa ser processado no sistema operativo *Android*, para qual o jogo foi construído.

Outro ponto a ser abordado neste capítulo, são os testes efetuados no ambiente *Android*, ao invés dos que tinham sido feitos até ao momento no ambiente de desenvolvimento do *Unity*.

4.1 Criação do *APK*

O *Unity* oferece ao programador, uma forma bastante simples de criar executáveis. Para tal, é necessário aceder ao menu *"Edit > Project Settings > Player"*. Neste ecrã, define-se a variável *"Company Name"*, o *"Product Name"*, que neste caso é o título do jogo, "*Supernova QuEST"* e ainda o ícone da aplicação, que será a imagem representativa do ficheiro executável no dispositivo móvel. Para este ícone, utilizou-se a imagem do planeta que aparece no ecrã inicial do jogo.

O passo seguinte é a criação do executável. Para isso, deve aceder-se ao menu *"File > Build Settings***"**, onde já está configurado que a plataforma de destino é o sistema operativo Android.

De seguida, liga-se o dispositivo móvel com o sistema operativo Android ao computador onde está a ser desenvolvido o projeto. Este será detetado automaticamente e surgirá na lista de dispositivos disponíveis, na opção *"Run Device".* Depois de selecionar o dispositivo, basta clicar em *"Build and Run"* para iniciar a geração do ficheiro *APK*.

No final deste processo, o jogo instala-se no dispositivo e é executado automaticamente.

A **Figura 92** ilustra a configuração usada para criar o *APK*. É também importante, assegurar que todas as cenas do jogo estão selecionadas e organizadas na ordem correta.

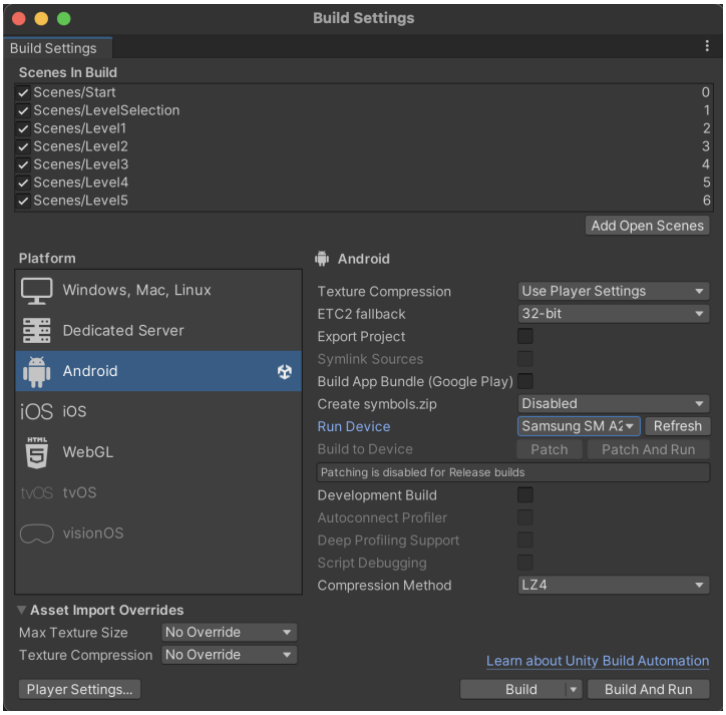

Figura 92 - Configuração de criação do APK

4.2 Teste em Android

De forma a iniciar os testes no sistema operativo Android, foi feita uma “build” para um dispositivo Samsung Galaxy A20e com a versão 11 do Android.

Instalada a aplicação e feito o seu arranque, verificou-se que o ecrã inicial surgiu da forma esperada, a música de fundo reproduziu de forma automática e todos os elementos visuais estavam presentes na posição correta.

Carregando no botão de “Play”, este direciona o jogador para o ecrã de seleção de capítulos. Quando é feita esta mudança de ecrã, pode-se observar o efeito de "Fader", seguido do ecrã de "Loading". Como o dispositivo que esta a ser usado, tem menos capacidade de processamento que a máquina usada para o desenvolvimento do jogo, o efeito de preenchimento da barra foi bem percetível, confirmando assim o correto funcionamento da mesma.

No ecrã de seleção de capítulos todos os elementos gráficos estavam de acordo com o design pretendido, verificou-se ainda uma alteração na música de fundo nesta transição entre ecrãs, indicadora de que tudo está correto quanto a reprodução de sons de fundo.

Como é uma sessão de jogo nova, o único capítulo disponível é o planeta “Rosara”, feita a sua seleção o jogador é levado para dentro da ação, sendo colocado ao comando da nave por predefinição "Azure Horizon".

A jogabilidade revelou-se estável, com a nave a responder muito bem ao toque no ecrã e os disparos também ocorrem com a cadencia esperada para esta nave.

Um problema identificado anteriormente no ambiente de desenvolvimento, era o comportamento incorreto da nave ao colidir com as extremidades do ecrã, este não se verificou no ambiente de Android.

Os inimigos surgem conforme o previsto com as cadencias desejadas, como também o “Lider” que apenas aparece no fim do tempo definido para o respetivo capítulo.

No entanto, verificou-se, que os tempos atribuídos aos capítulos com base na estrutura de dificuldade poderiam ser reduzidos.

Foi também feito um ajuste a frequência de "*spawn*" dos "*power-ups*", de forma a verificar o seu comportamento. De modo geral, todos funcionaram de forma correta apenas o “*power-up*” "Ultra Shoot", demonstrou poder ser considerado "*overpower*", porque em alguns dos casos foi possível eliminar um “Lider” com apenas alguns disparos, isto pode retirar parte da dificuldade prevista para o jogo.

Quanto às moedas mostrou ser um elemento interessante adicionado a jogabilidade, uma vez que, encoraja o jogador a não se focar apenas nos alienígenas, mas também nos asteroides com o objetivo de arrecadar moedas.

Estas moedas numa versão futura, servirão para comprar novas naves na loja do jogo. Contudo, essa funcionalidade ainda não se encontra incluída nesta versão, dado que a loja não foi implementada.

Nos capítulos seguintes, a dificuldade aumentou de forma progressiva. A quantidade de inimigos cresce e os líderes vão-se tornando mais desafiantes. A única exceção foi o último líder, “Toxicaris”, cuja velocidade e cadência de disparo são bastante elevadas, tornando a vitória quase impossível caso o jogador não apanhe um “*power-up*” antes da sua entrada na cena.

Os sons de fundo estiveram presentes em todos os capítulos, enquadrando-se bem na atmosfera de jogo, assim como, os efeitos sonoros que funcionaram de forma satisfatória, embora em alguns casos, nomeadamente nos “*power-ups*”, se tenha notado um ligeiro atraso na reprodução do som, embora não comprometa a jogabilidade, este aspeto poderá ser melhorado com ficheiros de áudio mais curtos ou de maior qualidade.

Relativamente às animações, tudo correspondeu ao comportamento esperado, as cores estavam bem aplicadas, os movimentos fluíam corretamente e nenhum elemento destoava, em linha com o que já havia sido validado no ambiente de testes de desenvolvimento.

De forma geral, o jogo comportou-se como esperado, a principal sugestão passa pela revisão da duração dos capítulos, que poderá ser encurtada. Uma possível solução seria a introdução de mais capítulos, com novos inimigos e líderes distintos, de forma a garantir maior variedade e equilíbrio na experiência de jogo à medida que o jogador progride.

A **Figura 93** encontra-se dividida em duas partes, à esquerda, é apresentada uma captura de ecrã do dispositivo móvel com o ícone da aplicação, no sistema operativo Android, já na direita, é exibido o ambiente de jogo do capítulo “Rosara” a correr no mesmo sistema operativo.

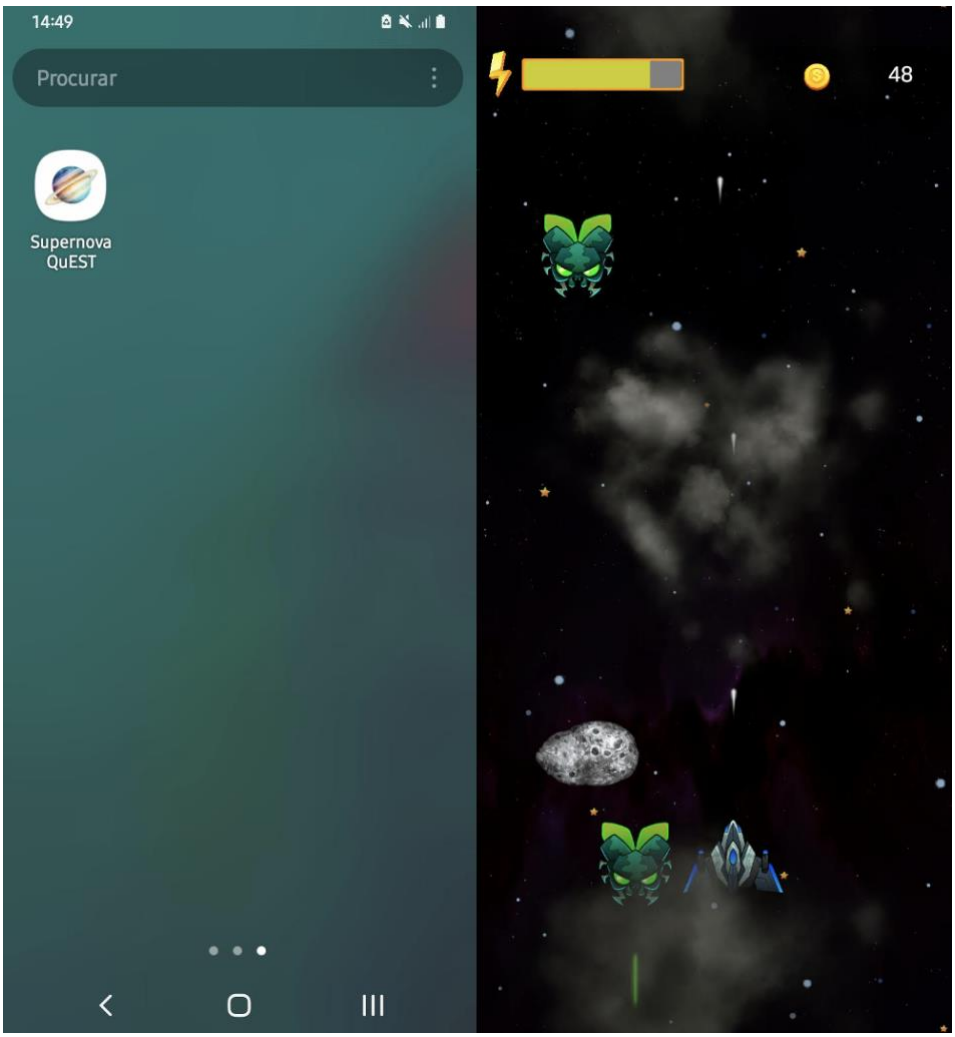

Figura 93 - Jogo no sistema operativo Android

5. Conclusão e Trabalho Futuro

Este capítulo, dedica-se a apresentar os principais contributos documentados neste relatório, os desafios ultrapassados, os objetivos alcançados e os que ficaram por alcançar.

