Master D

Programação de Videojogos com Unity 3D

Relatório Terceiro Projeto

Docente Pedro Silva

Trabalho realizado por:

- Fábio Dias

Índice

Índice de Figuras

Objectivos

Neste terceiro projeto, foi-nos indicado a realização do jogo *Space Invaders* em 2D.

O conceito do jogo é simples: o jogador possui uma nave que pode movimentar na horizontal, esta encontra-se na parte inferior do ecrã. Inimigos vão aparecendo na parte superior e o objectivo do jogador é eliminá-los antes que eles ou o eliminem ou cheguem ao seu lado do ecrã, caso contrário, o jogo acaba.

De forma a tornar o jogo mais dinâmico, alterei algumas partes do conceito, criando uma adaptação deste jogo.

Assim, o jogo também possui a componente 3D, permitindo ao jogador movimentar-se na vertical também, encontrando diversos tipos de inimigos, incluindo batalhas individuais contra uma nave mais poderosa que as outras, a chamada *Boss Battle*.

Criando objetivos para o jogador, o jogador vai ganhando moedas enquanto joga e pode gastá-las num mercado onde pode comprar corpos, propulsores e armas, melhorando a nave, tornando-a mais resistente, mais rápida e a infligir mais dano.

O jogo também guarda o progresso do jogador, permitindo assim retomar o seu jogo e não perder o conteúdo das horas investidas no jogo.

As mecânicas deste projecto: um jogador; o jogador controlará a nave com a tecla *A* para se movimentar para a esquerda e a tecla *D* para a direita. O jogador também usará a tecla *W* para subir e a tecla *S* para descer a nave durante a *Boss Battle*; o fim do jogo ocorre quando o jogador perde demasiados pontos de vida ou quando as naves inimigas chegam ao seu lado do ecrã; a velocidade das naves inimigas será aumentada conforme o tempo, dificultando o jogo.

Resolução

O jogo começa com o menu principal. Neste existem quatro botões no centro do ecrã e um icon no lado direito superior. De forma a ter noção do progresso do jogador, no lado esquerdo superior encontra-se uma aba com informações pertinentes ao progresso: jogos efectuados; pontuação máxima; moedas disponíveis; tiros disparados; inimigos mortos; *bosses* mortos.

Figura 1 - *Menu Principal*

O botão *Select Ship* leva o jogador a um novo menu onde podemos comprar partes da nave usando as moedas que fomos adquirindo e, também, equipá-las.

Caso ainda não tenhamos comprado a peça em específico, o seu valor aparecerá, juntamente com um botão *Buy*. Se o pressionarmos e possuirmos as moedas necessárias, esta é comprada e o botão desaparece assim como o seu valor. São substituídos por um texto onde se lê *Owned* e podemos agora usar essa peça.

Quando terminada a visita ao menu, o jogador pode pressionar o botão *Back* para voltar ao Menu Principal.

