INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Relatório de Estágio na Empresa Biodroid Entertainment

Diogo Santiago Lopes
(Licenciado)

Trabalho Final de Mestrado para obtenção do grau de Mestre em Engenharia de Redes de Comunicação e Multimédia

Orientador:

Professor Coordenador Doutor Arnaldo Joaquim Castro Abrantes

Júri:

Presidente: Professor Adjunto Doutor Paulo Manuel Trigo Cândido da Silva

Vogais:

Investigador Doutor João Paulo da Silva P. G. Magalhães

Dezembro de 2014

Resumo

O presente documento descreve primariamente o trabalho desenvolvido por um aluno do Mestrado em Engenharia de Redes de Comunicação e Multimédia, do Instituto Superior de Engenharia de Lisboa (ISEL), no contexto de um estágio na empresa Portuguesa de videojogos *Biodroid Entertainment*.

Foi objectivo do estagiário integrar-se na equipa de programação para desenvolver protótipos de videojogos ou de algum dos seus subsistemas, com o propósito de investigar qual o melhor caminho a tomar para cumprir determinado requisito ou funcionalidade de um videojogo.

Serão descritas as ferramentas principais utilizadas no cumprimento do objectivo referido. Nesse aspecto, por um lado o estagiário experimentou ferramentas mais maduras como o motor *Unity*. Por outro lado, foi colocado na vanguarda do desenvolvimento de videojogos com a introdução ao motor relativamente recente *Unreal Engine 4*.

Adicionalmente, com os conceitos que o estagiário foi aprendendo ao longo do seu estágio, foi também desenvolvido um jogo paralelamente ao trabalho feito na *Biodroid*, cujo desafio era implementar um agente que utilizasse inteligência artificial para aprender as acções feitas pelo seu adversário, no contexto de um jogo de luta. Será feito um esforço para tentar aplicar o fluxo normal de desenvolvimento de um videojogo, passando pelas várias fases e aplicando metodologias das várias equipas responsáveis, mesmo sendo um trabalho individual.

Abstract

The current document describes primarily the work developed by a master student of Communication Network and Multimedia Engineering, from *Instituto Superior de Engenharia de Lisboa* (ISEL), within an internship in the Portuguese videogame company *Biodroid Entertainment*.

The intern's objective has been one of integrating himself in the programming team to develop prototypes of either a videogame or one of its subsystems, with the purpose of investigating which would be the best path to take to fulfil a certain game's requirement or functionality.

The main tools used in the fulfilment of the mentioned objective will be described. In that aspect, on one hand the intern tried out more stable tools, such as the *Unity* engine. On the other hand, he was placed at the vanguard of videogame development with the introduction of the relatively recent *Unreal Engine 4*.

Additionally, with the knowledge the intern gathered throughout his internship, a game was also developed outside of *Biodroid*. The challenge was to implement an agent that, through artificial intelligence, would learn the actions and moves performed by his opponent, within a fighting game. An effort will be made as to try to apply a normal videogame development workflow, passing through the various development phases and applying methodologies of the various teams, despite being an individual project.

Agradecimentos

Independentemente do teor individual de uma tese, seria injusto dizer que a completei sozinho. Como tal, sinto a necessidade de declarar alguns agradecimentos:

Aos meus colegas iselianos, sem os quais nunca teria chegado a este ponto da minha vida académica. Não tendo feito projecto na Licenciatura, devo agradecer não só aos meus colegas e amigos com quem ainda convivo, mas também àqueles que me acompanharam tanto tempo antes do Mestrado. Especificamente, agradeço ao Luís Tavares, ao Luís Marques, ao Laudelino Lima, ao Fábio Peixinho e ao Délio Amaral, pela amizade e pelas variadas discussões, académicas ou não, ao longo de todo o meu percurso.

A toda a *Biodroid*, por ter aceitado mais um estagiário para trabalhar no seu seio, ensinando-me muito mais do que eu poderia esperar aprender em tão pouco tempo. Em específico, gostaria de agradecer ao João Magalhães por me ter orientado na minha estadia enquanto estagiário e por ter aceitado arguir este relatório de estágio.

À minha família, o meu suporte básico de vida, cujo apoio deu-me a força mental e física para elaborar todo este trabalho.

Ao José Amaral, o grande responsável pela aventura que foi trabalhar, elaborar e escrever este trabalho, em todas as suas componentes.

À minha mãe, pelas preocupações constantes pelo meu bem-estar geral e pelo seu apoio incondicional.

Ao meu pai, fonte infinita de sabedoria, na forma dos seus conselhos preciosos, e de amizade, na forma do seu apoio estóico nas situações mais caóticas.

Ao Carlos Junior, inicialmente um colega, depois um amigo e finalmente um irmão. Por ter sido como uma pedra basilar desde o início do Mestrado até aos derradeiros momentos finais da elaboração deste trabalho.

Índice

Índice de Figuras

"The secret of life is honesty and fair dealing. If you can fake that, you've got it made."

Anónimo

1 Introdução

Este relatório de estágio feito no contexto do Mestrado em Engenharia de Redes de Comunicação e Multimédia tem como objectivo descrever o percurso do autor do documento enquanto estagiário na empresa Biodroid Entertainment, desde Outubro de 2013 até Setembro de 2014.

Foi objectivo do autor participar no desenvolvimento de várias componentes de videojogos, fossem elas ferramentas auxiliares ou implementação de comportamentos e mecanismos para cumprir as regras de um videojogo.

1.1 Motivação

No sentido de reforçar a parceria entre o ISEL e a *Biodroid Entertainment*, foi proposta a possibilidade de um aluno de mestrado realizar a sua tese trabalhando na *Biodroid* como estagiário. O presente documento, no formato de um relatório de estágio, é o resultado dessa experiência.

Tendo em conta os conhecimentos adquiridos pelo estagiário no decorrer da Licenciatura de Engenharia de Redes de Comunicação e Multimédia e respectivo Mestrado, considerou-se que este teria as aptidões necessárias para realizar trabalho de programação nesta empresa de videojogos.

O objectivo inicial do estagiário foi de criar protótipos funcionais de jogos. Estes protótipos seriam guardados para eventual desenvolvimento futuro, no caso da ideia a ser testada ser suficientemente apelativa, em termos de potencial mercantil e exequibilidade.

1.2 Objectivos

O objectivo fundamental de qualquer estagiário é aprender um conjunto de capacidades que permita a sua eventual integração numa equipa de desenvolvimento de um produto. Como tal, no decorrer do estágio, partindo de uma base de conhecimento adquirida academicamente, o estagiário desenvolverá estas capacidades específicas à indústria dos videojogos, com o intuito de as aplicar profissionalmente. Esta aprendizagem envolverá não só conceitos de

programação de alto e baixo nível, mas também alguns conhecimentos relevantes na área da arte, nomeadamente os requisitos e limitações que afectarão os artistas no seu trabalho.

Adicionalmente, de forma independente, aplicar-se-ão os conhecimentos aprendidos para construir um protótipo para um jogo de luta, que tentará responder ao desafio de se é possível implementar um agente artificial que aprenda a lutar com as acções realizadas pelo humano, utilizando conceitos da área de inteligência artificial.

1.3 Resumo do trabalho

De uma forma geral, a totalidade do trabalho é bipartida.

Por um lado, tem-se o estágio a realizar na *Biodroid*, que é o foco principal deste documento. O estagiário teve variadas funções no decorrer do seu trabalho, o que lhe permitiu entrar em contacto com um elevado e abrangente número de tecnologias e conceitos relevantes, tanto a nível de programação como de arte, *game design* e produção.

Todo este conhecimento ajuda a enriquecer o contexto académico do estagiário, pois as experiências por ele vividas são, através deste documento (e não só), comunicadas aos interessados na área, sejam eles alunos ou professores.

A segunda grande parte do documento revolve sobre o protótipo referido nos Objectivos (ver acima). Para o desenvolvimento deste, propôs-se aplicar o mesmo fluxo de desenvolvimento que se aplicaria a um qualquer outro jogo, passando pelas várias fases de produção e simulando as várias equipas responsáveis pelo desenvolvimento.

1.4 Organização do documento

O documento tem a seguinte organização:

- **Introdução**: introduz-se o documento e o contexto do mesmo, explicitam-se os objectivos propostos, resume-se de modo geral as componentes maiores do trabalho realizado e apresenta-se a organização do documento pelos seus capítulos, sendo este texto integrante dessa parte.
- **Empresa e Negócio**: apresenta-se a estrutura da *Biodroid Entertainment* em termos das maiores divisões por equipas de desenvolvimento e explicam-se as tarefas

específicas existentes no seio de cada um desses núcleos. Adicionalmente, dá-se uma breve explicação sobre as principais fases de desenvolvimento de um videojogo.

- **Ferramentas utilizadas**: seria uma tarefa hercúlea descrever todas as tecnologias usadas no desenvolvimento de jogos, até porque podem variar de projecto para projecto. Consequentemente, neste capítulo descrevem-se apenas as ferramentas mais importantes directamente utilizadas pelo estagiário.
- **Estágio**: neste capítulo descreve-se com o máximo rigor possível as actividades realizadas ao longo do estágio. Como já foi referido, as actividades em que o estagiário esteve envolvido abrangeram diversos temas distintos uns dos outros, e portanto este capítulo tem várias divisões que parecerão descorrelacionadas, exactamente porque o são.
- **Protótipo Independente**: neste capítulo descreve-se extensivamente um protótipo para um jogo de luta, explicitando as várias fases de desenvolvimento e especificando pormenores tanto da construção dos conteúdos visuais como da implementação do código de programação.
- **Conclusão**: tiram-se as conclusões sobre o aproveitamento do estagiário de todo o trabalho realizado, tendo em conta a experiência, conhecimentos, capacidades, maturidade e outros eventuais benefícios que possam ter sido obtidos.

2 Empresa e Negócio

Neste capítulo ir-se-ão resumir os componentes chave de uma empresa de videojogos, tendo como base a *Biodroid*, explicando as equipas e fases típicas de desenvolvimento.

2.1 Organização

A estrutura empresarial da *Biodroid* assenta-se principalmente no desenvolvimento de videojogos, tendo paralelamente um departamento de aplicações móveis. O estagiário foi inserido no desenvolvimento de videojogos. Os intervenientes deste desenvolvimento concentram-se em cinco secções principais:

- Arte;
- Game Design;
- Produção;
- Programação;
- Garantia de Qualidade.

2.1.1 Arte

Os artistas são responsáveis pela criação de uma quantidade de conteúdos gráficos que serão usados como arte conceptual ou como partes integrantes do produto final.

Dependendo do jogo, a quantidade de conteúdos a construir pode variar. Por um lado pode ser apenas necessário criar alguns modelos simples de personagens com poucas animações, alguns objectos e um nível pequeno e simples. Por outro, pode ser preciso criar conteúdos mais complexos, com cenários de muito maior dimensão, com terreno, vegetação, decorações; personagens variadas com aspecto e animação mais realistas; texturas com um nível de detalhe superior; efeitos visuais complexos, etc.. Dito isto, a dimensão da equipa artística pode variar de projecto para projecto, podendo bastar um único artista para um projecto pequeno, e uma equipa inteira de artistas para projectos mais ambiciosos, cada um com tarefas específicas.