Foi verificada, a importância da construção de um “Game Design Document” na primeira fase do projeto. Através deste documento, a implementação do jogo tornou-se quase linear, uma vez que se teve bastante cuidado na sua redação. A construção do jogo, foi dividida em etapas que foram sendo compridas de forma satisfatória.

Foi interessante perceber que foram aplicados vários conhecimentos adquiridos durante o percurso académico, desde conceitos de programação orientada a objetos, como "design patterns", estudados nas cadeiras de programação. Foram igualmente aplicados conhecimentos adquiridos em cadeiras de interface pessoa máquina e computação gráfica, demonstrando a importância destas cadeiras para a concretização deste projeto. Em complemento, revelou-se crucial o conhecimento adquirido através de pesquisas na documentação da plataforma de *Unity*, como em blogs dedicados ao desenvolvimento de videojogos.

Quanto a trabalho futuro, a inclusão da *framework* "GamEducation" é fundamental tendo o contexto do projeto em questão.

A implementação do jogo "Supernova QuEST", mostrou-se muito desafiante devido a minha situação laboral, o que dificultou a integração da *framework* "GamEducation", como a própria conclusão do videojogo.

Esta *framework* está preparada para receber jogos através de ficheiros *APK*, contudo, para que essa integração seja possível é necessária a implementação de algum código adicional no projeto, uma vez que quando foi construída esta abordagem foi testada num ambiente de desenvolvimento Android nativo e não no *Unity*, que é a tecnologia usada para a construção deste videojogo. Foi então, realizada uma pesquisa com o objetivo de encontrar uma possível solução para este problema, de modo a facilitar a integração do jogo com a *framework*. Dentro das poucas soluções encontradas, a que pareceu mais fiável seria uma presente em um documento encontrado no manual do *Unity* com o seguinte título “Java and Kotlin source plug-ins” [27]. Contudo, não é garantido que esta abordagem seja a correta, dado que a mesma não foi testada.

Apesar dos vários desafios, o resultado obtido deste projeto é bastante satisfatório, tudo o que foi concluído teve bastante trabalho associado, dedicação e grande foco no detalhe.

O "Supernova QuEST", é um jogo que apresenta um elevado potencial para alcançar o seu público-alvo. Precisa apenas de alguns ajustes, para que possa ser distribuído em conjunto com a *framework* educacional "GamEducation" alcançando o seu objetivo primordial que é a educação.

6. Referências

[1] Unity Documentation, “Prefabs”, 6000.1. Accessed: Mar 2025. [Online]. Available:https://docs.unity3d.com/Manual/Prefabs.html

[2] Unity Documentation, “Introduction to scenes”, 6000.1. Accessed: Mar 2025. [Online]. Available:https://docs.unity3d.com/Manual/CreatingScenes.html

[3] Asset Store, “*2D Space Kit*”, Accessed: Nov 2024. [Online]. Available:https://assetstore.unity.com/packages/2d/environments/2d-space-kit-27662#content

[4] Asset Store, “Free Stylized 2D Space Shooter Pack”, Accessed: Nov 2024. [Online]. Available:https://assetstore.unity.com/packages/2d/free-stylized-2d-space-shooter-pack-245185#content

[5] Unity Documentation, “Introduction to GameObjects”, 6000.1. Accessed: Mar 2025. [Online]. Available:https://docs.unity3d.com/Manual/GameObjects.html

[6] Unity Documentation, “Canvas”, 2020.1. Accessed: Mar 2025. [Online]. Available:https://docs.unity3d.com/2020.1/Documentation/Manual/UICanvas.html

[7] Christina Creates Games, “Unity UI Canvas Anchors explained”, Accessed: May 2025. [Online]. Available:https://www.youtube.com/watch?v=Lf8gbfEHgcI

[8] Unity Documentation, “9-slicing”, 6000.1. Accessed: May 2025. [Online]. Available:https://docs.unity3d.com/Manual/sprite/9-slice/9-slicing.html

[9] OpenGameArt, “Explosions”, Author: Graul98, Accessed: May 2025. [Online]. Available:https://opengameart.org/content/explosions-1

[10] Unity Documentation, “CanvasGroup”, 6000.1. Accessed: May 2025. [Online]. Available:https://docs.unity3d.com/ScriptReference/CanvasGroup.html

[11] Medium, “Game Design Pattern — Using Singletons in Unity”, Author: Ray, Accessed: May 2025. [Online]. Available:https://medium.com/codex/game-design-pattern-using-singletons-in-unity-acbd05d8ac9d

[12] Asset Store, “Simple Button Set 01”, Accessed: May 2025. [Online]. Available:https://assetstore.unity.com/packages/2d/gui/icons/simple-button-set-01-153979

[13] Medium, “Unity - Fader”, Author: Benjamin Calvin, Accessed: May 2025. [Online]. Available:https://medium.com/@benjamin.calvin/unity-fader-9b2f1126f79f

[14] Unity Documentation, “AsyncOperation”, 6000.1. Accessed: May 2025. [Online]. Available:https://docs.unity3d.com/6000.1/Documentation/ScriptReference/AsyncOperation.html

[15] Unity Documentation, "PlayerPrefs", 6000.1. Accessed: May 2025. [Online]. Available:https://docs.unity3d.com/6000.1/Documentation/ScriptReference/PlayerPrefs.html

[16] PngTree, "Saturn Watercolor Watercolor Illustration With Solar System Planets", Accessed: May 2025. [Online]. Available:https://pngtree.com/freepng/saturn-watercolor-watercolor-illustration-with-solar-system-planets_13009451.html

[17] Unity Documentation, "ScriptableObject", 6000.1. Accessed: May 2025. [Online]. Available:https://docs.unity3d.com/6000.1/Documentation/Manual/class-ScriptableObject.html

[18] Medium, "Introduction to the Unity State Machine Pattern", Author: Murat Han Acipayam, Accessed: May 2025. [Online]. Available:https://mracipayam.medium.com/introduction-to-the-unity-state-machine-pattern-ad3bce7d987c

[19] PngTree, "Coin", Accessed: May 2025. [Online]. Available:https://pngtree.com/element/down?id=NTk4NjMwMQ==&type=1&time=1743268174&token=MWY3NzIwNDg5YTk4ZmZmY2ZmNmI1ODU0MjZkMjA5ODk=&t=0

[20] Asset Store, "Free – Sci-Fi and Cyberpunk Music Pack", Author: SLD Audio, Accessed: May 2025. [Online]. Available:https://assetstore.unity.com/packages/audio/ambient/sci-fi/free-sci-fi-and-cyberpunk-music-pack-264590

[21] Asset Store, "Universe Sounds Free Pack", Author: Marcin Klosowski, Accessed: May 2025. [Online]. Available:https://assetstore.unity.com/packages/audio/ambient/sci-fi/universe-sounds-free-pack-118865#content

[22] Asset Store, "Free Laser Weapons", Author: Daniel SoundsGood, Accessed: May 2025. [Online]. Available:https://assetstore.unity.com/packages/audio/sound-fx/weapons/free-laser-weapons-214929#content

[23] Asset Store, "Free Sounds Effects Pack", Author: Olivier Girardot, Accessed: May 2025. [Online]. Available:https://assetstore.unity.com/packages/audio/sound-fx/free-sound-effects-pack-155776#content

[24] Pixabay, "coin", Author: chieuk, Accessed: May 2025. [Online]. Available:https://pixabay.com/sound-effects/coin-257878/

[25] Pixabay, "Game Over", Author: Freesound Community, Accessed: May 2025. [Online]. Available: https://pixabay.com/sound-effects/083822-8-bit-quotgame-overquot-82872/

[26] Pixabay, "Victory", Author: RibhavAgrawal, Accessed: May 2025. [Online]. Available: https://pixabay.com/sound-effects/victorymale-version-230553/

[27] Unity Documentation, “Java and Kotlin source plug-ins”, 6000.1. Accessed: Jun 2025. [Online]. Available: https://docs.unity3d.com/6000.1/Documentation/Manual/AndroidJavaSourcePlugins.html

7. Anexos

7.1 Anexo I – Detalhes de programação

Na **Figura 94**, encontra-se a implementação da classe "AlienShoot". Esta contem três atributos: o "*speed*", do tipo "*float*", que dará a velocidade de descida do projétil disparado pelo inimigo, o atributo "*damage*", também do tipo "*float*", que indica o valor que será retirado a nave do jogador e por último o atributo que recebe a referência ao componente "*RigidBody2D*", responsável por aplicar a física ao tiro.

