UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Desenvolvimento de Jogos Multi Jogador: Um Caso de Estudo

Nuno Filipe Alexandre Carapito

Relatório de projeto para obtenção do Grau de Mestre em
Design e Desenvolvimento de Jogos Digitais
(2º ciclo de estudos)

Orientador: Prof. Doutor Frutuoso Silva

Covilhã, junho de 2016

Agradecimentos

Este relatório de projeto marca o fim de mais uma etapa académica na minha vida. O desenvolvimento de um projeto para a obtenção do grau de Mestre em Design e Desenvolvimento de Jogos Digitais representa um caminho longo e exigente. A motivação dada por algumas pessoas muito particulares deu-me coragem para continuar o meu trabalho, dia após dia. Gostaria de agradecer a todas as pessoas que direta ou indiretamente contribuíram para o meu trabalho ao longo do último ano.

Gostaria de começar por agradecer aos meus pais, a quem agradeço da forma mais profunda, por me terem sempre dado o melhor possível e me terem ajudado a conseguir alcançar os meus objetivos, não só na vida académica, como em todas as áreas pessoais e profissionais.

Agradecer também à minha irmã, Sofia, que foi incansável durante toda a minha vida académica, apoiando-me sempre nos bons e maus momentos, dando-me força e coragem.

Agradecer à minha namorada, Marília, por saber estar sempre presente nos momentos mais importantes da minha vida. O seu apoio, paciência e ajuda durante as alturas mais difíceis do desenvolvimento deste projeto foram cruciais para terminar esta fase da minha vida.

Agradecer a todos os meus amigos, tanto aos do mestrado como aos de Engenharia Informática, por todo o apoio, diversão, companheirismo e entreajuda.

Por último, mas não menos importante, expresso o meu agradecimento ao meu orientador Prof. Doutor Frutuoso Silva, por ter aceitado supervisionar o meu projeto, pela amizade, disponibilidade, compreensão e por me dar a oportunidade de desenvolver um projeto de forma a ganhar experiência em Desenvolvimento de Jogos Digitais.

A todos o meu sincero obrigado,

Nuno Filipe Alexandre Carapito

Resumo

O desenvolvimento de jogos digitais tem vindo a aumentar em todo o mundo. Nesta área, não são só empresas que desenvolvem jogos, existem também pessoas individuais ou pessoas organizadas em pequenas equipas que competem no mercado. Estes últimos, normalmente, contam com o apoio de motores de jogo (*software* desenvolvidos por outras empresas), que têm como principal objetivo acelerar o desenvolvimento de jogos. Os dois motores de jogos mais usados a nível mundial são o *Unity* e o *Unreal Engine*, sendo usados por centenas de milhares, se não milhões, de pessoas. Estes motores de jogo apresentam diversas limitações, no entanto, estas não são impeditivas de criar e lançar um jogo para o mercado.

Nos jogos digitais, a componente multi jogador tem vindo a ganhar uma importância enorme. Nos dias que correm, é raro um jogo sair para o mercado sem ter esta componente multi jogador. Embora o modo local seja cada vez menos usado (mas que ainda existe), o modo como estes jogos são jogados é normalmente *online*, visto que quase todas as casas dos países desenvolvidos têm acesso à Internet. Os jogos multi jogador tanto podem ser jogador contra jogador, como em modo cooperativo, em que vários jogadores se ajudam entre si de forma a completar desafios maiores.

A programação da componente multi jogador pode ser um grande desafio para os programadores, visto ser preciso ter em consideração várias componentes, que, caso não sejam verificadas, podem introduzir falhas no jogo e dar vantagens aos jogadores que as descubram. Para além disso, dependendo do tipo de arquitetura de rede usada no seu desenvolvimento, acrescem alguns problemas adicionais, a que é preciso dar a devida atenção.

Assim, este relatório apresenta o resultado da comparação entre dois tipos de arquiteturas usadas no desenvolvimento de um jogo multi jogador: a arquitetura *peer-to-peer* e cliente/servidor. Para este teste foi desenvolvido um jogo usando um motor de jogo e verificado qual das duas arquiteturas de rede é melhor para o desenvolvimento de um jogo de estratégia multi jogador. Serão descritos todos os passos tomados no seu desenvolvimento, os principais problemas encontrados, bem como as estratégias utilizadas para os ultrapassar.

Palavras-chave

Programação, jogos digitais, multi jogador, arquitetura cliente-servidor, arquitetura *peer-to-peer*, motor de jogo, *Unreal Engine*, *Unity*.

Abstract

The development of digital games has been rising all over the world. In this area, not only companies develop games, but there are also some individual people or people organized in small teams that are competing in the market. The last ones are usually helped by game engines, softwares developed by other companies, which main objective is to speed up the development of games. The two most used worldwide game engines are Unity and Unreal Engine, being both used by hundred of thousand, if not millions of people. This game engines have several limitations, but those don't stop people from creating and launching the game in the market.

In the digital games, the importance of the multi-player component has been increasing. Nowadays, it's rare that a game goes to the market without a multi-player component. Even though the local mode is barely used (but still exists), the online mode is the mode that is normally used, since almost all houses of the developed countries has access to the Internet. The multi-player games can be player versus player, or in collaborative mode in which several players help each other to complete bigger challenges.

Programming the multi-player component can be a challenge to the programmers, because it's necessary to give some attention to several details, which in case they are not checked, they can introduce several flaws to the game and give advantage to the players that find them. Beside that, depending the type of network architecture used on the development of the game, there may appear some adittional problems, which require proper attention.

This report lists the result of the comparison between two types of network architectures used in the development of a multi-player game, the peer-to-peer architecture and client/server architecture. For this test, it was developed a game with the help of a game engine, and checked which of the two architectures is better to the development of a multi-player strategy game. There will be also a description of the steps taken, what were the main problems encountered and how they were overcome.

Keywords

Programming, digital games, multi-player, client/server architecture, peer-to-peer architecture, game engine, Unreal Engine, Unity.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Este documento foi elaborado no âmbito da Unidade Curricular de Dissertação, Projeto ou Estágio, do segundo ano do segundo ciclo de estudos em Design e Desenvolvimento de Jogos Digitais da Universidade da Beira Interior na Covilhã, Portugal. Neste documento é apresentado todo o trabalho e pesquisa feita durante o desenrolar desta unidade curricular.

Inicialmente será contextualizado o projeto na área em que se enquadra e a sua motivação, seguido do problema e dos objetivos. Por fim, é introduzida a abordagem que vai ser usada para a resolução do problema, tal como a organização do documento.

1.1 Enquadramento e Motivação

O desenvolvimento de jogos digitais tem vindo a aumentar em todo o mundo. Nesta área, não são só empresas que desenvolvem jogos, existem também pessoas individuais ou pessoas organizadas em pequenas equipas que competem no mercado. Estes últimos, normalmente, contam com o apoio de motores de jogo (*software* desenvolvidos por outras empresas), que têm como principal objetivo acelerar o desenvolvimento de jogos. Os dois motores de jogos mais usados a nível mundial são o *Unity* e o *Unreal Engine*, sendo usados por centenas de milhares, se não milhões, de pessoas. Estes motores de jogo apresentam diversas limitações, no entanto, estas não são impeditivas de criar e lançar um jogo para o mercado.

Nos jogos digitais, a componente multi jogador tem vindo a ganhar uma importância enorme. Nos dias que correm, é raro um jogo sair para o mercado sem ter esta componente multi jogador. Embora o modo local seja cada vez menos usado (mas que ainda existe), o modo como estes jogos são jogados é normalmente *online*, visto que quase todas as casas dos países desenvolvidos têm acesso à Internet. Os jogos multi jogador tanto podem ser jogador contra jogador, como em modo cooperativo, em que vários jogadores se ajudam entre si de forma a completar desafios maiores.

A programação da componente multi jogador pode ser um grande desafio para os programadores, visto ser preciso ter em consideração várias componentes, que, caso não sejam verificadas, podem introduzir falhas no jogo e dar vantagens aos jogadores que as descubram. Para além disso, dependendo do tipo de arquitetura de rede usada no seu desenvolvimento, acrescem alguns problemas adicionais, a que é preciso dar a devida atenção.

Existem cada vez mais pessoas a produzir jogos digitais. Com uma quantidade elevada de tutoriais na Internet, uma documentação bastante completa e com uma comunidade que ajuda imenso, qualquer pessoa pode desenvolver um jogo individualmente. No entanto, existe falta de documentação, por parte das empresas que desenvolvem os motores de jogo, sobre como implementar diferentes arquiteturas de rede para jogos multi jogador *online*, e, entre as

diferentes arquiteturas, quais as vantagens de umas comparativamente às outras, bem como saber qual delas se deve implementar para o tipo de jogo desejado. O que motivou a realização deste trabalho foi tentar colmatar essa falha de informação, orientada aos jogos de estratégia multi jogador, e programar um jogo multi jogador usando as arquiteturas *peer-to-peer* e cliente/servidor.

1.2 Problema e Objetivos

O desenvolvimento de jogos digitais tem vindo a encontrar diversas barreiras ao longo dos anos, sendo por vezes impedidos de serem lançados devido a estas barreiras. Um exemplo disto é o *World Of Warcraft*, que encontrou uma barreira muito grande no lançamento do jogo a nível de arquitetura de rede, uma vez que cada servidor aguentava no máximo cinco mil jogadores ao mesmo tempo, tendo sido criados centenas de servidores diferentes. Com o passar do tempo, com maior capacidade de processamento por parte do *Hardware*, este limite tem sido aumentado e os servidores já chegam a ter cerca de dez mil jogadores e muitos deles interligados entre si, podendo os jogadores de um servidor jogar com jogadores de outro servidor. No entanto, continua a ser usada uma arquitetura cliente/servidor, que traz elevados custos para a empresa, que tem de sustentar toda a infraestrutura. Nos últimos anos tem sido feito um esforço a nível global para arranjar forma de transformar este tipo de arquitetura numa arquitetura *peer-to-peer*, na qual a informação é trocada diretamente entre os clientes. Para grandes empresas, habitualmente dotadas de equipas bastante numerosas, pode ser fácil implementar qualquer uma das arquiteturas, no entanto, para equipas pequenas, esta dificuldade pode atrasar bastante o desenvolvimento e subsequente lançamento do jogo. Para além disso, podem começar a desenvolver o jogo para um tipo de arquitetura e no fim verificarem que não está a funcionar como desejavam, tendo de refazer novamente toda esta implementação, visto não existir qualquer documentação que ajude este tipo de pessoas a escolher o melhor caminho.

O principal objetivo deste projeto é o desenvolvimento de um jogo de estratégia multi jogador *online*, em que serão aplicados os dois tipos de arquitetura: a cliente/servidor e *peer-to-peer*. Com este jogo pretende-se verificar qual dos dois tipos de arquitetura funciona melhor num jogo desenvolvido em *Unity*. Para além disso, também é pretendido que o jogo seja intuitivo e fácil de perceber por todos os jogadores, sem ser necessário algum tipo de conhecimento prévio. Os jogadores poderão criar as suas próprias partidas e jogar com os seus amigos, que se ligam ao criador da partida através do endereço *Internet Protocol (IP)* do mesmo. Um fator crítico no desenvolvimento deste jogo aponta para o facto de não existir muita documentação do *Unity* sobre o desenvolvimento de jogos multi jogador *online*, usando qualquer uma das duas arquiteturas.

1.3 Abordagem Adotada para a Resolução do Problema

Para conseguir os objetivos descritos na secção anterior, o trabalho deste projeto foi dividido nas seguintes tarefas:

- A primeira tarefa consistiu em compreender os conceitos básicos dos principais tipos de arquitetura de rede em jogos multi jogador, de modo a preparar as tarefas subsequentes;

- A segunda tarefa compreendeu a pesquisa de publicações académicas e da indústria de forma a perceber os principais problemas relativos a cada uma das arquiteturas e como os resolver;
- A terceira tarefa consistiu em estudar quais os motores de jogo disponíveis e quais os mais usados pelo mercado atual. Para além disso, foi criado o *Game Design* do jogo, a ser implementado na tarefa seguinte;
- A quarta tarefa focou-se no desenvolvimento do jogo para possibilitar o estudo sobre qual das duas arquiteturas funciona melhor no *Unity*;
- Na quinta e última tarefa realizaram-se dois testes ao jogo com vários utilizadores, cada um referente a uma arquitetura implementada. Nesta tarefa foram tiradas as conclusões deste projeto, a partir do resultado de um questionário feito aos jogadores.

1.4 Organização do Documento

Este relatório de projeto está organizado em cinco capítulos principais. O corpo do relatório é composto por três capítulos, precedido pelo capítulo Introdução e sucedido pelo capítulo Conclusão e Trabalho Futuro. O conteúdo e organização dos capítulos deste relatório podem ser sumariados da seguinte forma:

- Introdução

  Neste capítulo, o atual, é feita a contextualização do problema deste projeto ao ser introduzido o enquadramento, a motivação e os objetivos do trabalho realizado. A abordagem adotada para a avaliação do problema também é delineada neste capítulo. Por fim, é descrita a organização e estrutura deste relatório;

- Jogos Multi Jogador

  No segundo capítulo é feita uma análise ao aparecimento dos jogos, com especial destaque aos jogos que têm uma componente multi jogador. É também feita uma análise aos tipos de arquitetura multi jogador que serão testados no desenvolvimento do projeto e ao motor de jogo que foi usado no desenvolvimento do projeto;

- Implementação

  No terceiro capítulo é delineado o método adotado para analisar o problema descrito na secção 1.2. Em primeiro lugar é exposto um breve resumo do *game design* do jogo, seguido da apresentação de alguns detalhes relativos à construção do jogo, onde se mostra como foi criada toda a sua estrutura e o porquê de ser criado assim;

- Testes e Resultados

  No quarto capítulo procede-se à apresentação dos testes efetuados com vários utilizadores, usando as duas arquiteturas que serão testadas, bem como uma análise dos seus resultados;

- Conclusões e Trabalho Futuro

  O capítulo cinco conclui este projeto, incluindo algumas observações finais sobre o problema abordado neste trabalho e a apresentação das principais conclusões deste projeto. São também apresentadas possíveis direções para o trabalho futuro.

Capítulo 2

Jogos Multi Jogador

Este capítulo aborda o estudo aprofundado sobre o desenvolvimento de jogos multi jogador, onde é feita uma análise sobre os dois modos de jogo multi jogador e quais foram os primeiros a aparecer. De seguida, é feita uma análise às duas arquiteturas de rede usadas neste tipo de jogos e quais os seus principais problemas.

O presente capítulo apresenta também uma introdução de alguns conceitos que serão usados ao longo deste relatório, bem como uma breve introdução ao motor de jogo usado para programar o jogo usado nos testes, e a razão por ele ter sido usado.

2.1 Introdução

Ano após ano, o número de jogos publicados tem vindo a aumentar. Centenas de milhões de pessoas por todo o mundo jogam desde os jogos mais simples, como o *Candy Crush*, aos mais complexos, como o *World of Warcraft*, *Counter Strike*, *The Sims*, *League of Legends*, entre outros. Segundo a *U.S. Entretainment Software Association (ESA)* [(ES15], por cada casa americana existem em média dois jogadores e, dentro de cada uma destas casas, a idade média dos jogadores é de trinta e cinco anos. Este mesmo relatório mostra que o uso de jogos de computador nas casas da população americana continua a aumentar, tendência que se tem vindo a repetir, segundo o mesmo relatório dos anos anteriores.

Hoje em dia, a produção de jogos digitais nas grandes empresas envolve o trabalho de centenas de pessoas, entre os quais artistas, designers, programadores, músicos, entre outros. No início dos anos 90 do século passado, a empresa *ID Software* desenvolveu um conjunto de jogos comerciais, tais como *Commander Keen*, *Wolfenstein 3D* e *Doom*, com uma equipa de cinco a seis pessoas a tempo inteiro. Desde então, as equipas começaram a crescer e em 2000 já havia equipas com centenas de pessoas. O orçamento da maioria dos jogos comerciais populares é desconhecido, mas estima-se que jogos como o *World of Warcraft* tenham custado de vinte a cento e cinquenta milhões de dólares [HMvdV+13]. No entanto, o rápido aumento nos custos de produção originou o desmembramento de várias empresas internacionais, como por exemplo a *Interplay Entretainment* (criadora do *Fallout*).

2.2 Jogos Multi Jogador: Modo *Online* e *Local*

Os jogos multi jogador são jogos nos quais duas ou mais pessoas podem jogar no mesmo ambiente e ao mesmo tempo. Estes jogos permitem uma interação entre jogadores, quer seja em parceria ou em competição, oferecendo uma componente social que não existe nos jogos de um só jogador. Este tipo de jogos pode funcionar em dois modos distintos: o modo local e o modo *online*.

Os jogos multi jogador em modo *online* são os jogos que requerem que os jogadores se liguem a um servidor central (independente da arquitetura de rede usada), de forma a conseguir jogar com outros jogadores, utilizando uma ligação à Internet. Nos jogos multi jogador em modo local, os jogadores podem usar *Local Área Network (LAN)/Wireless Local Area Network(WLAN)*, ou até mesmo Bluetooth, para criar uma rede local com outros jogadores, não sendo necessária uma ligação à Internet.

2.2.1 O Aparecimento dos Jogos Multi Jogador

Os jogos multi jogador tiveram um aparecimento discreto, uma vez que os primeiros eram simples jogos só para um jogador, mas com características que eram visíveis em comunidades, como por exemplo, listas de melhores pontuações, torneios e *chats*. Isto permitia aos jogadores a possibilidade de comparar os seus resultados com outros jogadores, bem como competir para ver quem conseguia obter mais pontos. Este tipo de jogos foram apelidados de "jogos competitivos orientados à comunidade" [BN13].

Os jogos multi jogador apareceram há mais de quarenta anos. Uns dos primeiros foram o *Empire* de 1973 e o *Spasim* de 1974, suportando oito e trinta e dois jogadores, respetivamente. Antes destes foram desenvolvidos jogos multi jogador, mas com um limite máximo de dois jogadores, como por exemplo o *Tennis for Two* e o *SpaceWar!*. No entanto, os recursos do jogo tinham de ser partilhados por todos os jogadores. Anos mais tarde, começaram a ser lançados jogos com base numa ligação de rede, como por exemplo o *Midi Maze*. Após estes, começaram a aparecer os jogos mais conhecidos, como por exemplo o *Doom*, um *First-Person Shooter*, com dois estilos de jogo, *Deathmatch* e *Arena*. Posteriormente, os mais conhecidos a surgir foram o *Counter-Strike*, o *Halo* e o *Unreal Tournament*.

Após estes jogos surgiram os jogos baseados em turnos. Tal como o nome sugere, significa que os jogadores teriam de esperar que o turno do outro jogador acabasse para que pudessem jogar. Um exemplo típico deste tipo de jogos são os jogos de tabuleiro, como por exemplo o xadrez. Estes jogos são tipicamente calmos, pois não precisam de muita interatividade, comparativamente ao que requerem os jogos de ação.

Hoje em dia, o desenvolvimento de jogos com uma componente multi jogador em tempo real está cada vez mais popular. Para a maioria das empresas é impensável lançar um jogo sem este tipo de componente, uma vez que esta componente atrai mais jogadores, permitindo que o jogo dure mais tempo, conseguindo assim obter mais lucro.

2.3 Arquiteturas para Jogos Multi Jogador

Com o avançar do tempo e da tecnologia, a quantidade de jogos multi jogador tem vindo a crescer, não só em popularidade mas também em escala. Existem duas grandes arquiteturas de redes de computadores para jogos multi jogador: a arquitetura cliente/servidor e a arquitetura *peer-to-peer*.

Atualmente, na maioria dos jogos multi jogador é usada a arquitetura cliente-servidor. Neste tipo de arquitetura existe um servidor central e todos os clientes (ou seja, todos os

jogadores) se ligam a este servidor para poderem começar a jogar. Todas as ações tomadas pelos jogadores são enviadas para o servidor sob a forma de uma mensagem. O servidor processa de forma sequencial as mensagens recebidas e vai responder aos clientes quais são os efeitos das suas ações, também através de mensagens. A comunicação só ocorre entre o servidor e cliente e nunca entre os vários clientes. Este tipo de arquitetura começou a ser usada por ser bastante conveniente a nível de implementação e segurança, não só pela maior facilidade de implementação, mas também pela capacidade de evitar que os clientes façam batota, visto ser o servidor que toma as decisões mais importantes.

No entanto, com o aumento da complexidade dos *Massively Multiplayer Online Games (MMOGs)*, está a aparecer um problema grave na arquitetura clássica cliente-servidor (principalmente nos *Massively Multiplayer Online Role-Playing Games, MMORPG*), devido ao elevado custo do preço em *hardware* e *data centers*. Nos últimos tempos, tem sido feita uma pesquisa intensa para adaptar a arquitetura *peer-to-peer* aos *MMOGs*, para espalhar a carga pelas máquinas dos jogadores [HADF12]. Neste tipo de arquitetura, cada um dos jogadores funciona tanto como cliente quanto como servidor, existindo assim uma troca de informação entre os vários jogadores, sem que haja necessidade de um servidor central. No entanto, só a informação mais importante pode ser trocada com os vários jogadores, caso contrário, vai existir uma quantidade de informação bastante grande para ser trocada entre os vários jogadores, o que pode causar alguns problemas. É uma arquitetura bastante difícil de implementar, para além de também ser muito difícil prevenir que alguns jogadores façam batota, visto que os jogadores com experiência em redes podem alterar a informação dos pacotes de rede que enviam aos outros jogadores, sendo assim beneficiados (tal não é possível na arquitetura cliente/servidor pois é o servidor que faz todos os cálculos).

2.3.1 Principais Problemas dos Jogos Multi Jogador

Segundo Badar et al.[BN13], os principais problemas dos jogos *online* são:

- Latência;
- Perda de pacotes;
- Pouca largura de banda.

A latência e o limite na largura de banda são os dois principais problemas nos jogos *online*. Relativamente à perda de pacotes, era um grave problema até há uns anos atrás. No entanto, já foram criados métodos para evitar que a perda de um pacote afete uma partida entre vários jogadores.

A largura de banda é definida como a quantidade de dados que pode ser transmitida do ponto A para o ponto B, num determinado período de tempo, sendo chamada na gíria por "limite de tráfego". Embora nos países mais desenvolvidos as larguras de banda já sejam bastante elevadas, existem ainda países com bandas bastante reduzidas. Para além disso, existe também um limite bastante reduzido nas redes móveis, sendo do interesse do jogador que a quantidade de dados a serem transmitidos seja o mais baixo possível. As equipas de desenvolvimento têm de ter este aspeto em consideração quando estão a desenvolver jogos multi jogador *online*.

Outro problema é a latência, que é o tempo demorado entre o envio de um pedido (pacote de dados) e a chegada da sua resposta. Os jogos em tempo real necessitam de baixas latências, visto que um atraso no tempo de resposta pode causar degradação na experiência de jogo dos utilizadores. Os jogos baseados em turnos aguentam elevadas latências sem que sejam afetados por um atraso [BN13]. *Michael Powers* [Pow06] afirma que o principal problema de um jogo em tempo real é a latência da rede. Um jogo com elevada latência vai afetar os jogadores, levando a um elevado nível de inconsistência, podendo até dar vantagem injusta para alguns deles, removendo assim a igualdade entre todos.

Segundo Wang et al.[WJS09], os efeitos da latência em jogos *online* podem ser categorizados da seguinte maneira:

- Eficiência da rede;
- Consistência visual;
- Consistência do mundo de jogo;
- Igualdade.

Para resolver o problema de eficiência da rede pode ser usado a técnica de *Content Addressable Network(CAN)*, que é um sistema distribuído *peer-to-peer* e descentralizado que mapeia chaves n-dimensionais em valores. O benefício principal de usar esta técnica advém da menor necessidade de pacotes relativamente a uma transmissão simples. Outra solução é gravar em *buffer* as mensagens do estado do jogo, e, em vez de se enviar várias mensagens individualmente, enviar um conjunto delas. Ao agrupar as mensagens de evento do jogador e ao transmiti-las em intervalos causa menos problemas na rede.

Para resolver o problema da consistência visual existe uma técnica chamada *Dead Reckoning Technique* [WJS09]. Normalmente, num jogo em tempo real, existem bastantes objetos, tais como criaturas, veículos, entre outros. Para que todos os jogadores possam receber uma atualização de todos os objetos é necessário transmitir uma quantidade elevada de informação entre o servidor e os clientes. Isto causa um enorme consumo de banda de rede e aumenta o risco de latência. Esta técnica extrapola a posição exata do objeto ao saber a sua última posição e velocidade. Com esta técnica, os clientes apenas precisam de dizer ao servidor as mudanças de velocidade dos objetos sobre o controlo do utilizador.

Relativamente à consistência do mundo de jogo, existe um método chamado de *Bucket Synchronisation Mechanism*. Este método recolhe todas as mensagens de eventos do jogo e guarda-as num balde (*bucket* em inglês, o nome do método vem daí). Em conformidade com um intervalo específico, todas as mensagens no balde são processadas e é criada uma visão local do estado global. Esta é a melhor forma de sincronização para jogos multi jogador *online*, usando o sistema *peer-to-peer* [WJS09].
Pode-se também usar um algoritmo de tempo sempre que forem encontradas inconsistências num estado de jogo local. Com a ajuda do algoritmo, o jogo pode voltar atrás para um estado anterior que seja mais consistente.