Neste último caso, a equipa de arte seria composta tipicamente por um *lead artist* que é responsável por definir o aspecto geral que o jogo deverá ter e garantir que os conteúdos estão

a ser criados na direcção artística correcta. Tipicamente os restantes membros da equipa estão divididos em duas áreas principais:

- 2D: esta área está mais relacionada com conteúdos no formato de imagens, ou conteúdos a duas dimensões, para generalizar. Inclui principalmente:
    - **Artistas conceptuais**: o objectivo destes artistas é criar representações de cenários, objectos, personagens, terrenos, etc., que são usados por vários membros da equipa ou para ter noções iniciais do aspecto do jogo ou para construir os modelos 2D ou 3D requeridos;
    - **Artistas de interface**: responsáveis por criar as imagens que serão usadas nos componentes da interface gráfica do jogo, tais como botões, menus e afins;
    - **Artistas de texturas**: criam as texturas que são aplicadas aos modelos do jogo;
    - **Artistas de *sprites***: criadores de imagens de personagens e objectos para jogos 2D.
- 3D: esta área foca-se na construção de modelos, animações e ambientes a três dimensões. Os seus constituintes são:
    - **Modeladores:** artistas que criam as malhas 3D dos personagens e objectos do jogo;
    - **Animadores**: responsáveis por adicionar animações às malhas 3D, o que pode implicar deformar a malha com um sistema de ossos (exemplo: personagens) ou adicionar-lhe efeitos de movimento para interacção (portas, armadilhas, etc.);
    - **Artistas de nível**: criadores dos níveis onde o jogo decorrerá, que pode envolver a construção de terreno, edifícios, posicionamento de luzes, decorações, colocação de personagens, entre outras actividades.

Alguns artistas podem realizar mais do que uma das tarefas atrás referidas, inclusivamente podendo fazer trabalho tanto de 2D como de 3D (por exemplo, um artista pode modelar um personagem e texturizá-lo de seguida).

Os artistas têm de trabalhar em conjunto com os *game designers*, especialmente aqueles com influência mais directa no jogo, como é o caso dos criadores de níveis para garantir as características correctas dos conteúdos consoante os objectivos propostos para o nível (posicionamento, quantidade, etc.).

2.1.2 Game Design

Estabelecem o conceito, as regras e as mecânicas do jogo. As suas responsabilidades são abrangentes, listando algumas:

- definição dos controlos do jogo, através dos quais o jogador interage com o sistema;
- determinação da resposta do jogo para cada acção do jogador;
- definição do comportamento pretendido para as várias mecânicas programadas, tais como física, inteligência artificial, interacções entre objectos, etc.;
- construção dos ambientes de jogo em conjunto com os artistas;
- escrita da narrativa/história do jogo, caso aplicável;
- ajustamento das variáveis associadas às várias mecânicas do jogo até convergirem para um balanço adequado ou no mínimo aceitável;
- construção do *Game Design Document* (GDD).

Um *game designer* poderá ter várias tarefas e inclusivamente pode ter algumas funções de arte ou mesmo de programação. É comum um *game designer* desenhar níveis conceptuais para serem criados. Por outro lado, pode ser possível colocar um *designer* a programar numa linguagem de *scripting* de muito alto nível que não envolva conhecimentos profundos de programação. De facto, algumas plataformas ajudam nesse sentido, como é o caso do *Unreal Engine 4*, que fornece programação através de *Blueprints*, como será referido (página 11).

2.1.3 Produção

Os produtores são responsáveis pela gestão do bom desenvolvimento de um videojogo. São o principal meio de comunicação entre a equipa de desenvolvimento (através dos seus líderes) e intervenientes externos, tais como os investidores. São responsáveis por delimitar prazos às várias equipas de desenvolvimento. Isto não significa que não possam fazer parte de uma dessas equipas, mas para projectos de maior dimensão é provável que toda a gestão necessária ocupe grande parte do tempo de um produtor. Certificam-se da qualidade e do cumprimento dos prazos e objectivos do jogo, ou das suas entregas intermédias.

2.1.4 Programação

Implementam todo o código necessário para satisfazer os requisitos descritos no GDD. Pela sua aproximação às tecnologias de informática, são os mais aptos para aconselhar a restante

equipa sobre o que é possível fazer em termos de capacidade computacional. Também aqui podem haver tarefas específicas a cada programador, tais como:

- **física**: definição de um motor de forças e colisões que os objectos em cena podem utilizar; este tema pode envolver conhecimentos profundos de física e de simulação;
- **inteligência artificial (IA):** a implementação dos comportamentos pretendidos para os jogadores artificiais, sendo um tema que pode requerer uma considerável quantidade de trabalho pela complexidade dos processos associados à inteligência artificial;
- **interface gráfica**: trabalhando em conjunto com *designers* e artistas, o programador de interface gráfica tipicamente define a ligação entre cada interacção e a função correspondente no motor do jogo;
- **mecânicas de jogo (*gameplay*):** pode-se considerar este tema como sendo mais genérico, pois a implementação das mecânicas do jogo pode abordar vários assuntos gerais de programação, sendo também que pode não haver uma linha bem definida entre o que é programação das mecânicas e programação doutro tema, como a IA;
- **motor gráfico**: esta tarefa implica trabalhar a um nível mais baixo de programação, particularmente perto da placa gráfica e de linguagens de *shading*; implica também conhecimento profundo de álgebra devido à complexidade e eficiência necessárias dos métodos utilizados num renderizador típico.

2.1.5 Garantia de Qualidade

A garantia de qualidade de um jogo é realizada primariamente pelos testadores de jogos. O objectivo destes é encontrar com rigor o número máximo de erros possível durante o desenvolvimento, e com particular atenção na fase final antes do lançamento/entrega final. Um testador documenta os erros encontrados, dando o máximo de informação possível à equipa responsável pelo problema.

2.2 Fases de desenvolvimento

As fases principais do desenvolvimento de videojogos são listadas a seguir, sendo que cada uma tem subfases possíveis:

- Pré-produção;
- Produção;
- Pós-produção.

2.2.1 Pré-produção

Nesta fase visiona-se toda a ideia do jogo, quais as suas regras e características principais. É uma fase em que todas as equipas do desenvolvimento preparam uma eventual fase de produção. Os produtores ocupam-se com a definição de documentos a escrever e com horários e prazos para o desenvolvimento vindouro. A equipa de arte cria modelos e imagens conceptuais para preparar a restante equipa para a aparência do jogo. A equipa de programação foca-se em criar protótipos e tecnologia que suporte ou exclua as ideias propostas. Estas ideias estarão descritas no *Game Design Document*, ou GDD, que deve ser o foco principal da equipa de *design* nesta fase.

O GDD é um documento criado inicialmente na fase de pré-produção, mas que é constantemente editado ao longo do projecto, à medida da sua evolução. É o documento central para todas as ideias a implementar, incluindo assuntos como: história, personagens, cenários, mecânicas, interface gráfica, som, inteligência artificial, controlos, entre outros. Um GDD pode variar muito de projecto para projecto, pois alguns dos aspectos mencionados podem nem ter relevância. A importância do documento prende-se pelo facto de ser um esforço conjunto de todas as equipas de desenvolvimento, e portanto toda a informação necessária sobre o jogo deve poder ser retirada de lá.

2.2.2 Produção

Assumindo que todos os requisitos estão cumpridos (em termos financeiros, tecnológicos e de recursos humanos), inicia-se a fase de produção. É a fase mais longa do desenvolvimento, não sendo coincidência o facto de ser a fase onde todos os conteúdos são criados, tanto em termos de arte como de programação.

A não ser que o conteúdo a criar seja parecido a conteúdo de jogos anteriores, a equipa de arte terá de construir os modelos e texturas, sob supervisão rigorosa dos supervisores, nomeadamente os produtores e o(s) artista(s) *lead*.

A equipa de programação implementará todos os comportamentos requeridos, podendo ou não aproveitar código desenvolvido em protótipos na fase de pré-produção. Assim que os objectivos iniciais estiverem cumpridos, será função da equipa corrigir erros descobertos por testadores e de adicionar eventuais funcionalidades novas propostas pelos *designers*.

Os *game designers* continuam a evoluir o GDD, adicionando-lhe novas informações e ideias de implementação ou, pelo contrário, retirando conceitos que se determinaram obsoletos.

Os produtores garantem o bom desenvolvimento do jogo, gerindo os prazos para a equipa e supervisionando os conteúdos que vão sendo criados pelas várias equipas.

Por último, os testadores têm a responsabilidade de encontrar escrupulosamente os erros, falhas e incoerências em todos os aspectos do jogo, com maior ênfase para os comportamentos implementados pela programação. É um trabalho moroso pois implica testar várias vezes o mesmo ambiente ao serem adicionadas pequenas alterações, para garantir que essas alterações não tiveram outras consequências indesejadas.

2.2.3 Pós-Produção

O trabalho de pós-produção é essencialmente de manutenção do jogo depois de acabado e lançado, corrigindo, se a plataforma-alvo o permitir, erros que só tenham sido detectados posteriormente.

3 Ferramentas utilizadas:

Neste capítulo vão-se descrever algumas das principais ferramentas utilizadas no decorrer de todo o trabalho descrito no documento.

3.1 Unity

O *software Unity* é uma plataforma de desenvolvimento de jogos para várias plataformas. Está preparado para ser o mais abrangente possível em termos das escolhas que os criadores de jogos podem fazer. Quer-se com isto dizer que os desenvolvedores de jogos podem construir os conteúdos todos num qualquer programa de modelação/animação 3D e importá-los para o editor do *Unity*. Outrossim, o jogo pode ser publicado para um conjunto de plataformas diferentes, sendo que segue o paradigma de *"write once, run everywhere"*, ou seja, independentemente da linguagem de programação usada e dos formatos dos conteúdos importados, a construção e exportação dos jogos é abstraída para o criador, bastando seleccionar a plataforma pretendida.

O IDE do *Unity* permite fazer apenas operações básicas de transformação aos objectos importados, considerando que estes já devem ter sido construídos externamente. O objectivo no IDE é então de construir os níveis dos jogos, colocando os seus vários elementos, tais como estruturas, personagens, decorações, luzes, etc**.**.

Descrevem-se de seguida algumas das funcionalidades disponíveis na plataforma:

- **Motor de física**: os objectos na cena, ou ambiente de jogo, podem ter associada uma componente de física, e a partir desse momento passam a fazer parte dos cálculos do motor de física, podendo então ser afectados por gravidade, colisões, atrito, e outros efeitos, existindo um conjunto decente de parâmetros controláveis;
- **Múltiplas linguagens de programação**: um programador pode escolher a linguagem em que irá programar os seus *scripts*. As opções actuais são C#, *Javascript* ou *Boo* (baseado em *Python*). O *Unity* permite que um mesmo projecto use *scripts* escritos em linguagens diferentes, permitindo, por exemplo, que mais que um programador com conhecimentos diferentes possa trabalhar no mesmo projecto;

- ***Mecanim***: um mecanismo de controlo de animações de personagens humanóides com uma armação associada. Permite criar camadas de animações para definir que animação afectará uma certa parte do corpo do personagem;
- ***NavMesh***: Navegação de personagens: cria uma área de navegação que os agentes de jogo podem utilizar para determinar o melhor caminho para um certo ponto na área;
- Outros mecanismos e efeitos: partículas, rastos, tecidos, etc..

O *Unity* é um *software* maioritariamente grátis. No entanto, algumas funcionalidades mais avançadas só estão disponíveis na versão Pro, que requer a aquisição de uma licença. A maioria destas funcionalidades limitadas são estéticas, o que implica que se pode fazer a mecânica de um jogo utilizando todo o poder da plataforma, independentemente da versão. Adicionalmente, as licenças de certas plataformas-alvo (*iOS*, *Android*) de exportação têm de ser compradas à parte por um custo mais elevado.

Como já foi referido, uma das vantagens da utilização do *Unity* é a possibilidade de construir um mesmo jogo para múltiplas plataformas, mantendo quase todos os elementos iguais no ambiente de desenvolvimento (tendo em conta mesmo assim que o hardware entre plataformas pode ser diferente; felizmente o *Unity* tem opções para ajustar a complexidade de um jogo para plataformas distintas). O tipo de plataformas possíveis é muito abrangente, incluindo a exportação para *web browsers* (*Unity Web Player*), dispositivos móveis (*tablets*, *smartphones*), computadores (PC, *Linux*, *Mac*) e, por fim, consolas (PS3, Xbox, Wii-U).