Quanto a métodos, temos o "*Start*()", onde é definida a velocidade e direção dos disparos. Temos também, o método "*OnTriggerEnter*()", que deteta colisões com a nave do jogador, aplicando o dano correspondente que deve ser retirado desta. Por último, o método "*OnBecameInvisible*()" que faz com que o projétil seja removido do jogo quando este se torna invisível de forma a poupar recursos de processamento.

```
using UnityEngine;

0 references
public class AlienShoot : MonoBehaviour
{

    1 reference | 1 reference
    [SerializeField] private float speed, damage;
    2 references
    private Rigidbody2D rigidBody2D;
    0 references
    void Start()
    {
        rigidBody2D = GetComponent<Rigidbody2D>();
        rigidBody2D.velocity = Vector2.down * speed;
    }

    0 references
    private void OnTriggerEnter2D(Collider2D collider2D)
    {
        if (collider2D.CompareTag("Player"))
        {
            collider2D.GetComponent<PlayerStats>().takeDamage(damage);
            Destroy(gameObject);
        }
    }

    0 references
    private void OnBecameInvisible()
    {
        Destroy(gameObject);
    }
}
```

Figura 94 - Classe "AlienShoot"

A **Figura 95**, apresenta a implementação da classe que cria os alienígenas no ecrã de jogo, denominada de “AlienSpawner”.

Esta classe, tem a destacar os seguintes atributos: o “alien” do tipo “*GameObject*”, o atributo “alienSpawnTime” este do tipo “*float*”, que define o intervalo de tempo entre cada criação destes “aliens”, o “spawnerPositionY” que determina a posição na vertical de onde os inimigos vão ser criados, entre outros.

No que diz respeito aos métodos destaca-se o “AlienSpawn()”, que é o responsável por criar “aliens” no campo de jogo. Existe um temporizador interno “alienTime”, que quando o tempo de “*spawn*” é atingido é criada a instância do “alien” pretendido, numa posição aleatória no eixo de X. No fim, é feito o “*reset*” ao valor do temporizador interno e o método é constantemente chamado a cada “frame”, dado que o mesmo faz parte da função “Update()” do *Unity*.

```
using System.Collections;
using UnityEngine;

0 references
public class AlienSpawner : MonoBehaviour
{
    1 reference
    [SerializeField] private GameObject alien;
    1 reference
    [SerializeField] private float alienSpawnTime;
    4 references
    private Camera mainCamera;
    2 references | 2 references | 2 references | 3 references
    private float maxLeft, maxRight, spawnerPositionY, alienTime;

    0 references
    void Start()
    {
        mainCamera = Camera.main;
        StartCoroutine(SetBoundaries());
    }

    0 references
    void Update()
    {
        AlienSpawn();
    }

    1 reference
    private void AlienSpawn()
    {
        alienTime += Time.deltaTime;
        if (alienTime >= alienSpawnTime)
        {
            Instantiate(alien, new Vector3(
                Random.Range(maxLeft, maxRight), spawnerPositionY, 0),
                Quaternion.identity);
            alienTime = 0;
        }
    }

    1 reference
    private IEnumerator SetBoundaries()
    {
        yield return new WaitForSeconds(0.4f);
        maxLeft = mainCamera.ViewportToWorldPoint(new Vector2(0.15f, 0)).x;
        maxRight = mainCamera.ViewportToWorldPoint(new Vector2(0.85f, 0)).x;
        spawnerPositionY = mainCamera.ViewportToWorldPoint(new Vector2(0, 1.2f)).y;
    }
}
```

Figura 95 - Classe "AlienSpawner"

A **Figura 96**, representa a classe “Asteroid”. Esta contém como atributos principais o “minSpeed”, o “maxSpeed” e o “rotateSpeed”, todos eles do tipo “*float*”, estes serão os responsáveis por definir a velocidade e a rotação dos asteroides.

Quanto a métodos temos o “Start()”, onde é atribuída uma velocidade aleatória ao asteroide. O método “Update()”, que dá uma rotação constante ao asteroide em torno do seu eixo de Z, dando a este um efeito visual interessante e realista a queda do mesmo.

Esta classe inclui os métodos “Hitted()” e “Destroyed()”, herdados da classe abstrata “Enemy”, estes são responsáveis por despoletar as animações correspondentes ao impacto e a destruição do asteroide.

```
0 references
public class Asteroid : Enemy
{
    1 reference | 1 reference | 1 reference
    [SerializeField] private float minSpeed, maxSpeed, rotateSpeed;
    2 references
    private float speed;

    0 references
    void Start()
    {
        speed = Random.Range(minSpeed, maxSpeed);
        rigidBody2D.velocity = Vector2.down * speed;
    }

    0 references
    void Update()
    {
        transform.Rotate(0, 0, rotateSpeed * Time.deltaTime);
    }

    0 references
    private void OnTriggerEnter2D(Collider2D collider2D)
    {
        if (collider2D.CompareTag("Player"))
        {
            // decrease the power of the player
            collider2D.GetComponent<PlayerStats>().takeDamage(damage);
            // asteroid destroyed
            Destroy(gameObject);
        }
    }

    2 references
    public override void Hitted()…

    2 references
    public override void Destroyed()…

    0 references
    private void OnBecameInvisible()…
}
```

Figura 96 - Classe "Asteroid"

A **Figura 97**, mostra a implementação da classe "AsteroidSpawner".

Os seus atributos são os mesmos que foram usados na classe "AlienSpawner", a única exceção é o atributo "asteroids" que no caso é um "*array*" de "*game objects*", no qual se colocam os tipos de "*prefabs*" respetivos aos mesmos.

Quanto a métodos, destaca-se o “CreateAsteroid()”. Este é o responsável por fazer uma seleção aleatória de um dos tipos de asteroides passados no “*array*” de asteroides, o qual define a posição de criação destes no topo do ecrã fora da visão do jogador e aplica uma rotação aos mesmos. Outro aspeto que este método faz, é uma variação na escala de tamanho entre 0.9 e 1.1, trazendo para o jogo diversidade nos asteroides gerados.

```
public class AsteroidSpawner : MonoBehaviour
{
    2 references
    [SerializeField] private GameObject[] asteroids;
    1 reference
    [SerializeField] private float spawnerTime;
    3 references
    private float timer = 0f;
    2 references
    private int asteroidType;
    4 references
    private Camera mainCamera;
    2 references | 2 references | 2 references
    private float maxLeft, maxRight, spawnerPositionY;

    0 references
    void Start()…

    0 references
    void Update()
    {
        timer += Time.deltaTime;
        if (timer > spawnerTime)
        {
            CreateAsteroid();
            timer = 0f;
        }
    }

    1 reference
    private void CreateAsteroid()
    {
        asteroidType = Random.Range(0, asteroids.Length);

        GameObject myObject = Instantiate(
            asteroids[asteroidType], new Vector3(Random.Range(maxLeft, maxRight),
             spawnerPositionY, 0), Quaternion.Euler(0, 0, Random.Range(0, 360)));

        float asteroidSize = Random.Range(0.9f, 1.1f);
        myObject.transform.localScale = new Vector3(asteroidSize, asteroidSize, 1);

    }
```

Figura 97 - Classe "AsteroidSpawner"

A **Figura 98**, ilustra a classe “Enemy”. Esta classe, serve como base para a criação dos diferentes inimigos presentes no jogo e contém dois atributos principais: o “health” e o “damage”, do tipo “*float*”. Um representa a vida que o inimigo possui e o outro o dano que este irá descontar no caso de receber danos feitos por parte da nave do jogador.

Quanto a métodos, o único que tem uma implementação é o “Damaged()”, que recebe o valor de dano que irá ser debitado da “health” do inimigo. Quando a vida deste atinge o valor igual ou inferior a zero, o método “Destroyed()” é invocado, de forma a destruir o inimigo do ecrã de jogo.

É feita a declaração dos métodos “Hitted()” e “Destroyed()”, estes estão definidos como virtual, o que permite que as subclasses possam personalizar a sua implementação da forma desejada, como por exemplo, a adição de animações e efeitos visuais.

```
using UnityEngine;

4 references
public class Enemy : MonoBehaviour
{
    2 references | 2 references
    [SerializeField] protected float health, damage;
    2 references
    [SerializeField] protected Rigidbody2D rigidBody2D;

    1 reference
    public void Damaged(float damage)
    {
        health -= damage;
        Hitted();
        if (health <= 0)
        {
            Destroyed();
        }
    }

    3 references
    public virtual void Hitted(){}

    3 references
    public virtual void Destroyed(){}
}
```

Figura 98 - Classe "Enemy"

A classe “Insetroid”, está ilustrada na **Figura 99**, esta herda a classe “Enemy”.

Entre os seus atributos principais destacam-se: o “speed”, que define a velocidade com que o inimigo se move na vertical; o “shootTime”, que estabelece o intervalo entre disparos; o “shootPoint”, que define o local de onde os projéteis são criados; o “shoot”, que corresponde ao objeto do projétil a ser instanciado e o “shootTimer”, um temporizador que controla a cadência dos disparos.