Relativamente ao último problema, a igualdade é uma parte essencial para que um jogo seja justo e divertido para os jogadores. Não deve haver vantagens de uns jogadores sobre outros

devido à latência. Uma das formas de resolver este problema é com o método chamado *Sync-MS*, que promove a igualdade nos jogos ao balancear o tempo de resposta. Visto que clientes de todo o mundo experienciam latências variadas, alguns jogadores podem tomar ações de acordo com a última mensagem, antes de outros jogadores terem sequer recebido essa mesma mensagem. Idealmente, todos os clientes devem receber as mensagens de atualização do servidor ao mesmo tempo e isto é conseguido através do mecanismo *Sync-out*: ele coloca as mensagens de atualização em linha de espera e aguarda que as mensagens tenham chegado a todos os clientes. Só após todos os clientes terem recebido a mensagem é que ela é entregue ao jogo. Como resultado, todos os jogadores podem reagir ao mesmo tempo.

Sheldon et al.[SEB+03] partilharam na sua pesquisa o impacto da latência nos jogos de estratégia, referindo que não existem efeitos percetíveis quando a latência aumenta no intervalo de 0 milissegundos a 500 milissegundos. Latências por volta de 800 milissegundos causam degradação da experiência do jogo e transmitem informação errada ao jogador. No entanto, a frequência de atualizações nos jogos de estratégia não precisa de ser assim tão elevada, pelo que se pode "esconder" a latência, sem que o jogador repare nela.

2.4 Motores de Jogo

Um motor de jogo (*Game Engine* em Inglês) é um *framework* desenhado para a criação e desenvolvimento de videojogos. É usado principalmente por equipas pequenas para criação de jogos para computadores, consolas e dispositivos móveis. A principal funcionalidade de um motor de jogo é que engloba um motor gráfico para gráficos *2D* e *3D*, um sistema de física, sistema de som, sistema de *script*, animação, sistema de inteligência artificial, *networking*, entre outros. O processo de desenvolvimento de jogos é, assim, acelerado, ao reusar/adaptar o mesmo motor de jogo para vários jogos, ou para lançar o jogo em várias plataformas.
Atualmente existem muitos motores de jogo disponíveis. Eles variam em muitos aspetos, desde a principal linguagem de programação, à linguagem de *scripting*, se é *cross-platform*, se é orientado a *2D*/*3D*, a plataforma alvo e o tipo de licença.

Segundo a *statista* [Sta14], os cinco motores de jogo mais utilizados no Reino Unido são:

- *Unity 3D* (62%);
- *Unreal Engine* (12%);
- *Cocos2d* (9%);
- *CryEngine* 3(5%);
- *Marmalade* (5%).

Neste projeto foi usado o *Unity*, cujos detalhes vão ser descritos na próxima subsecção. Relativamente ao *Unreal Engine*, é um motor de jogo desenvolvido pela *Epic Games* que foi usado pela primeira vez em 1998. Desde então tem vindo a ser melhorando, estando atualmente na versão 4. Tem como linguagens principais o C++ e o *UnrealScript*. Já o *cocos2D* foi lançado em 2008 e é *open source*. Este tem várias versões, cujas diferenças entre si são as plataformas alvo e as linguagens de programação (*Python, C++, ObjectiveC, c♯, Javascript*, entre outras). O *CryEngine* é o motor de jogo usado no *Far Cry*, desenvolvido pela *Crytex*. Foi tornado gratuito

a 19 de agosto de 2011 e tem como linguagens principais o *C++, Lua e o c♯*. Por fim, o *Marmalade* é um motor de jogo desenvolvido pela *Marmalade Technologies Limited*. Tem como linguagens de programação *C/C++* e suporta um grande leque de plataformas.

2.4.1 *Unity*

O *Unity* (também conhecido por *Unity 3D*) é um motor de jogo desenvolvido pela *Unity Tecnologies*. A primeira versão foi lançada a 8 de Junho de 2005. Esta primeira versão foi limitada a *Mac OS*, tanto para criação, como para publicação de jogos.

O *Unity* está dividido em duas versões: a versão pro, que tem um custo de 1500 dólares, e a versão gratuita, que pode ser usada não só para fins comerciais, como para fins educacionais.

O *Unity* foi o motor de jogos usado para o desenvolvimento do projeto. Inicialmente foi pensado em utilizar ou o *Unity* ou o *Unreal Engine*. No entanto, na altura, o *Unreal Engine* não era gratuito (foi tornado gratuito a 2 de Março de 2016) e tinha uma curva de aprendizagem bastante elevada. O *Unity* tinha algumas vantagens comparativamente aos concorrentes: não só tinha uma curva de aprendizagem bastante baixa, como também tinha uma grande quantidade de suporte *online* e muitos tutoriais por onde se podia aprender. Para além disso, tinha a possibilidade de programar os *scripts* em c♯, o que neste caso era uma mais valia.

2.5 Conclusão

Este capítulo introduziu vários conceitos relacionados com os jogos multi jogador. Inicialmente, foi feita uma introdução sobre o estado geral dos jogos na atualidade, de forma a contextualizar a evolução da produção de videojogos. Esta área tem vindo a evoluir ao longo dos últimos anos e a tendência é de continuar a aumentar.

De seguida, foi feita uma introdução a um tipo específico de jogos, os jogos multi jogador, diferenciando-os em dois modos, o modo *online* e o modo local. Após isso, foi feita uma análise ao aparecimento deste tipo de jogos.

Foi então feita uma introdução a dois tipos de arquiteturas de rede que podem ser usadas nos jogos multi jogador, mostrando quais são as principais vantagens e desvantagens entre os dois modos. Embora a arquitetura *peer-to-peer* tenha algumas vantagens comparativamente à arquitetura cliente-servidor (como por exemplo o nível de latência), ainda peca por ser muito difícil evitar que os jogadores façam batota, como explicado anteriormente.

A última parte deste capítulo foi dedicada a fazer uma introdução aos motores de jogo, fazendo uma breve referência aos motores mais usados. Foi também feita uma breve introdução ao motor de jogo que foi usado neste projeto (*Unity*), cujo desenvolvimento é descrito no próximo capítulo.

Capítulo 3

Implementação

3.1 Introdução

No desenvolvimento de todos os jogos o primeiro passo é pensar no *Game Design* do mesmo. Os *Game Designers* são responsáveis pela conceção, criação e coordenação de todo o jogo, ou seja, definem o jogo, definem quais os seus objetivos, as regras, como é jogado, entre outros. Para além do Game Design existem outras áreas, tais como o *Game Art*, *Game Sound* e *Game Programming*, que são elaborados por outros elementos da equipa das empresas.
Neste capítulo procura-se explicar detalhadamente como foi pensado o *Game Design* do jogo e a forma como foi criado, a ordem em que foram implementadas as várias componentes, as dificuldades encontradas e como é que foram ultrapassadas.

3.2 *Game Design*

Um *Game Design Document* é um documento muito importante na preparação e no desenvolvimento de um jogo. Este documento contém toda a informação de um jogo, desde o seu nome, conceito, género, audiência, progressão no jogo, estrutura de níveis, objetivos, mecânicas, entre outros. Este documento é fulcral tanto para grandes empresas, como para pequenas empresas, porque permite que todos os trabalhadores olhem para o jogo da mesma maneira e não da maneira que cada um acha que é. Para além disso, ajuda também durante todo o seu desenvolvimento, fazendo com que não haja desvios do rumo traçado inicialmente.
O *Warcraft 3* é um jogo desenvolvido pela *Blizzard Entretaintment* em 2002, que recebeu a sua primeira e única expansão em 2003. É um jogo de estratégia em tempo real (com alguns elementos *Role-Playing Game(RPG)*) para computador. Foi um jogo muito bem recebido pela comunidade, não só por ser o terceiro jogo da saga *Warcraft*, mas também por trazer uma inovação no modo multi jogador, a adição de criaturas controladas pelo computador. Além disso, trazia também um editor de mapas, no qual a própria comunidade podia criar mapas customizados e jogar com os outros a partir da plataforma *online (battle.net)*. Estes mapas continuam a ser jogados nos dias de hoje.
A partir da customização dos mapas nasceu um dos mais conhecidos mapas do *Warcraft 3*, o *Defense of the Ancients: Allstars (DOTA)*. Este mapa tem sido atualizado ao longo dos anos, tendo a sua última atualização sido lançada em 27 de Março de 2015. Foi este mapa que fez surgir dois dos maiores *Multiplayer Online Battle Arena (MOBA)* da atualidade, o *League of Legends* e mais tarde o *Dota 2*.
Foi com esta premissa que foi pensando desenvolver um jogo multi jogador baseado no mapa do *Warcraft 3*, o *Vampirism Fire*, criando assim um *Standalone* com funcionalidades parecidas a esse mapa. Esse mesmo mapa já foi criado também no *Starcraft 2* e, tal como o *Dota*, continua a sofrer atualizações, tendo a última sido lançada a 15 de Março de 2016.
O *Vampirism Fire* é um jogo multi jogador composto por duas equipas. A primeira equipa é composta por um máximo de oito jogadores (daqui em diante chamados de humanos) e a

segunda equipa é composta por um máximo de dois jogadores (daqui em diante chamados de vampiros). Estes números podem ser alterados dependendo dos resultados dos testes realizados com os jogadores.
O objetivo dos humanos é angariar recursos pelo mapa, construir a sua base num dos vários pontos disponíveis, defendê-la com torres, que atacam o inimigo automaticamente, e treinar um herói, de forma a derrotar os vampiros. O objetivo dos vampiros é encontrar as bases dos humanos pelo mapa e destruí-las, ganhando mais poder por cada unidade/edifício que destroem.

Cada humano tem uma unidade principal (o trabalhador), que é o que tem a possibilidade de criar edifícios/reparar edifícios. Cada jogador só tem um trabalhador e não pode produzir mais. Para além desta unidade, existe o recoletor, que serve para apanhar os recursos (árvores neste caso). Tem ainda uma unidade especial, o herói, que durante o desenvolvimento vai ser um tanque. O vampiro apenas tem a sua unidade principal, o herói.
O trabalhador pode construir três tipos de edifícios: a refinaria, que vai servir para construir recoletores e estes irem recolher madeira; a fábrica de guerra, que vai permitir o jogador construir os seus tanques; e a torre de defesa, que irá atacar os inimigos mal estes se aproximem dela.
Existem dois tipos de recursos neste jogo: a madeira, que serve para construir novos edifícios, melhorar os mesmos e construir unidades; e a população, que vai limitar o treino de unidades. Cada humano só pode possuir no máximo um herói. A população do jogo pode ser aumentada até um máximo de duzentos através da construção de refinarias.
Ganham os humanos se matarem os vampiros ou, por outro, lado ganham os vampiros se destruírem todos os trabalhadores.
Vai ser também implementada uma sala de entrada, que vai possibilitar os jogadores escolherem a qual das duas equipas querem pertencer (vampiros ou humanos) e esperarem por mais jogadores. Estando o *Game Design* concluído é possível, assim, passar à programação do jogo.

3.3 *GameObjects* Iniciais

O *Unity* trabalha com base em *GameObjects*, que é a classe base das cenas do *Unity*. São objetos fundamentais que podem representar personagens, cenário, adereços, entre outros.
No entanto, sozinhos não são muito úteis, sendo sempre acompanhados por uma série de componentes que implementam a real funcionalidade do objeto. Uma das componentes que é obrigatória em todos os *GameObjects* é o *Transform*. Esta componente representa a posição, a orientação e o tamanho do objeto e não é possível ser removida. Para além do *Transform*, existem muitas outras componentes, tais como formas primitivas (cubos, esferas,...), câmaras, luzes, áudio, interface de utilizador, entre outras
Por exemplo, para adicionar uma câmara à cena, é preciso adicionar uma componente do tipo câmara ao objeto (ver imagem 3.1).

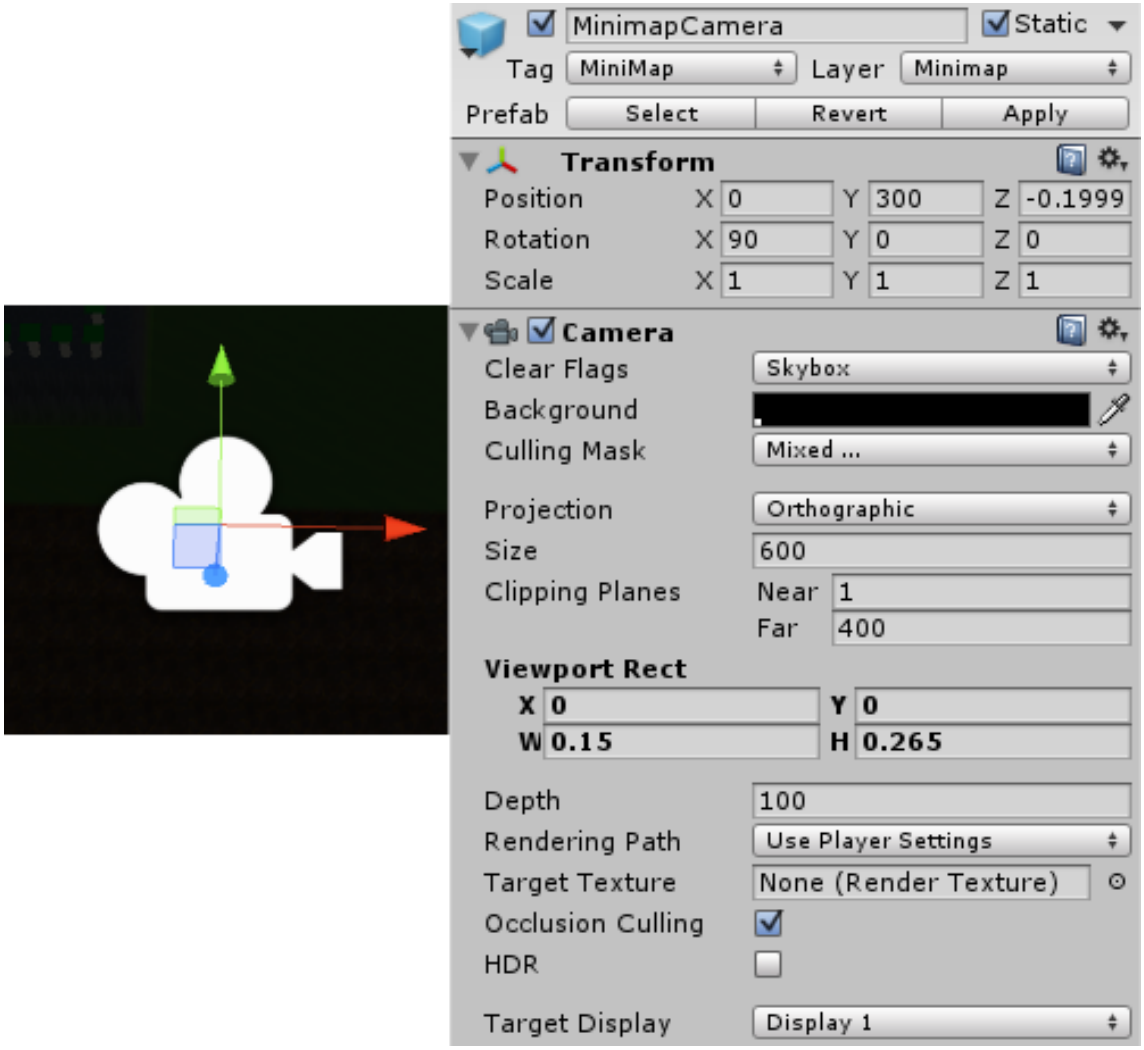

Figura 3.1: Exemplo de um objeto com uma componente do tipo câmara.

O primeiro passo para o início do desenvolvimento do jogo foi a criação de um novo projeto *3D* no *Unity*. Este projeto foi inicialmente criado com o *Unity* 5.2 e foi posteriormente atualizado para a versão 5.3.1 e depois para a 5.3.4 (de sublinhar que nenhuma destas atualizações teve um impacto direto no desenvolvimento do jogo, as alterações representam apenas melhorias no desempenho do motor de jogo).
O projeto foi dividido em duas fases: inicialmente foi programada toda a jogabilidade e só no fim foi implementado o modo multi jogador.

Para começar o desenvolvimento do jogo foi criada uma nova cena e nela foram criados cinco objetos:

- Uma câmara (câmara principal);
- Uma luz (sol);
- Um plano (vai servir de chão);
- Um cubo;
- Um objeto vazio com o *script User Input* (vai ser explicado na secção 3.5).

3.4 Objetos Globais

Tal como foi definido no Game Design do jogo, este tem unidades, edifícios e recursos como objetos no qual o jogador pode interagir com eles com algumas das suas unidades. No entanto, eles têm semelhanças entre si (tanto as unidades como os edifícios podem ser selecionados, todos eles devem reagir quando se carrega com o botão direito, entre outros).

Assim, foi possível definir todos estes objetos como ramificações de um objeto global e depois criar classes específicas para cada um deles, herdando as propriedades desta classe principal (ver imagem 3.2). Assim, foi criado o *script* "principal" *WorldObjects* e três *scripts*, que vão herdar as propriedades deste: o *"Units"*, o *"Buildings"* e o *"Resources"*.

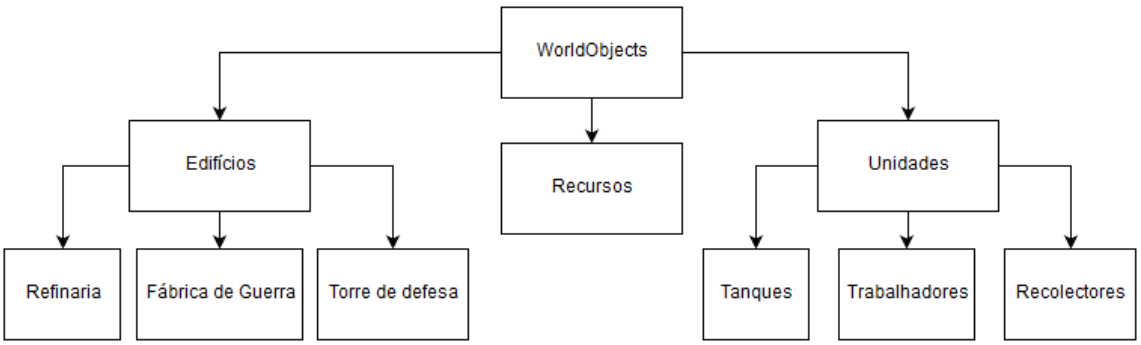

Figura 3.2: Hierarquia dos objetos globais.

Na sua definição foi preciso dizer que ele herda as funções do *WorldObjects* e não do *MonoBehaviour* (ver excerto 3.1).

```
public class Building : WorldObjects
```

Excerto de Código 3.1: Herança da classe *Building*.

No *script WorldObject* foram adicionadas um conjunto de variáveis como "`protected`" (ver excerto 3.2) e uma lista de funções, tais como "`SetSelection`", "`GetActions`" e "`PerformActions`". Visto serem do tipo *protected*, estas variáveis não podem ser acedidas por outros *scripts*, apenas por ele próprio e pelos *scripts* que pertencerem à sua hierarquia.

```
protected int Cost;
protected int SellPrice;
protected int Health;
protected int MaxHealth;
protected string Name;
protected Texture2D Texture;
protected bool CurrentlySelected;
```

Excerto de Código 3.2: Variáveis do *script WorldObject*.

3.5 *User Input*

3.5.1 Movimento da Câmara

A maioria dos jogos de estratégia tem um mapa grande e por isso é necessário que a câmara se mova. Quando o rato está na borda do ecrã, a câmara deve-se mover nessa direção. Quando é usado o botão de *scroll*, a câmara vai descer ou subir, o que permite fazer o *zoom in* ou *zoom out*, dependendo da direção em que foi feito o *scroll*. Para isto foram criadas variáveis estáticas, com o intuito de servir no movimento da câmara e tornar mais fácil alterar a velocidade, altura máxima, entre outros. No futuro, estas variáveis também vão permitir que os jogadores possam alterar estes valores nas opções de jogo, caso assim o desejem:

- `ScrollSpeed` e `ScrollWidth` - Velocidade de *scroll*;
- `MinCameraHeight` e `MaxCameraHeight` - Altura máxima e mínima da câmara;
- `MapLimitXPositivo`, `MapLimitXNegativo`, `MapLimitZPositivo`, `MapLimitZNegativo` - Posições máximas que a câmara pode ter.

A função `MoveCamera` vai tratar do movimento da câmara. Nesta função vão ser verificadas as coordenadas do rato no ecrã e vai ser usada a variável `ScrollWidth` previamente definida. Se a posição do rato for entre zero e o valor da variável, a câmara vai descer. Se estiver entre o tamanho máximo do ecrã e entre esse mesmo valor menos o valor da variável, a câmara vai subir. O mesmo acontece para o lado esquerdo e direito. A câmara só se vai movimentar se o valor estiver entre o valor das variáveis `MapLimit`.

3.5.2 Interface Gráfica

Neste tipo de jogo, a interface gráfica é um elemento chave que vai mostrar ao jogador tudo o que ele pode fazer. Para este jogo foram pensadas várias alternativas. A escolhida foi ter uma barra a toda a largura na parte inferior do ecrã, que iria conter o mini mapa, informações sobre as unidades/edifícios selecionados e os botões para poder controlar as unidades/edifícios. No topo do ecrã foi implementada também uma barra que vai mostrar os recursos que o jogador tem. Neste caso, vai mostrar a população usada/total e a quantidade de madeira disponível.
Existem duas possibilidades para desenhar o *User Interface(UI)* no *Unity*. Ele tem uma classe *(GUI)* que permite desenhar todo o tipo de objetos (retângulos, imagens, texto, etc). Inicialmente, esta foi a classe utilizada para desenhar a primeira versão da interface. No entanto, esta interface era de certa forma limitada e era necessário muito trabalho para implementar funcionalidades simples, como por exemplo os botões. A meio do projeto esta classe foi abandonada, tendo-se implementado toda a interface usando a nova *UI* gráfica.

3.5.2.1 Classe *GUI*

Para desenvolver a *UI* presente na imagem 3.3 foram criadas duas imagens com um tamanho de dezasseis por dezasseis com a cor desejada para ter como fundo (neste caso, amarelo escuro e azul escuro). Foi criada uma pasta com o nome de *HUD (Heads-Up Display)* (onde se colocou toda a informação sobre o *UI*) e dentro desta foi criada uma pasta chamada *"Skins"*. Dentro desta pasta criaram-se duas *GUISkins*, uma com o nome de *"OrderSkin"* e outra com o nome de *"ResourceSkin"*.

O *Unity* permite que sejam criadas *GUI Skins*, que é uma forma fácil de customizar uma série de componentes usadas em interfaces gráficas e que são usadas várias vezes. Basta alterar uma componente na *skin*, que todos os objetos, em que ela estiver a ser usada, vão sofrer essa alteração. A cada *skin* foi atribuído uma imagem background, previamente criada.
Foi criado um *GameObject* vazio, com o nome de *HUD*. Também foi criado um *script* com o mesmo nome, que irá desenhar toda a interface (ver excerto 3.3).

```
private void DrawOrdersBar()
{
GUI.skin = orderSkin;
GUI.BeginGroup(new Rect(Screen.width - ORDERS_BAR_WIDTH, RESOURCE_BAR_HEIGHT,
    ORDERS_BAR_WIDTH, Screen.height - RESOURCE_BAR_HEIGHT)); //Rectangle
GUI.Box(new Rect(0, 0, ORDERS_BAR_WIDTH, Screen.height - RESOURCE_BAR_HEIGHT),
    "");
//Desenhar o fundo do ecrã. começa no topo do grupo, e não do ecrã. A string
    vazia significa que não queremos mostrar texto nenhum lá escrito
GUI.EndGroup(); //Fecha o grupo
}
```

Excerto de Código 3.3: Função que desenha a barra do fundo do *UI*.

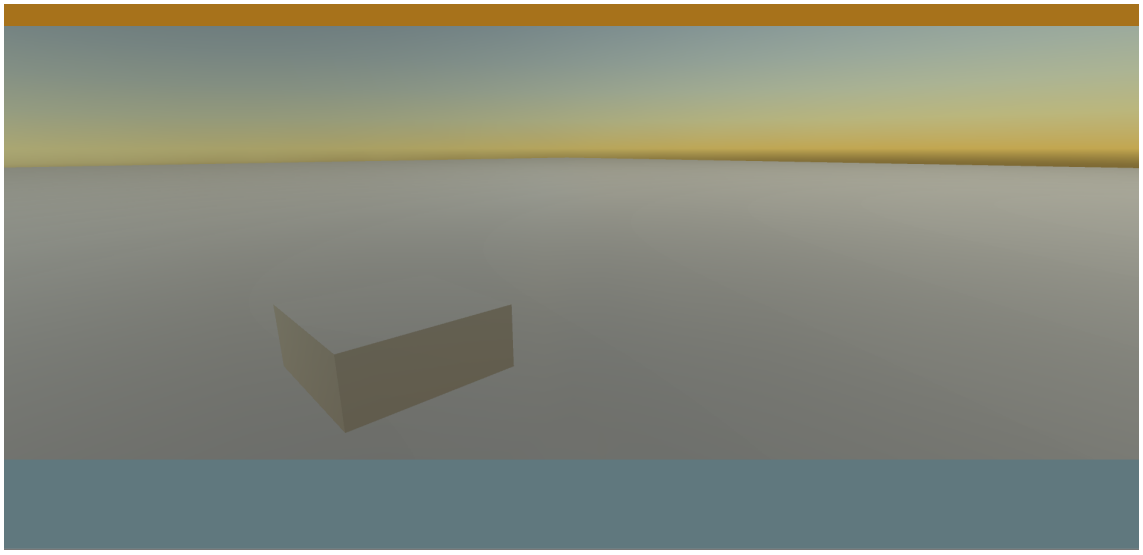

Figura 3.3: *Screen* do desenvolvimento inicial da *UI*.