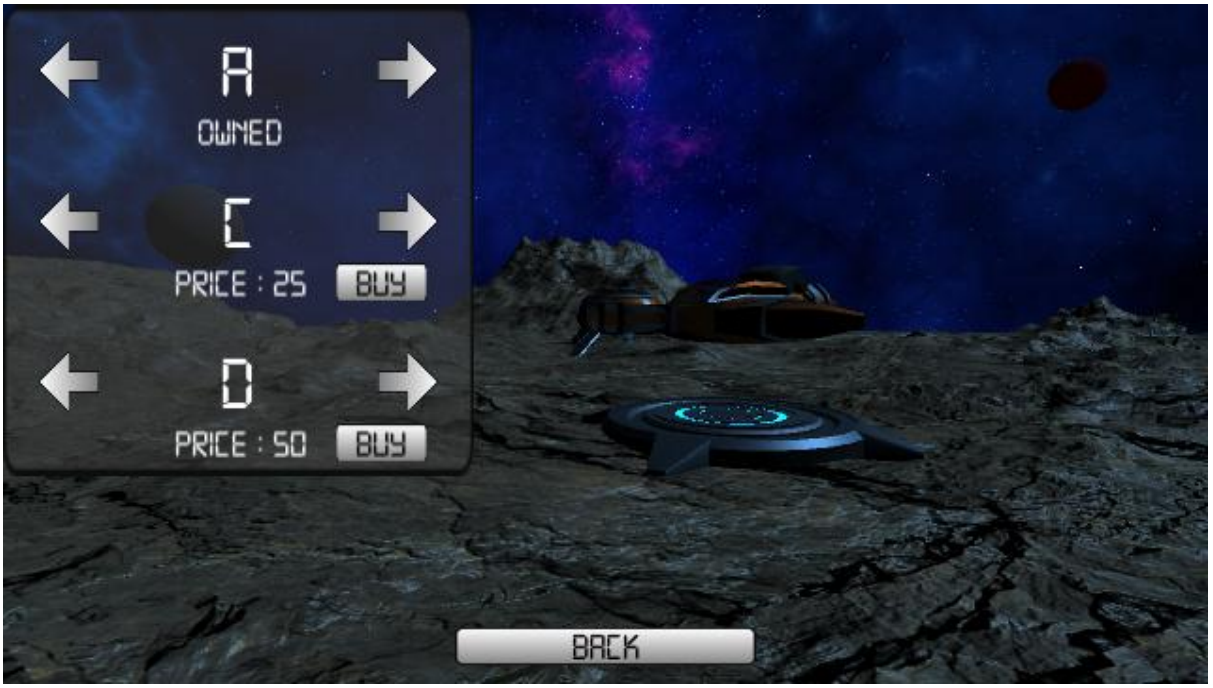

Figura 2 - *Menu Select Ship*

Ao pressionarmos o ícone no lado direito superior, acedemos ao menu de definições onde podemos mudar o volume da música e o volume dos efeitos sonoros.

Tem ainda um botão *Credits* que nos leva aos créditos do jogo.

Figura 3 - Menu Opções

Figura 4 - Menu Créditos

Carregando no botão *Play*, o jogo começará. Os menus desaparecem, a nave roda na direção do horizonte e parte para a sua aventura. O ecrã ficará escuro e um nível será carregado.

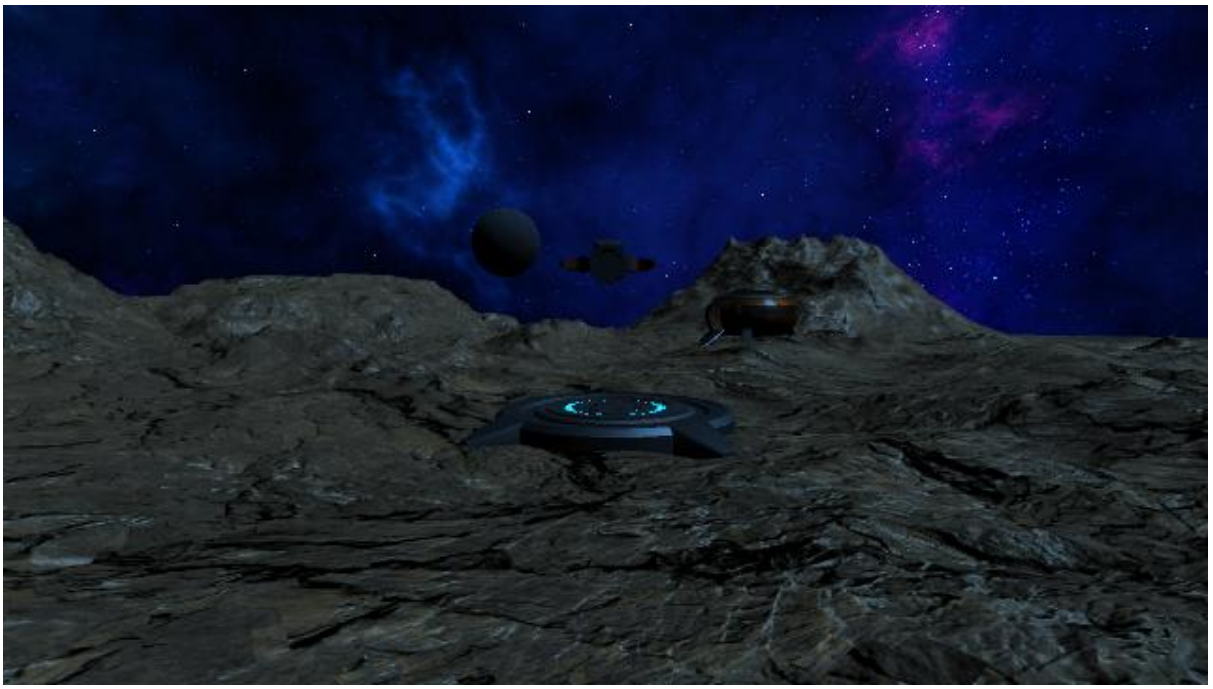

Figura 5 - Animação para Começo do Jogo

Ao começar o jogo, o ecrã ficará mais claro e começa uma pequena animação da nave a entrar na visão do jogador, vinda da parte inferior do ecrã até ao centro, e volta para a parte inferior do ecrã.