No que diz respeito a métodos, o “*Start*()” é responsável por iniciar o movimento do inimigo na vertical, aplicando-lhe uma velocidade através do seu componente físico. O método "*Update*()", faz a gestão dos disparos efetuados por cada “*frame*”, para tal, faz o incremento do temporizador interno, quando este atinge o valor definido no "shootTime", os disparos são então instanciados, na posição pré-definida no "shootPoint".

```
using UnityEngine;

0 references
public class Insetroid : Enemy
{

    1 reference | 1 reference
    [SerializeField] private float speed, shootTime;
    1 reference
    [SerializeField] private Transform shootPoint;
    1 reference
    [SerializeField] private GameObject shoot;
    3 references
    private float shootTimer;

    0 references
    void Start()
    {
        rigidBody2D.velocity = Vector2.down * speed;
    }

    0 references
    void Update()
    {
        shootTimer += Time.deltaTime;
        if (shootTimer >= shootTime)
        {
            Instantiate(shoot, shootPoint.position, Quaternion.identity);
            shootTimer = 0;
        }
    }

    0 references
    private void OnTriggerEnter2D(Collider2D collider2D)
    {
        if (collider2D.CompareTag("Player"))
        {
            collider2D.GetComponent<PlayerStats>().takeDamage(damage);
            Destroy(gameObject);
        }
    }

    2 references
    public override void Hitted()…

    2 references
    public override void Destroyed()
    {
        // Animation
        Destroy(gameObject);
    }

    0 references
    private void OnBecameInvisible()
    {
        Destroy(gameObject);
    }
}
```

Figura 99 - Classe "Insetroid"

A **Figura 100**, mostra a implementação da classe "Missil". Esta trata do funcionamento dos mísseis disparados pela nave do jogador.

São três os atributos pertencentes a esta classe: o "speed", que determina a velocidade com que o míssil se desloca, no caso na vertical, o "damage", que indica o valor de dano que será retirado ao inimigo em caso de colisão com o mesmo e por fim o "rigidBody2D", que aplica o sistema de física ao mesmo.

Entre os métodos, destaca-se o “OnTriggerEnter2D”, este deteta as colisões com objetos do tipo "Enemy", decrementando o valor de "damage" ao mesmo, através do método “Damaged()”, pertencente a classe “Enemy”.

```
using UnityEngine;

0 references
public class Missil : MonoBehaviour
{
    1 reference | 1 reference
    [SerializeField] private float speed, damage;
    1 reference
    [SerializeField] private Rigidbody2D rigidBody2D;

    0 references
    void Start()
    {
        rigidBody2D.velocity = transform.up * speed;
    }

    0 references
    private void OnTriggerEnter2D(Collider2D collider2D)
    {
        Enemy enemy = collider2D.GetComponent<Enemy>();
        enemy.Damaged(damage);
        Destroy(gameObject);
    }

    0 references
    private void OnBecameInvisible()
    {
        Destroy(gameObject);
    }
}
```

Figura 100 - Classe "Missil"

A implementação do script "ParallaxEffect" é mostrada na **Figura 101**. Este script é aplicado nos fundos usados para os capítulos do jogo, tornando-os dinâmicos.

Os três atributos pertencentes a este script são: a "speed", que controla a que velocidade os fundos se movimentam, o "backgroundHeight", onde é armazenado o valor da altura do fundo utilizado e por último a "startPosition", que define a posição inicial de onde o fundo vai iniciar a sua movimentação.

No método de "*Start*()", é definido o valor da posição inicial e da altura do fundo, no método "*Update*()" é onde se efetua o movimento descendente do fundo, tendo como base o atributo "speed". Quando este atinge o limite vertical, é então reposicionado para a sua posição inicial começando de novo o ciclo, de forma a proporcionar o efeito pretendido.

```
using UnityEngine;

0 references
public class ParallaxEffect : MonoBehaviour
{
    1 reference
    [SerializeField] float speed;
    2 references
    private float backgroundHeight;
    3 references
    private Vector3 startPosition;

    0 references
    void Start()
    {
        startPosition = transform.position;
        backgroundHeight = GetComponent<SpriteRenderer>().bounds.size.y;
    }

    0 references
    void Update()
    {
        // Vector3.down -> Vector3(0, -1, 0)
        // speed -> 5
        // Time.deltaTime -> calculated based on time, not frames
        transform.Translate(Vector3.down * speed * Time.deltaTime);

        // transform.position = value will decreasing with the upddate() call
        // startPosition.y = 0
        // backgroundHeight = 20.48
        // startPosition.y - backgroundHeight = -20.48
        if (transform.position.y < startPosition.y - backgroundHeight)
        {
            transform.position = startPosition;
        }
    }
}
```

Figura 101 - Classe "ParallaxEffect"

A **Figura 102**, apresenta a implementação da classe "PlayerController". É nesta, que se define o comportamento da nave do jogador com base na interação através do toque no ecrã.

Os seus principais atributos são: a “maincamera”, o “offset” e todos os atributos pertencentes a definição dos limites do ecrã de jogo.

Quanto a métodos destaca-se o “*Update*()”, este é o responsável por detetar os toques feitos no ecrã e atualizar a posição da nave em correspondência.

O método “*OnEnable()*” e “*OnDisable()”*, são parte da *API – Application Programming Interface* “*EnhancedTouch*” garantindo o correto controlo do toque ao longo do ciclo de vida do componente.

```
public class PlayerController : MonoBehaviour
{
    6 references
    private Camera mainCamera;
    3 references
    private Vector3 offset; // distance from te touch to the spaceship
    2 references | 2 references | 2 references | 2 references
    private float maxLeft, maxRight, maxUp, maxDown;
    0 references
    void Start()
    {
        mainCamera = Camera.main;
        StartCoroutine(SetBoundaries());
    }

    0 references
    void Update()
    {
        if (Touch.activeTouches.Count > 0) // verify if is touching the screen
        {
            if (Touch.activeTouches[0].finger.index == 0) // verify if only one finger is touching to prevent teleporting the ship with a second finger touching
            {
                Touch myTouch = Touch.activeTouches[0]; // store the active touch
                Vector3 touchPosition = myTouch.screenPosition; // get the touch position
                touchPosition = mainCamera.ScreenToWorldPoint(touchPosition); // transform to world points
                // to be possible to control the ship without clicking over the ship
                if (Touch.activeTouches[0].phase == TouchPhase.Began) // verify if the phase of the first touch is Began
                {
                    offset = touchPosition - transform.position; // distance between two positions
                }
                if (Touch.activeTouches[0].phase == TouchPhase.Moved) // verify if the phase of the first touch is Moved
                {
                    transform.position = new Vector3(touchPosition.x - offset.x,
                     touchPosition.y - offset.y, 0); // made the transform.position with the offset value subtracted
                }
                // Clamp is used to constrain a value within a specified range (value, min, max);
                transform.position = new Vector3(
                    Mathf.Clamp(transform.position.x, maxLeft, maxRight),
                    Mathf.Clamp(transform.position.y, maxDown, maxUp), 0);
            }
        }
    }

    0 references
    private void OnEnable()
    {
        EnhancedTouchSupport.Enable();
    }

    0 references
    private void OnDisable()
    {
        EnhancedTouchSupport.Disable();
    }

    1 reference
    private IEnumerator SetBoundaries()
}
```

Figura 102 - Classe "PlayerController"

A **Figura 103**, apresenta a implementação da classe “PlayerShooting”.

Esta classe inclui quatro atributos: o “missil” que representa o objeto a ser disparado pela nave; o “shootingPoint” que indica o local onde os mísseis são lançados pela nave; “shootingTimer” onde é definido o intervalo entre cada disparo e por fim o “timer” um temporizador interno usado na classe.

Quanto a métodos, destaca-se o “Update()” onde é feita a contagem do tempo para que o tiro seja disparado com a cadencia definida previamente. Quando este tempo é atingido, é então chamado o método “Shoot()” que instancia um novo míssil na posição especificada na variável “shootingPoint”.

```
using UnityEngine;

0 references
public class PlayerShooting : MonoBehaviour
{
    1 reference
    [SerializeField] private GameObject missil;
    1 reference
    [SerializeField] private Transform shootingPoint;
    4 references
    [SerializeField] private float shootingTimer;
    2 references
    private float timer;

    0 references
    void Start()
    {
        timer = shootingTimer;
    }

    0 references
    void Update()
    {
        shootingTimer -= Time.deltaTime;
        if (shootingTimer <= 0)
        {
            Shoot();
            shootingTimer = timer;
        }
    }

    1 reference
    private void Shoot()
    {
        Instantiate(missil, shootingPoint.position, Quaternion.identity);
    }
}
```

Figura 103 - Classe "PlayerShooting"

A **Figura 104** ilustra a implementação da classe "PlayerStats". Esta classe é responsável por gerir o "power" da nave controlada pelo jogador.

Existem apenas dois atributos: o "powerMax", no qual é determinado o valor máximo de vida da nave, e o outro atributo é o "power", que guarda o valor atual do "power" da nave.

O método “takeDamage()” recebe o “damage” como parâmetro, o qual é decrementado no valor de “power” da nave, quando este valor é inferior ou igual a zero, é feita a chamada ao método “Destroy()” de forma a destruir o próprio objeto.

```
using UnityEngine;

3 references
public class PlayerStats : MonoBehaviour
{
    1 reference
    [SerializeField] private float powerMax;
    3 references
    private float power;

    0 references
    void Start()
    {
        power = powerMax;
    }

    3 references
    public void takeDamage(float damage)
    {
        power -= damage;
        if (power <= 0)
        {
            Destroy(gameObject);
        }
    }
}
```

Figura 104 - Classe "PlayerStats"

A **Figura 105**, mostra as alterações necessárias feitas na classe "PlayerShooting" de forma a atualizar a barra de "power" presente na *UI* do jogo.

Desse modo, foi necessário importar a biblioteca "UnityEngine.UI", o que permite fazer uso deste tipo de componentes, como a "Image" usada na variável "powerFill".

No método “takeDamage()”, é feito o cálculo da quantidade de preenchimento que a barra irá remover tendo em conta o "damage" passado. Para tal, é feita a divisão entre o valor de “power” e “powerMax”, de forma a retornar um valor entre 0 e 1 que corresponde ao pretendido pelo “fillAmount”.

Esta linha de código facilmente identificada na figura, deve ser também colocada dentro do método de “Start()”, de forma que quando o jogo se inicia a barra esteja com o preenchimento máximo.