3.5.2.2 *UI* Gráfica

O *Unity* na versão 4.6 introduziu uma nova *UI*, que, para além de ser muito mais rápida de implementar (é implementada através do *Inspector*), tem maior suporte e permite mais funcionalidades (como por exemplo, *sliders*, campos para introduzir informação, menus, texto, entre outros (ver imagem 3.4)). A funcionalidade mais importante para este projeto foi a possibilidade de fazer *resize* automático, conforme a resolução que estiver a ser usada.

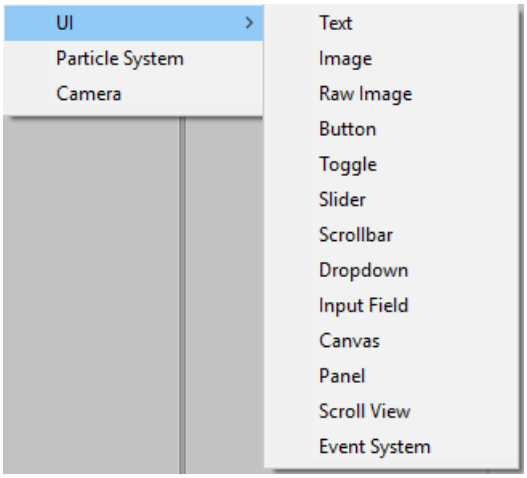

Figura 3.4: Funcionalidades da nova *UI*.

Assim, para desenvolver esta nova *UI*, foi criado um objeto do tipo *Canvas* (ver imagem 3.5). Dentro deste foram criados quatro painéis: um que contém todos os elementos da barra do topo do ecrã (barra de recursos); outro que engloba os elementos da barra de fundo do ecrã (barra de ações); um com o nome de *PausePanel*, que contém os objetos do menu de Pausa (opções, sair, etc) e outro com o nome de *VictoryPanel*, que engloba os objetos do menu de vitória.

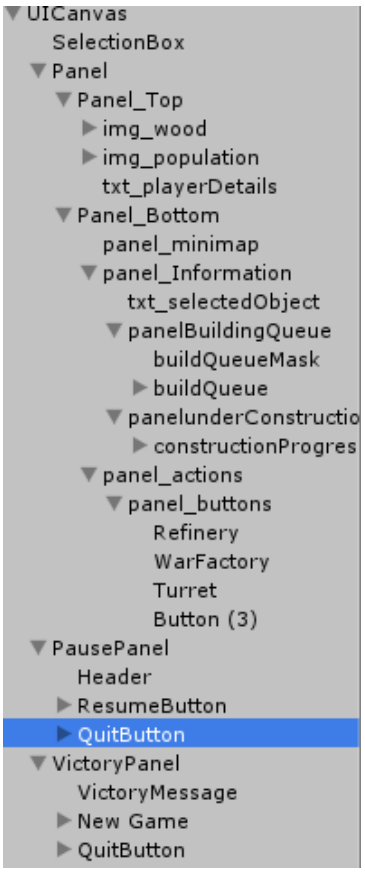

Figura 3.5: Hierarquia da *UI*.

Para mostrar ao jogador as ações que pode fazer neste novo *UI* gráfico foi preciso criar quatro objetos do tipo botões, filhos de um *GameObject* vazio com o nome de *"panel_buttons"*. Estes botões vão mostrar todas as ações que o jogador pode fazer quando tiver uma

unidade/edifício selecionado. Foi também criada a função `DrawActions`, que recebe um *array* de *strings* com as ações que a unidade/edifício selecionado pode fazer. Por cada ação deste *array* vai ser mostrado um botão, será colocada a imagem da ação e o nome do botão será substituído pelo nome da ação. Cada um destes botões tem uma função `OnClick`, que chama a função `HandleButton` e envia como parâmetro o nome do botão. Se um jogador tiver alguma unidade selecionada, ao carregar no botão será feita a ação dessa unidade (através da função `PerformAction`).

Foram também criados cinco *gameObjects* do tipo imagens, filhos de um *GameObject* vazio com o nome de *"buildQueue"*. Cada uma destas imagens vai representar uma unidade em fila de espera para ser criada e serão desenhadas pela função `DrawBuildQueue`. Por cada uma das unidades, que vão estar em fila para serem criadas, vai ser mostrada a imagem desta mesma. Será também mostrada uma máscara na primeira posição, que representa o tempo de treino da unidade e que diminui conforme o tempo que falta para estar concluído o processo.

Ao clicar numa destas imagens é chamada a função `CancelTrain`. Nesta função será verificado se o jogador tem algum edifício selecionado e se este pertence ao jogador. Se pertencer, vai ser verificado o nome da imagem que foi clicada (*Image1*, *Image2*, ...) e cancelado o treino da unidade respetiva, que está na fila de espera, na posição do número da imagem (a terceira unidade se for carregado na *Image3*, por exemplo).

Uma funcionalidade que a nova *UI* do *Unity* traz (e que acabou por ser um *bug* no desenvolvimento do jogo) é a possibilidade de usar teclas do teclado como se fosse o botão esquerdo do rato. Isto é, caso o jogador tivesse carregado num botão, poderia carregar no espaço para continuar a "carregar" nesse botão. Para resolver isto, foi necessário definir a propriedade *Navigation* dos botões para "*None*".

3.5.2.3 Cursores

É importante para o jogador saber as ações que pode fazer. Uma das formas de transmitir isto ao jogador é a partir do cursor do rato. O jogo foi desenvolvido com sete tipos de cursores diferentes (ver imagem 3.6):

- Atacar - Mostrado sempre que o jogador tenha uma unidade/edifício selecionado que possa atacar e o cursor esteja em cima de uma unidade/edifício de um jogador de outra equipa;
- Recolher Recursos - Sempre que o jogador possuir um recoletor selecionado e o cursor se encontrar em cima de um recurso;
- Mover unidade - Estado normal do cursor;
- Mover câmara - Quando o cursor estiver nas bordas do mapa e a câmara se estiver a mexer;
- Selecionar - Quando o cursor se encontrar em cima de uma unidade/edifício;
- *Rally Point* - Quando o jogador estiver a alterar o *Rally Point* de um edifício;
- Proibido - Quando o movimento não for possível (descrito na secção 3.16).

Figura 3.6: Imagem dos Cursores usados no jogo.

Para isso foram criados variáveis do tipo `Texture2D`. A variável `activeCursor` vai conter a imagem que deverá ser mostrada pelo jogo no momento presente e é atualizada sempre pela função `DrawMouseCursor` do *script HUD*. Nesta função, se o rato estiver em cima do *UI*, vai ser desenhado o cursor normal do mapa. Caso contrário, vai ser desenhado o cursor da ação que estiver a decorrer. Existem também oito variáveis que vão guardar as imagens das ações definidas anteriormente, que vão ser usadas pelo cursor.

Foi também criada uma `skin` para ser mais fácil atribuir as imagens ao cursor, bem como a função `UpdateCursorAnimation`. Esta função serve para as ações que têm mais que uma imagem (por exemplo, a ação de ataque tem duas imagens) e vai trocando as imagens entre si, de forma a parecer que o cursor está animado.

De forma a enumerar todos os tipos de ações que são precisas para este projeto e ficar tudo organizado foi criado na pasta *RTS* um *script* com o nome de *Enum*. Para começar, foram lá definidos os tipos de ações que o rato pode representar.
Na função `GetCursorDrawPosition` é verificado em que sítio do mapa o cursor está. Nesta função é chamada a função `SetCursorState`, que é usada para trocar a imagem do cursor facilmente.

Para o cursor de ataque foi necessário fazer com que a imagem deste mude caso esteja em cima de um inimigo. Para isso, foi modificada a função `SetHoverState` no *script WorldObjects*, na qual é verificado se a unidade/edifício selecionado é do jogador. Se sim, é verificado se o cursor está em cima de uma unidade/edifício. Se puder atacar (função `CanAttack`, ver excerto 3.4) e a unidade/edifício for de um jogador inimigo, é colocado o cursor de ataque; caso contrário, é colocado o cursor de mover.

A função `CanAttack` foi definida no *WorldObjects*, que devolve um booleano a `false`. Posteriormente, foi feito um `override` em cada *script* individual. Por exemplo, no *script Tank*, a função `CanAttack` vai devolver `true` (ver excerto 3.4).

```
public override bool CanAttack() {
    return true;
}
```

Excerto de Código 3.4: Função *CanAttack* do *script Tank*.

3.5.3 Cor dos jogadores

Nos jogos multi jogador, cada jogador tem uma cor que é diferente das cores dos outros jogadores da partida. Esta cor ajuda o jogador a identificar as suas unidades/edifícios e as unidades/edifícios dos outros jogadores. Para isso, foi criada a função `SetTeamColor`, onde é verificado em todos os objetos (unidades e edifícios) quais das suas componentes têm um *script TeamColor* adicionado. Se tiver, vai mudar a cor do material desse objeto para a cor do

jogador. O *script TeamColor* é um *script* vazio, serve apenas para indicar quais as componentes que devem ser pintadas.

No *script Player* foi criada uma nova variável global do tipo `Color`, para que se possa guardar a cor que um determinado jogador tem. A função `SetTeamColor` também é chamada na função `Construct`, para que, quando a construção de um edifício acabar, seja alterada a cor do edifício.

3.5.4 Health Bar

Nos jogos de estratégia é muito importante transmitir ao jogador o estado de vida das suas unidades/edifícios, isto é, permitir que o jogador saiba de uma forma visual (sem se interessar com os números) a quantidade de vida que as suas unidades/edifícios possuem no momento. Foi implementada uma barra que diminui e muda de cor conforme a vida da unidade. Esta vida só é mostrada quando o jogador está selecionado, para não pesar demasiado no ecrã de jogo. Inicialmente foi construída com a *UI* antiga (secção 3.5.4.1) e só depois foi migrada para a nova interface (secção 3.5.4.2).

3.5.4.1 *GUI*

Para criar esta barra, foi adicionada uma chamada à função `CalculateCurrentHealth`, na função `DrawSelectionBox` do *script WorldObjects*. Nesta nova função vai ser calculada a percentagem de vida da unidade, valor este utilizado na atribuição da cor da barra de vida (ver imagem 3.7).

Após isto era necessário saber a que percentagem é que a cor da barra mudava. Foi definido que se a vida da unidade/edifício fosse superior a 65%, era de cor verde; se fosse entre 65% e 35%, era amarelo e se fosse inferior, era vermelha. Foram criadas três variáveis do tipo `Texturas2D`, que iriam guardar três texturas de dezasseis por dezasseis com a cor verde, amarela e vermelha.

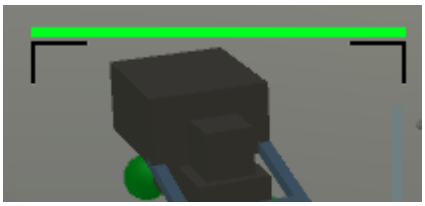

Figura 3.7: Barra de vida no *GUI*.

3.5.4.2 *UI*

Com a nova *UI* do *Unity* a barra de vida foi muito mais simples de ser implementada. Foi criado um objeto do tipo *canvas* e um objeto do tipo imagem como filho do *canvas* e foi adicionado este *canvas* como filho de cada unidade/edifício. Assim, quando uma unidade for selecionada, é mostrada a barra de vida ou escondida caso seja desselecionada. No entanto, era importante colocar o *canvas* sempre numa perspetiva perpendicular com a câmara. A função `CameraFacingBillBoard` (*script* criado pela comunidade do *Unity*) veio resolver isso, colocando o *canvas* sempre numa perspetiva perpendicular, independentemente do ângulo da câmara.

No entanto, tal como no caso anterior, a barra de vida precisa de mudar de cor de acordo com a quantidade de vida que a unidade/edifício tem. Para isso entrou a função `ChangeColorByScale`,

em que é definida a cor para quando a vida estiver no máximo (verde), a cor para quando a vida estiver no fim (vermelho) e o valor máximo e o valor mínimo. Quando a barra de vida está a ser mostrada, a quantidade de vida da unidade vai ser comparada com os valores definidos como máximo e mínimo e atribuir assim uma cor à barra, que varia entre as duas cores definidas.

Figura 3.8: Barra de vida no *UI*.

3.5.5 Programação dos Botões do Rato

Existem jogos em que apenas é usado o botão esquerdo do rato para tudo, enquanto que noutros também se usa o botão direito, que serve para selecionar unidades/edifícios. Neste jogo, o botão esquerdo do rato serve para selecionar/desselecionar unidades, escolher o local onde vão ser construídos os edifícios e interagir com a interface gráfica. O botão direito serve para dar instruções, tais como mover, atacar, recolher recursos e definir *RallyPoints*.

Um jogo de estratégia precisa de ter algo que permita selecionar ou desselecionar unidades. Para isso, pode ser usado o rato, as teclas ou até mesmo um comando (para os jogos de consolas). Foram criadas duas funções no *script UserInput* para tratar dos inputs do jogador. Se o jogador carregar com o botão esquerdo do rato (tem o valor zero), vai ser chamada a função `LeftMouseClick`; caso carregue com o botão direito do rato (tem o valor um), vai ser chamada a função `RightMouseClick`. Estas funções são chamadas na função `Update` do mesmo *script* (ver excerto 3.5).

```
if(Input.GetMouseButtonDown(0))
    LeftMouseClick();
else if(Input.GetMouseButtonDown(1))
    RightMouseClick();
```

Excerto de Código 3.5: Função para resolver dos cliques do rato.

Dentro da função `LeftMouseClick` é verificado se o rato está dentro da área de jogo e não em cima do interface (através da função `MouseInBounds`). Caso esteja, vai verificar se algum objeto é atingido (fazendo um `RayCast` a partir da câmara até bater no primeiro objeto). Se o objeto atingido pelo `RayCast` não for o chão, coloca-se essa unidade/edifício como selecionado. Se houver alguma unidade/edifício selecionado, o que estava selecionado deixa de estar e um novo é colocado como selecionado. Para mostrar ao jogador que a unidade/edifício está selecionado é desenhada uma caixa à volta da unidade/edifício, que é escondida quando este for desselecionado.

Para isso foi adicionado o seguinte na função `Update` do *script WorldObjects* (ver excerto 3.6).

```
if(currentlySelected)
  DrawSelection();
```

Excerto de Código 3.6: Desenha a caixa de seleção se o jogador tiver alguma unidade/edifício selecionado.

Inicialmente, a função `DrawSelection` criava uma caixa do tipo *GUI* onde era aplicado uma `Skin`. Esta `skin` tinha uma imagem carregada que era transparente e com uma borda preta (ver imagem 3.9).

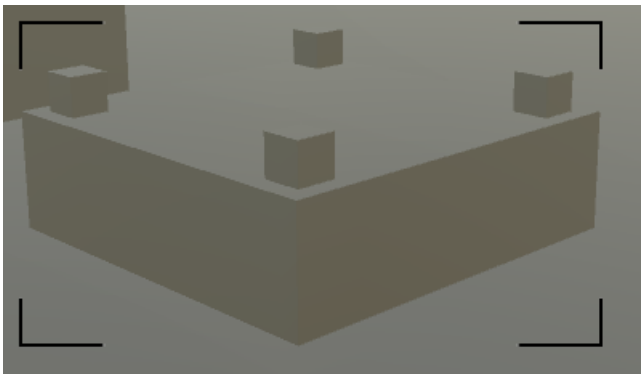

Figura 3.9: Caixa de seleção usando o *GUI*.

Na nova versão da interface foi criado um novo material, com uma imagem de um círculo azul desenhado e o resto dela transparente. Este material foi aplicado a um plano, que foi colocado como filho de todas as unidades e de todos os edifícios. Após isso, foi apenas preciso trocar a função `DrawSelection`, que vai ativar/desativar o *Renderer* do plano, caso a unidade/edifício estivesse selecionado ou não (ver imagem 3.10).

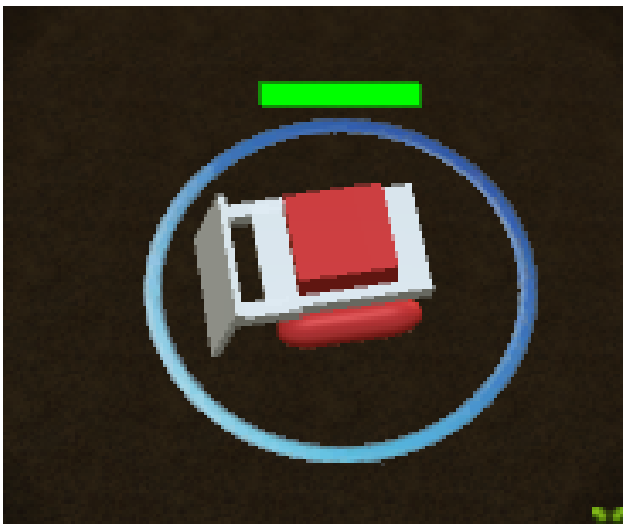

Figura 3.10: Caixa de seleção usando o *UI*.

Estando a seleção de uma unidade implementada, faltava implementar uma funcionalidade muito importante nos jogos de estratégia. Nos jogos de estratégia, quando o jogador carrega em cima do mapa e arrasta o rato (enquanto continua a premir o botão esquerdo

do rato), é desenhado no ecrã de jogo um quadrado (ver imagem 3.11). Assim que o botão esquerdo é largado, o jogo vai analisar quais as unidades/edifícios que estão dentro da caixa de seleção e vai selecioná-los. Um pormenor interessante nesta funcionalidade é que, caso existam unidades/edifícios dentro desta caixa, só é selecionado um dos tipos, com base no tipo de objeto que foi primeiramente selecionado.

Figura 3.11: Caixa de seleção do *Starcraft 1*.

Para implementar a caixa de seleção foi necessário criar um *GameObject* do tipo *Image* e colocá-la dentro do *Canvas* do *HUD* (esta imagem vai ser a caixa de seleção do jogo). De seguida, no *script HUD*, foi criada a função `DrawSelectionBox`, que vai verificar o estado do clique do botão esquerdo do rato. Quando o jogador carrega nele é gravada a posição inicial do rato e a imagem previamente criada é movida para essa posição. Enquanto o jogador estiver a premir o botão esquerdo, o tamanho da caixa de seleção é alterada, conforme a posição do rato.

Quando o jogador deixa de carregar no botão esquerdo do rato é feito um *reset* ao tamanho e posição da caixa. Faltava apenas verificar se existia algum objeto dentro desta área, antes de fazer o *reset*. Esta verificação é feita na função `CheckIfSelected`, na qual é chamada a função auxiliar `GetBounds`. A função `GetBounds` recebe a posição de um objeto e vai verificar se a posição deste está dentro da posição inicial e a posição final do rato. Se estiver, devolve `true`; se não, devolve `false`. Esta verificação funciona em qualquer sentido, isto é, funciona caso o jogador selecione de cima para baixo, da esquerda para a direita ou ao contrário.

A função `CheckIfSelected` vai ser chamada na função `Update` do *script WorldObjects*. Quando o jogador deixar de premir o botão esquerdo do rato, o jogo vai verificar se a unidade/edifício se encontra dentro da caixa de seleção. Se estiver, esta unidade/edifício vai ser selecionado. Caso contrário, não faz nada.
No entanto, só uma unidade é que estava a ficar selecionada, embora visualmente estivesse a mostrar várias unidades selecionadas (a *selecionBox* era mostrada, mas só uma

unidade respondia aos comandos). Para permitir que várias unidades/edifícios fossem selecionados, teve de ser criado uma variável do tipo lista de *WorldObjects* com o nome de `selectedObjects` e substituir todo o código de seleção de unidades/edifícios em todos os *scripts*.

Na função `LeftMouseClick` não só era colocada a unidade/edifício como selecionado, mas também era adicionada à lista. Na função `CheckIfSelected` do *script WorldObject* foi feita uma alteração também no mesmo sentido, em que se o objeto estivesse dentro desta caixa, era adicionado à lista. Relativamente às ações a serem apresentadas quando havia várias unidades/edifícios selecionados, as ações mostradas são as da unidade/edifício na posição zero da lista (basicamente é o líder do grupo). É também importante dar destaque à alteração na função `RightMouseClick`, em que, por exemplo, se existirem várias unidades selecionadas e for dada a instrução de se moverem para uma posição, esta instrução é dada a todas as unidades que estão presentes na lista de unidades selecionadas.

Um problema que estava a surgir era que, quando um trabalhador planeava uma construção, enquanto ele se movia até ela, era possível selecionar o edifício (que só estava planeado ainda) e fazer as ações do mesmo. Foram feitas verificações adicionais para que apenas fosse possível selecionar um edifício e só fosse mostrada a `selectionBox`, se este tivesse mais que zero pontos de vida. As ações deste só vão ser mostradas se o edifício não estiver sobre construção.

3.6 Treino de Novas Unidades

Para criar unidades é preciso ter um edifício que tenha como ação a criação de unidades.

Para o treino de unidades foi usada a classe `Queue` do c♯, que de certa forma é parecida à classe `List`. No *script building* foram adicionadas três variáveis, uma do tipo `Queue<string>` (*string* que serve para guardar o nome das unidades que vão ser treinadas), outra que demonstra o tempo que demora cada unidade a ser treinada e o `spawnPoint`.

A função `ProcessBuildQueue` é chamada na função `Update` e serve para processar a lista de espera de treino das unidades (ver excerto 3.7). De forma a ser processada a fila de espera de treino do edifício é preciso verificar se existe alguma unidade para ser treinada. Se existir, o tempo de treino da unidade atual é incrementado. Caso este tempo seja superior ao progresso máximo, a unidade é criada.

```
protected void ProcessBuildQueue() {
    if(buildQueue.Count > 0) {
        currentBuildProgress += Time.deltaTime * ResourceManager.BuildSpeed;
        if(currentBuildProgress > maxBuildProgress) {
            if(player) player.AddUnit(buildQueue.Dequeue(), spawnPoint,
                transform.rotation);
            currentBuildProgress = 0.0f;
        }
    }
}
```

Excerto de Código 3.7: Função que processa a lista de espera do treino de unidades.

De forma a ter uma lista de unidades/edifícios que podem ser criados pelo jogador, foi criado um *GameObject* com o nome de *GameObjectList*, que vai conter um *script* com o mesmo nome. Tem também quatro variáveis: três *arrays* de *GameObjects*, em que um deles vai conter todos os edifícios, outro todas as unidades e outro todos os *WorldObjects*, e a quarta variável vai ter o *GameObject* do jogador.
Neste novo *script* foi adicionada a garantia de que ele é único em cada jogo e não é criado várias vezes por algum motivo (ver excerto 3.8).

```
private static bool created = false;
void Awake() {
    if(!created) {
        DontDestroyOnLoad(transform.gameObject);
        ResourceManager.SetGameObjectList(this);
        created = true;
    } else {
        Destroy(this.gameObject);
    }
}
```

Excerto de Código 3.8: Garantia de que apenas existe um objeto deste tipo no jogo.

Para deixar o objeto preparado para o futuro foram logo criadas todas as funções `Get`, para todas as variáveis (`GetBuilding`, `GetUnit`, `GetWorldObject`, `GetPlayerObject` e `GetBuildImage`).
Na fábrica de guerra foi implementado o treino de um tanque. Para isso, foi criado o *script WarFactory*, *script* que herda as propriedades do *script Building* e foi adicionado o seguinte código (ver excerto 3.9). Foi também criado um *script* com o nome de *Tank* para o tanque, que vai herdar as propriedades do *script Unit*.

```
public class WarFactory : Building {
    protected override void Start () {
        base.Start();
        actions = new string[] { "Tank"};
    }
}
```

Excerto de Código 3.9: Definir ações da fábrica de guerra.

Falta apenas mostrar ao jogador na barra da interface a informação de que pode treinar um tanque. Foi assim criada a função para mostrar as ações do edifício. Nesta função é chamada a função (`IsOwnedBy`), que devolve `true` se o edifício pertencer ao jogador, ou `false` se não. Se pertencer ao jogador, é chamada a função `DrawActions` no *script HUD*. Nesta função vai ser criado um `UI` com botões e vai ter um número de botões igual ao número de ações que o edifício tiver.

Isto era o suficiente para mostrar as ações do edifício, faltando apenas criar a unidade. Para isso, no *script WarFactory*, foi adicionada a função `PerformAction`, que recebe uma *string* (o nome da unidade a ser construída) e é chamada a função `CreateUnit` (ver excerto 3.10). A função `CreateUnit` vai colocar uma unidade na lista de espera. Esta unidade só é colocada na lista caso o jogador tenha população livre.

```
public override void PerformAction(string actionToPerform) {
    base.PerformAction(actionToPerform);
    CreateUnit(actionToPerform);
}
```

Excerto de Código 3.10: Função que decide o que fazer com os cliques nas *actions*.

Visto que um jogador poderia ter uma quantidade elevada de unidades, a hierarquia do *Unity* iria ficar muito confusa. Por isso foram criados dois *GameObject* vazios, um com o nome de *Units* e outro com o nome de *Buildings* (ver imagem 3.12). Dentro de cada um destes vão aparecer todas as unidades e todos os edifícios, respetivamente, daquele jogador.

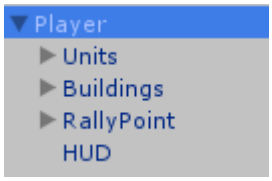

Figura 3.12: Hierarquia do *GameObject Player*.