3.2 Unreal Engine 4

O *Unreal Engine* é outra plataforma de desenvolvimento de jogos. Actualmente está na sua quarta iteração, com o *Unreal Engine 4* (UE4), que tem como alvo as gerações mais recentes de consolas (apesar de ser possível exportar um jogo para outras plataformas também). Como o *Unity*, este motor de jogo também abstrai o criador de uma quantidade de componentes como física, inteligência artificial, som, programação, etc.. No caso da programação, o UE4 tem duas particularidades relevantes:

- O código fonte está disponível, sendo possível modificá-lo e recompilá-lo para implementar novos módulos do motor ou alterar os existentes, caso seja necessário;

- A programação das mecânicas de jogo pode ser feita através de uma linguagem visual de alto nível chamada "*Blueprints*", reduzindo a necessidade de conhecimentos formais de programação para se criarem jogos.

Como os padrões de qualidade deste motor são maiores comparativamente aos do *Unity*, o poder computacional necessário para usar o IDE também é maior. Por outro lado, o licenciamento do *Unreal* é feito mensalmente e é mais barato.

3.3 *Git*

O *Git* é um sistema distribuído de controlo de versões. O objectivo de um sistema de controlo de versões é de gerir vários tipos de informação, permitindo manter várias versões (ou revisões) de um conjunto de documentos ou ficheiros. Estas versões podem ser comparadas, restauradas a versões anteriores ou fundidas numa só. O facto de o *Git* ser distribuído implica que as versões mais actualizadas dos ficheiros não estão num único local central. Na verdade, todos os membros desse repositório terão uma cópia local dos ficheiros, permitindo que haja falhas nalgum nó da rede distribuída sem perda das versões. Como os dados são sempre locais, a única comunicação que existe entre membros do repositório são as alterações feitas nos ficheiros modificados (no limite pode ser o repositório inteiro no caso de um membro novo do repositório). Todas as outras operações são rápidas pois não envolvem nenhum servidor central para actualizar.

3.4 *OpenGL*

O *OpenGL* (*Open Graphics Library*) é uma API (*Application Programming Interface*) independente da plataforma e da linguagem, utilizada para realizar renderizações através da placa gráfica. As funções definidas pela API só têm o objectivo de desenhar gráficos e não de os representar num visualizador, nem de controlar interacções com o utilizador.

Para complementar esta API foram criadas bibliotecas adicionais:

- **GLUT (*OpenGL Utility Toolkit*)**: biblioteca primariamente usada para gerir as interacções I/O com o sistema operativo e com o utilizador, permitindo a construção de janelas de visualização dos gráficos gerados pelo *OpenGL* e a detecção de *input* do teclado e rato. Sendo que o GLUT foi descontinuado, a biblioteca que é estável

denomina-se de *freeglut*, totalmente baseada no GLUT, corrigindo apenas alguns erros e não adicionando funcionalidades novas.

- **GLFW**: como o GLUT, é uma biblioteca de gestão de I/O, mas é mais leve e mais recente que o GLUT e *freeglut*.

3.5 *MinGW*

O *MinGW* (*Minimalist GNU for Windows*) é uma versão do sistema operativo GNU feito para funcionar com Windows. Uma das utilidades incluídas neste ambiente de desenvolvimento é o compilador GCC para C++. No contexto da empresa, a equipa de programação não estava satisfeita com o compilador fornecido pelo *Visual Studio* da *Microsoft* na sua versão mais recente, pois faltavam-lhe alguns mecanismos de C++ que eram úteis. O *MinGW* pode utilizar a versão de C++ que se pretender.

4 Estágio

Neste capítulo desenvolver-se-á sobre o estágio realizado na *Biodroid*, especificando os projectos e outras actividades levadas a cabo pelo estagiário.

4.1 Protótipo Inicial:

Apesar da função do estagiário ser de programação, este não foi imediatamente inserido na equipa de programação. Ao invés disso, o estagiário foi colocado a criar um protótipo de um jogo. Este protótipo inicial seria desenvolvido por uma equipa pequena, composta pelo estagiário, autor deste documento, por um *game designer*, também estagiário, por um artista e por um orientador, que neste caso foi o *Lead Game Designer* da empresa.

Como programador único deste projecto, seria função do estagiário implementar toda a mecânica desejada para o jogo. Isto requereu um pequeno período de adaptação à ferramenta a utilizar, que neste caso foi a plataforma *Unity*, e por conseguinte, a linguagem de programação associada, o C#. Durante este período de aproximadamente uma semana, o *game designer* construiu a ideia do jogo de modo a poder fornecer um GDD.

Note-se que este projecto não tem como objectivo a criação de um protótipo no sentido esperado do conceito. Este protótipo deverá ser completo o suficiente não só para provar o funcionamento das mecânicas base, mas também para poder estar pronto para ser lançado para produção, caso seja adequado para esse fim. Neste caso também serviu como treino para ambos os estagiários envolvidos.

Neste jogo desenvolvido o jogador tem o controlo de um personagem alienígena que tem como objectivo roubar bolos de uma fábrica, resolvendo puzzles e evitando os inimigos. O jogo é pensado para plataformas móveis, e por isso a interacção possível pelo utilizador é limitada. O utilizador poderá apenas direccionar o seu personagem pelos níveis, em princípio através de toque. Como a plataforma de desenvolvimento utilizada é um computador, simulou-se o toque com o clique do rato normal.

4.1.1 Movimento e Interacção

O movimento do personagem é dirigido por um plano (uma sequência de movimentos) construído pelo jogador. Esse plano tem representação visual no nível, e o jogador pode mudá-lo, criando um novo plano, consoante os factores dinâmicos do nível.

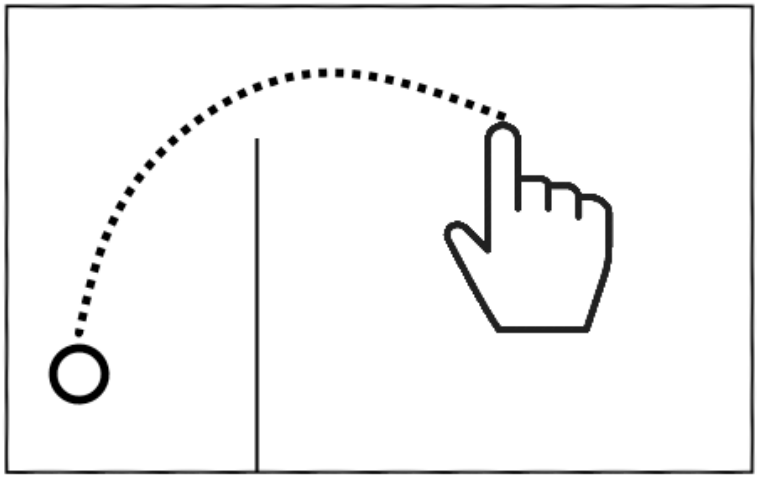

Figura 4.1: Representação de uma sequência de movimentos (plano)

A interacção do jogador com o ambiente é feita através de um mecanismo de *RayCast*, em que, metaforicamente, um raio é lançado consoante um vector, implicando uma certa direcção e magnitude. A direcção é a mesma da câmara de jogo (a partir da qual o jogador observa o ambiente). A magnitude pode ser considerada um número praticamente grande (na ordem dos milhares), apenas para simular um vector de magnitude infinita. O objectivo disto é assegurar que o raio colida com um qualquer objecto do nível.

Em maior parte dos casos, o objecto com que o raio irá colidir será o chão. Este facto é essencial para o jogador poder estabelecer o plano de movimento. Este plano é uma sequência de posições que o personagem segue iterativamente, convergindo sempre para a seguinte. Por motivos de complexidade computacional, esta sequência tem de ter um grau de discretização suficientemente grande para o seu processamento ser viável em aplicações móveis. Tomando a Figura 4.1 acima como referência, nota-se que a sequência não contínua de pontos fornece uma aproximação satisfatória para o trajecto do personagem.

Sendo um jogo com uma grande componente de puzzle, haverão outras interacções possíveis com os objectos em cena. Exactamente como o chão, estas interacções serão activadas pelo mecanismo de *RayCast*, sendo que cada objecto atingido por um raio terá um comportamento diferente definido, aproveitando o paradigma de programação orientada a objectos. Alguns exemplos de objectos com interacção disponível são o bolo (o objectivo do jogo) e as caixas (utilizadas para vários fins).

4.1.2 Inteligência Artificial

O jogo tem inimigos que patrulham os níveis. O comportamento desejado para estes inimigos é que, ao detectarem o personagem do jogador, o persigam com o intuito de o apanhar e terminar o jogo. O inimigo poderá perder o personagem de vista, colocando-o numa fase de procura atenta. Caso não o encontre depois de um certo tempo, voltará à sua patrulha normal. Torna-se clara a necessidade de inteligência artificial para implementar estes comportamentos. Para os inimigos, este problema divide-se em duas partes:

- **Movimento**: o inimigo tem de conseguir navegar pelo nível evitando naturalmente os obstáculos e, se possível, tomar o melhor caminho para um destino. Para este tipo de problemas é típico usarem-se mecanismos de procura em espaço de estados.
- **Acções**: consoante a informação sensorial que tiver, um inimigo deve poder tomar decisões sobre que acção tomar. Para o nível de complexidade que se pretende dos inimigos, um mecanismo de estímulo-reacção para cada percepção sensorial do nível será suficiente.

Abordar a primeira parte do problema é mais simples, especificamente pelo facto da plataforma *Unity* já fornecer uma ferramenta que abstrai todo o processo de procura em espaço de estados. A *NavMesh* é uma área de jogo pré-calculada (*baked*) sobre a qual um objecto do jogo pode realizar pedidos de resolução de caminhos de um ponto a outro. Esta malha de navegação é criada a partir dos objectos já presentes em cena, particularmente a malha que estiver a ser utilizada como chão do nível. Os restantes objectos podem ser considerados relevantes ou não para o cálculo. A Figura 4.2 mostra um exemplo de uma malha de navegação possível.

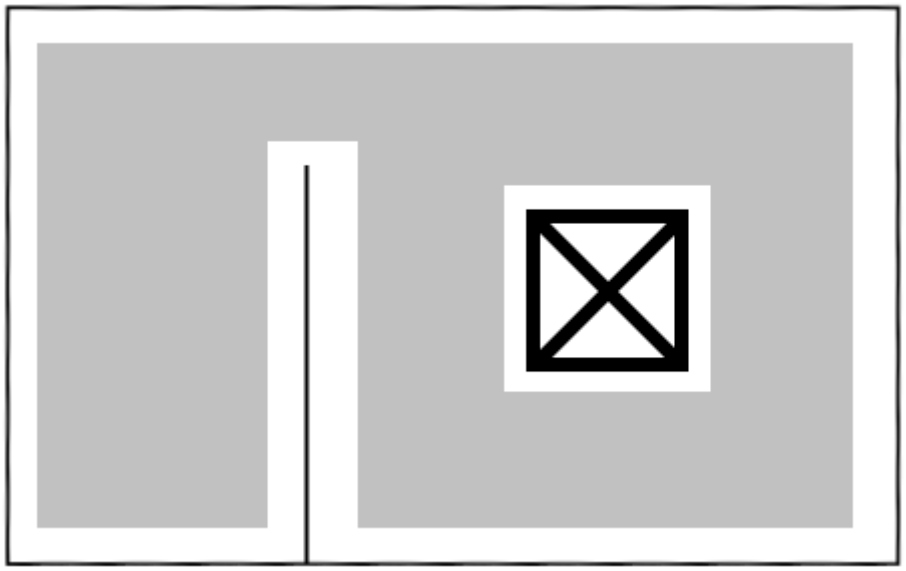

Figura 4.2: Exemplo de uma área de navegação (a cinzento); o quadrado negro no meio representa um qualquer obstáculo

Pode-se notar na figura que existe uma certa margem para os obstáculos e paredes por parte da malha de navegação. Esta é uma das várias propriedades mutáveis na configuração do cálculo.