PlayerStats.cs

Assets > Scripts > PlayerStats.cs > PlayerStats > takeDamage

```
1   using UnityEngine;
2   using UnityEngine.UI;
3
    3 references
4   public class PlayerStats : MonoBehaviour
5   {
        2 references
6       [SerializeField] private Image powerFill;
        3 references
7       [SerializeField] private float powerMax;
        5 references
8       private float power;
9
        0 references
10      void Start()
11      {
12          power = powerMax;
13          powerFill.fillAmount = power / powerMax;
14      }
15
        3 references
16      public void takeDamage(float damage)
17      {
18          power -= damage;
19          powerFill.fillAmount = power / powerMax;
20          if (power <= 0)
21          {
22              Destroy(gameObject);
23          }
24      }
25  }
26
```

Figura 105 - Classe "PlayerStats"

A **Figura 106**, mostra a implementação da animação de explosão na classe "Insetroid". Esta explosão, ocorre no momento anterior a destruição do alienígena.

Existem duas chamadas ao método “Destroy()” dentro desta classe, no método "OnTriggerEnter2D()" e o "Destroyed()". Nestes dois é feita a criação da instância da animação através da chamada ao método “Instantiate()” do *Unity*, ambas as linhas estão identificadas na imagem.

Desta forma, a animação de explosão é reproduzida e após um “delay” de dois segundos definidos no script de “Explosion”, a animação é destruída e de seguida o objeto “Insetroid” é removido da cena.

Assets > Scripts > Insetroid.cs > Insetroid > Destroyed

```
using UnityEngine;

0 references
public class Insetroid : Enemy
{

    1 reference | 1 reference
    [SerializeField] private float speed, shootTime;
    1 reference
    [SerializeField] private Transform shootPoint;
    1 reference
    [SerializeField] private GameObject shoot;
    3 references
    private float shootTimer;

    0 references
    void Start()
    {
        rigidBody2D.velocity = Vector2.down * speed;
    }

    0 references
    void Update()
    {
        shootTimer += Time.deltaTime;
        if (shootTimer >= shootTime)
        {
            Instantiate(shoot, shootPoint.position, Quaternion.identity);
            shootTimer = 0;
        }
    }

    0 references
    private void OnTriggerEnter2D(Collider2D collider2D)
    {
        if (collider2D.CompareTag("Player"))
        {
            collider2D.GetComponent<PlayerStats>().takeDamage(damage);
            Instantiate(explosionPrefab, transform.position, transform.rotation);
            Destroy(gameObject);
        }
    }

    2 references
    public override void Hitted()
    {
        //Animation
    }

    2 references
    public override void Destroyed()
    {
        // Animation
        Instantiate(explosionPrefab, transform.position, transform.rotation);
        Destroy(gameObject);
    }

    0 references
    private void OnBecameInvisible()
    {
        Destroy(gameObject);
    }
}
```

Figure 106 - Classe "Insetroid"

Na **Figura 107**, encontra-se a implementação da animação “Hitted” na classe “PlayerStats”.

Para o efeito, foi criada a variável "animator" do tipo "*Animator*". Será através desta, que é possível fazer usar do método "SetTrigger" responsável por ativar esta animação. Para tal, dentro do método “takeDamage” é verificado se a animação pode ser reproduzida, caso seja verdade é passado o valor de "Hitted" pelo “*trigger*” definido no “*Animator Controller*”.

De forma a evitar ciclos de repetição da animação, foi criada a "*coroutine*" "AntiSpamAnimation" que controla a “*flag*” "canPlayAnimation", impondo um tempo de espera de 0.15 segundos entre a alteração do seu valor, de forma que este ciclo não ocorra.

```
    public void takeDamage(float damage)
    {
        if (shield.isProtected)
        {
            return;
        }

        power -= damage;
        powerFill.fillAmount = power / powerMax;

        if(canPlayAnimation)
        {
            animator.SetTrigger("Hitted");
            StartCoroutine(AntiSpamAnimation());
        }

        // decrease ultra shoot
        playerShooting.DecreaseShooting();

        if (power <= 0)
        {
            EndGameManager.endGameManager.gameOver = true;
            EndGameManager.endGameManager.StartResolveSequence();
            Instantiate(explosionPrefab, transform.position, transform.rotation);
            Destroy(gameObject);
        }
    }

    1 reference
    public void AddPower(int powerAmount) ...

    1 reference
    private IEnumerator AntiSpamAnimation()
    {
        canPlayAnimation = false;
        yield return new WaitForSeconds(0.15f);
        canPlayAnimation = true;
    }
}
```

Figura 107 - Implementação da animação "Hitted"

A **Figura 108**, apresenta a implementação da classe “WinCondition” responsável pelo controlo da duração de um capítulo e pela cessação da criação de inimigos na cena de jogo.

A classe contém três atributos: o “levelTime” utilizado para guardar o tempo estipulado para o capítulo, o “spawer” é um “*array*” que vai receber os “*game objets*” responsáveis pela criação dos inimigos e o “timer” que é uma variável auxiliar de contagem do tempo decorrido.

De forma a implementar a lógica definida para a duração dos capítulos, foi usado o método “Update()”. É neste, que é feito o incremento do tempo até que este seja igual ao tempo definido em “leveltime”. Caso esta condição se verifique, os "*spawners*" dos inimigos são desativados.

Esta classe, ainda irá sofrer alterações após a introdução do “Lider” no jogo.

```
0 references
public class WinCondition : MonoBehaviour
{
    1 reference
    [SerializeField] private float levelTime;
    2 references
    [SerializeField] private GameObject[] spawner;
    2 references
    private float timer;

    0 references
    void Update()
    {

        timer += Time.deltaTime;

        // verify if the timer achieved the level time
        if(timer >= levelTime)
        {
            // desactivate all the game objects spawned
            for (int i = 0; i < spawner.Length; i++)
            {
                spawner[i].SetActive(false);
            }

        }
    }
}
```

Figura 108 - Classe "WinCondition"

A **Figura 109**, mostra a implementação da classe “PanelController”.

Esta classe contém três atributos: o “canvasGroup” que recebe o componente utilizado para controlar a visibilidade e a interatividade do grupo de elementos da *UI* criada, o "winPopUp" e o "losePopUp" que recebem as referências aos objetos criados para ambos.

Foram ainda definidos dois métodos, “ActivateWinPopUp” e “ActivateLosePopUp”, responsáveis pela ativação dos respetivos *“pop-ups”*. Ambos os métodos tornam o grupo de UI visível ao definir o valor da propriedade “*alpha*” para um e ativam o objeto correspondente através da função “*SetActive(true)”.*

Esta classe irá sofrer ajustes futuros, nomeadamente, com a inclusão de um “*manager*”, que será responsável por fazer a correspondência entre os eventos do jogo e os elementos da interface.

```
2 references
public class PanelController : MonoBehaviour
{
    2 references
    [SerializeField] private CanvasGroup canvasGroup;
    1 reference
    [SerializeField] private GameObject winPopUp;
    1 reference
    [SerializeField] private GameObject losePopUp;
    0 references
    void Start()…

    1 reference
    public void ActivateWinPopUp()
    {
        canvasGroup.alpha = 1;
        winPopUp.SetActive(true);
    }

    1 reference
    public void ActivateLosePopUp()
    {
        canvasGroup.alpha = 1;
        losePopUp.SetActive(true);
    }
}
```

Figura 109 - Classe "PanelController"

O script responsável pela criação do "*Fader*", foi chamado de "FadeEffect" e está apresentado na **Figura 110**.

Fazem parte deste script vários atributos responsáveis por reproduzir este efeito, são eles: o "fadeEffect", uma vez que, é necessário aplicar o padrão “*singleton*” a esta classe de forma a manter apenas a existência de uma instância desta classe durante o jogo; a "canvasGroup", referente ao componente que permite controlar a opacidade desta interface essencial para criar o efeito; o "changeValue", que é o valor que define o valor de "*alpha*" e que deve ser alterado a cada iteração; a "waitingTime", que indica o tempo de espera em cada iteração e a "fadeStarted", funciona como uma "*flag*", que impede que os efeitos de “fadeIn” e “fadeOut” aconteçam simultaneamente.

Os métodos “FadeIn()” e “FadeOut()” são ambos “*Coroutines*”, o primeiro diminui o valor do “*alpha*” até que o ecrã fique totalmente transparente, o segundo faz o processo inverso, aumenta o valor de "*alpha*" até que passe de transparente para uma cor opaca.

Dentro do método de "FadeOut()" e depois de ser reproduzido o efeito de "*fade*", é feita a chamada ao "LoadScene()" onde é passado como argumento o "*level*", que corresponde ao índice do ecrã para onde vai transitar. De seguida, é aplicado um pequeno intervalo e é feita a chamada ao método “FadeIn()”, de forma a iniciar o mesmo efeito já dentro do novo ecrã.

A função "FadeOut()", é chamada dentro do método "FaderLoad()" que por sua vez será usada na classe "ButtonController".

Para que tudo funcione corretamente é necessário atualizar esta classe "ButtonController", substituindo a chamada direta ao método "LoadScene()" pela utilização desta nova classe "FadeEffect", sendo agora possível fazer acesso ao "FaderLoad()".

```
public class FadeEffect : MonoBehaviour
{
    public static FadeEffect fadeEffect;
    [SerializeField] private CanvasGroup canvasGroup;
    [SerializeField] private float changeValue;
    [SerializeField] private float waitingTime;
    [SerializeField] private bool fadeStarted;

    private void Awake()
    {
        if (fadeEffect == null)
        {
            fadeEffect = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }

    void Start()
    {
        StartCoroutine(FadeIn());
    }

    public void FaderLoad(int level)
    {
        StartCoroutine(FadeOut(level));
    }

    IEnumerator FadeIn()
    {

        fadeStarted = false;
        while(canvasGroup.alpha > 0)
        {
            canvasGroup.alpha -= changeValue;
            yield return new WaitForSeconds(waitingTime);
        }
    }

    IEnumerator FadeOut(int level)
    {
        if (fadeStarted)
        {
            yield break;
        }
        fadeStarted = true;
        while(canvasGroup.alpha < 1)
        {
            canvasGroup.alpha += changeValue;
            yield return new WaitForSeconds(waitingTime);
        }

        SceneManager.LoadScene(level);
        yield return new WaitForSeconds(0.2f);
        StartCoroutine(FadeIn());
    }
}
```

Figura 110 - Classe "FadeEffect"

A **Figura 111**, mostra a implementação do ecrã de "Loading" no jogo.