Neste momento, ao carregar no botão para criar um tanque, o que iria acontecer seria que a cada *X* segundos iria ser criado um tanque. Este tanque era criado com base na função `Instantiate` do *Unity*, que cria um objeto igual ao objeto fornecido. Neste caso, vai criar uma cópia do objeto do tanque.
Durante o treino da unidade não existia nenhuma forma visual de dizer quanto tempo faltava para concluir o seu treino. Para ser mostrado, na função `DrawOrdersBar` no *Script HUD* é chamada a função `DrawBuildingQueue`. Nesta função vai ser mostrada a imagem das unidades que estão em fila de espera e é colocada uma imagem transparente (uma espécie de máscara), que vai reduzindo o seu tamanho conforme o tempo que falta para concluir o treino (ver imagem 3.13).

Figura 3.13: Botão de treino (esquerda) e máscara para mostrar o tempo que falta para acabar o treino (direita).

A lista de espera de treino de unidades não tinha nenhum limite máximo, isto é, um jogador poderia meter mil unidades na lista de espera e não fazer mais nada o resto do tempo. Isto seria uma grave falha no jogo, pois não só o jogador podia colocar em fila de treino uma quantidade elevada de unidades, mas se, por algum motivo, ele quisesse cancelar o treino, teria de cancelar mil unidades.
Para resolver isto foi feita a verificação do tamanho da lista de espera na função `CreateUnit`. Se o tamanho da lista de espera for inferior a cinco, é colocado uma nova unidade na lista, caso contrário, é ignorado.

Para cancelar o treino de uma unidade da lista de espera é feito um processo muito semelhante. Foi criada a função `CancelUnit`, que vai verificar se o tamanho da lista de espera é superior a zero. Se for, vai cancelar o treino de uma unidade na posição desejada. Se a unidade cancelada for aquela que está a ser construída, o tempo de treino é colocado a zero.

Relativamente aos recoletores foi necessário indicar a qual refinaria estes devem entregar os recursos. Isso foi feito quando a unidade é criada (posteriormente será mudado para que o recoletor vá entregar os recursos à refinaria mais próxima). A referência da refinaria é enviada como parâmetro função `AddUnit`. No caso das outras unidades, o parâmetro enviado é `NULL`.

3.7 Mover Unidades e dar Ordens

Neste tipo de jogos é fulcral que as unidades se movam. Neste caso, uma unidade que esteja selecionada vai-se mover através do clique do botão direito do rato.

Para começar, foi adicionada uma verificação à função `MouseHover`, que vai mudar o cursor do rato de acordo com o dono do edifício. Isto é, na função vai ser verificado quem é o dono do objeto selecionado. Se for do próprio jogador, a imagem do cursor do rato vai deixar de estar no estado de selecionar e vai passar para o estado de mover (através da função `SetHoverState`).

Para além disso, no *script Unit*, foi criada a função `RightMouseClick` que, juntamente com a função `StartMove`, vai tratar do movimento da unidade. A função `StartMove` vai indicar qual é a posição de destino do objeto e a função `MakeMove` (chamada na função `Update`) vai fazer com que a unidade se mova efetivamente para a posição desejada (verificando em que direção é que ela tem que se mover e alterando a posição da unidade progressivamente).

Foi também criada a função `TurnToTarget`, que, tal como o nome indica, faz com que o objeto rode sobre si próprio, de forma a ficar frontal para o sítio para onde se vai mover.

A função `StartMove` determinava o destino de uma unidade até uma determinada posição. No entanto, isto não iria funcionar para os recoletores, visto que eles vão até um objeto e não até uma posição definida do objeto. Por isso, foi preciso fazer uma atualização à função `StartMove` no *script Unit*, para que o recoletor se mova para a árvore (ver excerto 3.11). Em vez de se substituir a função `StartMove`, foi criada uma nova versão do `StartMove`, com argumentos diferentes. Nesta é enviado como parâmetro um objeto que vai ser o destino.

```
public void StartMove(Vector3 destination, GameObject destinationTarget) {
    StartMove(destination);
    this.destinationTarget = destinationTarget;
}
```

Excerto de Código 3.11: Nova função `StartMove`, que move a unidade até um objeto.

Faltava ajustar a posição de destino da unidade, quando ela se mover até um objeto, para garantir que ele não ia contra o objeto. Para isso, no *script Unit* foi criada a função `CalculateTargetDestination`, que é chamada no fim da função `TurnToTarget`. Nesta função vai ser verificada a posição dos quatro cantos do objeto (seja um edifício, recurso ou unidade) e desvia o local de destino da unidade um determinado valor, garantindo que não existe colisão (ver imagem 3.14).

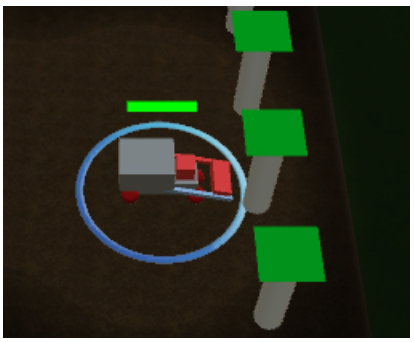

Figura 3.14: O recoletor não colide contra a árvore, mantendo uma distância mínima.

3.8 Construção de Novos edifícios

Tal como foi definido no *Game Design*, o trabalhador vai ser a unidade que vai construir novos edifícios. Para isso, no *script Worker*, vão ser dadas duas ações que ele pode ter (ver excerto 3.12).

```
actions = new string[] {"Refinery", "WarFactory", "Turret"};
```

Excerto de Código 3.12: Ações do trabalhador.

Para além disso, este trabalhador vai ter no seu *script* a função `setBuilding`, que serve para definir um edifício como projeto atual do trabalhador (aquele em que ele está a trabalhar), dizer-lhe para mover para a localização do edifício e definir a variável `building` a `true` (significa que está a construir um edifício).

Visto que a função para mostrar as ações no *UI* já estava feita do treino de unidades (3.6), as ações do trabalhador já eram mostradas na *UI*. Ao clicar num dos botões do *UI* vai ser chamada a função `CreateBuilding` (ver excerto 3.13). Nesta função vai ser criada uma versão temporária do edifício escolhido na posição do rato, para que o jogador possa escolher onde o quer construir. Esta versão temporária vai também ser transparente, para mostrar ao jogador que esta não é a versão final do seu edifício.

Também são desligados todos os `colliders` para garantir que não existem colisões com qualquer um dos objetos do terreno (função `SetColliders`, que recebe um booleano para definir todos os `coliders` do edifício a `true` ou a `false`). São também criados dois novos materiais, um com a cor verde e um com a cor vermelha, para mostrar ao jogador onde é que ele pode e não pode construir o edifício. Estes materiais são usados na função `SetTransparentMaterial`, que vai guardar o tipo de material anterior à alteração (para que seja possível voltar aos materiais originais) e mudar o material para verde/vermelho.

```
public void CreateBuilding(string buildingName, Vector3 buildPoint, Unit
    creator, Rect playingArea) {
    GameObject newBuilding = (GameObject)Instantiate(ResourceManager.
        GetBuilding(buildingName), buildPoint, new Quaternion());
    tempBuilding = newBuilding.GetComponent< Building >();
    if (tempBuilding) {
        tempCreator = creator;
        findingPlacement = true;
        tempBuilding.SetTransparentMaterial(notAllowedMaterial, true);
        tempBuilding.SetColliders(false);
        tempBuilding.SetPlayingArea(playingArea);
    } else Destroy(newBuilding);
}
```

Excerto de Código 3.13: Função que cria um edifício temporário.

Foi feita uma atualização à função `MouseHover`, em que se a variável `findingPlacement` estivesse a `true` (variável que indica se o jogador está a construir um novo edifício ou não), o edifício temporário iria atualizar a sua posição para a posição do rato.
Faltava apenas verificar se a posição do edifício é legal para ser construída, construí-la e cancelar o posicionamento do edifício temporário. Foi criada a função `CanPlaceBuilding`, que inicializa a variável do tipo booleana `canPlace` a `true`. Nesta função vai ser verificado se o edifício pode ser construído na posição atual, verificando se existem colisões dos cantos do edifício com outros objetos (ver excerto 3.14). No fim, é devolvido o valor da variável `canPlace`, que vai ser `false` caso exista colisões, ou `true` caso contrário, podendo deste modo o edifício ser construído.

```
bool canPlace = true;
foreach(Vector3 corner in corners) {
  GameObject hitObject = WorkManager.FindHitObject(corner);
  if(hitObject && hitObject.name != "Ground") {
     WorldObject worldObject = hitObject.transform.parent.GetComponent<
         WorldObject>();
    if(worldObject && placeBounds.Intersects(worldObject.GetSelectionBounds()))
         canPlace = false;
  }
}
return canPlace;
```

Excerto de Código 3.14: Função para verificar as colisões entre edifícios.

Sabendo que é possível ou não construir o edifício naquela posição, basta apenas construí-lo. Se for possível construir o edifício naquela posição, na função `LeftMouseClick` é chamada a função `StartConstruction`. Nesta função vai ser atribuído o edifício ao jogador e vão ser reiniciados os `colliders` do edifício. Vai ser também instruído ao trabalhador que a construção foi iniciada, atribuindo-lhe esta mesma construção.
Este edifício é inicializado com zero de vida e com a variável `needsBuilding` a `true`. Na função `OnGUI` do *script HUD* é também chamada a função `DrawBuildingProgress`, que vai desenhar

uma barra de construção.
Foi criada uma função com o nome de `Construct` para que o trabalhador pudesse construir o edifício. Esta função serve para aumentar os pontos de vida do edifício, e, caso os pontos de vida atuais sejam superior ao máximo de vida (que significa que ele está completamente construído), recuperar as cores originais.

Foram adicionadas novas verificações à função `RightMouseClick` do *script Worker* do trabalhador. Caso seja carregado com o botão direito em cima de um edifício que esteja a ser construído, o trabalhador vai até ao edifício e vai começar a construí-lo. Foi também adicionada a condição em que se o jogador estiver à procura de um local para o edifício e carregar com o botão direito em qualquer sítio, a construção é cancelada e o edifício temporário é cancelado (destruído na realidade).

No entanto, existia um problema neste momento: caso um jogador criasse um edifício temporário e o trabalhador nunca fosse até ele para ser construído (fosse mandado para ir para outro sítio), o edifício temporário iria ficar lá. Isto não deveria acontecer, pois o facto de o jogador mandar o trabalhador para algum lado é sinónimo de querer cancelar o edifício (isto não acontece se o edifício já tiver mais que um de vida). Outro caso era quando o jogador mandava o trabalhador fazer a construção A e depois logo de seguida a construção B. Ele ia diretamente para a construção B e a construção A ficava para sempre inacabada.

Para resolver este problema foi necessário alterar a função `StartConstruction`, na qual, quando um jogador mandar um trabalhador fazer uma construção, se ele já tiver uma construção pendente, ele vai cancelar a antiga e prosseguir com a nova. Para implementar o cancelamento da construção, quando se carrega com o botão direito no chão, faz-se apenas uma chamada à função `CancelWorkerJob`. Se ela devolver `true`, a construção é cancelada, caso contrário não o é.
A função `CancelWorkerJob` devolve `true` se o edifício tiver zero de vida, ou `false` caso contrário. Isto porque o jogador pode querer que o seu trabalhador pare uma construção para ir fazer outra, mas já tinha gastado tempo a fazer aquela construção.

Nos jogos de estratégia é possível criar uma lista de espera na construção de edifícios. Se o jogador disser ao seu trabalhador para construir um edifício enquanto tem a tecla *"SHIFT"* premida, os outros edifícios são colocados na lista de espera. Assim que o construtor acabe a construção do primeiro, ele vai construir o edifício que está a seguir na lista de espera.
Para implementar esta funcionalidade foi criada uma variável do tipo lista com o nome de `FutureProjects`, que contém todos os projetos do trabalhador. Se o trabalhador não tiver nenhum projeto atual, ele vai verificar a lista de projetos futuros, verificar o edifício na primeira posição, removê-lo da lista e defini-lo como seu projeto atual. Ele faz esta atribuição na sua função `Update`.

Na função `setBuilding` também é feita uma alteração, para que caso o trabalhador já tenha um projeto atual, este novo é adicionado na lista de projetos futuros.
A implementação da tecla *shift* foi feita na função `Update` do `script UserInput`. Nesta é verificada se a tecla *shift* está premida. Se sim, é chamada a função `StartConstruction` com um argumento do tipo booleano a `true`. Caso contrário, é chamada a mesma função com um argumento do tipo booleano a `false`. Faltava apenas alterar a função `StartConstruction`.

Resumidamente, se a tecla *Shift* não estiver premida, é feita construção normal, que tinha feito até agora. Caso contrário, vai ser criado aquele edifício e questionado ao jogador onde quer colocar o próximo edifício temporário e assim sucessivamente.

Um problema que foi encontrado foi quando se começou a testar a construção de edifícios. O teste que verificava se era possível construir uma unidade em determinado sítio (função `CanPlaceBuilding`), apenas verificava os cantos e acontecia quando se tentava construir edifícios muito grandes e muito pequenos. Para isso foi feito uma atualização à função `CanPlaceBuilding`, que vai fazer novas verificações. Não só foram adicionados mais pontos para fazer a verificação (pontos intermédios), como também foram feitas verificações de se o `collider` do edifício colidia com o `collider` da caixa de seleção, significando que está demasiado próximo e por isso não pode ser construído. Estas colisões só vão ser verificadas com edifícios que estejam dentro da *fog of war* (secção 3.19), caso contrário, seria possível descobrir onde é que os inimigos tinham as suas bases.

Aproveitando o facto de se estar a trabalhar com edifícios, foram logo implementados os *upgrades*. É um sistema que é muito visto em vários tipos de jogos, no qual um jogador pode aumentar o nível do seu edifício para garantir melhorias, não só mais pontos de vida, mas também novas funcionalidades. Para a melhoria do edifício foi criada uma nova ação e um novo botão no *UI*. A melhoria do edifício custa cada vez mais recursos e vai demorando cada vez mais tempo. Um jogador só vai poder fazer uma melhoria se o edifício não estiver a sofrer nenhuma melhoria, nem se lá estiver a ser treinada nenhuma unidade.

Já que tinha sido criado um novo botão para a melhoria do edifício, foi também implementado um novo botão para cancelar a ação do edifício. Caso o edifício estivesse a fazer uma melhoria, a melhoria era cancelada. Caso fosse o treino de uma unidade, era o treino que era cancelado. Caso estivesse a ser construído, a construção seria cancelada. Este botão apenas tinha de verificar qual era a ação que estava a decorrer no edifício e cancelá-la. Ao cancelar a ação, o custo dos recursos é devolvido ao jogador.

3.9 Atacar Inimigos

Para atacar um inimigo é preciso ter uma unidade/edifício do próprio jogador selecionado e carregar com o botão direito em cima de uma unidade/edifício de outro jogador. Para fazer isso, foi preciso atualizar a função `RightMouseClick`, adicionando uma verificação de que se o nome do jogador, que é dono da unidade selecionada, for diferente do jogador dono da unidade, em que foi clicado com o botão direito, e ela puder atacar, vai chamar a função `BeginAttack`, que vai tratar de tudo relativamente ao ataque.
Nesta função (ver excerto 3.15) é verificado se o alvo está a uma distancia razoável, através da função `TargetInRange`. Se esta devolver `true`, significa que o alvo está perto e pode começar a atacar. Caso contrário, a unidade vai mover-se para perto do alvo. As unidades que podem atacar têm uma variável com o nome de *WeaponRange*, que indica o alcance da unidade. Esta distância é calculada com a função `FindNearestAttackPosition`.

```
protected virtual void BeginAttack(WorldObject target) {
    this.target = target;
    if(TargetInRange()) {
        attacking = true;
        PerformAttack();
    } else
      AdjustPosition();
}
```

Excerto de Código 3.15: Função que verifica se uma unidade/edifício pode atacar.

Nesta função é enviado como parâmetro o alvo (*WorldObject*). A partir disto sabe-se a posição dele e pode-se calcular o vetor entre a unidade/edifício do jogador e o alvo. Este vetor dá a distância ao alvo, pela qual se vai subtrair o alcance da unidade. A unidade de ataque vai colocar-se a 90% do alcance da sua arma, para que, caso o alvo se mova, não seja preciso ajustar a posição tão rapidamente (no caso de ser uma torre de defesa, como ela não se pode mexer, não poderá atacar).
Assim que a unidade está perto do alvo é possível começar a atacar. Para isso, foi usada a função `PerformAttack`. Nesta função vai ser verificado se o alvo está a uma distância certa e se não estiver vai mover a unidade em questão. Caso a unidade/edifício não esteja virada para o alvo, esta vai rodar sobre si para que esteja de frente para o alvo. Caso esteja pronta a disparar, ela vai fazê-lo. Foi apenas preciso adicionar uma condição extra para verificar se o alvo existe (ele vai ser destruído se a vida dele chegar a zero).

A unidade/edifício tem um tempo de recarregamento para que não esteja sempre a disparar. A função `ReadyToFire` usa uma variável auxiliar que é incrementada continuamente (`Time.deltaTime`, que é incrementado na função `Update`). Se for superior ao tempo de recarregamento da unidade/edifício, devolve `true` e o valor da variável é colocada a zero.

Falta apenas implementar o efeito visual do ataque (um míssil, por exemplo) para que o jogador saiba quando é que a sua unidade está a atacar ou a ser atacada. Para isso, no *script Tank*, foi criada a função `UseWeapon`. Nesta função vai ser instanciado um *Projectile* e vai ser lhe definido o alvo, que é o alvo da unidade (ver excerto 3.16).

```
protected override void UseWeapon () {
    base.UseWeapon();
    Vector3 spawnPoint = transform.position;
    spawnPoint.x += (2.1f * transform.forward.x);
    spawnPoint.y += 1.4f;
    spawnPoint.z += (2.1f * transform.forward.z);
    GameObject gameObject = (GameObject)Instantiate(ResourceManager.
        GetWorldObject("TankProjectile"), spawnPoint, transform.rotation);
    Projectile projectile = gameObject.GetComponentInChildren< Projectile >();
    projectile.SetRange(0.9f * weaponRange);
    projectile.SetTarget(target);
}
```

Excerto de Código 3.16: Função que instancía um projétil.

Foi criada uma nova classe chamada *Projectile*, que tem como variáveis velocidade, dano, distância e um alvo. A função `Update` é a que vai fazer toda a magia. Se este míssil colidir contra alguma unidade/edifício inimigo, vai causar dano a esse objeto (através da função `TakeDamage`) e destruir-se. Caso contrário, vai continuar a mover-se em linha reta.

A função `TakeDamage` no *script WorldObjects* serve para reduzir os seus pontos de vida num determinado valor. Caso a vida desse *WorldObjects* chegue a zero, ele destrói-se a si próprio.
Para finalizar, foi criado um *GameObject* vazio com o nome de *TankProjectile* e dentro dele foi criada uma cápsula, que vai ser o míssil. Foi adicionado o *script TankProjectile* como componente deste novo objeto e foi criado um *Prefab*, para que ele possa ser reutilizado várias vezes ao longo do jogo.
Um *Prefab* é um *template* de um objeto normal. Imaginando que um jogo tem cem unidades iguais: se por acaso fosse preciso mudar alguma coisa numa das unidades, seria preciso fazer essa alteração cem vezes; se tivesse sido criado um *prefab* dessa unidade, bastava fazer a alteração no *prefab* para que a alteração fosse replicada nessas cem unidades e nas seguintes.
Neste momento, existia um *bug*: quando era dada uma ordem ao tanque para se mover, ele não só se movia, como também rodava (devido à inteligência artificial, como vai ser explicado no capítulo 3.14). Para resolver este problema, bastou na função `Update` colocar a verificação de apenas rodar se ele não se estivesse a mover.
Para ter a certeza que isto não voltava a acontecer, na função `ShouldMakeDecision` foi adicionada uma condição que se a variável `moving` fosse `true`, ele não iria fazer nada.

A implementação do ataque nas torres de defesa também foi bastante simples, visto que era igual ao que tinha sido feito para o *script Tank*. Foram então criados dois *scripts* novos, *TurretProjectile*e e *Turret*. O *TurretProjectile* vai herdar do *script Projectile* e não vai ter nenhuma função nem variável. O *script Turret* vai herdar do *script Building* e vai ter as funções iguais ao *script Tank*. A única diferença é que vai ser instanciado um projétil do tipo *TurretProjectile* e não do tipo *TankProjectile*.

3.10 *RallyPoint*

O *rallypoint* (ou ponto de encontro em português) é o local no terreno para onde as unidades vão quando acabam de ser treinadas. O jogador pode colocar este ponto em qualquer sítio do terreno, seja num recurso, numa unidade/edifício, ou no chão, e a unidade ao completar o seu treino deve mover-se até lá.
Para criar um marcador visual para o ponto de encontro foram usados dois cubos e foi aplicado um material sobre eles, de forma a criar uma bandeira. Depois foi criado um *Prefab* e inserido como filho do *GameObject Player*.
Foi criado um *script* com o nome de *RallyPoint*, onde foram adicionadas as funções para mostrar e para esconder a bandeira do ponto de encontro (função `Enable` e `Disable`).
Visto que só faz sentido a bandeira ser mostrada se existir algum edifício selecionado, foi alterado o código da função `SetSelection` do *script Building*, em que a bandeira é mostrada quando o edifício é selecionado e escondido quando é desselecionado. Assim, já é possível mostrar e esconder a bandeira. Faltava apenas permitir ao jogador trocar a posição do *rallypoint*.

No *script* dos três edifícios foi colocado como ação o *RallyPoint*. Assim, aparece um botão na interface do utilizador para ele poder escolher o sítio onde o ponto de encontro será colocado. Ao carregar nesse botão, a imagem do cursor do jogador é mudado para uma bandeira (ver imagem 3.15) e ao carregar num local do mapa, o ponto de encontro é alterado. Esta alteração é efetuada através da função `LeftMouseClick` do *Script UserInput*, que tem lá uma condição para quando o estado do cursor for do tipo *RallyPoint*.

Figura 3.15: Cursor que é mostrado ao jogador quando ele deseja alterar a posição do rallyPoint.

Para além disso, se o jogador tiver um edifício selecionado e carregar com o botão direito num sítio do terreno, o ponto de encontro deverá ser alterado para esse local. Para permitir isso foi preciso fazer `override` da função `RightMouseClick` no *script Building* e fazer uma chamada à função `SetRallyPoint`. Esta função foi criada no *Script Building* e serve para alterar a posição do ponto de encontro (ver excerto 3.17).

```
public void SetRallyPoint(Vector3 position) {
    rallyPoint = position;
    if(player && player.human && currentlySelected) {
        RallyPoint flag = player.GetComponentInChildren< RallyPoint >();
        if(flag) flag.transform.localPosition = rallyPoint;
    }
}
```

Excerto de Código 3.17: Função que vai alterar a posição do ponto de encontro.

No entanto, há edifícios que não vão ter ponto de encontro, visto não irem criar unidades. Um exemplo deste tipo de edifícios é a torre de defesa. Então, para além de não ter a `action SpawnPoint` (ou seja, não aparece um botão para mudar o ponto de encontro), na função `Update` é chamada a função `Disable` de forma a que a bandeira não apareça quando este edifício estiver selecionado.

Uma opção que também foi importante adicionar no jogo foi a possibilidade de o jogador colocar o ponto de encontro nos recursos, para que quando os recoletores forem criados possam ir recolher o recurso automaticamente.
Para isso bastou alterar a função `RightMouseclick` no *script Building*, para permitir o jogador colocar o ponto de encontro num recurso. Foi preciso também alterar a função `StartMove` no *script Harvester*, para verificar que, caso o ponto de encontro esteja num recurso, ele mover-se-á até ao recurso e começará a recolhê-lo.

No entanto, independentemente do sítio do ponto de encontro, a unidade aparecia sempre do mesmo lado do edifício. Isto não acontece nos jogos de estratégia, visto que a unidade é criada do lado do edifício mais próximo do ponto de encontro. Para isso, foi alterada a função `SetRallyPoint`, de forma a verificar o ponto do edifício que estava mais perto do ponto de encontro (esta alteração vai calcular um ponto entre a posição do edifício e o ponto de encontro, a uma distância definida previamente).

3.11 Recursos

Tal como foi dito na secção de Game Design (3.2) este jogo tem dois tipos de recursos:

- Madeira - Para construir e fazer melhorias aos edifícios e produzir novas unidades;
- População - Permitir fazer novas unidades.

Para isso foram criadas três variáveis: `StartWood`, `startPopulation`, `PopulationLimit`, para definir a população inicial, a madeira inicial e definir o máximo da população durante todo o jogo (por exemplo, se não quisermos que a população passe de duzentos, como acontece no *Starcraft*). Não existe uma variável para o máximo de madeira, pois não existe um valor máximo que o jogador possa ter.

O primeiro passo para a implementação foi a criação dos recursos que um jogador pode colecionar (madeira, neste caso). Foi criado um *script* com o nome *Resources*, que herda o *script WorldObjects*. Este Recurso vai ter um tipo, a capacidade de recursos e a quantidade de recursos que ainda tem disponível.
Embora neste jogo só exista um tipo de recursos, no *Script Enum* foi criado uma variável que vai conter todos os tipos de recursos que podem existir, isto para que, se num futuro, quiser ser implementado outro tipo de recurso, seja mais fácil. Os recoletores quando são criados, o tipo de recurso que vão recolher é do tipo *"Unknown"*. Foi feito desta forma para que nenhum jogador consiga alguma vantagem por estar logo definido como madeira. Para além disso, a quantidade não pode ser inferior a zero.