Após esta animação, uma animação surge a informar em que ronda o jogador se encontra.

Figura 6 - Início do Jogo

A partir daqui os inimigos começam a surgir na parte superior do ecrã e movem-se na direcção do jogador, ou seja, movem-se de cima para baixo.

Figura 7 - Aparecimento dos Inimigos

De cinco em cinco rondas aparecerá o *Boss*. Esta unidade é uma nave mais poderosa do que todas as outras.

Com o seu aparecimento, uma animação começará com um *Warning*. A cor do fundo fica vermelha e a visão muda para trás da nave, chegando assim à componente 3D.

Figura 8 - Ronda 5

Figura 9 - *Warning*

Destruindo o *Boss*, uma nova animação começa indicando *Threat Clear*, enquanto a camera volta para a visão superior, passando novamente para a componente 2D.

Figura 10 - Threat Clear

Durante o jogo, podemos carregar na tecla *Escape* e acedemos ao menu de pausa. Neste encontram-se quatro botões: *Resume Game* que retoma o jogo; *Restart Game* que anula todo o progresso da sessão de jogo actual e recomeça o jogo; *Options* que abre um novo menu onde encontramos as opções para mudar o volume da música e efeitos sonoros; *Quit* que sai desta sessão de jogo e não grava qualquer progresso.

Neste menu de pausa também podemos verificar as informações atuais como a ronda atual, o número de inimigos que esta ronda possui, os pontos feitos até agora e as moedas ganhas.