Em termos práticos, para o movimento do inimigo resta apenas usar alguns elementos fornecidos pela API do *Unity*. O objecto de jogo que representa o inimigo terá de ter um componente *NavMeshAgent* de modo a utilizar a malha de navegação calculada. É através deste componente que se irá definir em código qual o destino pretendido pelo agente. Todo o processo restante de movimento será da responsabilidade do motor de jogo.

Contudo, como foi dito, é necessário determinar um destino para o qual se irá calcular um caminho. O fornecedor deste destino será a outra parte da inteligência artificial do inimigo, ou seja, o mecanismo de estímulo-reacção.

A arquitectura para esta parte é simplesmente uma máquina de estados. O inimigo encontra-se num estado inicial e consoante os eventos que detecta irá transitar de estado, como se mostra na Figura 4.3.

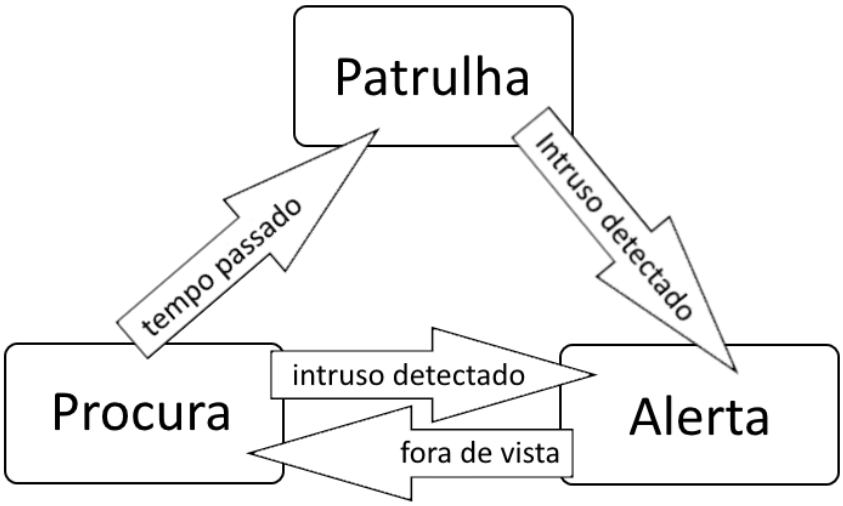

Figura 4.3: Máquina de estados do agente inimigo

O inimigo tem três estados possíveis:

- **Patrulha**: este é o estado inicial por predefinição. O inimigo segue um caminho pré-estabelecido repetidamente. Ao detectar um intruso, transita para o estado de alerta;
- **Alerta**: o inimigo descobriu um intruso e persegue-o enquanto o tiver em vista. Se o alvo estiver demasiado tempo fora da sua visão, transita para o estado de procura;
- **Procura**: Durante um período de tempo, o inimigo procura nos arredores do último sítio onde viu o intruso. Se o encontrar, transita de novo para o estado de alerta. Senão, depois do período de tempo passar, transita para o estado de patrulha.

A patrulha é estabelecida como um conjunto ordenado de pontos de referência no nível. O inimigo seguirá iterativamente estes pontos. Para tal, bastará utilizar a malha de navegação através da API do *Unity*, como já foi explicado, definindo progressivamente o novo destino. De igual forma, quando o inimigo está em estado de alerta, terá de definir o seu destino como sendo a posição do intruso. Este destino será o utilizado pelo NavMeshAgent (referido acima na página 17).

Nos três estados há um conceito fulcral para o funcionamento da inteligência artificial: a visão. Esta é conseguida através de algumas operações vectoriais. Existe um conjunto de verificações necessárias para determinar se um inimigo consegue ou não ver um objecto.

A visão do inimigo terá uma certa abrangência limitada, caracterizada por um ângulo e por uma distância. Esta primeira verificação trata de determinar se um certo objecto está contido nesse cone de visão. Para isso, calcula-se o vector existente entre o inimigo e o alvo.

$$V_{diff} = P_{alvo}(x, y, z) - P_{inimigo}(x, y, z)$$

A ordem de subtracção é importante, pois vai-se calcular o ângulo Θ existente entre V_{diff} e $V_{frontal}$, que é o vector representante de frente em termos relativos (ou locais) do inimigo, o que implica que V_{diff} tem de estar correctamente direccionado. Este ângulo calculado e a magnitude do vector V_{diff} serão utilizados para esta verificação. Formalmente:

$$\Theta \leq \Theta_{inimigo} \wedge mag(V_{diff}) \leq dist\_visão_{inimigo}$$

$\Theta_{inimigo}$ é o ângulo de visão máximo definido para o inimigo, sendo que $dist\_visão_{inimigo}$ é o análogo mas para a distância de visão máxima. A Figura 4.4 mostra figurativamente esta verificação.

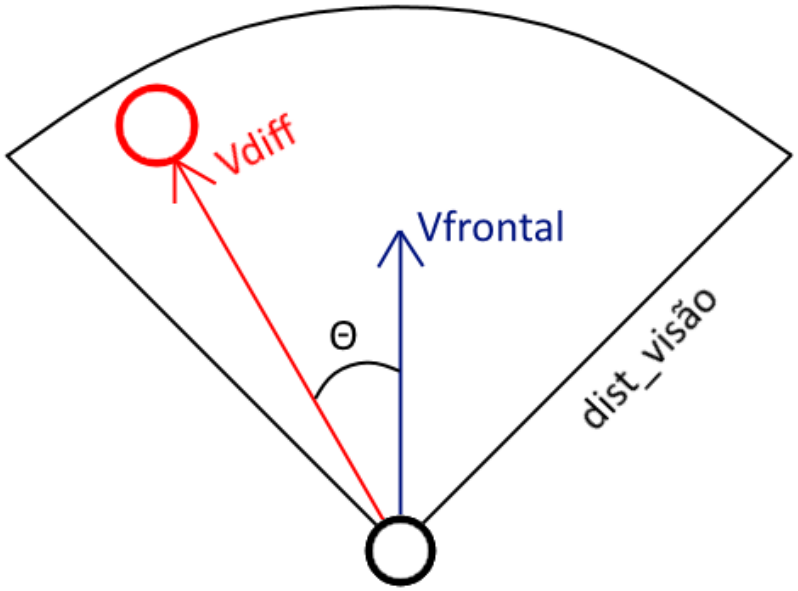

Figura 4.4: Representação da visão do agente inimigo

Estes cálculos não são suficientes para todas as situações, pois não têm em conta eventuais obstáculos entre o inimigo e o alvo que inibissem a visão. Assumindo que a primeira

verificação se valida, poder-se-á então lançar um raio (*RayCast*) na direcção de V_{diff} e determinar se a colisão que ocorre se trata do alvo ou de um outro objecto.

Se as duas verificações forem válidas, o personagem do jogador é considerado como detectado.

4.1.3 Controlo de Jogo

É usual existir um módulo num jogo para implementar o seu controlo e a sua lógica de funcionamento geral. Neste módulo, chamado de controlador, estão explicitadas as condições e os mecanismos de vitória ou derrota. Também armazena informações sobre o estado global do jogo, se necessário. Por exemplo, pode guardar a informação de se a porta final (para acabar o nível) está fechada ou aberta, sendo que esta variável de jogo será modificada quando o jogador acabar todos os objectivos do nível. Pode também ser responsável pelo controlo do tempo, possibilitando a pausa e retoma do jogo.

4.1.4 Conclusão do protótipo

O jogo, na sua versão actual, ficou feito no espaço de um mês de trabalho, quando se decidiu que o estado em que estava era suficiente como prova de conceito para um jogo do género. Serviu como um excelente método de aprendizagem para a plataforma Unity, mas mais relevante foi a vaga noção que forneceu do método de trabalho típico no desenvolvimento de um jogo, que viria a provar-se útil nas fases finais do estágio.

4.2 Ferramentas Auxiliares

Numa equipa de desenvolvimento de jogos é usual construírem-se ferramentas de suporte proprietárias para auxiliar nalgum aspecto do desenvolvimento. Estas ferramentas poderão não ser só usadas por programadores, o que implica que deve haver cuidado na usabilidade (facilidade de utilização) da ferramenta.

Durante um período de tempo, o estagiário esteve envolvido na implementação de algumas ferramentas deste género. Este trabalho vai em desacordo com o propósito inicial do estágio, ou seja, trabalhar em protótipos. No entanto, deve-se ter em conta que foi nesta altura que se deu a inserção do estagiário na equipa de programação e consequentemente era compreensível

que se testasse o nível de conhecimentos do estagiário. Por outro lado, não existia mais nenhuma ideia de protótipo pronta a ser implementada.

Foram apresentadas ao estagiário as várias ferramentas usadas pela empresa, sendo que a mais relevante nesta altura foi o *software Git*. É através deste *software* que o controlo de versões é mantido, permitindo uma fácil difusão e manutenção dos conteúdos para toda a equipa, mesmo que vários membros da equipa estejam a trabalhar no mesmo código ou ficheiro.

4.2.1 Traduções

A primeira ferramenta em que o estagiário trabalhou lidava com traduções entre línguas dos textos na interface gráfica do jogo. Esta ferramenta já estava maioritariamente feita, pois já era resultado nas necessidades de um jogo anterior. Tinha, no entanto, algumas particularidades que poderiam ser melhoradas. Estes melhoramentos seriam importantes para a utilização da ferramenta em jogos futuros.

De modo muito geral, todo o processo relativo às traduções funciona da seguinte forma: é criado um ficheiro Excel com todas as expressões que se pretendem traduzidas. Esse ficheiro é enviado a um tradutor profissional, que o devolve preenchido com as traduções. Este novo ficheiro é convertido para XML para poder ser utilizado por qualquer aplicação (assumindo que tem um analisador de XML). A aplicação, que neste caso será um jogo, obterá os dados desse ficheiro XML e mantê-los-á em memória numa estrutura de dados, como por exemplo, um dicionário. Assim, para realizar uma tradução, bastará procurar no dicionário a expressão desejada.

A linguagem de programação utilizada para esta ferramenta foi o *Python*, que tem um grande conjunto de bibliotecas relativamente simples de usar para processar dados em XML ou em Excel.

Num nível abstracto, a sequência normal para obtenção do ficheiro XML é o seguinte:

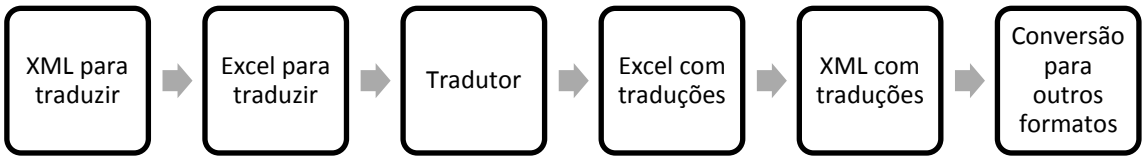

Figura 4.5: Sequência típica da concepção de um ficheiro de traduções

O processo pode começar em qualquer um dos dois primeiros passos, sendo que as traduções que se quiserem fazer podem estar ou num ficheiro XML ou num ficheiro Excel. No entanto, é estritamente necessário que exista um ficheiro Excel para ser enviado para o tradutor, pois este pode não ter as competências para trabalhar com XML directamente. Será então sempre necessária uma conversão do formato XML para Excel. Quando o tradutor devolve este ficheiro Excel preenchido, gera-se o novo ficheiro XML. Este novo ficheiro será utilizado para poder ser convertido para qualquer formato desejado, dependendo da plataforma alvo.

Cada aplicação (leia-se, videojogo) terá de importar a informação XML para uma estrutura armazenada em memória, para rápido acesso. Essa estrutura, por questões de eficiência, é um dicionário, em que a cada expressão de entrada corresponderá uma tradução. As expressões a serem traduzidas estão escritas directamente no código, implicando que o inglês será a linguagem por omissão. Para realizar a operação de tradução será necessário chamar uma função que receba a expressão em inglês e que a indexe no dicionário. Cada objecto guardado no dicionário é um conjunto de expressões traduzidas para línguas diferentes.