O único tipo de recurso coletável é madeira. Para isso, foi criado um *GameObject* com o nome *"tree"* com dois *GameObject filho*, um cilindro a simular o tronco da árvore e um cubo a simular a copa. Uma das componentes da árvore é um *script* com o nome de *WoodDeposit* (ver excerto 3.18). Os *GameObject filho* tem uma componente, um *script* com o nome de *Wood* (não tem qualquer variável ou função, ver excerto 3.19).

```
public class WoodDeposit : Resource {
  protected override void Start()
  {
    base.Start();
    resourceType = ResourceType.Wood;
  }
}
```

Excerto de Código 3.18: *Script WoodDeposit*.

```
public class Wood : MonoBehaviour {
}
```

Excerto de Código 3.19: *Script Wood*.

A unidade que vai recolher os recursos é o recoletor. Esta unidade é um objeto composto por umas pás, pneus e uma caixa (tudo isto ficcional, no futuro serão trocados os modelos). Esta unidade tem um *script* com o nome de *Harvester*, que também tem herança do *script Unit*.

Neste *script* existem quatro variáveis globais: uma que vai dizer que tipo de recurso é que ele está a recolher; outra que indica a capacidade máxima que ele pode recolher; outra que indica se está a recolher recursos ou a entregar e uma que indica a quantidade de recursos que tem consigo.
Quando o jogador tem um recoletor selecionado e o rato está em cima de um recurso que não esteja vazio, a imagem do cursor vai mudar para o estado de recolher recursos. Se o jogador carregar com o botão direito em cima dele, o trabalhador vai começar a recolher recursos. A função `StartHarvest` (ver excerto 3.20) vai fazer com que o recoletor se mova até ao recurso e comece a recolhê-lo. Caso o jogador carregue com o botão direito do rato noutro sítio, o recoletor vai parar de recolher recursos (função `StopHarvest`).

```
private void StartHarvest(Resource resource) {
    resourceDeposit = resource;
    StartMove(resource.transform.position, resource.gameObject);
    //we can only collect one resource at a time, other resources are lost
    if(harvestType == ResourceType.Unknown || harvestType != resource.
        GetResourceType()) {
        harvestType = resource.GetResourceType();
        currentLoad = 0.0f;
    }
    harvesting = true;
    emptying = false;
}
```

Excerto de Código 3.20: Função `StartHarvest`.

No *script harvester* também foi atualizada a função `Update`. O recoletor vai colecionar recursos se a variável `harvest` estiver a `true`. Se a variável `emptying` estiver a `true`, ele vai mover-se até ao centro de depósitos e entregar os recursos. A função `Update` chama a função `Collect` e a função `Deposit`, de acordo com a variável que estiver a `true` (ver excerto 3.21). É impossível ter as duas variáveis a `true` ao mesmo tempo.

```
private void Collect() {
    float collect = collectionAmount * Time.deltaTime;
    //make sure that the harvester cannot collect more than it can carry
    if(currentLoad + collect > capacity) collect = capacity - currentLoad;
    resourceDeposit.Remove(collect);
    currentLoad += collect;
}
private void Deposit() {
    currentDeposit += depositAmount * Time.deltaTime;
    int deposit = Mathf.FloorToInt(currentDeposit);
    if(deposit >= 1) {
        if(deposit > currentLoad) deposit = Mathf.FloorToInt(currentLoad);
        currentDeposit -= deposit;
        currentLoad -= deposit;
        ResourceType depositType = harvestType;
        player.AddResource(depositType, deposit);
    }
}
```

Excerto de Código 3.21: Função para recolher e depositar recursos.

Após isso, foi necessário fazer um ajuste para não acontecer nada se for mandado ao recoletor recolher madeira a uma árvore que já não tem madeira, ou quando o rato está em cima dessa mesma árvore. Estas alterações foram feitas no *script Unit*, na função `RightMouseClick` e `SetHoverState`. Também foi implementado para que as unidades se possam mover até um sítio onde exista uma árvore, mas que já não tenha recursos.

3.12 Áudio

Para a implementação do som foi criada a pasta *Audio* na pasta *Assets*. Nesta pasta foram adicionados todos os sons do jogo. Também foi criada na pasta *Scripts* uma pasta com o nome de *Audio*. Dentro desta pasta foi criado um *script* com o nome de *AudioElement*.

Este *script* vai servir como um *Audio Manager* para todos os objetos do jogo. Existe uma série de razões para o fazer deste motivo. Uma delas é que o *Unity* apenas permite que uma componente do tipo *AudioClip* seja adicionado a um objeto. Isto foi problemático, visto que vários objetos do jogo vão ter mais que um som.
A ideia chave é que quando for criado um *AudioElement*, também seja criado um novo *GameObject* vazio. Este objeto vazio vai conter uma série de objetos filhos e cada um deles vai conter um *SoundClip*. Isto é necessário para que se possa tocar vários sons ao mesmo tempo e poder alterar os volumes de cada um deles, sem que os outros sejam afetados.

Depois, estes *gameObjects* são anexados às unidades/edifícios. Isto significa que se o objeto se mover pelo mapa, o objeto de *AudioElement* também se vai mover pelo mapa, pois vai ficar sempre na mesma posição que o objeto pai. Isto é importante quando no futuro do projeto se quiser implementar som *3D*.

Tendo os objetos auxiliares criados, faltava apenas adicionar os sons pretendidos aos *WorldObjects*. No *script WorldObject* foram adicionadas as seguintes variáveis (ver excerto 3.22).

```
public AudioClip attackSound, selectSound, useWeaponSound;
public float attackVolume = 1.0f, selectVolume = 1.0f, useWeaponVolume = 1.0f;
protected AudioElement audioElement;
```

Excerto de Código 3.22: Variáveis de som do *WorldObjects*.

No fim da função `Start` foi inicializado o som. Foi criada uma lista de *AudioClips*, que contém os três sons *(attack, select e useweapon)* e uma lista de variáveis do tipo `float`, que vão guardar o volume de cada um dos sons. Após isso, basta apenas tocar os sons. Por exemplo, na função `SetSelecion`, foi adicionado código para fazer com que o som de ser selecionado seja tocado (ver excerto 3.23). Na função `BeginAttack` é tocado o som de atacar e na função `UseWeapon` o som de disparar.

```
if(audioElement != null)
  audioElement.Play(selectSound);
```

Excerto de Código 3.23: Código que vai fazer com que o som de select seja tocado.

Para as unidades foram criados alguns sons auxiliares, como, por exemplo, o som das unidades a moverem-se. Para o recoletor foram adicionados dois sons: um que vai ser tocado quando for iniciada a recolha de recursos e outro quando a unidade entregar os recursos. Para o trabalhador foi adicionado um som, que vai ser tocado quando acabar de construir o edifício. Para colocar tudo a funcionar, falta apenas arrastar os sons da pasta para os *Prefabs* das unidades.
Visto o jogo ser multi jogador e não haver pausas, não foi preciso implementar a condição de que só pode tocar um som se a velocidade do jogo for superior a zero, visto que a velocidade do jogo vai ser sempre superior a zero.

3.13 Condições de Vitória

Tudo o que tem um início tem de ter um fim. Este jogo não foi exceção e as condições de vitória e derrota já foram definidas no *game design*, no subcapítulo 3.2.
Para começar a implementar estas condições, foi criada uma condição simples, apenas para verificar se estava tudo a funcionar como era suposto. Inicialmente foi apenas adicionada a condição em que um jogador ganhava se angariasse vinte unidades de madeira.

Foi criado um *script* com o nome de *VictoryCondition*. Este tem uma variável, que é um *array* com todos os jogadores do jogo e que contêm as condições de vitória para cada jogador (isto é, pode-se definir condições diferentes para cada um dos jogadores, o que no futuro vai ser útil, porque a condição de vitória dos humanos é diferente da dos vampiros).
No entanto, para isto funcionar foi criado o *script GameManager* onde vai ser gerido tudo o que acontece no jogo. Este novo *script* foi adicionado a um objeto vazio e na função `Update` vai ser verificado se algum jogador ganhou e, caso essa resposta seja afirmativa, acabar o jogo (ver excerto 3.24).

```
void Update() {
  if(victoryConditions != null) {
    foreach(VictoryCondition victoryCondition in victoryConditions) {
      if(victoryCondition.GameFinished()) {
         Time.timeScale = 0.0f;
        Screen.showCursor = true;
        hud.enabled = false;
      }
    }
  }
}
```

Excerto de Código 3.24: *Script* que vai verificar se alguma das condições de vitória está concluída.

Para acabar de implementar as condições de vitória faltava criar o *script* da condição de vitória. Neste caso foi criado um *script* com o nome de *AccumulativeMoney*, com uma função chamada `PlayerMeetsConditions`, que vai devolver `true` se o jogador existir, não estiver morto e o número de recursos for superior ao limite definido(ver excerto 3.25).

```
public class AccumulateMoney : VictoryCondition
{
  public int amount = 20;
  private ResourceType type = ResourceType.Wood;
  public override bool PlayerMeetsConditions(Player player)
  {
    return player && !player.IsDead() && player.GetResourceAmount(type) >=
        amount;
  }
}
```

Excerto de Código 3.25: Condição que vai acabar o jogo se algum jogador tiver vinte de madeira.

Faltava apenas criar a função `isDead` e `GetResourceAmount` no *script Player*, funções que vão devolver se o jogador está morto e a quantidade de recursos de um jogador, respetivamente. O jogador vai estar morto se não tiver o trabalhador.

Bastava arrastar o *script AccumulateMoney* para o objeto *VictoryConditions* para que tudo ficasse a funcionar.

Para implementar as condições de vitória para que uma equipa ganhe quando outra equipa não tiver trabalhadores, bastou criar um *script* que devolve `true` caso todos os jogadores de uma equipa estejam mortos (verificado usando a função `isDead`).

3.14 Inteligência Artificial

Para finalizar a versão inicial do jogo foi preciso adicionar inteligência artificial. Um exemplo de inteligência artificial que as unidades/edifícios devem ter é atacar sozinhas quando tiverem um inimigo por perto. Para começar, foi preciso atualizar a função `Update` do *script WorldObject*, sendo adicionadas as funções `ShouldMakeDecision` e `DecideWhatToDo`.

A função `ShouldMakeDecision` vai fazer com que só sejam tomadas decisões de *X* em *X* segundos. Tem a variável `timeSinceLastDecision`, cujo valor vai ser incrementado por `Time.deltaTime`. Caso esta variável seja superior ao valor definido (por exemplo, dois segundos), ele vai devolver `true` e vai então ser chamada a função `DecideWhatToDo`, em que vai ser definido o que fazer. Vai ser verificada a sua posição e ver quais são os objetos que estão à sua volta, através da função `FindNearbyObjects`. Nesta função vai ser usado o motor de física do *Unity*, onde são procurados `colliders` numa área esférica (*OverlapSphere*), que estejam a uma determinada distância e vai adicioná-los a uma lista.

A base da inteligência artificial está feita. Falta, no entanto, ser mais específico, ou seja, o que é que cada unidade deverá fazer. Os recursos nunca vão tomar decisões sozinhos, por isso, no *script* `WorldObjects` foi feito um *override* a função `ShouldMakeDecision`, que vai devolver sempre `false`.
Outro pormenor importante é que as unidades não devem tomar decisões caso estejam a movimentar-se para algum local ou a rodar. Nesse caso, a função `ShouldMakeDecision` do *script Unit* vai devolver `false` caso a variável `moving` ou a variável `rotating` estejam a `true`. No trabalhador, para além de ser afetado pelas condições anunciadas anteriormente, se estiver a construír algum edifício, vai devolver `false`. No recoletor é igual e não deverá tomar ações caso esteja a recolher recursos ou a depositá-los.
Para começar a programação destas ações, foi verificado o caso das unidades que podem atacar. Caso a unidade possa atacar, vai ser verificado quais são os *WorldObjects* que estão próximos dela. Se existir algum, vai verificar o mais próximo dele e vai-se movimentar para perto dele, para o começar a atacar (ver excerto 3.26).

```
if(CanAttack()) {
    List< WorldObject > enemyObjects = new List< WorldObject >();
    foreach(WorldObject nearbyObject in nearbyObjects) {
        Resource resource = nearbyObject.GetComponent< Resource >();
        if(resource) continue;
        if(nearbyObject.GetPlayer() != player) enemyObjects.Add(nearbyObject);
    }
    WorldObject closestObject = FindNearestWorldObjectInListToPosition(
        enemyObjects, currentPosition);
    if(closestObject) BeginAttack(closestObject);
}
```

Excerto de Código 3.26: Função que verifica os objetos num raio à sua volta e ataca o que estiver próximo.

Para os trabalhadores o código vai ser muito semelhante, apenas com a diferença que ele não vai verificar qual o *WorldObject* mais perto para atacar, mas os edifícios mais perto num

determinado raio e que estejam em construção.
Com auxílio da mesma função `FindNearestWorldObjectInListToPosition` vai ser verificar qual o edifício mais perto e mover-se até ele, para o começar a construir.
O mesmo acontece com o recoletor. Ele vai verificar quais os recursos perto dele num determinado raio, verificar qual é o mais próximo e começar a recolher esse recurso.
Acabando isto, a inteligência artificial está completa. Visto ser um jogo multi jogador e que todos os jogadores vão ser humanos, não foi preciso ter uma componente avançada de inteligência artificial, pois não existe a possibilidade de jogar contra o computador.

3.15 Terreno

O terreno nos jogos é muito importante. Não só permite aos jogadores escolherem diferentes táticas em diferentes partidas, mas também traz vantagens e desvantagens aos jogadores.
Neste caso, o mapa do jogo foi baseado no mapa *Vampirism Galactic* do *Starcraft 2* (ver imagem 3.16). É composto por quatro níveis de altura. A cada um destes níveis foi atribuído uma cor (textura) para que inicialmente fosse mais fácil ser diferenciado. Posteriormente, basta substituir a textura da cor por uma outra textura e é alterada no mapa todo, sendo mais fácil.

- 0 - Verde - Altura padrão. Todos os jogadores começam no centro do mapa no qual a altura é zero;
- 10 - Azul - Primeiro nível de altura;
- 25 - Vermelho - Segundo nível de altura;
- 40 - Laranja - Terceiro e último nível de altura.

As seguintes imagens ilustram o mapa original e a divisão por alturas (ver imagem 3.17). Tudo o resto que não está marcado tem altura zero.

Figura 3.16: Mapa original do *Starcraft 2*.

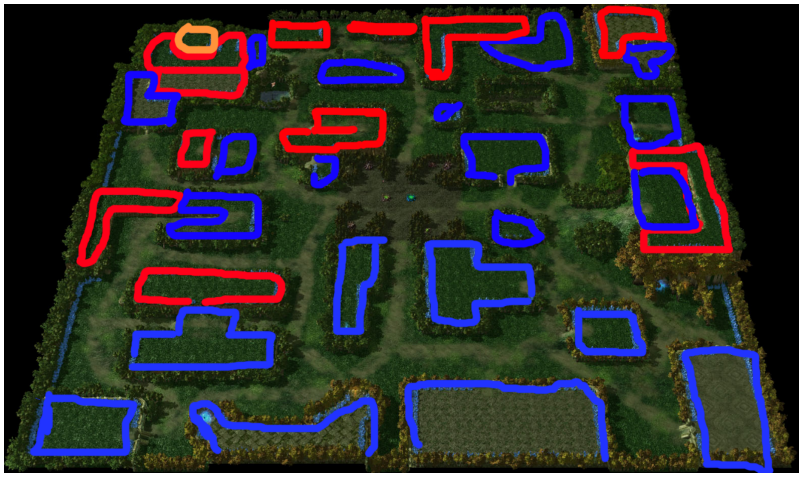

Figura 3.17: Divisão das plataformas do mapa por alturas.

O terreno foi construído usando a ferramenta *Terrain* do *Unity*. Foi criado um terreno *(GameObject -> Terrain)*. Após isto, é mostrado um terreno totalmente plano e uma série de ferramentas disponíveis para trabalhar neste terreno (ver imagem 3.18). Podem ser definidas as opções do terreno, o material para o mesmo, assim como ligar/desligar o `collider` com as árvores. Neste caso, como as árvores são feitas à parte e não são feitas no *terrain*, é indiferente.

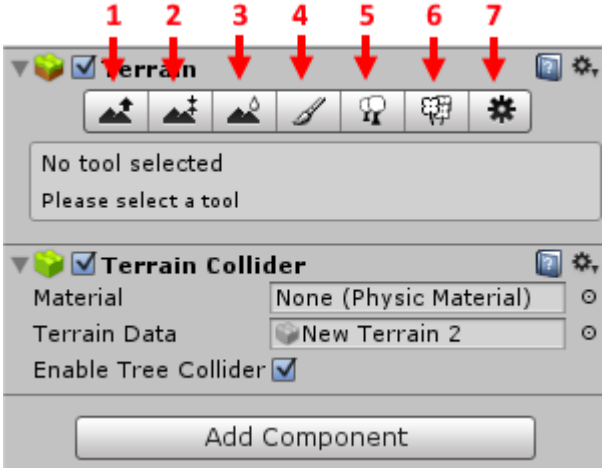

Figura 3.18: Ferramentas de construção de terrenos no *Unity*.

Para além disto, tal como mostra a imagem 3.18, o *Unity* oferece uma série de ferramentas para que o terreno possa ser trabalho, tais como:

1. Levantar/baixar terreno sem um valor definido;
2. Definir uma altura que se vai elevar o terreno;
3. Alisar altura;
4. Pintar texturas;
5. Inserir/remover árvores;
6. Adicionar relva;
7. Definições do terreno.

Todas estas opções, à exceção das definições, são inseridas no terreno com base em *brushes*. O *Unity* oferece uma série de *brushes*, mas podem ser adicionadas mais de acordo com o interesse do utilizador (basta adicionar as imagens com o nome de "*Brush_*" seguido do número da *brush*, começando em zero [Far13]).

As definições do terreno permitem configurar uma série de opções, desde o terreno em si (sombra, grossura, distância, material), assim como árvores e relva, opções do vento e resolução do mapa. O terreno para este jogo foi definido com uma largura e um comprimento de quinhentos *pixeis*, o que fez com que a resolução do *heightmap* fosse quinhentos e treze *pixeis* (é sempre uma potência de dois, somado de um).

Inicialmente foi usada a ferramenta número dois para os três tipos de alturas definidos previamente (apenas três, visto que tudo o resto vai estar à altura zero). Começou-se por delinear as plataformas do jogo de uma forma grosseira e sem muita precisão. Assim que estas plataformas estavam todas definidas, foi feito o *download* do *heightmap* em *Raw* (ver imagem 3.19), para que pudesse ser trabalhado com uma ferramenta externa, de forma a garantir que todas as plataformas se encontravam a uma altura correta e que não existiam falhas no mapa(fendas)[Uni15b].

Figura 3.19: *Heightmap* do terreno criado.

Um *heightmap* é uma imagem bidimensional utilizada para guardar valores de elevação de uma superfície. Estas imagens são sempre representadas numa escala de cinzento, onde o preto representa a altura mais baixa (zero) e o branco a mais alta (neste caso a altura máxima é seiscentas unidades, que foi o que ficou definido das definições do terreno).

Tal como se pode verificar na imagem 3.19, em algumas plataformas existia uma variação de cor, o que não podia acontecer (visto que as plataformas são completamente planas). Para fazer as rampas foi preciso criar uma variação gradual entre a cor das duas plataformas. Após criar isto, bastou alterar o nome do objeto para *Ground*, de forma a que os movimentos com o botão direito no terreno programados previamente continuassem a funcionar.

3.16 *Navigation Meshes*

Quando é dada a uma unidade a instrução de se mover para um lugar, é preciso primeiro verificar se é possível ir até lá e qual é o caminho mais curto. A forma de implementar isto foi através do sistema de navegação do *Unity* [Car14].

O *Unity* tem um sistema de navegação que possibilita o movimento das personagens pelo terreno de jogo de forma inteligente. Este sistema de navegação usa *Navigation Meshes (NavMeshes)*. Estas *meshes* são criadas automaticamente a partir da geometria do cenário.

O *Unity* também oferece *Navmesh Agents* e *NavMesh Obstacles*, que vão interagir com a *NavMesh*. O *Navmesh Agent* é aplicado às unidades que se movem pelo terreno e é responsável por fazer mover a unidade pela cena, encontrando caminhos na *NavMesh*. O *Navmesh Obstacle* oferece uma forma dinâmica de inserir um obstáculo na *NavMesh*, fazendo com que os agentes não consigam passar por ele (no caso deste jogo, vão ser os edifícios).

A *NavMesh* vai dividir o terreno em triângulos e fazer a ligação entre eles, permitindo às unidades evitarem os obstáculos e verificarem se existe algum caminho possível até à posição desejada. Para aplicar uma *NavMesh* ao terreno foi preciso ir a *Window* e depois a *Navigation* (ver imagem 3.20).

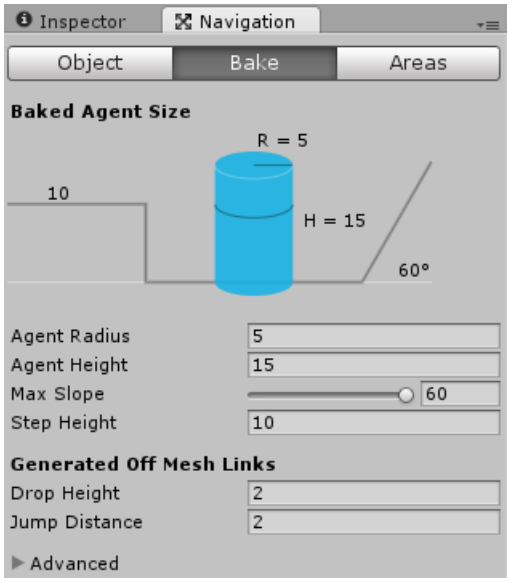

Figura 3.20: Janela de *Navigation* do *Unity*.

No menu *bake* existe uma série de opções para configurar a *NavMesh*:

- *Agent Radius* indica qual é a largura de um agente. Deve ser colocada a maior largura de todos os agentes, de forma a que eles não colidam contra os cantos das plataformas;
- *Agent Height* é a altura do agente. Se este valor for muito pequeno, vai impedir que os agentes subam as rampas. Se for muito grande, todas as inclinações do mapa serão consideradas como rampas;
- *MaxSlope* indica o máximo de inclinação que as rampas podem ter;
- *Step Height* indica a diferença de altura entre os vários níveis (neste caso as plataformas).

O *Drop Height* e o *Jump Distance* não foram usados no projeto, visto as unidades não poderem cair de plataformas, nem saltar entre plataformas. Após ser feito *bake* da *NavMesh*, ele vai dividir o mapa em regiões, como é possível ver na imagem 3.21. O caminho que for possível seguir é mostrado no terreno com a cor azul. É importante reparar que à volta de cada plataforma existe uma borda onde não é possível caminhar, borda esta criada pela opção *Agent Radius*.

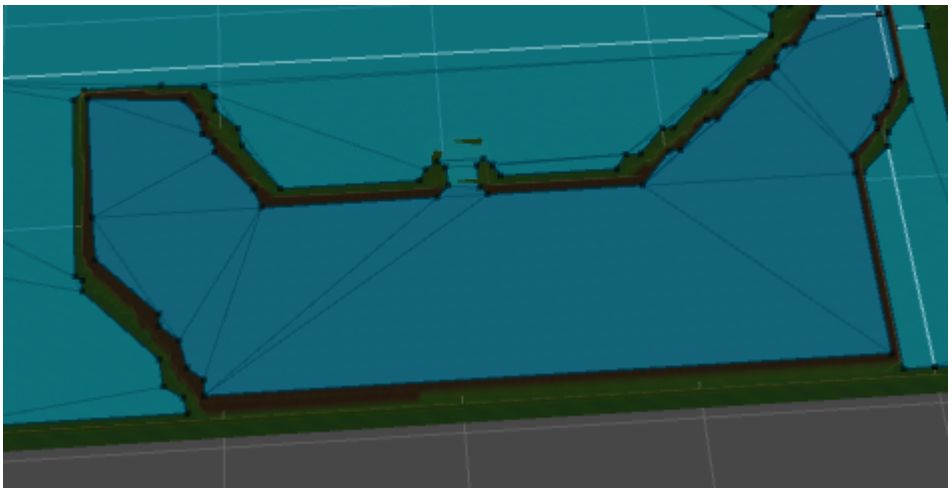

Figura 3.21: Exemplo da divisão do mapa após *bake* do terreno.

Para definir uma unidade como *Agent* foi preciso adicionar uma componente do tipo *Navmesh Agent* às unidades. Foi também preciso fazer o mesmo para os edifícios, mas adicionar uma *Navmesh Obstacle*, em vez de ser *Navmesh Agent*.
Para ter acesso à componente `agent` por código e dizer para se mover até uma posição, basta declarar uma variável do tipo *NavmeshAgent*, ir buscar a componente do objeto e indicar qual é o destino (ver excerto 3.27).

```
agent = GetComponent<NavMeshAgent>();
agent.setDestination(Place)
```

Excerto de Código 3.27: Definir posição destino do agente.

Eram apenas estas as alterações que tinham de ser implementadas na função `MakeMove`. Para verificar se uma unidade tinha chegado ao destino, bastava fazer a verificação entre a posição atual da unidade e o destino do `agent`.
Estando esta parte concluída, faltava implementar os *Navmesh Obstacles*. Tal como o nome indica, tratam-se de obstáculos. Estes obstáculos têm duas formas de funcionar: caso o obstáculo esteja esculpido na malha, vai criar uma área que não é possível caminhar à sua volta; caso não seja esculpido, será um obstáculo móvel e que poderá ser empurrado pelo agente, se este estiver no seu caminho.