Foram feitas algumas alterações na classe "FadeEffect", na qual se colocaram dois novos atributos denominados de "loadingScreen" e "loadingBar". Será através destes que se fará a referência ao ecrã de "Loading" e a barra de progresso respetivamente.

O método que sofreu mais atualizações foi o “FadeOut()”, logo após ser feita a transição é criada uma variável do tipo "*AsyncOperation*", é através desta que se inicia o carregamento da nova cena de forma assíncrona com recurso ao método "*LoadSceneAsync*()". Este tipo de carregamento permite que o jogo continue a funcionar normalmente sem interrupções durante o processo de "*loading*".

De modo a evitar que a nova cena seja ativada automaticamente ao atingir os 90%, a propriedade “*allowSceneActivation*” é definida como “*false*”, é sim ativado o ecrã de "Loading" através do “*setActive(true)*”.

Durante o carregamento é executado um ciclo “*while*”, que se prolonga até que a operação esteja concluída. Dentro desse ciclo é feita a atualização da variável “loadingBar” através do valor de "*progress*", este pode variar entre 0 e 0.9. Para que este valor, possa representar um valor de percentagem real na barra, é dividido por 0.9, por exemplo, quando o "*progress*" atinge 0.45 a barra irá apresentar o valor de 50% de preenchimento.

Quando o valor de "*progress*" for igual a 0.9 é chamada o método "*allowSceneActivation*", ao qual se atribui o valor de "*true*", permitindo agora sim a transição para a nova cena.

A instrução "*yield*" a retornar "*null*" é feita para garantir que a atualização da barra de progresso seja feita “*frame*” a “*frame*”, criando assim uma animação fluida para o jogador.

Por fim, é importante garantir que o ecrã de "Loading" seja desativado no método de "FadeIn()", uma vez que, este elemento é exclusivamente necessário durante a execução do “FadeOut()”. Com esta medida, assegura-se que o ecrã de “Loading” não permanece visível após a nova cena ser carregada.

```
2 references
public static FadeEffect fadeEffect;
4 references
[SerializeField] private CanvasGroup canvasGroup;
2 references
[SerializeField] private GameObject loadingScreen;
2 references
[SerializeField] private Image loadingBar;
2 references
[SerializeField] private float changeValue;
2 references
[SerializeField] private float waitingTime;
4 references
[SerializeField] private bool fadeStarted;

0 references
private void Awake()

0 references
void Start()

0 references
public void FaderLoad(int level)

2 references
IEnumerator FadeIn()
{
    // with the load screen here I've to desactivate the load screen
    loadingScreen.SetActive(false);

    fadeStarted = false;
    while(canvasGroup.alpha > 0)
    {
        if (fadeStarted)
        {
            yield break;
        }
        canvasGroup.alpha -= changeValue;
        yield return new WaitForSeconds(waitingTime);
    }
}

1 reference
IEnumerator FadeOut(int level)
{
    if (fadeStarted)

    fadeStarted = true;

    while(canvasGroup.alpha < 1)

    AsyncOperation asyncOperation = SceneManager.LoadSceneAsync(level);
    asyncOperation.allowSceneActivation = false;
    loadingScreen.SetActive(true);
    loadingBar.fillAmount = 0;

    while (asyncOperation.isDone == false)
    {
        loadingBar.fillAmount = asyncOperation.progress / 0.9f;

        if (asyncOperation.progress == 0.9f)
        {
            asyncOperation.allowSceneActivation = true;
        }
        yield return null;
    }

    StartCoroutine(FadeIn());
}
```

Figura 111 - Atualização da classe "FadeEffect"

A implementação apresentada na **Figura 112**, refere-se ao script “ButtonIcons” associado ao objeto “LevelContainer” que integra o ecrã de seleção de capítulos.

Este script dispõe de três atributos o "levelButtons", que contém as referências para todos os botões que representam os capítulos disponíveis no jogo o "lockedIcon", corresponde ao elemento gráfico que mostra um planeta com tonalidade negra e alguma transparência e o "firstLevelIndex" que indica o índice do primeiro capítulo.

Todo o comportamento desta classe, é definido no método “*Awake*()” onde é feito o "*GetInt*()" do índice do último nível desbloqueado pelo jogador através da leitura da chave armazenada no "*PlayerPrefs*", colocando este valor na variável "unlockedLevel".

Posteriormente, é executado um ciclo “*for*” que percorre todos os elementos do “array” “levelButtons, avaliando para cada um deles se o seu índice somado ao "firstLevelIndex" é inferior ou igual ao índice do nível desbloqueado de forma a colocar esse botão como "interactable" ou não, neste segundo caso é atribuída a imagem de "lockedIcon" ao botão.

```
public class ButtonIcons : MonoBehaviour
{
    4 references
    [SerializeField] private Button[] levelButtons;
    1 reference
    [SerializeField] private Sprite lockedIcon;
    2 references
    [SerializeField] private int firstLevelIndex;

    0 references
    private void Awake()
    {
        int unlockedLevel = PlayerPrefs.GetInt(EndGameManager.endGameManager.levelUnlock, firstLevelIndex);

        for (int i = 0; i < levelButtons.Length; i++)
        {
            if (i + firstLevelIndex <= unlockedLevel)
            {
                levelButtons[i].interactable = true;
            }
            else
            {
                levelButtons[i].interactable = false;
                levelButtons[i].image.sprite = lockedIcon;
            }
        }
    }
}
```

Figura 112 - Classe "ButtonIcons"

A **Figura 113**, apresenta a implementação da classe “SO_Spawner” que serve de base para a criação dos “*ScriptableObjects*”: “PowerUpSpawner” e “CoinsSpawner”.

Esta classe possui dois atributos, o “gameObjects” que é um “*array*” onde serão definidos os objetos a serem instanciados no jogo e o "spawnFrequency", que determina a frequência de criação desses mesmos objetos.

O único método presente nesta classe é o "Spawn()", este recebe como parâmetro uma posição que irá definir a posição de onde o objeto vai ser instanciado.

Esta geração de objetos, é feita de forma randómica numa escala de 0 a 100, para que seja possível definir uma frequência de criação destes objetos.

Outro aspeto aleatório dentro do método, é a escolha de qual objeto a ser criado. Por exemplo, no caso dos “power-ups” existe a possibilidade de gerar três objetos diferentes, atribuindo assim alguma dinâmica a este comportamento.

```
[CreateAssetMenu(fileName ="Spawner", menuName = "ScriptableObject/Spawner")]

2 references
public class SO_Spawner : ScriptableObject
{
    2 references
    public GameObject[] gameObjects;
    1 reference
    public int spawnerFrequency;

    2 references
    public void Spawn ( Vector3 spawnPosition)
    {
        int randomChance = Random.Range(0, 100);

        if(randomChance > spawnerFrequency)
        {
            int randomObject = Random.Range(0, gameObjects.Length);
            Instantiate(gameObjects[randomObject], spawnPosition, Quaternion.identity);

        }

    }

}
```

Figura 113 - Classe "SO_Spawner"

A **Figura 114**, mostra a implementação da classe “PowerPlus”.

Nesta classe existem apenas dois atributos: o "powerAmount", referente ao nível de "*power*" a ser restabelecido pela nave e o “audioClip” responsável por produzir um som quando este objeto é recolhido.

No método “OnTriggerEnter2D” verifica-se se este objeto colidiu com o "Player", caso seja verdade, é feita a adição do valor de "powerAmount" ao jogador através do método “AddPower()” proveniente da classe "PlayerStats", de seguida, é tocado um som que identifica este "power-up" e por último o objeto é destruído.

```
public class PowerPlus : MonoBehaviour
{
    1 reference
    [SerializeField] private int powerAmount;
    1 reference
    [SerializeField] private AudioClip audioClip;

    0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            PlayerStats playerStats = collision.GetComponent<PlayerStats>();
            playerStats.AddPower(powerAmount);
            AudioSource.PlayClipAtPoint(audioClip, transform.position, 1f);
            Destroy(gameObject);
        }
    }

    0 references
    private void OnBecameInvisible()
    {
        Destroy(gameObject);
    }
}
```

Figura 114 - Classe "PowerPlus"

A implementação da classe “Shield”, encontra-se representada na **Figura 115**.

É através desta, que se faz a gestão da do objeto "Shield" e onde se coloca a barra de duração desta proteção ativa no ecrã de jogo.

A classe "Shield", é composta por três métodos principais: o "Damage()" responsável por decrementar o número de impactos de defesa, o "Repair()" utilizado para quando a nave volta a recolher o mesmo “*power-up*” e a barra seja restituída e o "OnTriggerEnter2D" que trata de colisões, no caso mais concretamente com colisões entre a mesma e elementos do tipo "Enemy". Foi usado o método "*TryGetComponent()*", que tenta obter um componente do tipo "Enemy", o uso do "*out*" é necessário para a utilização deste método, este é mais seguro de usar ao invés do "*GetComponent()*", pois evita exceções.

Seguindo a lógica dentro do método "OnTriggerEnter2D", quando este inimigo embate com a "shield" este é automaticamente destruído e é retirada a "shield" um "hit" do total de três. Existe uma exceção, quando o inimigo é um "Lider" a "shield" é automaticamente destruída.