Apresenta-se de seguida um exemplo da estrutura XML que armazena as traduções:

```
<root>
  <keylang>eng</keylang>
  <langs> eng de es fr it</langs>
  <tr>
    <key>Tap to play</key>
    <de>Zum Spielen antippen</de>
    <es>Toca para jugar</es>
    <fr>Appuyez pour jouer</fr>
    <it>Premi per giocare</it>
  </tr>
</root>
```

Figura 4.6: Exemplo de estrutura XML

"*keylang*" é o elemento que define qual a língua chave a ser usada (no exemplo, é o inglês). "*langs*" é uma sequência de caracteres que informa quais as línguas disponíveis no ficheiro. Por fim, cada elemento "tr" é um nó único de tradução, sendo que para cada expressão diferente que se queira traduzir haverá um elemento deste tipo.

A explicação anterior abrange o que estava previamente feito. Achou-se relevante colocá-la neste relatório para dar contexto ao leitor sobre o trabalho feito pelo estagiário, explicado de seguida.

A estrutura XML apresentada terá de ser modificada para permitir uma melhor generalização das traduções:

Em primeiro, existe uma imposição de linguagem chave, e isso deveria ser feito no contexto de cada jogo. Poderá haver um jogo cujo código tenha texto escrito em francês e então seria necessário indexar no dicionário por essa língua (ao invés do inglês pré-definido).

Por consequência, o elemento que deveria corresponder à língua inglesa é chamado na verdade de "*key*". Este facto é muito limitativo pois poderá querer-se, numa fase de desenvolvimento, "traduzir" a própria língua inglesa. Um exemplo seria se se quisesse mudar uma frase na interface sem ter que recompilar o código de novo, considerando que as traduções são exteriores à compilação.

Por último, a definição das línguas disponíveis pode parecer redundante, considerando que a mesma informação já está disponível nos elementos de tradução, mas é preferível que exista uma explicitação das linguagens disponíveis em vez dessa informação estar dependente dos elementos de tradução (que podem eventualmente ser falíveis). Outrossim, será mais conveniente em termos estruturais colocar cada língua como um elemento XML ao invés de colocar todas as línguas numa sequência de caracteres única. Caso contrário seria necessário fazer-se a separação das línguas através de código adicional.

No final, a estrutura deverá ter o seguinte aspecto:

```
<root>
  <langs>
    <lang>eng</lang>
    <lang>de</lang>
    <lang>es</lang>
    <lang>fr</lang>
    <lang>it</lang>
  </langs>
  <tr>
    <key>Tap to play</key>
    <eng>TAP TO PLAY!</eng>
    <de>Zum Spielen antippen</de>
    <es>Toca para jugar</es>
    <fr>Appuyez pour jouer</fr>
    <it>Premi per giocare</it>
  </tr>
</root>
```

Figura 4.7: Nova estrutura XML, com a mesma informação

Nesta estrutura nova, existe a separação entre a língua chave e o inglês. A maior vantagem disto é o facto de se poder traduzir do inglês para o inglês. Explicando melhor, como o texto a traduzir está directamente no código, a eventual necessidade de correcção de uma expressão implica a recompilação do código, que pode ser um processo demorado. Para evitar isto, pode-se, por exemplo, definir o inglês como língua chave e pedir a sua tradução para inglês. Em maior parte dos casos irá originar a mesma expressão, mas em situações como a representada acima a expressão em inglês apresentada será diferente da chave.

4.2.2 Serialização

No contexto de projectos futuros a serem realizados pela empresa, poderia ser necessário criar um serializador de objectos proprietário. Um serializador é um mecanismo que transforma estruturas de dados ou outros objectos de programação num formato que pode ser guardado ou transportado. Normalmente, este formato é binário. Para o mecanismo funcionar, a entidade descodificadora tem de conhecer o método pelo qual os dados foram serializados, para garantir que não existe perda ou corrupção de dados.

Uma utilização potencial deste mecanismo seria de estabelecer envio de dados arbitrários entre a placa gráfica (GPU) e o processador (CPU) do dispositivo.

Mais uma vez, o trabalho do estagiário seria de complementar algumas falhas que o serializador proprietário já existente tinha, particularmente nos tipos de dados que poderia receber para serializar. O estagiário foi introduzido à programação em C++, com foco para a utilização de *templates*.

Os *templates* são uma funcionalidade da linguagem C++ que permite a utilização de tipos de objectos genéricos numa classe ou função. Deste modo, o tipo de objecto fica abstraído na função, e o foco do código centra-se no comportamento desejado. Estes *templates* ajudam na generalização de código, reduzindo substancialmente a quantidade de código que o programador tem de escrever manualmente.

A dificuldade principal que o estagiário sentiu nesta fase foi a familiarização com o código já feito, pois seria necessário que o código estivesse bem presente e compreendido para o estagiário poder fazer as alterações necessárias adequadamente. Como se disse, nem todos os tipos de dados podiam ser serializados no estado em que o código estava. O processo seguido pelo estagiário foi de aplicar a mesma lógica de serialização já feita para alguns tipos para os restantes. Contudo, não foi possível serializar todos os tipos, particularmente ponteiros do C, pois não se sabia explicitamente a dimensão dos dados a serializar, o que é um dado absolutamente necessário para realizar a serialização.

4.3 Onda

No contexto de um projecto vindouro, o estagiário foi instruído para voltar para o seu propósito inicial de criar protótipos. Desta vez o objectivo não era de criar um jogo inteiro,

mas sim de implementar subsistemas de um jogo, ou seja, alguns dos seus componentes mais específicos.

A intenção para o projecto em causa é criar num ambiente virtual em tempo real uma onda realista, simulando o seu comportamento para ambientes diferentes (praia, recife, etc.).

Sendo um problema complexo, é indispensável percorrer todas as alternativas possíveis para implementar a onda. Dito isto, vão-se explicar de seguida as várias alternativas e soluções encontradas e discutir as eventuais vantagens e desvantagens consequentes.

4.3.1 Onda no *Unity*

Sendo que o estagiário era mais experiente na plataforma *Unity*, decidiu-se que o primeiro teste a implementar seria nessa plataforma. A preocupação inicial para o problema de simular a deformação de uma onda é a capacidade computacional pesada necessária para a conseguir. A ideia da equipa de programação era de processar dados na placa gráfica (GPU), mas numa fase de testes não se pode descartar a hipótese de verificar se a própria CPU seria capaz de fluidamente efectuar os cálculos precisos.

4.3.1.1 Deformação pela CPU

O primeiro passo é de gerar a malha a deformar procedimentalmente. A intenção é poder definir-se dinamicamente o número de vértices que se pretende que a malha tenha, de modo a poder-se imediatamente testar se uma malha com complexidade espacial equivalente seria viável ou não num ambiente de jogo. Outra opção menos eficiente seria criar manualmente um conjunto de malhas 3D.

Em fase de testes não é relevante qual a deformação aplicada à malha, e por isso foi apenas aplicada uma operação de seno a cada vértice (Figura 4.8). A operação de seno não só permite deformar a malha de modo a haver *feedback* visual como também é uma operação computacionalmente pesada, o que neste momento é desejável para determinar a fluidez do ambiente em situações de processamento mais intenso.

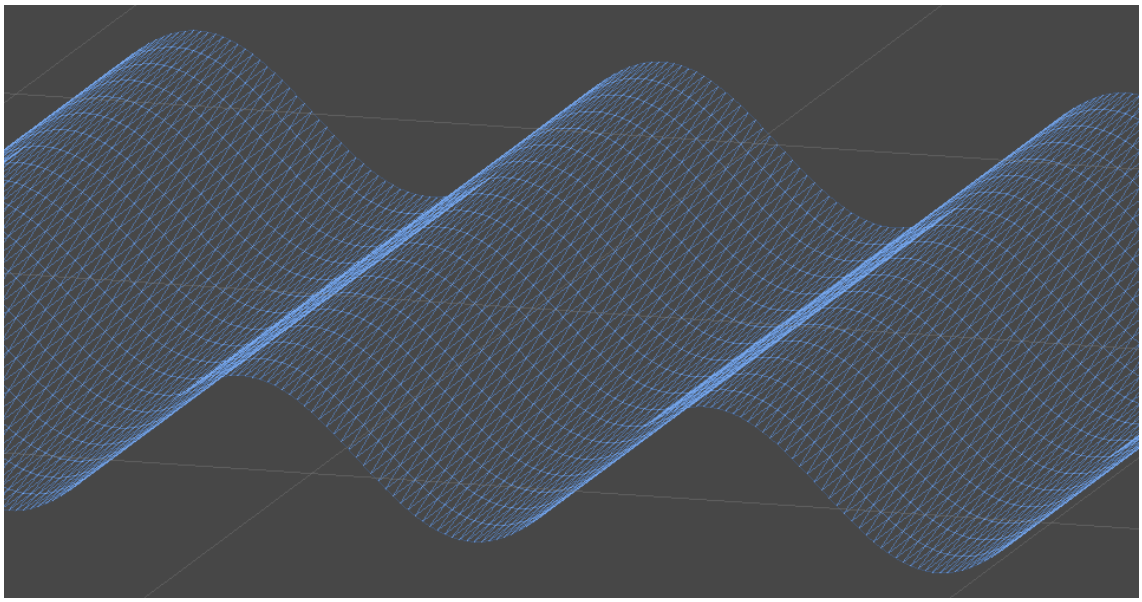

Figura 4.8: Malha deformada por uma função de seno

De modo a determinar objectivamente a performance dos cálculos feitos, mediu-se o número de tramas processadas por segundo, em primeiro com o ambiente virtual vazio (com a deformação desactivada), e de seguida com a malha a deformar activa. O primeiro caso atingiu um número de fps (*frames per second*, tramas por segundo) de aproximadamente 300. Com a malha activa esse número decresceu para 220fps, o que para este caso implica uma diferença superior a 25%.

Isto permite concluir que apesar do mecanismo de deformação da onda poder, de facto, funcionar completamente através da CPU, a carga computacional requerida poderá ser excessiva. A segunda opção, já referida como sendo a mais provável de vir a ser implementada, é utilizar a capacidade de processamento intensiva e altamente paralela da GPU.

Ainda no contexto da CPU, faça-se notar que não se aplicou nenhuma deformação semelhante a uma onda, tendo-se optado pela função seno. No entanto, a animação da deformação a usar já havia sido criada por um artista. Esta animação é composta por pontos com duas dimensões, ou seja, para cada instante de tempo, ou trama, apenas se tem uma representação 2D da animação da onda. Seria trabalho do programador juntar e ligar as várias tramas numa malha 3D. Assumindo que as coordenadas providenciadas pela animação são respectivas aos eixos xx e zz, numa malha rectangular corresponderia uma trama diferente da animação ao

eixo yy. De seguida mostra-se a representação de uma só trama da animação de deformação (Figura 4.9) e um exemplo de aplicação da animação numa malha 3D no Unity (Figura 4.10).