No entanto, os obstáculos traziam um novo problema: visto existir uma área em que não era possível caminhar à volta destes, ao dizer a uma unidade para se mover até uma posição, ela nunca iria conseguir chegar lá e por isso nunca iria parar. Para isso foi preciso verificar qual é o sítio mais perto da posição desejada e se há possibilidade de se mover até lá. Isto é conseguido através da função `GetClosestPlace` que é chamada na função `StartMove`.

Nesta função vai ser verificado o tamanho do obstáculo, afastando-se assim do obstáculo num valor igual ao seu tamanho. No entanto, nesta função vai ser verificado de que lado é que a unidade se aproxima do obstáculo. Por exemplo, se for do lado esquerdo, ele deve mover-se X unidades a menos do que a opção de destino desejada. Caso seja do lado direito, a posição final deverá corresponder a X unidades a mais do que a opção de destino desejada.

Para dar mais informação ao jogador foi implementada uma imagem para o cursor de impossibilidade de movimento. Simplesmente é uma cruz vermelha que indica ao jogador se pode mover ou não mover a unidade até determinado sítio. Para ver isto, foi feita a verificação se existe um *Walkable path* na *NavMesh*. Caso não haja, é mostrado a imagem de impossibilidade. Para verificar isto, foi criada uma função chamada `CheckIfMovePossible` que é chamada na função `OnMouseHover`.

3.17 *Steering Behaviours*

Os *Steering Behaviours* (comportamentos de direção, em português) são algoritmos que ajudam personagens autónomas a ter movimentos de uma forma realista, usando forças simples para produzir movimentos parecidos com a realidade e navegação de improviso relativamente ao terreno envolvente. Não são baseadas em estratégias complexas que envolvem planear o caminho ou fazer cálculos difíceis, mas sim, simplesmente, em informações locais, tal como as forças que estão a ser exercidas em objetos vizinhos.
A implementação dos comportamentos de direção pode ser conseguida através de vetores matemáticos. Visto que as forças influenciam a velocidade de uma unidade e a sua posição, usar vetores é um bom método para as representar.

Existem pelo menos nove tipos de algoritmos de comportamentos de direção. Neste projeto foram utilizados dois: seguir um líder e evitar colidir contra outras unidades [Bev13].
No primeiro caso, se existe mais do que uma unidade selecionada, vai ser escolhida uma delas como líder. Quando for ordenado às unidades selecionadas movimentarem-se até uma posição, elas não vão todas para a posição marcada. O líder vai para a posição indicada e as outras vão para posições próximas dessa. No caso de haver três unidades selecionadas, existia a possibilidade de os dois seguidores irem para a mesma posição, colidindo. Deste modo, o segundo algoritmo vai evitar isto.

Para isso foram criadas cinco variáveis no *script Units*. Duas delas são do tipo booleano, uma para indicar se é o líder ou não e outra para indicar se a unidade já alterou o seu caminho. As outras três são do tipo inteiro, que vão servir para dizer a que distância quer que os seguidores fiquem atrás do líder, qual a separação entre unidades vizinhas e qual o máximo de separação entre unidades vizinhas.

A função `FollowLeader` é chamada sempre que existe um novo pedido de movimento (através da função `StartMove`). Nesta função vai ser feita a verificação de se existe mais que um objeto selecionado. Se sim e caso a unidade não seja o lider, se ela se encontrar em movimento e ainda não tiver sido feita esta modificação, calcular-se-á o vetor de velocidade da unidade. Vai-se normalizar esse vetor e dizer que o destino dessa unidade vai ficar X unidades atrás da posição do líder (ver imagem 3.22).

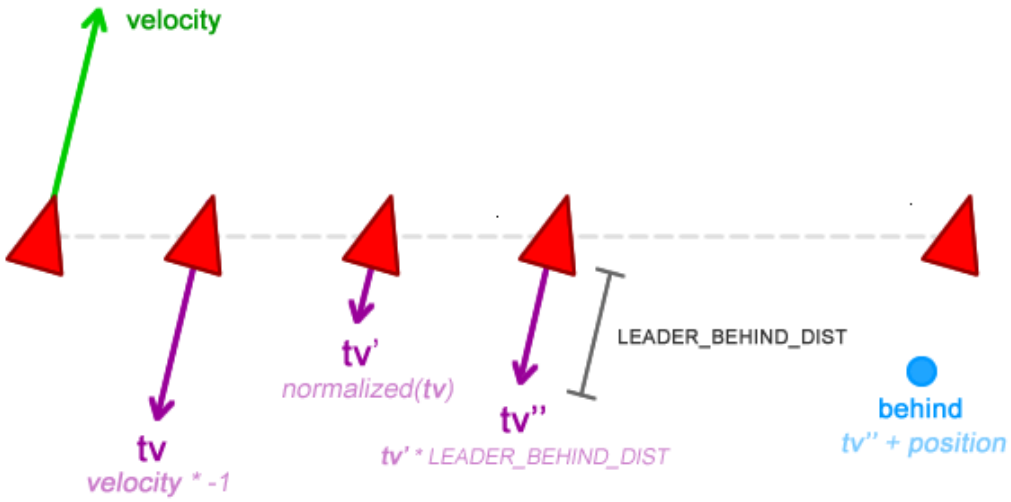

Figura 3.22: Algoritmo usado para calcular a posição dos seguidores do *leader*.

Tendo as unidades a não colidir com o líder, faltava fazer com que eles não colidissem entre si. As unidades ao seguir um líder tendem a mover-se de forma semelhante, formando uma "bola". Esta tendência pode ser mudada ao usar o algoritmo de separação. Este algoritmo previne que um grupo de unidades se aglomerem, garantido distância entre elas. A função `AvoidCrowding` vai fazer isso mesmo: devolve um vetor a ser somado ao destino da unidade da função anterior. Na função `AvoidCrowding` vai ser verificada a distância entre a unidade atual e as unidades vizinhas. Caso esta distância seja maior que a distância predefinida (*SEPARATION_RADUIS*), é guardada a distância entre as unidades sob a forma de um vetor.

Após isso, ele vai normalizar este vetor e multiplicá-lo pela variável *Max_Separation*. Este vetor é devolvido para a função `FollowLeader`.
Caso a distância previamente calculada seja inferior ao valor predefinido, não é preciso alterar a posição das unidades, sendo o vetor declarado como um vetor nulo para não influenciar a função `FollowLeader`.

Após a implementação destes dois algoritmos as unidades já não vão colidir umas com as outras.

3.18 Mini Mapa

Todos os jogos de estratégia tem um mapa em tamanho pequeno (mini mapa) que mostra todo o terreno de jogo bem como as unidades/edifício de todos os jogadores. Oferece também algumas funcionalidades. Se um jogador carregar num sítio do mini mapa, a câmara move-se imediatamente para essa posição. Se um jogador mover o rato em cima do mini mapa enquanto carrega com o botão esquerdo do rato, a câmara de jogo vai-se movendo de acordo com a posição do rato.

A criação de um mini mapa no *Unity* é muito simples. Basta criar uma câmara que esteja posicionada por cima do terreno e que esteja voltada para ele. Neste caso, esta câmara foi posicionada na posição (0,300,0) e com uma rotação de (90,0,0). A opção mais importante desta câmara é o tipo de projeção. Foi escolhida uma projeção ortogonal para que a altura

das plataformas seja ignorada. Na opção *Culling Mask* foi adicionada as *Layers* dos objetos que queremos que seja *renderizada* por ela (por exemplo, ela não vai *renderizar* os edifícios temporários, pois poderia ser uma vantagem/desvantagem para os jogadores).
Relativamente ao *ViewPort Rect*, que simboliza a posição do ecrã em que é mostrado o que a câmara vê, o mini mapa vai ficar no canto inferior esquerdo. Por isso, vai ficar afastado do canto esquerdo do jogo 0.001 *pixeis* tanto em *X* como em *Y*. Vai ter uma largura de 0.155% do ecrã e um comprimento de 0.275% do ecrã (ver imagem 3.23).

Figura 3.23: Exemplo do mini mapa do jogo.

Para implementar a parte dos cliques no mapa foi criado uma variável do tipo booleana que vai ser `true` quando o rato estiver em cima do mini mapa e `false` quando não. Para saber isto, foram criados no painel do mini mapa dois eventos, um do tipo *onMouseEnter* e outro do tipo *onMouseExit*. Para além disso, implementou-se uma função para calcular a posição para onde a câmara deve ir, em que é feito um `raycast` da posição do mini mapa e é convertida para a posição da câmara principal.

No mini mapa faltava apenas implementar uma caixa que mostrava o *view frustum* da câmara de jogo. Foi implementado criando num novo *script (LineRenderer)*, que desenha quatro linhas de acordo com os quatro cantos da visão da câmara principal. Neste novo *script* foi criada a função `CreateLine`, que é chamada na função `Start` e são assim criadas quatro linhas brancas. No *script HUD* vai haver uma nova função a ser chamada na função `Update`, com o nome de `DrawMinimapViewPort`. Nesta função vai ser verificada a posição dos quatro cantos que a câmara principal está a ver e adicionar esses pontos numa variável do tipo lista com o nome de `pointList`. Esta lista depois vai ser usada pela função `Update` do *script LineRenderer*, que vai atualizar a posição inicial e final das quatro linhas criadas(ver imagem 3.24).

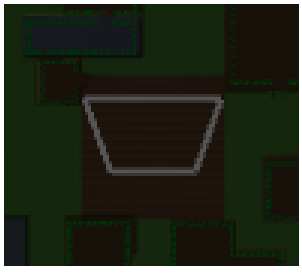

Figura 3.24: Caixa que mostra a região que a câmara principal está a ver.

3.19 *Fog Of War*

O *fog of war* foi um termo usado na guerra para traduzir incerteza sobre as capacidades e planos do inimigo. Nos jogos de estratégia, seja em tempo real, seja baseado em turnos, este desconhecimento é representado por uma névoa escura e é apenas visível uma área em redor das unidades/edifícios dos jogadores (ver imagem 3.25).

Figura 3.25: *Fog of war* no jogo *Age of Empires*.

Sem esta névoa o jogo tornar-se-ia desinteressante, pois os jogadores podiam ver as táticas dos jogadores adversários, defendendo-se delas. Para a implementação do *fog of war* no *Unity* foram precisas várias componentes [Tar13]:

- Uma câmara para*renderizar* apenas o *fog of war*;
- Um plano *(FogOfWarPlane)* para simular o *fog of war* (tem o *mesh collider* desabilitado para que não possa responder aos cliques);
- Um plano *(AppertureMask)* para simular a abertura no *fog of war* à volta das unidades.

Relativamente à câmara, ela tem algumas opções especiais. A opção *ClearFlags* vai ter uma cor sólida preta (basicamente a cada frame, vai fazer o *reset* de tudo o que vê, colocando uma cor preta). Na opção *Culling Mask* vai apenas mostrar objetos que tenham a *Layer "AppertureMask"*. Na opção *TargetTexture* é usada a textura *FogOfWarRT*, que é uma textura toda preta. Esta textura vai sendo atualizada pelo *AppertureMask* e reiniciada pela opção *ClearFlags*.

O plano que simula o *fog of war* é um plano simples, colocado à altura de vinte e dois. Tem como material o *FogOfWarPlane*, que tem como *shader* um *shader customizado*. Este *shader* é semelhante ao *shader* difuso, mas com simples alterações, principalmente a nível de *ZTest* (*renderização* em profundidade) e de cores.

O plano *AppertureMask* vai estar presente em todas as unidades/edifício. Para cada jogador, se a unidade/edifício lhe pertencer ou for da sua equipa, o *AppertureMask* dessa unidade/edifício vai ser *renderizado*; caso contrário, não. Isto vai fazer com que as unidades/edifícios da mesma equipa sejam visíveis a todos os membros daquela equipa, enquanto que as unidades/edifícios adversários vão estar escondidos.

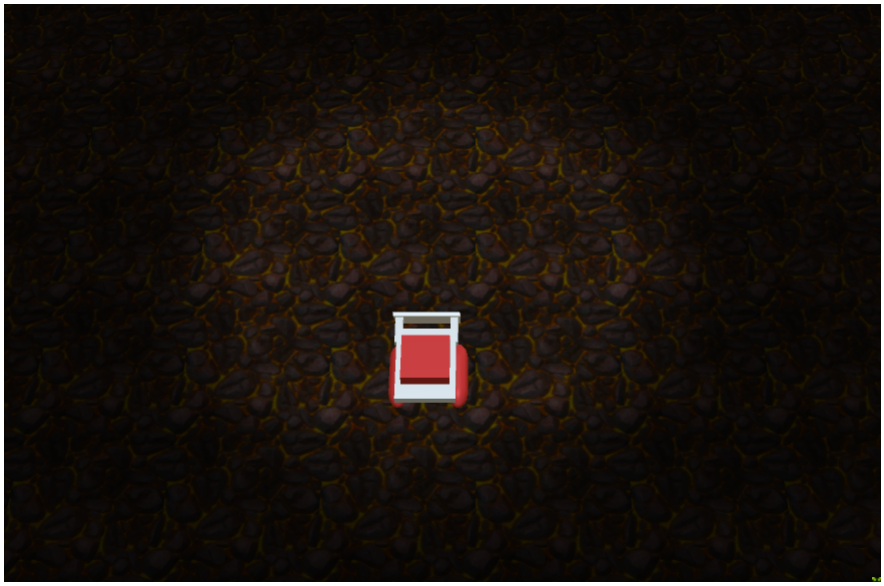

Figura 3.26: *Fog of war* implementado no jogo.

A implementação do *fog of war* trouxe alguns problemas: por um lado o cursor do rato mostrava o que estava para lá do *fog of war* (por exemplo, aparecia o cursor com a imagem de ataque), por outro lado era possível selecionar unidades não visíveis. Para resolver isso, foi preciso alterar o código de forma a verificar se, ao carregar com o botão esquerdo do rato no mapa, colidia com alguma `apperture mask`.
Para implementar esta alteração foi criada uma função que faz um `raycast` da câmara principal até ao infinito na posição do rato, verificando se encontra alguma `apperture mask`. Este `raycast` vai ignorar todas as *layers* e apenas focar-se na `layer AppertureMask`. Caso a função devolva `true`, pode-se fazer as ações normais. Caso contrário, não permite fazer nada.

3.20 *Tooltips*

Em todos os jogos existe uma dificuldade inicial por parte dos jogadores em aprender a jogar. Mesmo que exista um tutorial que ensine as bases do jogo ao jogadores, eles vão sentir a necessidade de saber o que é que cada unidade pode fazer, cada edifício, etc.
Para isso foram criadas umas *tooltips* (ver imagem 3.27), ou seja, umas caixas de texto que vão mostrar ao jogador o que é que cada ação faz ao passar com o rato por cima da botão. O jogador ao ter o trabalhador selecionado e passar o rato por cima da ação "construir refinaria", vai ser mostrado o nome do edifício, a imagem, a quantidade de recursos e uma descrição. Se algum destes campos não estiver a ser utilizado, esse campo é escondido. Esta informação foi guardada em quatro variáveis.

Relativamente à hierarquia no *Unity*, a *tooltip* faz parte do *Panel_actions* da interface. O *GameObject ToolTip* tem como componente uma imagem de fundo amarela (que vai criar aquele efeito aura) e vai conter o *script Tooltip*. Tem também cinco *GameObject* filhos: background, nome da ação, imagem, texto do custo da ação e a descrição.
O *script tooltip* é um *script* muito simples, que tem duas funções, uma para mostrar e outra para esconder o *GameObject* (são chamadas quando for posto o rato em cima de um botão do *UI* e quando for retirado). Depois o texto que é mostrado é alterado pela função `ChangeText`,

que é enviado como parâmetro *GameObject*, que chama a função, e, a partir do nome dele, é escolhido qual é o texto que deve aparecer.

Figura 3.27: Imagem do um *tooltip* do jogo.

3.21 Sistema de Grelha

Chegando a este ponto do projeto foi feita uma análise a tudo o que tinha sido feito e uma comparação com os jogos de estratégia normais. Uma possível falha que poderia ser grave prendia-se com o fato de no jogo desenvolvido o posicionamento dos edifícios não seguir nenhuma regra específica. Um jogador mais experiente poderia posicionar os edifícios numa forma estratégica, de forma a aproveitar o espaço da plataforma, enquanto que um jogador mais leigo já não.

Os jogos de estratégia usam um sistema de grelha para evitar estes problemas. Basicamente os edifícios têm posições pré-definidas em que podem ser construídos e assim todos os jogadores jogam com as mesmas regras. Esta grelha também auxilia os jogadores, ao mostrar os sítios onde podem e não podem construir.

A implementação deste sistema de grelha verifica se o edifício está num sítio plano (ou seja, não permite construir em rampas, o que antes era possível) e se é permitido construir nesse plano (por exemplo, se estiver encostado a um recurso, não é permitido construir).

Para isso, foi colocado como filho de todos os edifícios um *GameObject* vazio com o nome de *Grid*, composto por nove planos filho, formando um quadrado de três objetos por três objetos. A torre de defesa vai apenas ocupar o quadrado central, enquanto que a refinaria vai ocupar os três do meio e a fábrica de guerra vai ocupar todos. Os quadrados por baixo do edifício podem ter a cor verde ou vermelha, dependendo se o edifício pode ou não ser construído, respetivamente (ver imagem 3.28). Esta grelha só vai ser mostrada ao jogador se ele estiver à procura de um sítio para construir o seu edifício. Esta grelha também é visível por cima do *fog of war*.

Figura 3.28: Sistema de grelha do jogo relativamente à construção da refinaria.

3.22 *Networking*

Para começar a implementação do modo multi jogador foi criado um novo projeto de teste. Isto poderia ter sido feito numa cena à parte, mas assim houve a certeza que de não haveria conflitos no projeto atual.

Neste novo projeto foi criado uma cena e nela foram criados três *gameObjects* com *Box Colliders* e *Rigid Bodies* para fazerem de paredes e chão (duas na vertical e uma na horizontal).
Para além disso, foi criado um *Gameobject* vazio com o nome de *NetworkManager*, para gerir as ligações. Neste objeto foram adicionados duas componentes:

- *NetworkManager* - irá tratar de todas as ligações;
- *NetworkManager HUD* - irá mostrar um pequeno *HUD* (ver figura 3.29), para que os jogadores se possam ligar entre si.

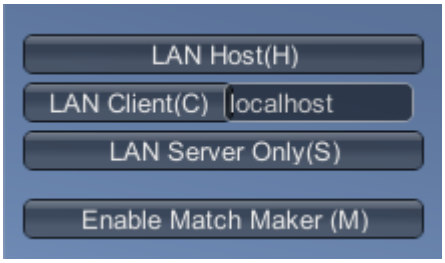

Figura 3.29: *Network HUD* do *Unity*.

A ideia neste teste foi ter uma bola a andar de um lado para o outro e ser sincronizado em ambos os clientes. Foi criado um *GameObject* vazio, onde depois foram adicionadas as seguintes componentes:

- *Box collider* e *Rigid Body*;
- *Script* para controlar o movimento;
- *Network Identity* (para ser considerado pelo *Unity* como um objeto que vai estar em todas as máquinas);
- *Unity Network Transform Script* (vai sincronizar o movimento entre todos os clientes).

A partir deste objeto criou-se um *prefab*. Sempre que um jogador se ligar à partida, o *Network Manager* instancia um objeto [Uni15c]. Caso o jogador se desligue, este objeto é apagado da partida. Esta informação foi adicionada no *Spawn Info*, na secção *Player Prefab* do *Network Manager*.
No entanto, ao testar o jogo assim, acontecia que o jogador que fosse o *hoster* (jogador que criou a partida, o seu jogo serve de servidor e de cliente) conseguia mover os dois objetos. Assim, no *script* que atualizava a posição da bola, foi preciso adicionar a seguinte verificação para tal não acontecer (ver excerto 3.28). A função `istLocalPlayer` devolve `true`, se aquele objeto pertencer ao jogador da máquina local, e `false`, se pertencer a outro.

```
if(!isLocalPlayer){
  return;
}
```

Excerto de Código 3.28: Verifica se o objeto pertence ao jogador da máquina local.

Passando à implementação real do modo multi jogador no jogo, foi preciso criar um *player prefab*, um objeto em que vai ser instanciado sempre que um cliente se ligar ao jogo. Este objeto foi o objeto *Player*. Para isso, bastou adicionar a esse objeto uma componente do tipo *Network Identity* e uma do tipo *Network Transform* e criar um *prefab* novo. Após isto, foi necessário criar uma nova cena com a finalidade de fazer a ligação entre os jogadores e mandá-los para a cena de jogo.

Esta nova cena terá o nome de *Lobby* e vai conter apenas um objeto vazio com duas componentes, o *Network Manager* e um *Network Manager HUD*. Foi também preciso adicionar algumas definições ao *Network Manager*. Para começar, foi preciso arrastar o novo *prefab* do jogador para o sítio onde diz *"Player Prefab"*. Isto seria suficiente caso não fossem aparecer novas unidades/edifícios durante o jogo, o que não é o caso. Como vão ser treinadas novas unidades/edifícios, vai ser preciso adicionar uma componente aos seus *prefabs*, o *Network Identity*. Após isso, foi preciso adicionar os *prefabs* à secção *Registered Spawnable Prefabs*. Uma última configuração foi adicionar a cena de jogo à secção *Online Scene* (ver imagem 3.30).

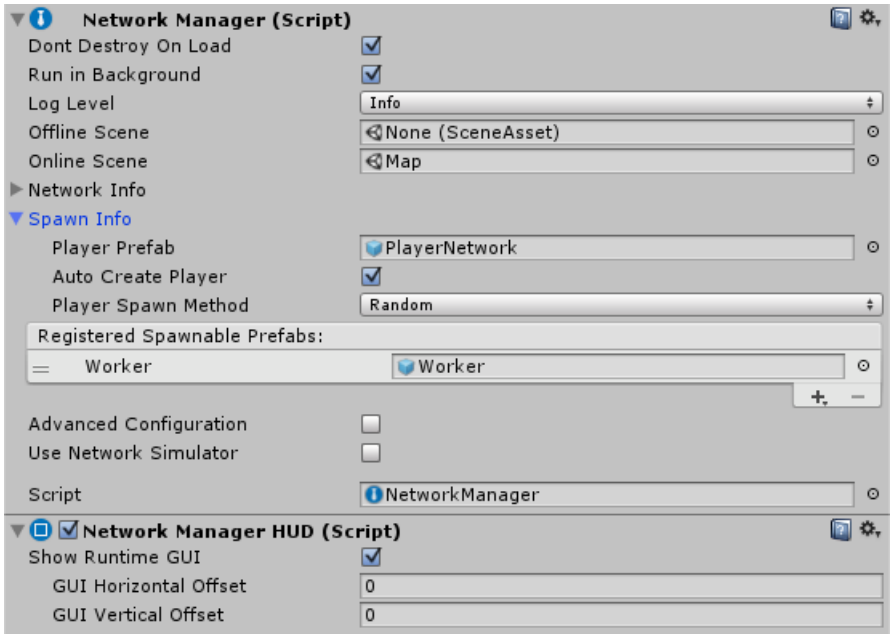

Figura 3.30: Configuração do *Network Manager*.

Um problema que começou a surgir nos primeiros testes foi que o painel de *HUD* estava a ser partilhado de algum modo entre os jogadores da partida, havendo funções que deixavam de funcionar. Para resolver este problema foi preciso alterar o *script HUD*. Em vez do *script* ter uma variável pública do tipo *GameObject*, foi declarada uma variável privada por cada *GameObject* do *HUD* e o *script* foi buscar as referências a partir da função `Awake`, usando-as posteriormente em todas as funções necessárias.

Com esta alteração os `triggers` dos botões também tinham deixado de funcionar. Até aqui, eles eram implementados diretamente no *inspector* do *Unity*, mas para o modo multi jogador foi preciso criar os `EventTriggers` por código. Para isso foi implementada a função `SetFunctionTrigger` e `SetTrigger`, que serve para criar os `triggers` de todos os botões e imagens (ver excerto 3.29).

```
private void SetFunctionTrigger(GameObject objectoTotrigger, EventTriggerType
    type, UnityAction<BaseEventData> functionName)
{
    EventTrigger trigger = objectoTotrigger.GetComponentInParent<EventTrigger
        >();
    EventTrigger.Entry entry = new EventTrigger.Entry();
    entry.eventID = type;
    entry.callback = new EventTrigger.TriggerEvent();
    entry.callback.AddListener(functionName);
    trigger.triggers.Add(entry);
}
```

Excerto de Código 3.29: Função para criar os *triggers* dos botões e das imagens.

Ao fazer *build* do projeto neste preciso momento e abrindo duas instâncias (dois jogos) era instanciado um objeto *player* para cada lado, sem unidades. Faltava implementar a criação de unidades. Para isso foi implementada a função `CmdSpawnWorker`, que vai instanciar o trabalhador (ver 3.30).

```
[Command] //Calls server to do something
void CmdSpawnWorker()
{
  GameObject instance = Instantiate(trabalhadorPrefab, transform.position,
      transform.rotation) as GameObject;
  instance.transform.parent = transform;
  NetworkServer.SpawnWithClientAuthority(instance,base.connectionToClient);
  RpcSetParent(instance);
}
[ClientRpc] //Server calls client
void RpcSetParent(GameObject instance)
{
  instance.transform.parent = transform;
}
```

Excerto de Código 3.30: Função para instanciar um trabalhador.