Figura 11 - Menu Pausa

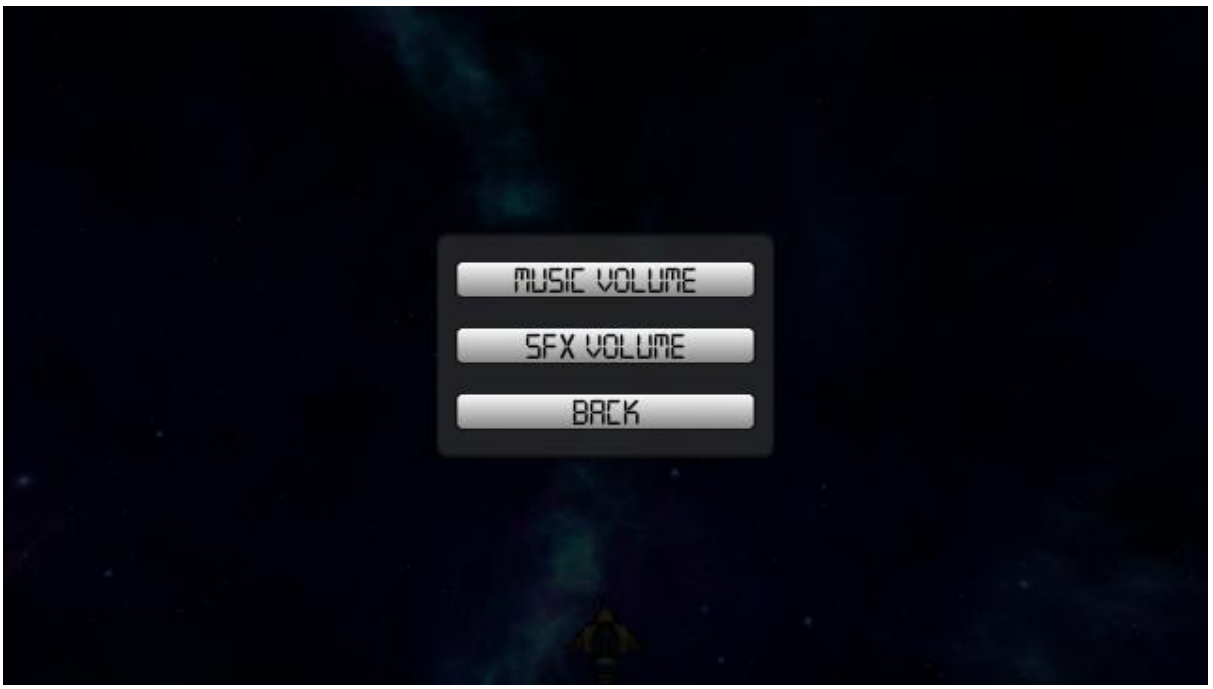

Figura 12 - Menu Opções

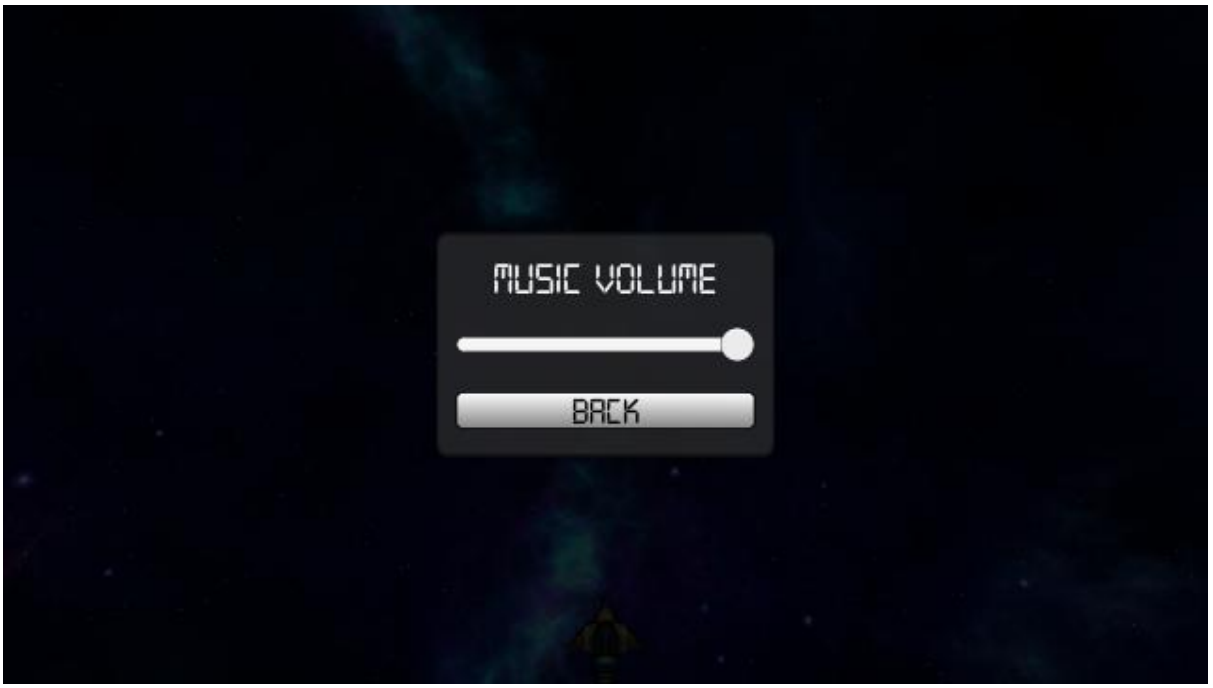

Figura 13 - Opções de Volume

Quando o jogo acabar, aparecerá um menu que resume a sessão de jogo, apresentando informações como a que ronda chegou, o número de pontos, quantas moedas ganhou, quantos mísseis disparou, quantos inimigos matou e quantos *Bosses* matou.

Pressionando o botão *Quit,* o jogador volta ao menu principal, e o seu progresso será atualizado.

Figura 14 - Menu Final

Figura 15 - Progresso Actualizado

Finalmente o botão *History* guarda as informações das últimas cinco sessões de jogo. Apresentadas numa lista vertical, que, ao pressionar, os detalhes aparecerão.

Figura 16 - Histórico sem Jogos

Figura 17 - Histórico com Um Jogo

Figura 18 - Histórico com Um Jogo e Detalhes

Por fim, o botão *Quit* do menu principal faz com que o jogo seja fechado.

Implementações

Neste projecto implementei algumas mecânicas que me desafiaram e que considero importante que sejam destacadas.