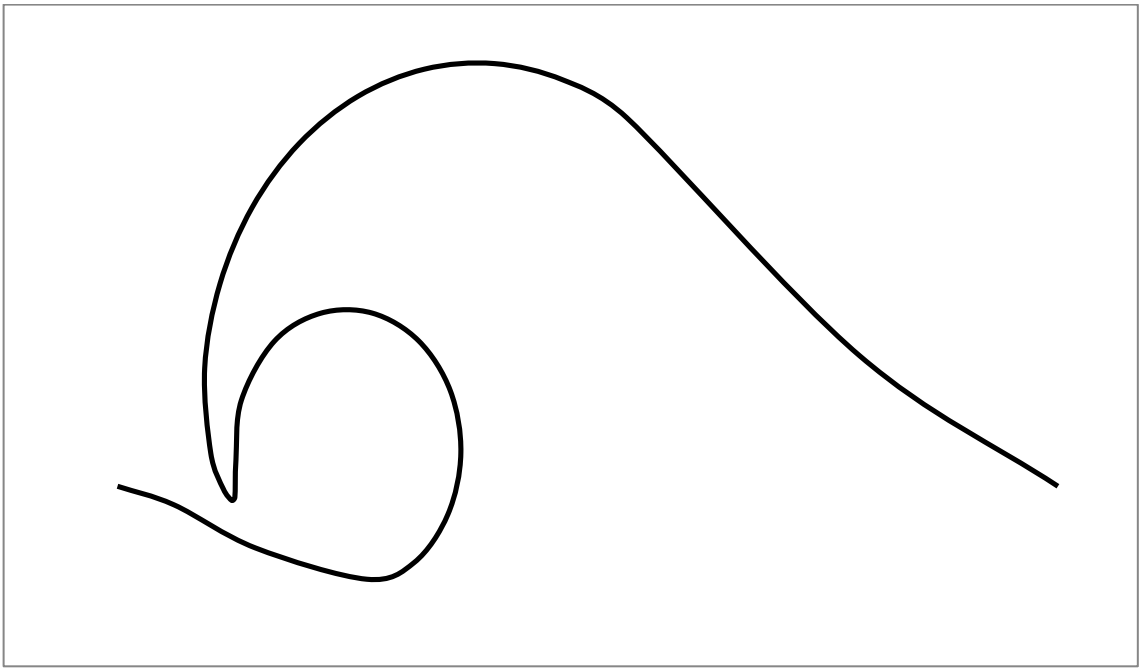

Figura 4.9: Visualização de uma trama, como se vista lateralmente

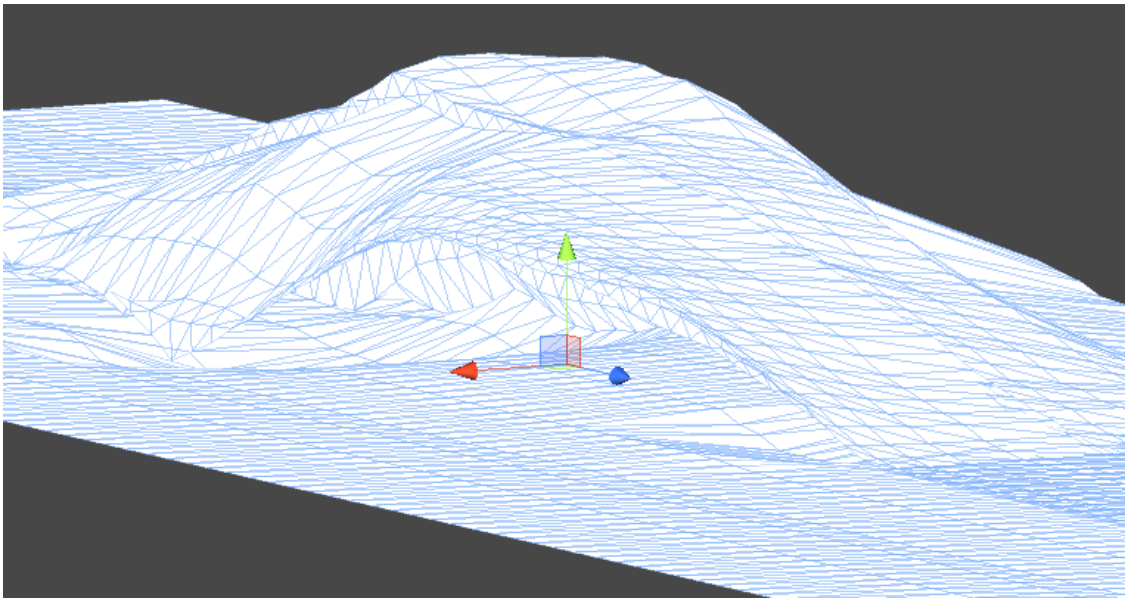

Figura 4.10: Malha 3D deformada pelos valores da animação; cada linha ao longo do rectângulo usa uma trama diferente da animação

4.3.1.2 Deformação pela GPU

Para utilizar a GPU no processamento em questão é necessário utilizar um *shader*[1].

A computação pesada, como é o caso do seno, feita no *shader* aumenta consideravelmente o número de fps que o *Unity* reporta. O problema com esta abordagem é passar a informação da animação da onda para o *shader*, pois, por limitação estabelecida, não existem *inputs* na forma de uma sequência de valores arbitrária. A resolução inicial foi utilizar *buffers* de dados específicos para computação geral em placas gráficas. No entanto, isto implicava necessariamente a utilização de DirectX11, significando que o *shader* poderia não funcionar em todos os dispositivos.

Para se tentar contornar este requisito colocou-se a hipótese de usar texturas. Os *shaders* conseguem ler e mapear texturas, normalmente para dar informação de cor a um objecto, no que se chama de *fragment shader*. No entanto, a informação presente nos pixéis de uma textura não tem necessariamente de ser de cor. Uma utilização típica de texturas para fins que não de definição de cor são os mapas de normais, por exemplo.

A ideia é codificar uma imagem RGBA (8 bits por canal) de forma a que cada pixel represente um número de vírgula flutuante de 32 bits, doravante chamado de *float*. Visualmente, a imagem final irá parecer ter cores aleatórias (Figura 4.11), mas o relevante é que os valores sejam coerentemente interpretados no *shader* do *Unity*.

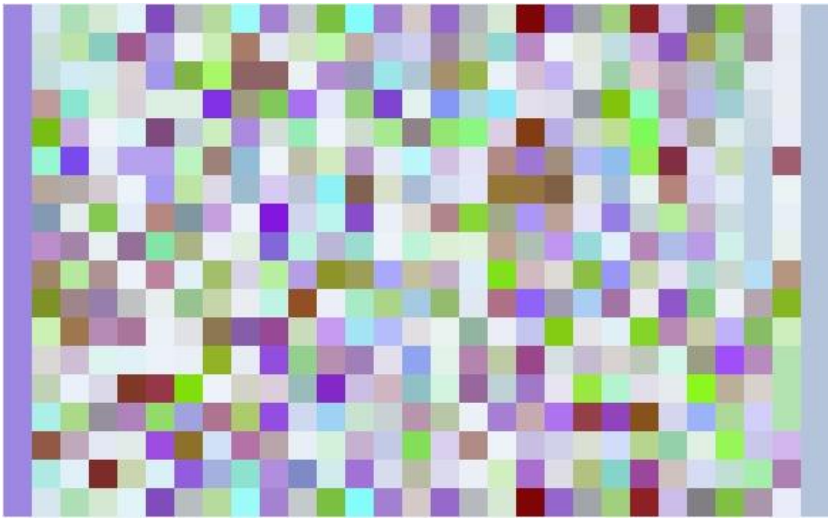

Figura 4.11: Valores da animação da onda convertidos para uma textura

[1] Um *shader* é um conjunto de instruções definíveis na placa gráfica que permite efectivamente a programação dentro da placa, permitindo cálculos e resultados gráficos controlados pelo programador.

De notar que a transformação de um *float* numa cor RGBA em *Unity* tem como pressuposto que a gama de valores está entre 0 e 1 (exclusive), o que implica que é necessário aplicar transformações aos valores tanto no codificador como no descodificador (por exemplo: o valor -19.75 teria de ser convertido para um valor entre 0 e 1). As operações de transformação feitas, neste caso, foram uma divisão por um valor normalizador alto o suficiente para colocar qualquer valor entre -1 e 1, seguida de uma divisão por 2 para colocar os valores entre -0.5 e 0.5 e finalmente a soma de 0.5 para atingir a gama final de 0 a 1. No descodificador as operações são feitas pela ordem inversa. Isto implica haver um conjunto de valores conhecidos nos dois lados, nomeadamente o valor normalizador.

A codificação realizada é linear: um *float* é composto de 32 bits, enquanto que cada canal de um pixel RGBA tem 8 bits. Logo, a cada um destes canais corresponderá uma partição da representação binária do valor *float*, como figurado a seguir (Figura 4.12):

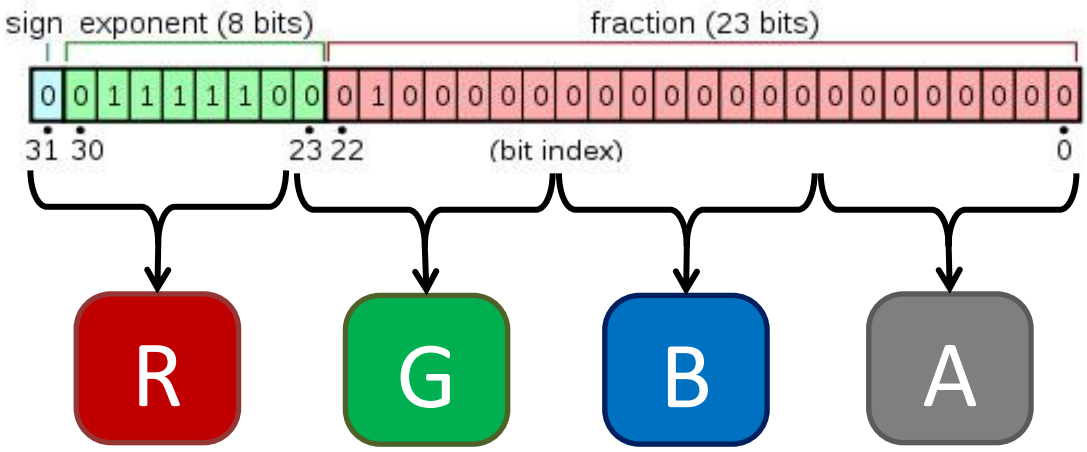

Figura 4.12: Mapa de conversão de *float* para RBGA

A textura está organizada por tramas e pontos da animação. Cada coluna de píxeis é respectiva a uma trama, sendo cada pixel dessa coluna correspondente ao valor dum certo ponto.

Dependendo da quantidade de informação necessária para ser transportada para o *shader*, poderá ser preciso criar mais que uma textura. Neste caso específico, como cada ponto da animação tem duas coordenadas, serão necessárias duas texturas, cada uma para mapear uma coordenada diferente. É seguro fazer isto porque as texturas partilham o mesmo mapeamento

UV[2], e por isso as coordenadas terão sempre os pares respectivos, como se mostra na Figura 4.13.

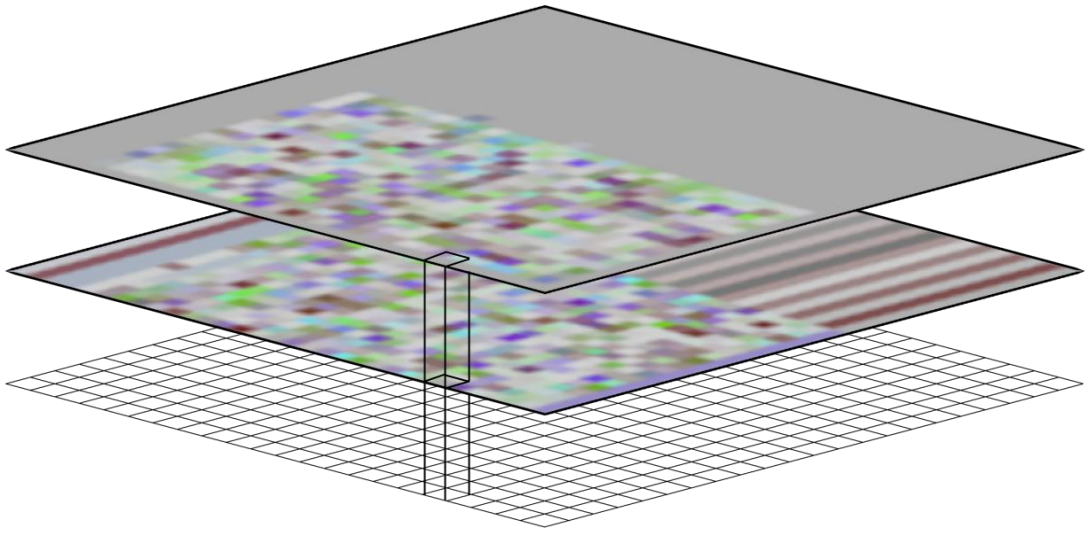

Figura 4.13: Visualização simplificada do mapeamento UV igual para as duas texturas; a malha 3D (camada de baixo) será afectada pelos valores das duas texturas (camadas superiores) na mesma coordenada UV

Note-se ainda pela Figura 4.13 que nem toda a textura está ocupada com valores da animação. Por convenção, é preferível que as texturas tenham uma dimensão em potência de dois. No entanto, a animação pode não ter um número de pontos que cumpra este requisito, sendo então necessário realizar o preenchimento de valores de *padding*[3]. Observa-se, por exemplo, a representação de cor do valor zero na área cinzenta da textura superior.