Outro aspeto a destacar, é a variável booleana "isProtected" utilizada na classe "PlayerStats". Esta variável, serve para indicar se a nave se encontra sob a proteção da "shield", caso isto seja verdade, o método "*takeDamage*()" não deverá ser executado, não permitindo que o jogador perca “power” quando esta protegido pela "shield".

```
public class Shield : MonoBehaviour
{
    3 references
    [SerializeField] private GameObject shieldBar;
    3 references
    [SerializeField] private Image shieldFill;
    6 references
    private int hits;
    3 references
    public bool isProtected = false;
    2 references
    private int shieldMax = 1;
    1 reference
    private float takeAmount = 0.33f;

    0 references
    private void OnEnable()
    {
        hits = 3;
        shieldBar.SetActive(true);
        isProtected = true;
        shieldFill.fillAmount = shieldMax;
    }

    3 references
    private void Damage()
    {
        hits -= 1;
        shieldFill.fillAmount -= takeAmount;

        if (hits <= 0)
        {
            hits = 0;
            isProtected = false;
            shieldBar.SetActive(false);
            gameObject.SetActive(false);
        }
    }

    1 reference
    public void Repair()
    {
        hits = 3;
        shieldBar.SetActive(true);
        shieldFill.fillAmount = shieldMax;
    }

    0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if(collision.TryGetComponent(out Enemy enemy))
        {
            // Verify if the shield hit a Lider
            if (collision.CompareTag("Lider"))
            {
                hits = 0;
                Damage();
                return;
            }
            // Destroy the enemy - 100 is to make sure that the enemy is destroyed
            enemy.Damaged(100);
            Damage();
        }
        else
        {
            // Destroy the projectile
            Destroy(collision.gameObject);
            Damage();
        }
    }
}
```

Figura 115 - Classe "Shield"

A **Figura 116**, apresenta a implementação do script “ShieldProtection”.

Este script, utiliza apenas o método “OnTriggerEnter2D” de modo a verificar se o objeto de “power-up” colide com o “Player”.

Caso esta condição seja verdadeira, é criada uma variável "playerShieldActivator" que recebe o componente da classe "PlayerShieldActivator" do objeto "Player". É através desta variável, que é possível fazer a chamada ao método "Activate()", que repara ou ativa a "shield" consoante o seu estado atual.

Este script contém ainda, a variável responsável por reproduzir um som sempre que este “power-up” é recolhido pelo jogador.

```
using UnityEngine;

0 references
public class ShieldProtection : MonoBehaviour
{
    1 reference
    [SerializeField] private AudioClip audioClip;

    0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            PlayerShieldActivator playerShieldActivator = collision.GetComponent<PlayerShieldActivator>();
            playerShieldActivator.Activate();
            AudioSource.PlayClipAtPoint(audioClip, transform.position, 1f);
            Destroy(gameObject);
        }
    }

    0 references
    private void OnBecameInvisible()
    {
        Destroy(gameObject);
    }
}
```

Figura 116 - Classe "ShieldProtection"

A atualização feita à classe “PlayerShooting”, encontra-se representada na **Figura 117**.

Passou-se a utilizar um “*array*” denominado de "shootingPoints", de forma a permitir que a nave possa disparar de vários pontos quando recolhe o “*power-up*” "Ultra Shoot".

Foram também criadas variáveis auxiliares, nomeadamente “upgradeShooting”, que funciona como condição no “*switch*” utilizado para controlar quantos pontos de disparo devem ser instanciados ou não. Esta variável tem, por predefinição, o valor zero, correspondente a um único ponto de tiro. De forma a fazer o controlo do valor mínimo e máximo do número de pontos de tiro, foram criadas a variável "upgradeMin" e "upgradeMax".

Relativamente aos métodos criados temos o "IncreaseShooting()", que permite aumentar o número de pontos de disparo ativos na nave, este método é chamado quando o “*power-up*” colide com a nave. O "DecreaseShooting()" faz o processo contrário, é chamado a partir da classe "PlayerStats" no método "takeDamage()", quando a nave sofre dano é reduzido um nível de pontos de tiro a nave.

Por último, o método “Shoot()” foi atualizado para suportar os novos pontos de disparo e gerir dinamicamente quais devem estar ativos. Esse controlo é realizado através de uma estrutura “*switch*”, que avalia o nível de disparo atual e por consequência o número de mísseis que devem ser instanciados.

```
public class PlayerShooting : MonoBehaviour
{
    3 references
    [SerializeField] private GameObject missil;
    4 references
    [SerializeField] private float shootingTimer;
    4 references
    [SerializeField] private Transform[] shootingPoints;
    1 reference
    [SerializeField] private AudioSource audioSource;
    2 references
    private float timer;
    7 references
    private int upgradeShooting = 0;
    2 references
    private int upgradeMax = 2;
    2 references
    private int upgradeMin = 0;

    0 references
    void Start()…

    0 references
    void Update()…

    1 reference
    public void IncreaseShooting()
    {
        upgradeShooting ++;

        if (upgradeShooting > upgradeMax)
        {
            upgradeShooting = upgradeMax;
        }
    }

    1 reference
    public void DecreaseShooting()…

    1 reference
    private void Shoot()
    {
        audioSource.Play();
        switch (upgradeShooting)
        {
            case 0 :
                for (int i = 0; i == 0; i++)
                {
                    Instantiate(missil, shootingPoints[i].position, Quaternion.identity);
                }
                break;

            case 1 :
                for (int i = 0; i <= 2; i++)
                {
                    Instantiate(missil, shootingPoints[i].position, Quaternion.identity);
                }
                break;

            case 2 :
                for (int i = 0; i < shootingPoints.Length; i++)
                {
                    Instantiate(missil, shootingPoints[i].position, Quaternion.identity);
                }
                break;

            default:
                break;
        }
```

Figura 117 - Atualização da classe "PlayerShooting"

A **Figura 118**, mostra a implementação da superclasse "LiderBaseState", que serve de base para a criação das subclasses que representam os quatro estados em que o "Lider" se pode encontrar.

A classe, inclui quatro variáveis de controlo da movimentação do líder e uma referência à classe "LiderController", de forma a permitir a invocação do método "ChangeState()".

Dentro do método "*Awake*()" é feita a atribuição as variáveis "liderController" e "mainCamera", já no método "*Start*()" são definidas as limitações de movimentação do líder.

Os métodos principais desta classe são o "RunState()" e o "StopState()", ambos virtuais. Para o primeiro, foi apenas criada a sua assinatura, no caso do segundo foi chamado o método "*StopAllCoroutines*()", que tal como o nome indica termina todas as "*coroutines*" associadas ao estado.

```
public class LiderBaseState : MonoBehaviour
{
    7 references
    protected Camera mainCamera;
    3 references | 3 references | 3 references | 3 references
    protected float maxLeft, maxRight, maxUp, maxDown;
    5 references
    protected LiderController liderController;

    0 references
    void Awake()
    {
        liderController = GetComponent<LiderController>();
        mainCamera = Camera.main;
    }

    3 references
    protected virtual void Start()
    {
        maxLeft = mainCamera.ViewportToWorldPoint(new Vector3(0.3f, 0, 0)).x;
        maxRight = mainCamera.ViewportToWorldPoint(new Vector3(0.7f, 0, 0)).x;
        maxUp = mainCamera.ViewportToWorldPoint(new Vector3(0, 0.6f, 0)).y;
        maxDown = mainCamera.ViewportToWorldPoint(new Vector3(0, 0.9f, 0)).y;
    }

    8 references
    public virtual void RunState()…

    9 references
    public virtual void StopState()
    {
        StopAllCoroutines();
    }
}
```

Figura 118 - Classe "LiderBaseState"

A **Figura 119**, apresenta a implementação da classe que controla as mudanças de estado do líder a qual se chamou de "LiderController".

As quatro variáveis presentes nesta classe, dizem respeito às quatro subclasses correspondentes aos estados em que o líder se poder encontrar: “liderEnter”, “liderFire”, “liderSpecial” e “liderDeath”.

Estas são criadas de forma a ser possível se usar o método "RunState()", este contém o comportamento específico para cada um dos estados.

Foram incluídas variáveis auxiliares de teste, com o objetivo de facilitar a verificação e construção dos comportamentos de todos os estados durante a fase de desenvolvimento dos mesmos.

Quanto a métodos é no "Start()" que é feita a chamada ao método "ChangeState()”. Neste é definido o estado inicial do líder, passando como argumento o valor "LiderState.Enter".

Contudo, o método principal desta classe é o "ChangeState()". Este, faz o controlo do estado em que o líder se encontra e encaminha para o estado que deve ser executado fazendo a execução do método RunState() correspondente.

```
public class LiderController : MonoBehaviour
    void Start()
    {
        ChangeState(LiderState.Enter);

        if (test)…

    }

    7 references
    public void ChangeState(LiderState liderState)
    {
        switch (liderState)
        {
            case LiderState.Enter:
                liderEnter.RunState();
                break;
            case LiderState.Fire:
                liderFire.RunState();
                break;
            case LiderState.Special:
                liderSpecial.RunState();
                break;
            case LiderState.Death:
                liderEnter.StopState();
                liderFire.StopState();
                liderSpecial.StopState();
                liderDeath.RunState();
                break;

            default:
                break;
        }
```

Figura 119 - Classe "LiderController"

A **Figura 120**, apresenta a implementação da classe “LiderEnter” que herda de “LiderBaseState”.

A classe contém apenas dois atributos o "speed", que determina a velocidade a que o líder se desloca e o "enterPoint", que representa a posição em que o líder irá interromper o seu movimento.

No método "*Start*()" é invocado o "base.Start()", de forma a se dar início à sequência de estados do líder e de seguida é passado o valor do ponto em que o líder vai interromper a sua deslocação.

A lógica principal deste estado de entrada, encontra-se dentro da implementação da "*coroutine*" "RunEnterState()", nesta é executado um ciclo "*while*" de forma a mover o líder através do "MoveTowards()".