Object Pooling

De forma a não prejudicar o fluxo do jogo, **Object Pooling** é uma óptima forma de minimizar as quebras de *frames per second* e manter o desempenho.

Para isto, criei cinco *Queue's*, uma para cada tipo de inimigo.

De seguida, inicializei-o e preenchi-os com *for's*.

Para utilizar um destes objectos, criei uma função *Get* com o tipo de inimigo a ser chamado. Para ser devolvido, criei uma função *Return* com o tipo de inimigo a ser devolvido.

Repeti estas funções para todos os tipos de inimigos.

Resumidamente, a função *Get* verifica se existe esse tipo de inimigo na *Queue*, se existir, é removido da *Queue* e é devolvido; caso não exista, é instanciado e devolvido.

A função *Return* insere o inimigo de volta na *Queue* e desactiva-o, ficando pronto para ser chamado novamente.

```
//Enemies Array Pooling
private Queue<GameObject> probes;
private Queue<GameObject> sentries;
private Queue<GameObject> tanks;
private Queue<GameObject> observers;
private Queue<GameObject> disruptors;

//Maximum Stack Size
private int maxPoolSize;
```

Figura 19 - Queue's e Número Máximo de Entidades

```
//Initialize Queue's
probes = new Queue<GameObject>();
sentries = new Queue<GameObject>();
tanks = new Queue<GameObject>();
observers = new Queue<GameObject>();
disruptors = new Queue<GameObject>();

//Set Maximum Stack Size
maxPoolSize = 40;
```

Figura 20 - Inicialização das Queue's

```
//Enqueue Inactive Probes Objects
for (int i = 0; i < maxPoolSize; i++)
{
    GameObject probe = Instantiate(probePrefab, enemiesParent);
    probes.Enqueue(probe);
    probe.SetActive(false);
}

//Enqueue Inactive Sentries Objects
for (int i = 0; i < maxPoolSize; i++)
{
    GameObject sentry = Instantiate(sentryPrefab, enemiesParent);
    sentries.Enqueue(sentry);
    sentry.SetActive(false);
}

//Enqueue Inactive Tanks Objects
for (int i = 0; i < maxPoolSize; i++)
{
    GameObject tank = Instantiate(tankPrefab, enemiesParent);
    tanks.Enqueue(tank);
    tank.SetActive(false);
}

//Enqueue Inactive Observers Objects
for (int i = 0; i < maxPoolSize; i++)
{
    GameObject observer = Instantiate(observerPrefab, enemiesParent);
    observers.Enqueue(observer);
    observer.SetActive(false);
}
```

Figura 21 - Preenchimento das Queue's com For's

```
public GameObject GetProbe()
{
    if(probes.Count > 0)
    {
        GameObject probe = probes.Dequeue();
        probe.SetActive(true);
        return probe;
    }
    else
    {
        GameObject probe = Instantiate(probePrefab);
        return probe;
    }
}

public void ReturnProbe(GameObject probe)
{
    probes.Enqueue(probe);
    probe.SetActive(false);
}
```

Figura 22 - GetProbe e ReturnProbe

Save System

Para o jogador não sentir que perdeu horas a jogar este jogo, implementei um sistema que grava o progresso do jogador.

Assim que o jogo é aberto, este é o primeiro *script* pessoal a ser corrido. Verifica se existe o ficheiro onde os dados são gravados. Se sim, este carrega os dados do ficheiro para as variáveis do *script*; caso este não exista, o *script* cria um ficheiro novo.

```
private void LoadGameDataFile()
{
    if (!System.IO.File.Exists(Application.persistentDataPath + "/GameData.xml"))
    {
        CreateGameDataFile();
    }

    //Load Game Data Document
    repository = System.IO.File.ReadAllText(Application.persistentDataPath + "/GameData.xml");

    //Open Game Data XML Document
    xmlData = XDocument.Parse(repository);
}
```

Figura 23 - LoadGameDataFile

```
private void CreateGameDataFile()
{
    #region Document Creation
    //Create Game Data XML Document
    XDocument xmlDocument = new XDocument(new XDeclaration("1.0", "utf-8", "yes"));
    #endregion

    Root Element

    Progress Element

    Spaceship Element

    Game History Element

    Settings Element

    //Save XML Document
    xmlDocument.Save(Application.persistentDataPath + "/GameData.xml");
}
```