É de notar que existe uma *tag* em cada uma das funções:
A *tag* `Command` serve para os clientes fazerem um pedido ao servidor. Neste caso, para instanciar uma nova unidade com autoridade do lado do cliente (ou seja, o cliente manda no objeto e pode comandá-lo diretamente sem fazer pedidos ao servidor).
A *tag* `ClientRpc` é uma chamada do servidor para todos os clientes. Neste caso, ele vai informar a todos os clientes que aquela determinada unidade vai ser filha do objeto *player*.
Para isto funcionar corretamente foi preciso fazer um import ao `UnityEngine.Networking` e

indicar que a classe herda o `NetworkBehaviour` em vez do `MonoBehaviour`.

Após novo teste verificou-se que o trabalhador era criado no jogo do primeiro jogador a entrar (o *hoster*) e não era criado no jogo do segundo jogador. Isto acontecia porque os clientes não se ligam ao mesmo tempo, criando um problema de sincronização de objetos e filhos. Para isso, foi criada uma `coroutine` que esperava cinco segundos antes de criar os trabalhadores, para que quando estes fossem criados já todos os clientes estivessem ligados.

Após esta mudança, o *core* já funcionava como deve ser, mas ainda eram necessárias algumas alterações. Para começar, foi preciso alterar em todos os *scripts* a forma como era verificado se uma unidade pertencia a um jogador. Em vez de verificar o nome do jogador, a verificação incide sobre a autoridade do jogador sobre o objeto (`.hasAuthority`). Se o jogador tiver autoridade sobre o objeto, a unidade/edifício é selecionado, movido, etc. Caso contrário, não é. Para isto funcionar como devido, foi preciso selecionar a opção *"Local Player Authority"* da componente *Network Identity* em todos os *prefabs*. Também foi alterada a forma como os edifícios eram construídos. Até aqui, era sempre instanciado um objeto local, o que já não fazia sentido. Foi então criada a função `CmdSpawnUnit` e `CmdSpawnBuilding`, para serem criadas unidades e edifícios na ligação do jogo. Para além desta função, foram implementadas as versões multi jogador de funções, que estavam a funcionar em modo local, enumerando-se aqui as principais:

- `CmdDeleteUnit` - para eliminar unidades do jogo;
- `RpcSetMoving`, `RpcSetCreator`, `RpcActivateNavmeshObstacle` - tal como o nome indica, para colocar uma unidade a mover-se, dizer a uma unidade qual é o edifício onde foi criada e para ativar uma *NavMeshObstacle*;
- `CmdSetBuildingCompleted` - para dizer que um edifício está concluído;
- `CmdIncrementResource` - para incrementar um tipo de recursos numa determinada quantidade;
- `CmdSetCurrentProject` - para dizer a um trabalhador qual é o seu projeto.

Faltava ainda fazer a sincronização do combate. Para isso, foram implementadas duas funções, a `CmdAttack` e a `RpcAttack`. Estas funções vão ser chamadas (a `CmdAttack` vai chamar a `RpcAttack`) sempre que alguma unidade/edifício sofra dano e é enviado como parâmetro da função o valor do dano e o `netID` da unidade (ver excerto 3.31). O `NetID` é um identificador único do objeto na ligação, ou seja, não há `NetID's` repetidos. Para colocar os ataques a funcionar sobre a ligação faltava apenas atualizar o *script* do projétil.

```
 public void RpcAttack(NetworkInstanceId netID, int value)
{
    GameObject objectAux = NetworkServer.FindLocalObject(netId);
    objectAux.GetComponent<WorldObjects>().hitPoints -= value;
}
```

Excerto de Código 3.31: Função para sincronizar o ataque entre os clientes.

O combate já estava a funcionar como deve ser mas os projéteis não estavam sincronizados em todos os clientes. Existiam duas possibilidades: a primeira era criar um projétil como um *prefab* de *Networking* e era instanciado e destruído como todos os outros objetos; a segunda era

sincronizar o alvo do ataque daquela unidade e instanciavam-se os projéteis em todos os clientes ao mesmo tempo. Só iria ser retirado dano no *script* do dono do objeto `(isLocalPlayer)`.
A primeira opção iria criar um peso muito grande na ligação. Se, por exemplo, houvesse trinta unidades a disparar entre si, seriam instanciados trinta projéteis de *X* em *X* segundos na rede, e a sua posição tinha de ser sincronizada para todos os clientes. Na segunda opção, bastava sincronizar o alvo e assim que tivesse um alvo o *script* ia chamar a função `CreateBullets`, criando os projéteis de forma local.
Um outro problema que a sincronização dos valores estava a trazer era que, devido ao tempo de resposta entre cliente/servidor, alguns edifícios ficavam com mais um ponto de vida do que o máximo possível, isto é, se o máximo fosse cinquenta, eles iriam ficar com cinquenta e um pontos de vida. Isto não acontecia todas as vezes. Foram implementadas funções em que a vida de um edifício era aumentada apenas se a vida com que ele ia ficar não fosse superior à vida máxima.
Faltava apenas fazer a sincronização dos recoletores. Para isso foram criadas duas funções: a `CmdCollect` e a `RpcCollect`, que iriam recolher e adicionar um recurso de um certo tipo a um jogador.

Chegando a esta parte do projeto foi preciso criar as duas versões para fazer os testes que vão ser explicados em detalhe no capítulo 4. Foi feita uma cópia do projeto para que fosse possível criar as duas versões. O desenvolvimento deste projeto seguiu a ideologia *peer-to-peer*, ou seja, a autoridade dos objetos está no lado dos jogadores. O servidor não manda nos objetos dos jogadores e serve apenas para guardar o *IP* de todos os jogadores. O cliente faz um pedido ao servidor através da `tag` "*command*", mas tudo o que essa função vai fazer é chamar uma outra função com a `tag` "*RpcClient*", de forma a enviar a informação para os outros jogadores.

Para a implementação da arquitetura cliente-servidor houve necessidade de fazer mais alterações. Inicialmente teve de ser retirada a autoridade dos objetos por parte dos clientes. Esta alteração foi simples, bastou tirar o visto de *"Local Player Authority"* em todos os *prefabs*.

Após isso teve de ser feita uma alteração a todas as funções, porque qualquer ação que fosse feita tinha de passar pelo servidor (atacar, mover, recolher recurso). Assim tiveram de ser alteradas as funções que tinham a *tag* `Command`, para que toda a ação fosse comandada pelo servidor e depois replicada por todos os jogadores. Visto que as funções já estavam todas feitas, foi uma alteração rápida.

O último aspeto que teve de ser levado em consideração foi a forma como estavam a ser feitas as verificações de a quem pertencia a unidade/edifício. Até aqui eram feitas a partir da função ``hasAuthority", o que agora iria retornar sempre `false`, porque quem tinha autoridade era o servidor. A função usada passou a ser `isLocalPlayer`, para que o jogador apenas pudesse controlar as suas unidades. O instanciamento de unidades passou a ser feito por `NetworkServer.Spawn` em vez de ser feito por `NetworkServer.SpawnWithClientAuthority`.

3.23 *Lobby Manager*

Chegando ao ponto final do projeto, faltava apenas criar algo para que os jogadores pudessem escolher a sua equipa (entre o lado humano e o lado vampiro).
Para isso, foi usado como base um *LobbyManager* do *Unity*, que é oferecido na *Asset Store* ([Uni15a]), com a premissa que no futuro estará disponível como um pacote que virá com a instalação do *Unity*. As alterações feitas a este *LobbyManager* ocorreram ao nível da implementação de novas funcionalidades sobre ele, já que ele trata de todas as ligações entre os jogadores (basicamente o que era feito com a componente *"Network Manager HUD"*, só que com um visual diferente).
Este *Lobby* foi muito simples de implementar, visto que bastou usar o `canvas` criado por ele. Dentro desse `canvas` existem vários painéis. Um no topo que mostra a informação da ligação, e o painel que vai interagir só com jogadores (ver imagem 3.31).

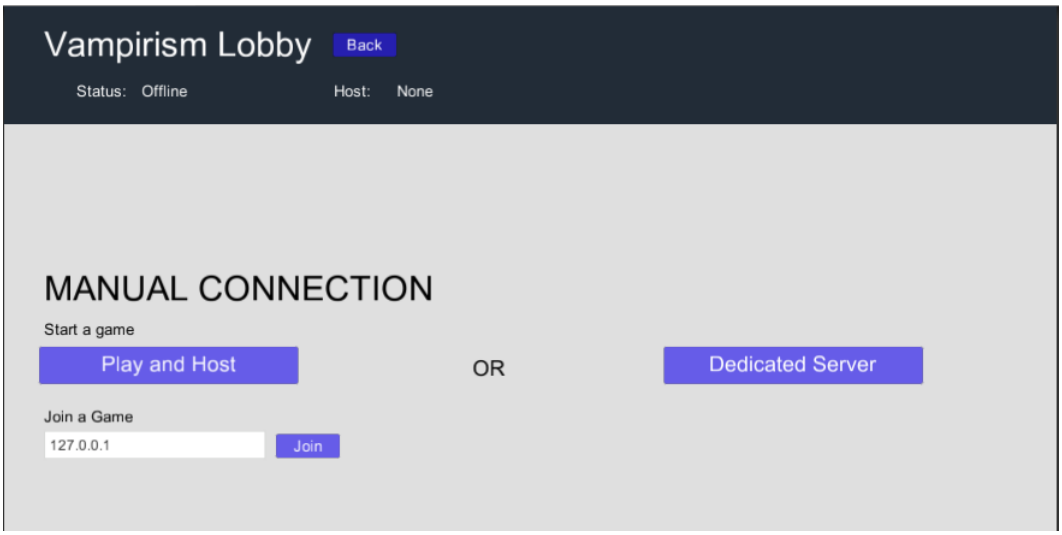

Figura 3.31: Ecrã inicial do *LobbyManager*.

Este painel inicial permite ao jogador criar um servidor e jogar ao mesmo tempo (*"Play and Host"*) criar um servidor dedicado (apenas vê) ou juntar-se a uma partida através do *IP* do servidor do jogo.
Ao selecionar qualquer uma das opções o jogador é levado para uma espécie de sala de espera, um painel onde vão estar os jogadores que se encontrarem ligados àquele servidor. Neste painel podem trocar o seu nome de jogador, a cor que as suas unidades vão ter e dizer se estão preparados ou não. Todas estas alterações são sincronizadas entre os jogadores (ver imagem 3.32). Podem também trocar de equipas, o que em algum momento será obrigatório a algum dos jogadores, visto que foi implementada uma condição na qual só é permitido um jogo começar quando houver um jogador de cada equipa. Isto poderá ser um problema um dia que o jogo seja lançado, pois poderão existir mais pessoas interessadas num dos dois lados, levando à falta de jogadores no outro. A solução passaria por começar tudo na mesma equipa e depois dentro do jogo ser sorteado aleatoriamente quem jogava de um lado ou do outro.

Mal haja um número de jogadores mínimo (pelo menos dois), um jogador em cada equipa e todos se encontrem prontos, o *hoster* pode dar início à partida, caso assim o deseje. É mostrado um pequeno contador regressivo de cinco segundos até os jogadores serem levados para o mapa de jogo, onde podem jogar. Após o jogo começar, não é possível que mais jogadores se juntem à

partida.
Esta implementação de duas equipas foi o que permitiu muitas novas regras no jogo, como por exemplo o facto de um jogador só poder ver as unidades da sua equipa, sendo as restantes tropas inimigas escondidas pelo *fog of war*.
Para além destas alterações previamente anunciadas, foram ainda feitas algumas alterações de forma a melhorar o sistema de *Lobby* (ver imagem 3.32):

- Correção das cores de fundo (dificultava os jogadores verem a sua informação);
- Se houver um número grande de jogadores, aparece uma *scrollbar*;
- Apenas o *hoster* pode começar o jogo e apenas se todos os jogadores estiverem prontos;
- Os jogadores podem cancelar o seu estado de pronto, mas não podem mudar a sua informação caso já estejam prontos;
- Os botões são desabilitados quando começa a contagem regressiva;
- O número de jogadores que estejam em modo preparado para a partida começar é aumentado automaticamente conforme o número de jogadores na partida.

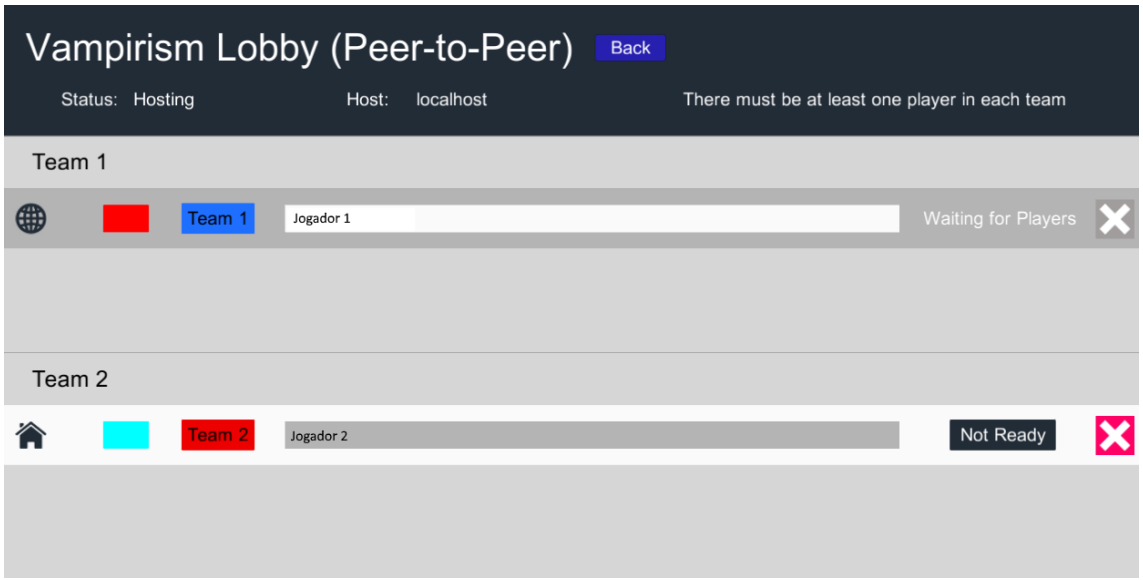

Figura 3.32: Versão final do *LobbyManager*, após alterações.

3.24 Conclusão

Neste capítulo, foi descrito com maior detalhe a implementação do jogo usando o motor de jogo *Unity*. Inicialmente, foi definido o *Game Design* do jogo, que serviu como base para o desenvolvimento do mesmo. De seguida, foi mostrada a implementação do jogo de uma forma gradual, começando nos objetos mais simples (mas fulcrais para o desenvolvimento inicial do jogo) até aos objetos mais complicados de implementar, que, embora não fossem funcionalidades obrigatórias, são pormenores importantes no jogo. Durante este desenvolvimento foram sempre explicadas todas as funções e as classes usadas.
No próximo capítulo serão expostos os testes realizados ao jogo usando as duas arquiteturas (cliente/servidor e *peer-to-peer*) e apresentados os seus resultados.

Capítulo 4

Testes e Resultados

O capítulo quatro é dedicado aos testes efetuados ao jogo com diversos utilizadores/jogadores, para avaliar se existe alguma diferença entre as duas arquiteturas (cliente/servidor ou *peer-to-peer*). É mostrado o protocolo dos testes, testes estes que foram realizados com quarenta utilizadores, seguido-se a apresentação dos resultados obtidos.

4.1 Introdução

No capítulo anterior foi relatado o desenvolvimento do jogo multi jogador, o qual foi testado por um grupo de utilizadores, de forma a obter uma avaliação sobre qual das arquiteturas é melhor para o desenvolvimento de jogos de estratégia multi jogador. Na secção 4.2 é apresentado o protocolo dos testes, enunciando todos os cuidados que foram tomados na realização dos mesmos, de forma a não influenciar os resultados. A partir destes testes espera-se saber se o jogo desenvolvido é de fácil compreensão e em qual das duas arquiteturas o jogo funciona melhor. Na secção 4.3 é feita uma análise dos testes efetuados na secção anterior, mostrando os gráficos com as respostas dos utilizadores.

4.2 Protocolo dos Testes

Nesta secção será explicado como foi feito o teste com os jogadores, que foi criado para verificar qual das duas arquiteturas se aplica melhor a um jogo de estratégia multi jogador *online*, usando o motor de jogo *Unity*.
Quarenta utilizadores realizaram os testes ao jogo desenvolvido, tendo avaliado as duas versões, cliente/servidor e *peer-to-peer*. Visto que a latência é um fator que influência este tipo de testes, estes não foram realizados presencialmente, mas sim com cada uma das pessoas em sua casa. Assim, os jogadores não se ligaram à sala de espera do jogo através de uma ligação local, mas sim através de uma ligação a partir da Internet, ou seja, tentando simular uma ligação real. Durante os testes, toda a conversação foi realizada mediante uma ligação *Skype* e todo o jogo foi gravado para futura análise.

Os testes foram aplicados a grupos de quatro pessoas. Este foi dividido em dois pequenos sub-testes, em que cada um avaliou um dos dois tipos de arquitetura. No teste da arquitetura cliente/servidor existiu um servidor dedicado para testar o jogo, enquanto que no teste da arquitetura *peer-to-peer*, como a comunicação é feita entre os jogadores, apenas houve necessidade de um servidor para fazer a sala de espera do jogo.

Foi construído um formulário usando o *Google Forms*, que continha informação importante para os participantes que iriam realizar o teste. No início do formulário fez-se uma introdução do teste aos participantes, enunciando a seguinte informação:

- É explicado o que vai acontecer durante o teste e o motivo de ele estar a ser realizado;
- É referido que este teste vai testar o jogo desenvolvido e não o participante;
- É dito que se o participante tiver dificuldades em usar ou compreender o que é pedido, deve referi-lo, visto que possivelmente outros participantes vão sentir a mesma dificuldade;
- O participante, se quiser, poderá desistir do teste a qualquer momento;
- Este teste terá a duração de cerca de trinta minutos;
- Todos os dados recolhidos serão usados apenas para a pesquisa e será mantido o anonimato, e que, se os mesmos forem publicados, serão compilados com a informação de todos os outros participantes.

Esta informação pode ser vista nas duas figuras abaixo.

Teste a um jogo desenvolvido no âmbito de um projecto final de Mestrado

Caro participante,

O meu nome é Nuno Carapito e sou aluno do 2º Ano do mestrado de Design e Desenvolvimento de Jogos Digitais na Universidade da Beira Interior.

No âmbito do Projecto Final de Mestrado, desenvolvi um jogo multi jogador com o objectivo de testar duas arquitecturas multi jogador.

Para além de testar as duas arquitecturas, pretendo encontrar formas de melhorar o jogo desenvolvido, não só a nível de utilização, mas também de percepção por parte do jogador sobre o que deve fazer, encontrar erros e descobrir melhorias de desempenho.

Este teste pretende avaliar o jogo, e não as suas capacidades.

Caso encontre dificuldades de compreensão ou utilização do jogo, o seu feedback é uma mais valia para todos os utilizadores, uma vez que é muito provável que as mesmas dificuldades sejam encontradas pelos outros colaboradores.

A prova tem a duração total de 30 minutos (2 testes de 15 minutos cada) e irá ser gravada para análise de comportamentos. Se se sentir incomodado em algum momento do teste pode parar imediatamente.

Solicito que no decorrer do teste verbalize todos os seus pensamentos. Isso irá ajudar-me a compreender melhor o porquê de fazer determinadas escolhas.

Estarei disponível para responder a qualquer questão somente após a conclusão do teste. No final do mesmo existirá também um pequeno questionário onde poderá inserir o seu feedback.

Toda a informação recolhida será usada apenas para motivos de pesquisa e será mantida em privado. Caso os resultados sejam publicados, estes serão compilados com a informação de outros utilizadores, salvaguardando-se o anonimato e total privacidade de todos os participantes.

Obrigado pela colaboração,
Nuno Carapito

Figura 4.1: Informação fornecida aos participantes no início do teste.

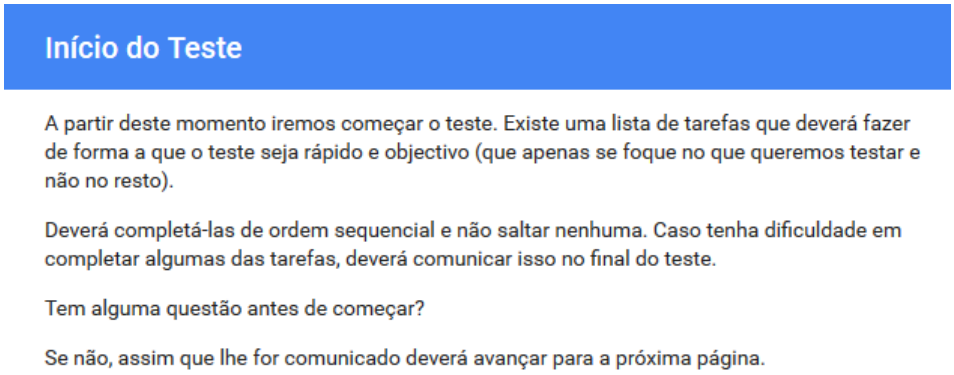

Figura 4.2: Informação fornecida aos participantes na segunda página do questionário.

De seguida, foi solicitado aos participantes o preenchimento de um pequeno inquérito com alguma informação (ver imagem 4.3 e 4.4), tal como o género, a faixa etária (dezoito aos vinte, vinte e um aos vinte e três, vinte e quatro aos vinte e seis, ou mais de vinte e sete anos de idade), se costuma jogar e, se sim, qual o tipo de jogos que joga, quantas horas por semana e se os jogos que joga são *online* ou não.

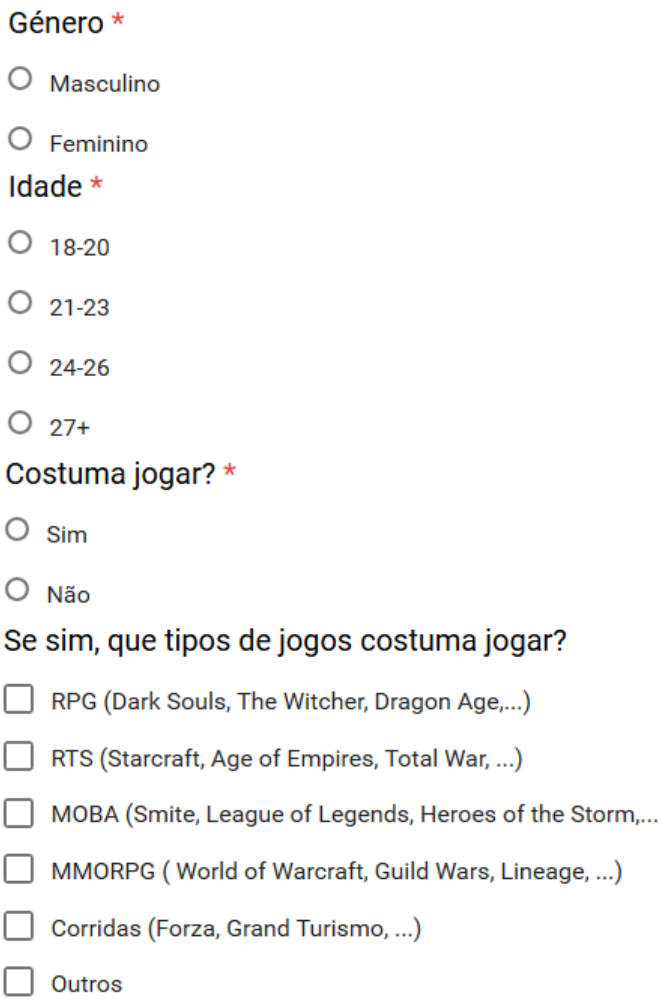

Figura 4.3: Página um do formulário apresentado aos participantes.

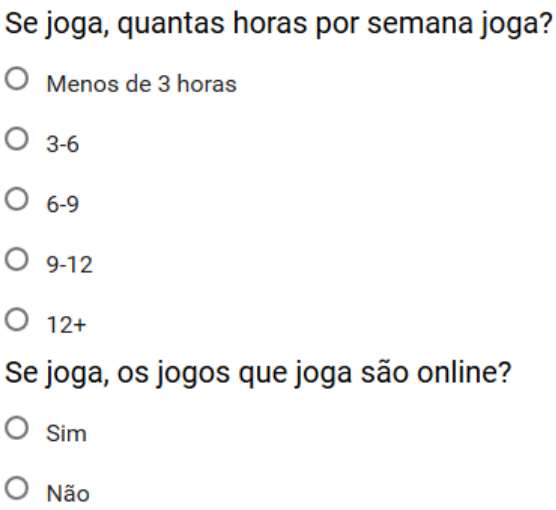

Figura 4.4: Página um do formulário apresentado aos participantes (continuação).

Após esta informação é anunciado o que irá acontecer a seguir, em que é mostrada uma lista de tarefas (ver imagem 4.5) que o participante deve completar por ordem sequencial, sem saltar nenhuma. Caso tenha alguma dificuldade em completar uma das tarefas, deverá apresentá-la no final do teste. É também perguntado se ele tem alguma dúvida antes de se dar início ao teste. Relativamente à lista de tarefas, elas foram construídas respeitando dois elementos chave (retirados do site `http://www.usability.gov`):

- Dizer sempre ao participante para fazer algo e não questioná-lo sobre como é que o faria;
- Não dizer ao participante onde clicar diretamente.