A utilização de texturas tem mais umas vantagens, especialmente devido a alguns mecanismos próprios de texturas em computação gráfica. Considera-se relevante fazer um aparte para explicar estes mecanismos.

No caso de uma coordenada UV sair fora do intervalo [0,1[o mecanismo de texturização tem várias opções de resolução. As opções mais típicas são ou definir o valor mapeado como

[2] Uma coordenada UV é uma coordenada adicional que um vértice de uma malha 3D pode ter que localiza o pixel ao qual o vértice está associado numa textura.
[3] Valores de *padding* são valores que são adicionados quando é necessário preencher alguma estrutura de dados incompleta de modo a respeitar certas dimensões fixas da estrutura.

sendo zero (*clipping*) ou ter apenas em conta a parte decimal da coordenada e repetir a textura (*repeating*). Esta última opção é útil em texturas que mostrem padrões ou repetições. É também possível aplicar transformações às coordenadas (translação, escala, rotação) de modo a permitir ajustá-las a certos contextos. A Figura 4.14 mostra um exemplo de uma transformação de escala, seguida de uma translação:

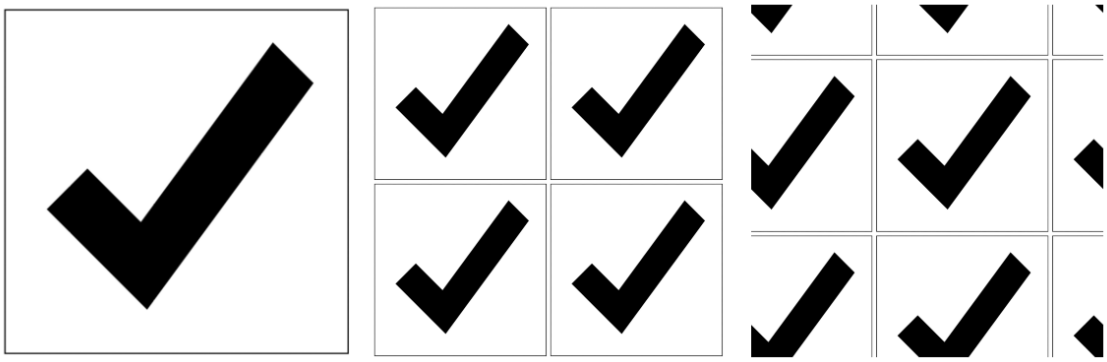

Figura 4.14: Exemplo de escala, repetição e translação de uma textura com recurso a transformações às coordenadas UV

Tendo em conta que neste exemplo o modo de texturização está estabelecido para repetir a textura, observa-se que o resultado final apresenta sem problemas visíveis o padrão esperado de repetição.

O mesmo conceito pode ser aplicado à animação da onda. Considerando que a animação está codificada numa textura, é possível adicionar pequenas transformações (neste caso na forma de uma translação) em cada ciclo de jogo de modo a simular o movimento da onda e da sua deformação.

Contudo, isto não implica necessariamente que a transição entre tramas seja visualmente suave, pois se não houver mais nenhum mecanismo, a onda teria a forma de degraus, derivado das posições transitarem bruscamente de uma para outra, mesmo entre tramas contíguas. Seria necessário determinar as posições interpoladas entre os valores das duas tramas entre as quais uma coordenada UV se insere.

Pode-se tirar proveito de um outro mecanismo comum de texturas: a filtragem. O problema identificado acima para a posição também ocorre para a cor. Existe um conjunto de técnicas de filtragem que permite determinar a melhor cor para uma determinada coordenada UV.

Particularmente, técnicas mais comuns como filtragem bilinear e/ou trilinear já fazem, em termos práticos, a interpolação necessária. Analogamente, o mesmo acontece para os valores de posição.

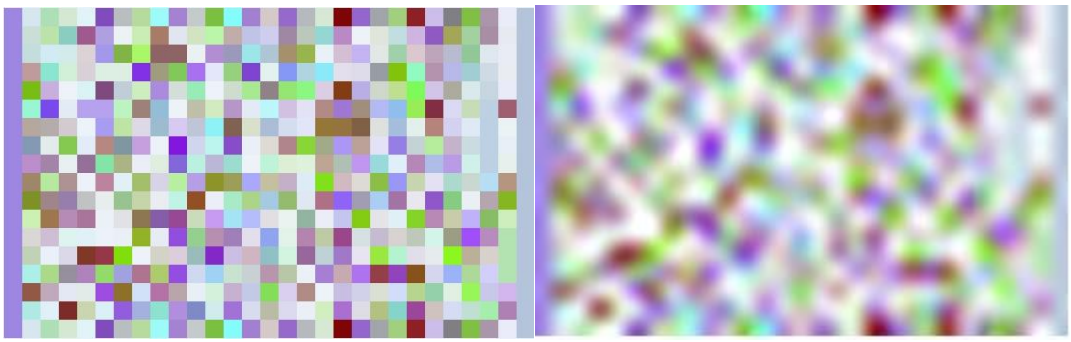

Figura 4.15: Textura sem filtragem e textura com filtragem

De um modo grosseiro, o que se observa na Figura 4.15 para as cores é também o que acontece para os valores *float* nelas codificados. Faça-se notar que desta maneira, o processo fica dependente da implementação (da filtragem) por parte de terceiros e portanto não é totalmente fiável.

Contudo, utilizar texturas para transportar dados também tem as suas desvantagens. A principal é a de que o acesso aos dados na textura fica inerentemente ligado às coordenadas UV de cada vértice. Torna-se mais difícil obter os dados de uma trama específica.

Para esta fase do desenvolvimento da onda, a plataforma essencial de desenvolvimento foi o *Unity*. Particularmente, a criação de *shaders* implica programação numa linguagem de computação gráfica. O *Unity* utiliza uma linguagem de alto nível deste tipo de programação, denominada *Shaderlab*. Pelas suas limitações, é mais usual utilizar-se *Shaderlab* em conjunto com outra linguagem de mais baixo nível, que neste caso foi a linguagem Cg[4].

Neste ponto era possível visualizar uma onda no ambiente do *Unity* utilizando *shaders* para deformar uma malha procedimentalmente gerada, sendo que as texturas com a informação das posições dos pontos tinham de ser importadas. Estas texturas foram criadas fora do contexto

[4] A linguagem Cg é uma linguagem de programação gráfica com sintaxe semelhante à do C, desenvolvida pela Nvidia. Foi entretanto descontinuada.

do *Unity*. A Figura 4.10 mostrada previamente foi tirada directamente do ambiente do editor e mostra um exemplo de uma onda já a utilizar os métodos explicados.

A continuação do trabalho implicou vários melhoramentos na onda, incluindo um melhor controlo da sua posição e escala e a fusão entre duas ondas diferentes para providenciar a outros membros da equipa (nomeadamente artistas ou *designers*) uma ferramenta de edição conveniente de uma onda.

Verificou-se que o método usado para movimentar a onda na malha teria bons resultados se se quisesse que as tramas da animação da onda deformassem a malha pela ordem em que estão na textura. Por exemplo, no caso de se querer especificar um intervalo de tramas para se usar, seria necessário de alguma forma transformar as coordenadas UV dos vértices da malha para os valores nesse intervalo pretendido. Neste ponto este controlo ainda precisaria de alguma maturação.

No entanto, há muito que o objectivo proposto já tinha sido atingido. Como prova de conceito, o protótipo cumpriu o requisito de ser possível animar a onda no ambiente do *Unity*, inclusive de mais de uma maneira diferente (em CPU e GPU).

4.3.2 Onda no *Unreal Engine 4*

Ao mesmo tempo que o estagiário estava envolvido no protótipo em *Unity*, um novo motor de jogo concorrente foi lançado: o *Unreal Engine 4* (UE4). A equipa de programação decidiu utilizar este novo motor para novos projectos, por um número de razões:

- É um motor muito recente, e por isso a empresa poder-se-á considerar na vanguarda da tecnologia de desenvolvimento de videojogos;
- A linguagem de programação é C++, linguagem na qual a maioria da equipa já tem experiência, trazendo também todos os benefícios da linguagem.
- Acesso a todo o código fonte, permitindo expandir as funcionalidades do motor para os objectivos específicos da empresa.

Este tipo de decisões tem os seus riscos. Apesar da equipa ter experiência em C++, existe um grande desconhecimento em relação à API (*Application Programming Interface*) do motor e às suas restantes particularidades. Por exemplo, o UE4 permite não programar uma única linha em C++, tendo a opção de uma linguagem gráfica própria denominada *Blueprints*. Teria

de haver um período de adaptação relativamente grande a este novo paradigma de programar através de nós e ligações. Mostra-se de seguida um exemplo do aspecto de uma *Blueprint* (Figura 4.16):

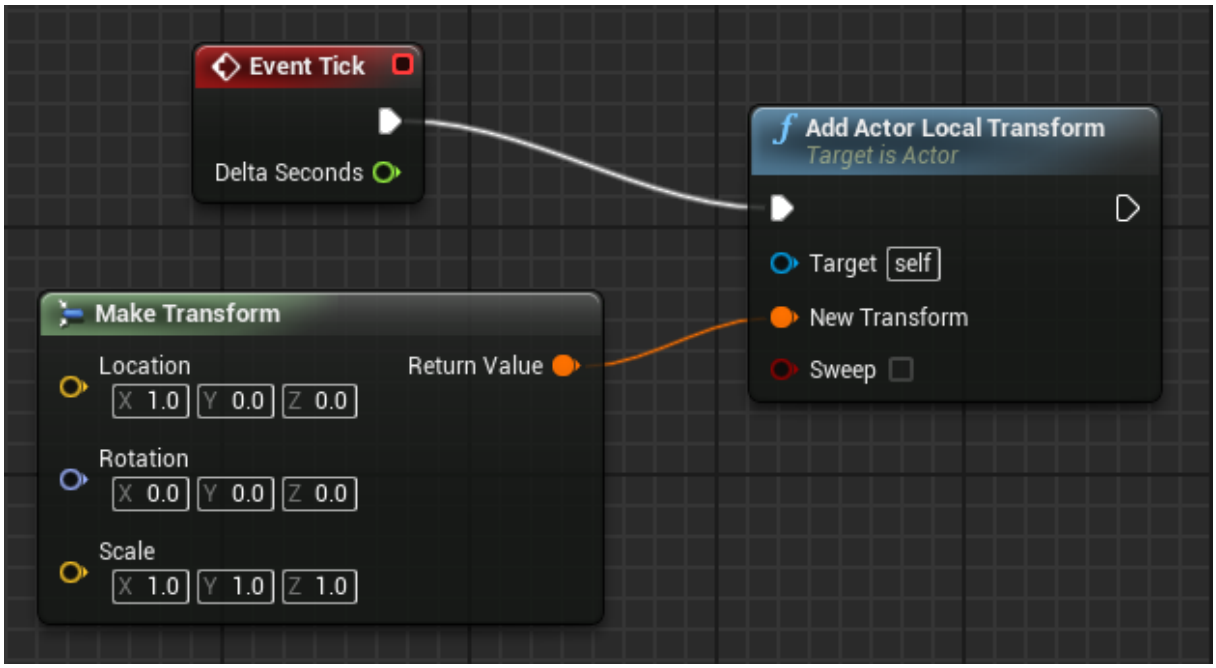

Figura 4.16: Exemplo de uma *Blueprint*; a sua função é de adicionar na coordenada X um valor a cada ciclo de jogo (*tick*), fazendo o objecto mexer-se

Esta linguagem é propícia a membros da equipa de desenvolvimento que não têm um contexto anterior de programação, sendo que estão abstraídos da sintaxe do C++, tendo mesmo assim um grande poder de implementação.