Quando o ponto é atingido pelo líder faz-se a mudança do estado de entrada para o estado de disparo, através do método "ChangeState()".

A execução desta "*coroutine*" é feita dentro do método "RunState()", que é chamado sempre que este estado é ativo, enquanto o método “StopState()” recorre apenas à implementação base feita na superclasse.

```
public class LiderEnter : LiderBaseState
{
    1 reference
    [SerializeField] private float speed;
    3 references
    private Vector2 enterPoint;

    2 references
    protected override void Start()
    {
        base.Start();
        enterPoint = mainCamera.ViewportToWorldPoint(new Vector2(0.5f, 0.7f));
    }

    2 references
    public override void RunState()
    {
        StartCoroutine(RunEnterState());
    }

    5 references
    public override void StopState()
    {
        base.StopState();
    }

    1 reference
    IEnumerator RunEnterState()
    {
        while (Vector2.Distance(transform.position, enterPoint) > 0.01f)
        {
            transform.position = Vector2.MoveTowards(transform.position, enterPoint, speed * Time.deltaTime);
            yield return new WaitForEndOfFrame();
        }

        liderController.ChangeState(LiderState.Fire);
    }

}
```

Figura 120 - Classe "LiderEnter"

A implementação da classe “LiderFire”, está detalhada na **Figura 121**. Esta classe herda de “LiderBaseState” respeitando desta forma o padrão de desenvolvimento “*State Machine*”.

Entre as variáveis principais encontram-se, o “missil” referencia ao “prefab” do míssil, a "shootingRate" que controla o intervalo de tempo entre os disparos, a "speed" que define a velocidade de deslocamento do líder e o "shootingPoints" um “*array*” onde se guardam os pontos de disparo do líder.

O principal método desta classe é o “*RunFireState*()”, que se divide em quatro partes na sua implementação.

A primeira parte, consiste na verificação do tempo de permanência no estado de disparo. Esta, é feita através das variáveis "fireStateTimer" e "fireStateExitTime" enquanto a primeira é menor ou igual a segunda, o líder mantém-se no estado de tiro.

A segunda parte, corresponde à movimentação dada ao líder este move-se continuamente em direção a uma posição aleatória, armazenada em "targetPosition". Assim que o líder atinge essa posição, move-se para outra posição aleatória criando assim uma movimentação imprevisível.

A terceira parte, diz respeito ao disparo efetuado pelo líder. Quando o “shootTimer” ultrapassa o valor de “shootRate”, o líder dispara os misseis a partir do “*array*” de "shootingPoints".

A quarta fase trata da transição de estado, quando o tempo definido por “fireStateExitTime” é atingido, é gerado aleatoriamente um valor entre zero e um. Se o resultado for zero, o “Lider” repete o estado de disparo, caso contrário, transita para o estado seguinte, correspondente ao ataque especial.

```
public class LiderFire : LiderBaseState
{
    1 reference
    [SerializeField] private GameObject missil;
    1 reference | 1 reference
    [SerializeField] private float shootingRate, speed;
    2 references
    [SerializeField] private Transform[] shootingPoints;

    2 references
    public override void RunState() …

    5 references
    public override void StopState() …

    1 reference
    IEnumerator RunFireState()
    {
        float shootTimer = 0f;
        float fireStateTimer = 0f;
        // Fire state exit time - random value between 5 to 10 seconds
        float fireStateExitTime = Random.Range(5f, 10f);

        Vector2 targetPosition = new Vector2(Random.Range(maxLeft, maxRight), Random.Range(maxDown, maxUp));

        while (fireStateTimer <= fireStateExitTime)
        {
            if (Vector2.Distance(transform.position, targetPosition) > 0.01f)
            {
                transform.position = Vector2.MoveTowards(transform.position, targetPosition, speed * Time.deltaTime);
            }
            else
            {
                targetPosition = new Vector2(Random.Range(maxLeft, maxRight), Random.Range(maxDown, maxUp));
            }

            shootTimer += Time.deltaTime;

            if (shootTimer >= shootingRate)
            {
                for (int i = 0; i < shootingPoints.Length; i++)
                {
                    Instantiate(missil, shootingPoints[i].position, Quaternion.identity);
                }
                // reset shoot timer
                shootTimer = 0;
            }
            // wait for the end of the frame
            yield return new WaitForEndOfFrame();

            // increase the fire state timer
            fireStateTimer += Time.deltaTime;

        }

        // when fire state timer end, should change of state to special or continue fire
        int randomState = Random.Range(0, 2);

        if (randomState == 0)
        {
            liderController.ChangeState(LiderState.Fire);
        }
        else
        {
            liderController.ChangeState(LiderState.Special);
        }
    }
}
```

Figura 121 - Classe "LiderFire"

A **Figura 122**, mostra a implementação da classe “LiderSpecial” que representa o estado de ataque especial do “Lider”.

Quanto às variáveis desta classe destacam-se as seguintes: a "waitime", que define o tempo de espera após a execução do ataque especial antes de transitar para o estado de disparo normal; o "specialShooting", que recebe o "*prefab*" correspondente ao ataque especial definido para o respetivo "Lider"; o "liderPosition", que determina o ponto do ecrã onde o líder se deverá posicionar para realizar o ataque.

Relativamente a métodos destaque-se o “RunSpecialState()” responsável por mover o “Lider” até à posição previamente definida no método de “Start()”, após alcançar essa posição o ataque especial é instanciado no ponto de disparo indicado em “shootingPoint”. Por último, é respeitado o tempo de espera definido no "waitTime" e é feita a transição para o estado de disparo normal novamente.

```
public class LiderSpecial : LiderBaseState
{
    1 reference | 1 reference
    [SerializeField] private float speed, waitTime;
    1 reference
    [SerializeField] private GameObject specialShooting;
    1 reference
    [SerializeField] private Transform shootingPoint;
    3 references
    private Vector2 liderPosition;

    2 references
    protected override void Start()
    {
        liderPosition = mainCamera.ViewportToWorldPoint(new Vector3(0.5f, 0.9f));
    }

    2 references
    public override void RunState()
    {
        StartCoroutine(RunSpecialState());
    }

    5 references
    public override void StopState()
    {
        base.StopState();
    }

    1 reference
    IEnumerator RunSpecialState()
    {
        while (Vector2.Distance(transform.position, liderPosition) > 0.01f)
        {
            transform.position = Vector2.MoveTowards(transform.position, liderPosition, speed * Time.deltaTime);
            // need to wait for the end of the frame
            yield return new WaitForEndOfFrame();
        }
        // generate the special attack
        Instantiate(specialShooting, shootingPoint.position, Quaternion.identity);
        // wait time to change to the fire state again
        yield return new WaitForSeconds(waitTime);
        liderController.ChangeState(LiderState.Fire);
    }

}
```

Figura 122 - Classe "LiderSpecial"

A **Figura 123**, mostra a implementação da classe "SpecialAttack", que determina o ataque especial que será conduzido pelo líder.

A destacar a nível dos atributos, temos o “*array*” "shootingPoints". Neste, é determinado de onde os vários disparos secundários deste ataque são instanciados.

No método de "Start", é definida uma velocidade e direção inicial do disparo principal de uma bola, ainda neste método é feita a chamada a "*coroutine*" "Explode". Esta "*coroutine*" aguarda um tempo aleatório e de seguida é feita a criação dos disparos secundários, em cada um dos pontos definidos no “*array*” associados a bola de tiro inicialmente criada. No exato momento em que estes disparos secundários são criados, a bola é então destruída o que produz um efeito de explosão.

Por fim, a classe verifica se o ataque colidiu com o jogador, caso isto seja verdade, é aplicado o dano através do método "takeDamage" e o objeto é destruído.

```
public class SpecialAttack : MonoBehaviour
    0 references
    void Start()
    {
        rigidBody2D.velocity = Vector2.down * speed;
        StartCoroutine(Explode());
    }

    1 reference
    IEnumerator Explode()
    {
        float randomTime = Random.Range(0.5f, 1.5f);
        yield return new WaitForSeconds(randomTime);

        for (int i = 0; i < shootingPoints.Length; i++)
        {
            Instantiate(shoot, shootingPoints[i].position, shootingPoints[i].rotation);
        }

        Destroy(gameObject);
    }

    0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            collision.GetComponent<PlayerStats>().takeDamage(damage);
            Destroy(gameObject);
        }
    }

    0 references
    private void OnBecameInvisible()
    {
        Destroy(gameObject);
    }
```

Figura 123 - Classe "SpecialAttack"

A classe "LiderStats" presente na **Figura 124**, é usada para gerir os "*stats*" por parte do líder. Esta classe herda de "Enemy", o que permite dar "*override*" aos métodos "Hitted" e "Destroyed".

A nível de atributos, temos o "liderController" que é uma referência ao "LiderController" e o "animator" de forma a utilizar a animação de “Damage”.

Quanto a métodos, temos a implementação do “Hitted”, onde é feito o "*trigger*" a animação de “Damage”, já no “Destroyed”, é feito o uso do “liderController”, de forma a fazer a mudança para o estado de “Death”. Por último, o "OnTriggerEnter2D", que trata das colisões entre o líder e a nave do jogador, caso esta colisão aconteça é aplicado dano a nave do jogador através da chamada ao componente “PlayerStats”.

```
public class LiderStats : Enemy
{
    1 reference
    [SerializeField] private LiderController liderController;
    1 reference
    [SerializeField] private Animator animator;

    2 references
    public override void Hitted()
    {
        //Animation
        animator.SetTrigger("Damage");
    }

    2 references
    public override void Destroyed()
    {
        liderController.ChangeState(LiderState.Death);
        Instantiate(explosionPrefab, transform.position, transform.rotation);
    }


    0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            collision.GetComponent<PlayerStats>().takeDamage(damage);
        }
    }
}
```

Figura 124 - Classe "LiderStats"