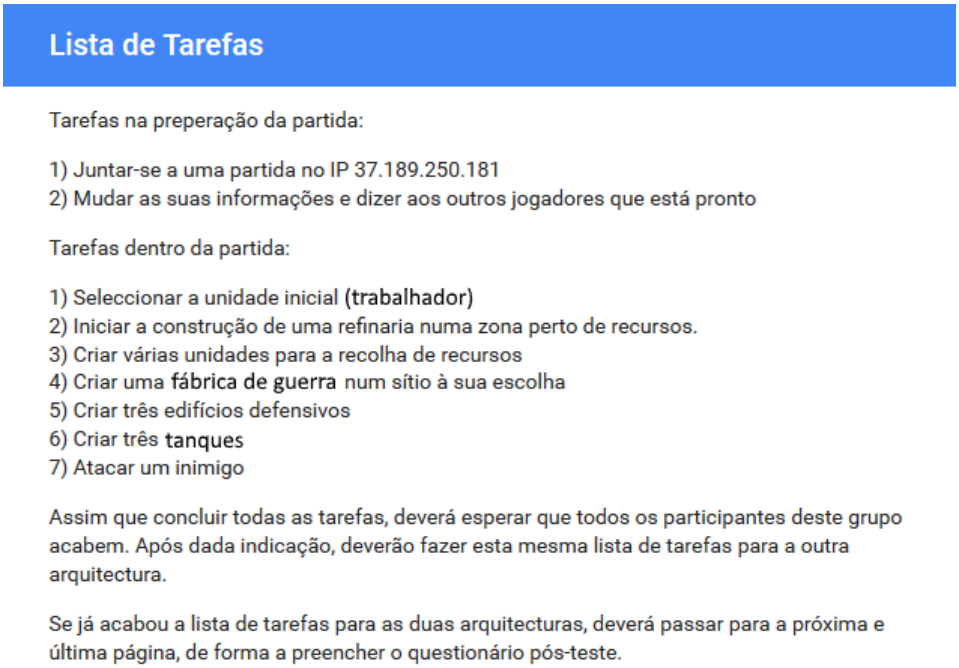

Figura 4.5: Lista de tarefas apresentadas aos participantes.

Quando o participante completa todas as tarefas deverá aguardar que todos os participantes o tenham feito de modo a poder mudar de jogo (para a outra arquitetura), fazendo a mesma lista de tarefas. Após terem completado todas as tarefas usando as duas arquiteturas, devem passar para a última página do formulário, onde é perguntado se o participante compreendeu todas as tarefas, os objetivos, se conseguiu perceber como interagir com o jogo, se encontrou algum erro no jogo e onde se comparam os dois tipos de arquitetura (ver imagem 4.6).

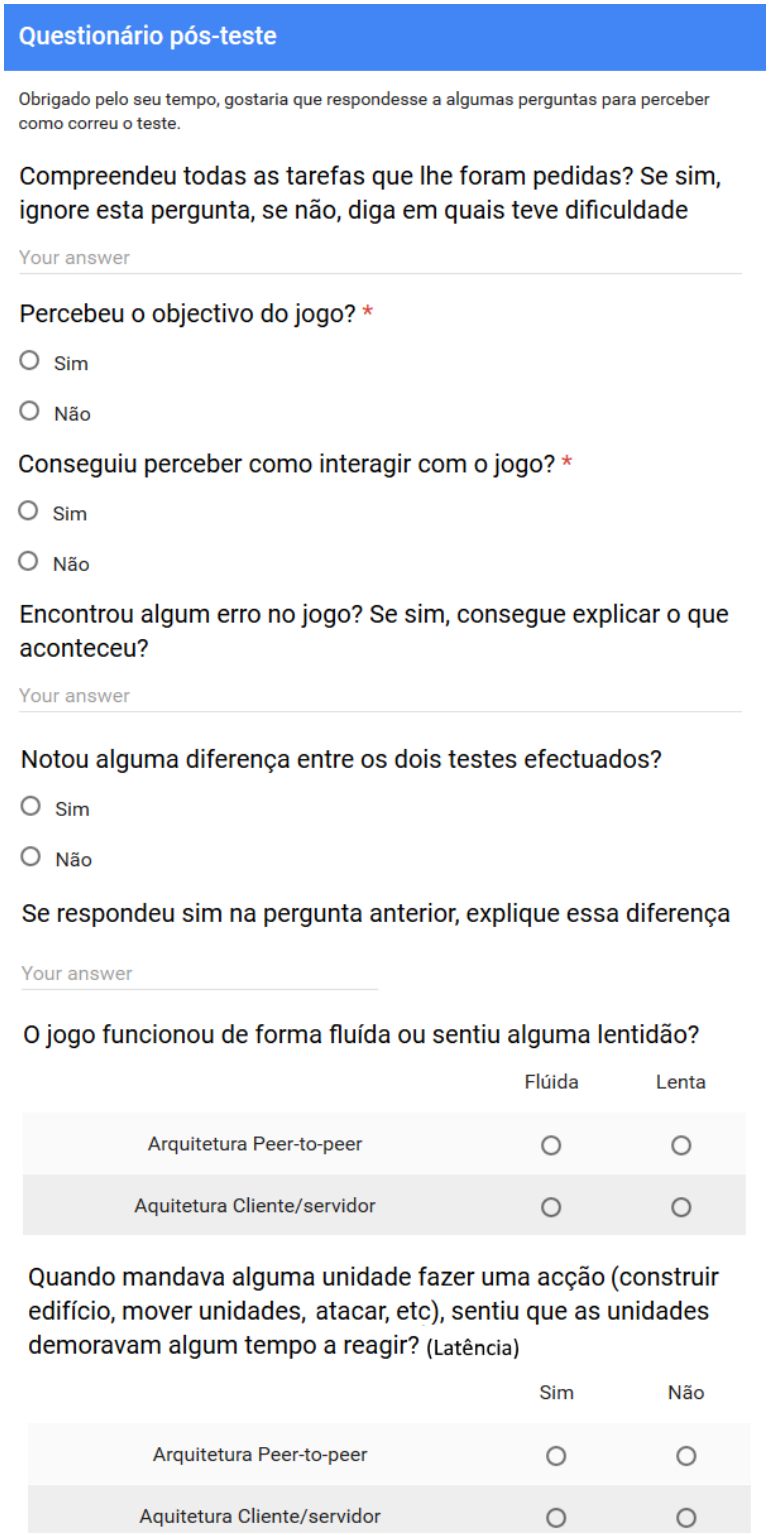

Figura 4.6: Página do questionário apresentado aos participantes no fim do teste.

4.3 Resultados

Dos quarenta participantes, três eram raparigas, duas delas com uma idade superior a vinte e sete anos e uma entre os dezoito e os vinte anos. Os restantes participantes eram rapazes, treze entre os dezoito e os vinte anos, catorze entre os vinte e um e os vinte e três, oito entre os vinte e quatro e os vinte e seis e dois com idade superior a vinte e sete anos (ver imagem 4.7).

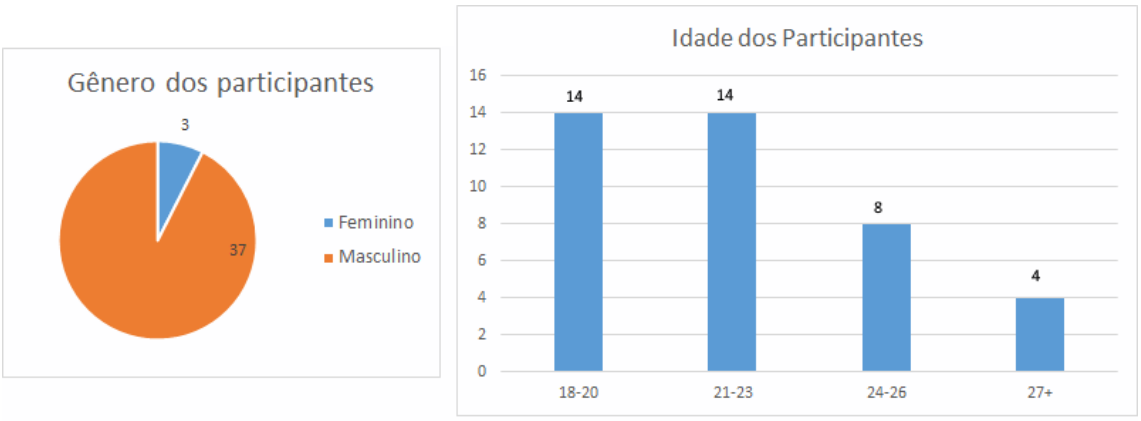

Figura 4.7: Gráficos relativos ao género e idade dos participantes.

Entre os participantes, cinco deles (quatro rapazes e uma rapariga) não costumam jogar no dia a dia (ver imagem 4.8). Dos restantes, dezasseis costumam jogar jogos de estratégia, enquanto que dezanove estão igualmente divididos entre *MOBA's* e *MMORPG*.

Figura 4.8: Gráfico relativo ao hábito de jogo dos participantes.

Por semana, três dos participantes jogam menos de três horas, sete jogam entre três a seis horas, dez jogam entre seis e nove horas, cinco jogam entre nove e doze horas e dez jogam mais de doze horas. Dos trinta e cinco participantes que jogam, dois não costumam jogar jogos *online* (ver imagem 4.9).

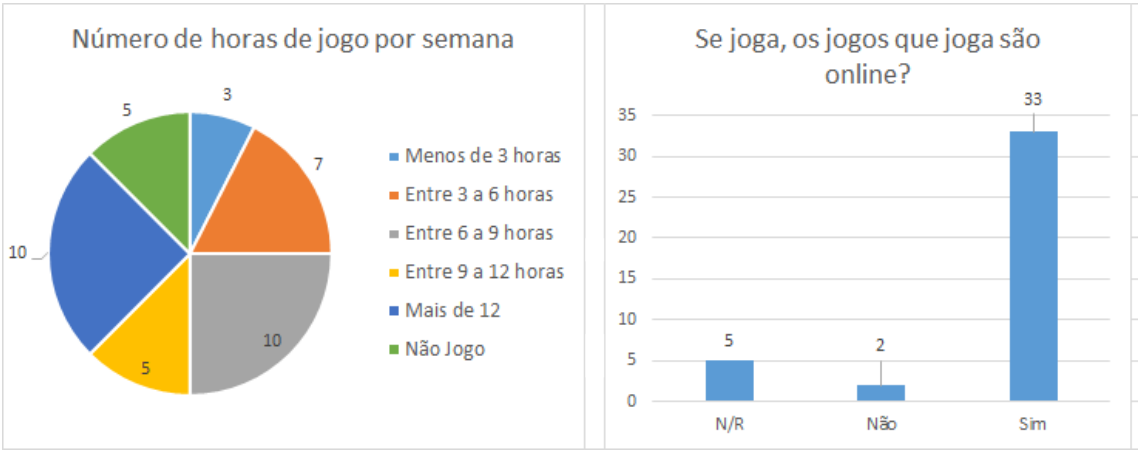

Figura 4.9: Gráficos relativos ao número de horas de jogo e ao modo como jogam dos participantes.

Relativamente às perguntas sobre o teste, dos quarenta participantes, trinta e nove perceberam o objetivo de jogo e um não. No entanto, todos eles perceberam como interagir com o jogo, inclusive os cinco que não costumam jogar. Relativamente à pergunta sobre se tinham percebido as tarefas que foram pedidas, todos os participantes compreenderam à exceção de um, que referiu que não tinha percebido como atacar as unidades adversárias (ver imagem 4.10).

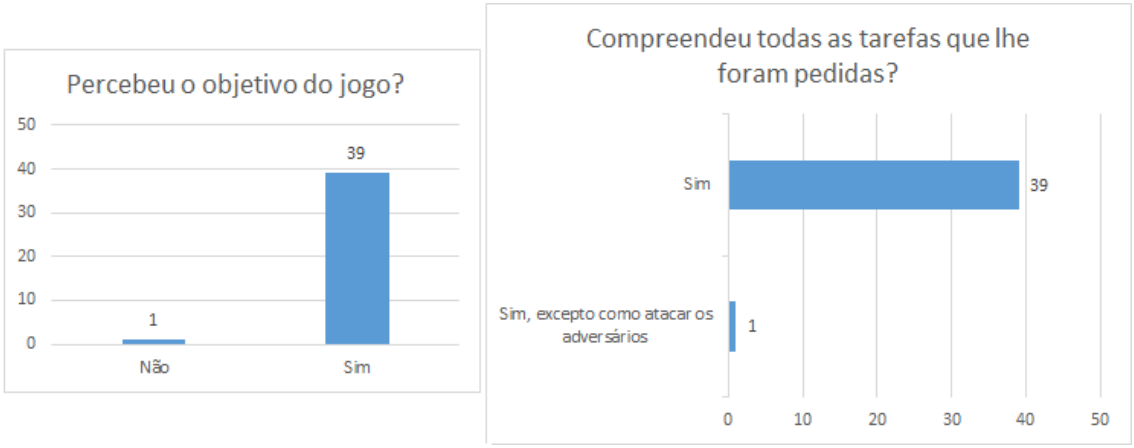

Figura 4.10: Gráficos relativos ao facto de que se o jogador percebeu as tarefas e o objetivo do jogo.

De modo geral, os problemas indicados pelos participantes podem dividir-se em três categorias:

- *Fog of war* - alguns participantes sentiram dificuldades em encontrar as suas unidades, visto que a área de abertura no *fog of war* estava demasiado pequena. Este problema intensificava-se nas plataformas;
- Criação de unidades nas bordas das plataformas - se um participante construísse um edifício na borda de uma plataforma e treinasse uma unidade, existia a possibilidade de ela ir para a parte de baixo da plataforma;
- Falhas no movimento das unidades e no ataque - algumas unidades não respondiam corretamente à ordem de ataque e não evitavam as colisões da melhor maneira.

Um participante não conseguiu jogar o modo cliente/servidor. Embora não exista total certeza sobre qual a origem do problema, o participante estava ligado à rede *MEO-WIFI*. Houve também três participantes que perderam a ligação com o servidor (modo cliente-servidor).

Estes três participantes estavam a partilhar a ligação à Internet com um quarto participantes, que ficava *online* enquanto os outros perdiam a ligação.

Em relação à pergunta se notaram diferença entre os dois testes efetuados, dezoito participantes responderam que não, enquanto vinte e duas afirmaram que sim (ver imagem 4.11).
Relativamente às diferenças encontradas, as respostas foram unânimes, os jogadores referiram que sentiram menos lentidão na arquitetura *peer-to-peer* comparativamente à arquitetura cliente/servidor, assim como uma maior fluidez do jogo e maior rapidez de resposta das unidades. Já na arquitetura cliente/servidor as unidades demoravam algum tempo a realizar as ordens dadas, quer fossem de movimento ou de ataque.

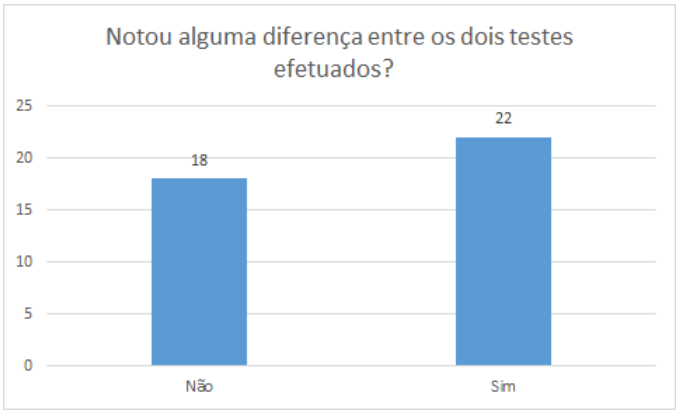

Figura 4.11: Gráficos relativos à diferença sentida pelos participantes entre os dois testes efetuados.

Relativamente à fluidez do jogo, trinta e quatro participantes disseram que a arquitetura *peer-to-peer* era fluída, enquanto que os outros seis disseram que sentiram alguma lentidão. Para além disso, dezassete deles disseram que a arquitetura cliente/servidor também era fluída, enquanto que os restantes vinte e dois disseram que era lenta. Um participante não respondeu a esta pergunta pois não conseguiu testar este modo (ver imagens 4.12).

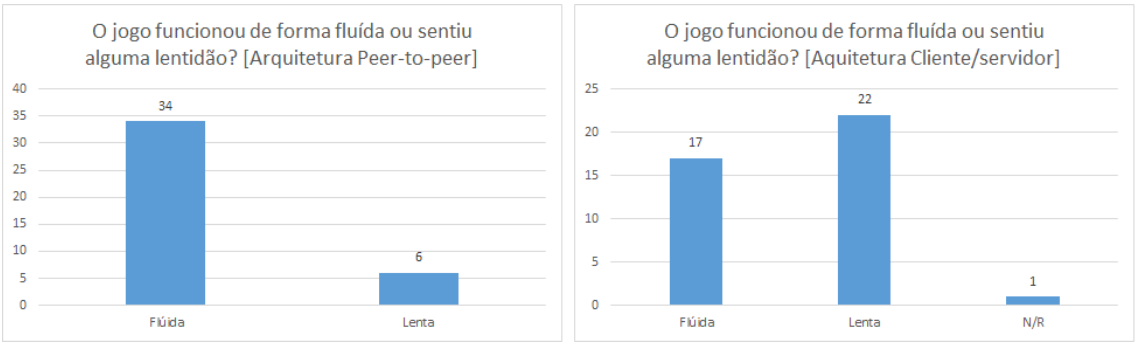

Figura 4.12: Gráficos relativos à lentidão/fluidez do jogo nas duas arquiteturas.

Já na última pergunta, a qual questionava se os participantes tinham notado alguma demora por parte das unidades em reagir quando era dada alguma ordem, vinte e seis referiram não terem sentido nenhuma lentidão, enquanto que catorze disseram que sim, relativamente à arquitetura *peer-to-peer*. Já na arquitetura cliente/servidor, dezasseis disseram que não sentiram nenhuma lentidão, enquanto que vinte e dois afirmaram que sim. Um participante não respondeu a esta

pergunta pois não conseguiu testar este modo (ver imagens 4.13).

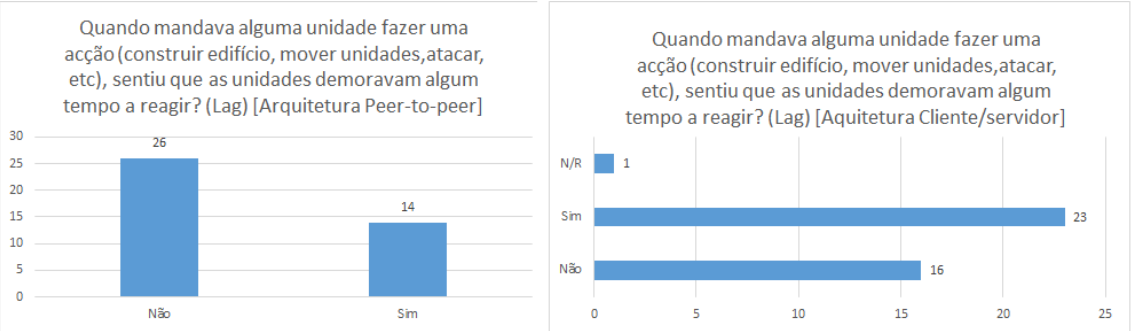

Figura 4.13: Gráficos relativos ao tempo de reação das unidades do jogo nas duas arquiteturas.

4.4 Conclusão

Neste capítulo foi descrito o teste do jogo realizado com quarenta utilizadores, de forma a tentar obter algumas conclusões sobre qual a melhor arquitetura de rede a ser usada. Foram também expostos os resultados obtidos com esse mesmo teste. No próximo capítulo serão tiradas algumas conclusões com base nos resultados obtidos e serão feitas algumas sugestões sobre o que poderá ser feito num trabalho futuro.

Capítulo 5

Conclusão e Trabalho Futuro

Este capítulo começa pela apresentação das principais conclusões retiradas dos testes efetuados no capítulo anterior, a partir do trabalho desenvolvido neste projeto. Após isso, são feitas algumas sugestões sobre o que pode ser realizado num trabalho futuro.

5.1 Principais Conclusões

Com base nos resultados obtidos através dos testes efetuados com quarenta participantes (secção 4.3), pode-se concluir que a arquitetura *peer-to-peer* do *Unity* funciona melhor para a maioria dos participantes, em que apenas uma pequena minoria sentiu alguma lentidão neste modo. Esta lentidão poderia não ser causada pela arquitetura de rede, mas pela ligação à Internet do participante ou por causa do participante estar a usar um computador com algumas limitações. Este resultado não é parecido com os resultados obtidos na arquitetura cliente-servidor, onde a maioria dos participantes disseram que este modo era lento.

De forma a tentar obter resultados mais conclusivos relativamente a estas duas arquiteturas, os participantes deveriam usar todos computadores iguais (ou seja, com especificações iguais) e usarem uma ligação à Internet igual, estando todos no mesmo local. Embora o servidor do jogo tenha estado alojado num servidor dedicado para este teste, usando uma ligação de *100mb* por segundo de *upload*, em que apenas uma pequena parte destes recursos foram usados, poderia ser usada uma ligação dedicada de *1gb* por segundo de *upload*.

O facto de o *Unity* apenas ter documentação e tutoriais de implementações usando a arquitetura *peer-to-peer* pode indicar que este seja a melhor arquitetura a ser usada na implementação de jogos multi jogador ou que deverá ser a arquitetura privilegiada pelos programadores.

5.2 Trabalho Futuro

Dados os objetivos iniciais para este trabalho conseguiu-se avaliar o problema encontrado inicialmente. Este relatório contém informação sobre como implementar um jogo de estratégia multi jogador *online*, usando dois tipos de arquitetura de rede, e um teste entre elas, tendo as conclusões sido referidas na secção anterior. Ainda existem algumas melhorias que podem ser feitas, como por exemplo alterações no sistema de salas, para que seja mais intuitivo para o jogador saber o que está a acontecer. Para além disso, poderia ser criado um sistema de pontos/níveis, em que os jogadores ganhavam experiência por cada jogo jogado, sendo, assim, de certa forma indicativo da qualidade de um jogador. Outra adição que poderia ser levada em conta é um sistema de *matchmaking* automático, isto é, os jogadores apenas têm que dar a instrução ao jogo que querem jogar e o jogo, com base no nível dos jogadores disponíveis (a partir da funcionalidade nova explicada anteriormente), juntá-los-á entre eles, de forma a que as partidas sejam equilibradas. Relativamente à comparação entre

os dois tipos de arquiteturas, poderá ser feito um teste usando um estilo de jogo diferente, como por exemplo o *MMORPG*, por ser um tipo de jogo que liga imensos jogadores ao mesmo tempo.

Relativamente ao jogo, neste momento existe um jogo de estratégia totalmente funcional, em que facilmente se adiciona outros modelos, mais unidades e edifícios e é possível lançar este jogo no mercado. Como trabalho futuro do jogo, falta concluir a implementação das funcionalidades referidas no *Game Design* criado para o jogo, tais como a implementação do herói, algumas magias para ajudar os heróis, um sistema de compras/venda de itens para o herói, algumas melhorias no sistema de *fog of war* e na grelha de construção e a implementação do som *3D*. É também preciso a ajuda de um designer para a criação de um sistema de *UI* mais intuitivo e apelativo para o jogador.

Bibliografia

[Bev13] Fernando Bevilacqua. Understanding steering behaviors: Leader following. http://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-leader-following--gamedev-10810, 2013. [Online; accessed 8-Fevereiro-2016]. 46

[BN13] M. Badar and T. Nikhil. Development guidelines for mobile multiplayer games. 2013. 6, 7, 8

[Car14] Attilio Carotenuto. Unity 3d ai: Navmesh navigation. https://www.binpress.com/tutorial/unity3d-ai-navmesh-navigation/119, 2014. [Online; accessed 5-Fevereiro-2016]. 44

[(ES15] Entertainment Software Association (ESA). Essential facts about the computer and video game industry. http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf, 2015. [Online; accessed 4-Janeiro-2016]. 5

[Far13] Sajid Farooq. Custom terrain brushes. http://answers.unity3d.com/questions/517169/custom-terrain-brushes.html, 2013. [Online; accessed 2-Fevereiro-2016]. 43

[HADF12] R. Humphrey, A. Allan, and G Di Fatta. Using spacial locality and replication to increase p2p network performance in mmo games. 2012. 7

[HMvdV+13] M. Hendrikx, S. Meijer, J. van der Velden, , and A. Iosup. Procedural content generation for games: a survey. 2013. 5

[Pow06] Michael Powers. Mobile multiplayer gaming, part 1: Real-time constraints. 2006. 8

[SEB+03] N. Sheldon, Girard E., S. Bord, M. Claypool, and E. Agu. The effect of latency on user performance in warcraft iii. 2003. 9

[Sta14] Statista. Leading game engines used by video game developers in the united kingdom (uk) 2014, 2014. Available from: http://www.statista.com/statistics/321059/game-engines-used-by-video-game-developers-uk/. 9

[Tar13] Sergey Taraban. Unity fog of war tutorial. http://www.gamedesignersvault.com/unity-fog-of-war-tutorial/, 2013. [Online; accessed 12-Fevereiro-2016]. 49

[Uni15a] Unity. Network Game Lobby (beta). https://www.assetstore.unity3d.com/en/#!/content/41836, 2015. [Online; accessed 1-Março-2016]. 57

[Uni15b] Unity. Terrain: Introduction to heightmaps. http://unity3d.com/pt/learn/tutorials/modules/intermediate/live-training-archive/introduction-to-heightmaps, 2015. [Online; accessed 3-Fevereiro-2016]. 43

[Uni15c] Unity. Using the networkmanager. http://docs.unity3d.com/Manual/UNetManager.html, 2015. [Online; accessed 19-Fevereiro-2016]. 52

[WJS09] A. I. Wang, M. Jarrett, and E. Sorteberg. Experiences from implementing a mobile multiplayer real-time game for wireless networks with high latency. 2009. 8