Figura 24 - CreateGameDataFile

```
public void LoadGameStats()
{
    //Progress Data
    XElement elementProgress = xmlData.Root.Element("PROGRESS");

    //Set Properties
    gameID = int.Parse(elementProgress.Element("GAME_ID").Value);
    highscore = int.Parse(elementProgress.Element("HIGHSCORE").Value);
    coins = int.Parse(elementProgress.Element("COINS").Value);
    enemies = int.Parse(elementProgress.Element("ENEMIES").Value);
    missiles = int.Parse(elementProgress.Element("MISSILES").Value);

    //Spaceship Data
    XElement elementSpaceship = xmlData.Root.Element("SPACESHIP");

    //Spaceship Selected Data
    XElement elementActiveSpaceship = elementSpaceship.Element("ACTIVE_SPACESHIP");

    //Set Spaceship Selected Parts
    bodySelected = elementActiveSpaceship.Element("SELECTED_BODY").Value;
    boostSelected = elementActiveSpaceship.Element("SELECTED_BOOST").Value;
    weaponSelected = elementActiveSpaceship.Element("SELECTED_WEAPON").Value;

    //Spaceship Purchased Data
    XElement elementPurchasedSpaceship = elementSpaceship.Element("PURCHASED_PARTS");

    //Set Spaceship Purchased Body Parts
    foreach (XElement bodyPart in elementPurchasedSpaceship.Element("PURCHASED_BODY").Elements("PURCHASED_BODY_PART")) {

        bodyPurchased.Add(bodyPart.Value);
    }
```

Figura 25 - LoadGameStats

```
private void SaveGameData()
{
    xmlData.Save(Application.persistentDataPath + "/GameData.xml");
}
```

Figura 26 - SaveGameData

Shop System

Para implementar a mecânica de possuir várias partes diferentes da nave controlada pelo jogador, foi preciso criar uma loja onde seria possível comprar essas partes e saber diferenciar quais as que pode comprar e quais as disponíveis para utilizar.

Desta forma, o ficheiro que possui os dados gravados tem de ser actualizado quando existe uma compra, assim como carregados quando o jogo começa.

Dividi as peças em duas secções: *Available*, que são as peças disponíveis para compra e *Purchased*, as peças já compradas.

Quando uma peça é comprada, dependendo da parte da nave mas, para esta demonstração, vamos usar o corpo da nave como referência, duas funções são chamadas: *AddPurchasedBodyPart,* que cria uma nova tag com o identificador desse corpo no ficheiro; e *RemoveAvailableBodyPart*, que remove a tag com o identificador desse corpo no ficheiro, impossibilitando assim uma segunda compra do mesmo corpo.

De forma a actualizar qual a peça escolhida, sempre que existir uma mudança numa peça para uma já comprada, esta é guardada, e, quando o botão *Back* é carregado, as peças serão actualizadas, não permitindo assim que o jogador jogue com peças que ainda não comprou.

```
#region Selected Parts

Selected Body Part

Selected Boost Part

Selected Weapon Part

#endregion

#region Purchased Parts

Purchased Body Parts

Purchased Boost Parts

Purchased Weapon Parts

#endregion

#region Available Parts

Available Body Parts

Available Boost Parts

Available Weapon Parts

#endregion
```

Figura 27 - Regiões de Peças

```
public void AddPurchasedBodyPart(string bodyPart)
{
    bodyPurchased.Add(bodyPart);

    xmlData.Root.Element("SPACESHIP").Element("PURCHASED_PARTS").Element("PURCHASED_BODY").Add(new XElement("PURCHASED_BODY_PART", bodyPart));

    SaveGameData();
}
```

Figura 28 - AddPurchasedBodyPart

```
public void RemoveAvailableBodyPart(string bodyPart)
{
    bodyAvailable.Remove(bodyPart);

    XElement elementBodyParts = xmlData.Root.Element("SPACESHIP").Element("AVAILABLE_PARTS").Element("AVAILABLE_BODY");

    foreach(XElement element in elementBodyParts.Elements("AVAILABLE_BODY_PART"))
    {
        if(element.Value == bodyPart)
        {
            element.Remove();
        }
    }

    SaveGameData();
}
```

Figura 29 - RemoveAvailableBodyPart

```
public string GetSelectedBodyPart()
{
    return bodySelected;
}

public void SetSelectedBodyPart(string body)
{
    bodySelected = body;

    UpdateSelectedBodyPart();
}

private void UpdateSelectedBodyPart()
{
    xmlData.Root.Element("SPACESHIP").Element("ACTIVE_SPACESHIP").Element("SELECTED_BODY").Value = bodySelected;

    SaveGameData();
}
```

Figura 30 - Funções da Região das Peças Seleccionadas

History System

De forma a dar a oportunidade ao jogador de rever o seu desempenho em jogos recentes, implementei um histórico que guarda as informações das últimas cinco sessões de jogo.

Para isto, criei uma função que cria uma sessão de jogo no final de cada jogo.

Para apresentar estes jogos, criei uma função que vai buscar a informação de cada sessão de jogo guardada no ficheiro.

```
public int GetNumberOfGameSessions()
{
    return xmlData.Root.Element("GAME_HISTORY_LIST").Elements("GAME_HISTORY").Count();
}

public List<string> GetGameSession(int gameSession)
{
    List<string> info = new List<string>();
    XElement gameSessionElement = xmlData.Root.Element("GAME_HISTORY_LIST").Elements("GAME_HISTORY").ElementAt(gameSession);

    foreach(XElement element in gameSessionElement.Elements())
    {
        info.Add(element.Value);
    }

    return info;
}
```

Figura 31 - GetNumberOfGameSessions e GetGameSession

```
public void CreateGameSessionHistory(int points, int coins, int missiles, int enemies, int bosses)
{
    //Get GAME_HISTORY_ELEMENT
    XElement gameHistoryList = xmlData.Root.Element("GAME_HISTORY_LIST");

    if (gameHistoryList.Elements("GAME_HISTORY").Count() >= 5)
    {
        //Get Last GAME_HISTORY Element and Delete it
        gameHistoryList.Elements("GAME_HISTORY").Last().Remove();
    }

    //Create Element GAME_HISTORY Standard Organization with GameID Attribute
    XElement gameHistory = new XElement("GAME_HISTORY");

    //Create Game History Stats Elements and Add to GAME_HISTORY Element
    gameHistory.Add(new XElement("GAME_ID", gameID.ToString()),
                    new XElement("POINTS", points.ToString()),
                    new XElement("COINS", coins.ToString()),
                    new XElement("MISSILES", missiles.ToString()),
                    new XElement("ENEMIES", enemies.ToString()),
                    new XElement("BOSSES", bosses.ToString()));

    //Add GAME_HISTORY to GAME_HISTORY_LIST first place
    gameHistoryList.AddFirst(gameHistory);

    UpdateGameStats(points, coins, missiles, enemies, bosses);

    SaveGameData();
}
```

Figura 32 - CreateGameSessionHistory

Dificuldades

Neste projecto senti dificuldade em organizar algumas funcionalidades.

Tentei organizar o código desde início, evitando o *código-esparguete* mas, mais depressa do que esperado, isto acabou por acontecer, o que dificultou a procura e resolução de bugs.

A própria motivação e empenho neste projecto foram desvanecendo com o passar do tempo. Não sei se foi por ser um projecto com várias mecânicas ou se foi por não estar agradado com o resultado que obtive ao longo do tempo.

Melhorias Futuras

Acredito que tenho de melhorar a organização do meu código.

Acredito que deveria melhorar a música feita por mim, investir mais tempo no seu desenvolvimento para obter um melhor resultado.

Sei que existem alguns bugs mas, devido à má-organização do código, não os consegui resolver. Algo que deve ser resolvido.

Acredito que deveria ter adicionado mais uma componente de customização como a cor dos materiais.

Faltaram efeitos especiais, algo que devia ter dedicado algum tempo também.

Conclusão

Embora seja um projecto pequeno, investi muito tempo neste projecto, criar assets, música e *sprites*

Acredito ter melhorado a minha gestão no que toca à distribuição de tempo e recursos em cada componente, sejam estes programação, arte, áudio e testes. Ainda assim, tenho de continuar a melhorar.

Gostei bastante de o desenvolver, por momentos foi frustrante e eventualmente perdi a motivação, mas, em geral, foi uma óptima experiência, com novas mecânicas, o que me fez sair da minha zona de conforto e procurar por mais.