Foi pedido ao estagiário que implementasse em UE4 o mesmo mecanismo que já tinha sido feito no *Unity*, ou seja, a deformação de uma malha 3D através de um *shader*. Explicam-se a seguir as várias diferenças que afectaram o tempo de adaptação do estagiário na transição do *Unity* para o UE4.

A primeira grande diferença é o método com que os *shaders* são criados no UE4. De maneira semelhante às *Blueprints*, o UE4 tem um editor de materiais incorporado, também baseado em nós e ligações. Mais uma vez, isto ajuda os artistas a abstraírem-se de linguagens de programação gráfica, permitindo uma aplicação mais directa e rápida dos efeitos pretendidos.

No entanto, quando o objectivo do *shader* envolve alguma complexidade, o editor de materiais apresenta limitações, como será explicado. A Figura 4.17 mostra um exemplo de um material simples no editor de materiais do UE4.

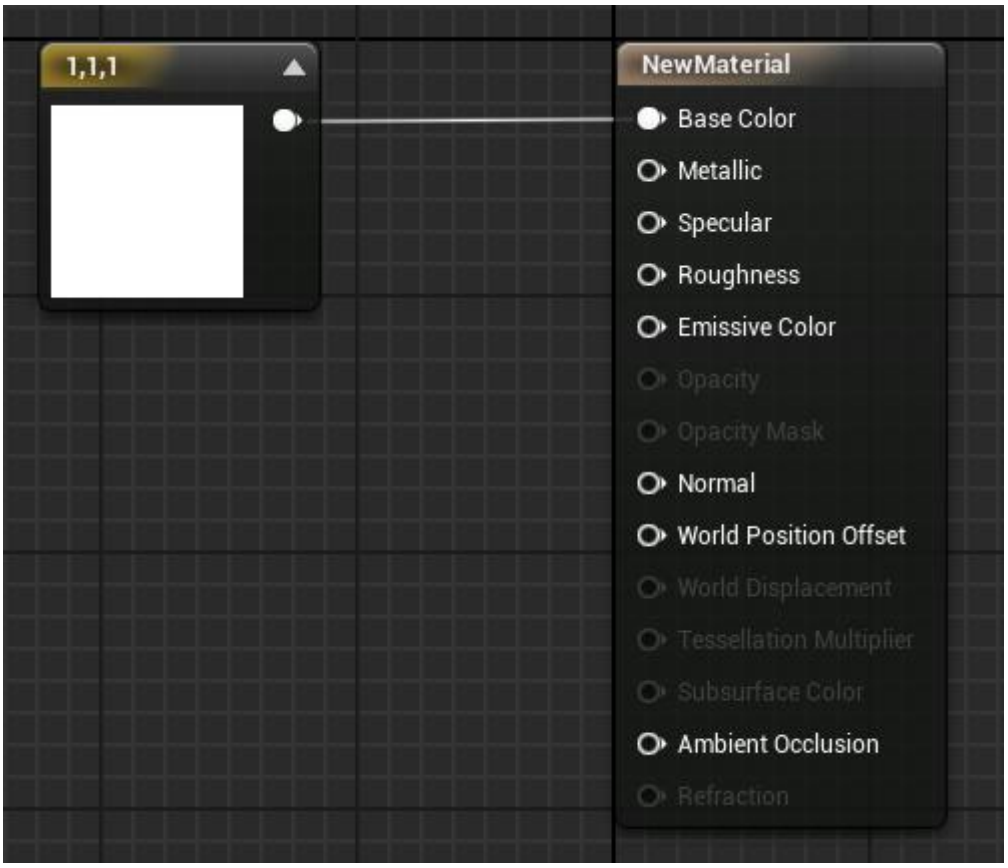

Figura 4.17: Exemplo de um material simples no UE4; apenas define a cor base como sendo branca; notem-se as várias outras ligações desconectadas, que poderiam ser usadas para definir o material como um metal, alterar normais, alterar posição dos vértices, etc.

Outra grande diferença detectada é a falta de documentação do código fonte. Sendo verdade que o *Unity* não disponibiliza o seu código, por outro lado a documentação da sua API está muito mais completa. O UE4, pelo contrário, tem ficheiros de documentação de C++ que dizem pouco mais que a relação de herança entre classes. Nota-se que existe um muito maior foco em documentar e criar suporte para *Blueprints* e outros mecanismos próprios do UE4 (nota: tenha-se em conta o momento em que o relatório foi escrito, pois a situação pode mudar; isto serve apenas para apresentar algumas dificuldades que o estagiário sentiu na altura).

Estas dificuldades de adaptação justificaram que o estagiário tenha despendido algum tempo na aprendizagem de algumas destas particularidades, nomeadamente o editor de materiais. Apesar de não ser programação no sentido mais usual da palavra, muitos conceitos da computação gráfica estavam presentes e eram necessários.

Como o *Unity*, o editor de materiais do UE4 não tem nenhuma maneira de receber sequências de valores arbitrários, particularmente para transportar os valores da animação. Como tal, decidiu-se reutilizar o conceito de fazer esse transporte através de texturas. No entanto, para reduzir a dependência de ferramentas exteriores ao motor, investigou-se uma forma de criar as texturas directamente dentro do UE4, e enviá-las para o editor de materiais, sendo que o único recurso necessário seria o ficheiro de texto com os valores da animação. O estagiário adaptou um analisador de texto para funcionar com a API do UE4, lendo os valores do ficheiro e colocando-os numa nova textura. Este analisador sofreu algumas alterações ao longo do desenvolvimento, pois foram sendo necessários mais valores por vários motivos, o que também implicou mais texturas a criar, como será explicado.

Como, de certo modo, se teve que reimplementar tudo o que já havia sido feito em *Unity*, decidiu-se resolver um problema que emergiu no protótipo anterior. O controlo da onda era complicado pelo facto de estar inerentemente associado às coordenadas UV da malha. A solução que se testou no UE4 foi utilizar uma textura adicional de controlo para auxiliar na definição de quais as tramas a seleccionar para cada vértice. Segue-se uma ilustração do problema e da solução:

Considere-se que a animação tem no total 20 tramas, numeradas de 1 a 20:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Utilizando as coordenadas UV a sequência das tramas a utilizar será linear e crescente. Modificar esta sequência pode ser complicado porque implica disturbar esta sequência implícita. Idealmente, seria possível realizar um controlo como exemplificado a seguir:

6 7 8 7 6 7 8 7 6 7 8 7 6 7 8 7 6 7 8 6 7 6 ...

Isto causaria que a malha se deformasse ciclicamente entre as tramas 6, 7 e 8 da animação, por exemplo. Para este propósito constrói-se a textura de controlo adicional referida acima que transporta este ciclo exemplificativo para o *shader*. Note-se: os valores contidos na

textura são os valores das tramas que deverão ser usadas. Esta textura pode ser mapeada com as coordenadas UV da malha, resultando na tradução da coordenada para o número da trama.

UV	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Textura de controlo	6	7	8	7	6	7	8	7	6	7	8	7	6	7	8	7	6	7	8	7

Figura 4.18: Tradução entre a trama correspondente à coordenada UV e a trama realmente desejada; apenas as tramas 6 a 8 é que são usadas para deformar a malha

Resumindo, numa primeira fase colocam-se as texturas de posição totalmente no *shader* e de seguida cria-se uma nova textura de controlo. O processo será, em cada vértice, obter a sua posição correspondente à trama especificada na textura de controlo.

Depois do tempo de adaptação, conclui-se que o editor de materiais é de facto uma ferramenta relativamente fácil de usar e de se adaptar. De notar que a *Epic*, criadora do UE4, disponibilizou tutoriais simples e claros que aceleram o processo de aprendizagem para este mecanismo. Não obstante esta vantagem, verifica-se que é uma ferramenta limitada pelo número de nós existentes que podem não ser suficientes para requisitos mais complexos. Existe um nó que permite a inserção de código HLSL[5] personalizado, mas na altura da elaboração deste relatório a sua utilização não é prática, pois é menos eficiente que os nós pré-definidos.

Como, em termos conceptuais, se provou que qualquer controlo pretendido se pode fazer através de texturas de controlo, continuou-se para se resolverem problemas mais técnicos.

A deformação da malha implica necessariamente o recálculo dos vectores normais dos vértices. O editor de materiais permite que o utilizador defina a normal de um vértice. No entanto, a informação das normais após a deformação da malha não deve ser calculada no UE4 em tempo real, pois atingir esse objectivo com a interface de nós e ligações pode ser uma tarefa colossal. Ao invés disso, decidiu-se utilizar normais pré-calculadas. No mesmo ficheiro

[5] HLSL –High-Level Shader Language: uma linguagem de programação gráfica, desenvolvida pela Microsoft.

de texto onde estão explicitadas as posições pode também estar a informação sobre as normais. Já se referiu que o analisador de texto usado para importar estes valores teve de ser actualizado várias vezes para acomodar as alterações do desenvolvimento. A inclusão das normais no ficheiro foi uma das primeiras razões para este facto acontecer.

É relevante referir que o UE4 suporta texturas com 4 canais de valores *float*, por alternativa aos típicos 4 canais RGBA 8 bits cada. Isto implica que cada textura passada para o *shader* pode transportar 4 valores *float* relevantes por pixel. Neste momento, têm-se 4 valores por pixel, 2 de posição e outros 2 de normais (relembra-se que no ficheiro só existe informação de 2 coordenadas). Isto significa que na verdade só é necessária uma textura para transportar todos os valores neste ponto, por oposição das 4 texturas necessárias no *Unity* para o mesmo efeito. Existiria uma textura apenas para a informação de controlo, assumindo que é apenas necessário um canal.

O grande benefício dos canais de cor das texturas poderem transportar valores *float* é que, analogamente ao que aconteceu com as normais, pode-se pré-calcular um grande conjunto de informações e inseri-las no ficheiro de texto a analisar, retirando essa necessidade do próprio motor em tempo real. Exemplos de informações que poderiam ser adicionadas são: vector de velocidade da onda naquele vértice, ângulo entre vértices vizinhos, controlo de emissão de partículas ou informações de controlo de tesselação (*tesselation*). Estas informações podem ser fundamentais na dinâmica e no aspecto visual da onda.

Retomando o assunto das normais, o leitor atento poderá ter notado que se referiu que as normais só têm duas coordenadas pré-calculadas. Ora, é necessário determinar de alguma maneira a terceira coordenada do vector normal. Inevitavelmente, este cálculo terá de ser feito em tempo real no UE4, pois só no ambiente de jogo é que serão determinadas as posições da coordenada em falta, requisito essencial para o cálculo correcto das normais. Este cálculo restante, no entanto, ficou por resolver neste protótipo.

4.3.2.1 Tesselação

Referiu-se anteriormente o conceito de tesselação. A tesselação é uma técnica que permite adicionar detalhe a uma malha 3D, criando novos vértices e triângulos consoante certas regras. Uma utilização típica de tesselação é de adicionar maior detalhe consoante a distância da câmara à malha. A criação de um maior número de triângulos não implica que se note automaticamente o aumento de qualidade da malha. É ainda necessário especificar que curva

esses novos elementos devem seguir ao serem criados e posicionados. Tome-se o seguinte exemplo:

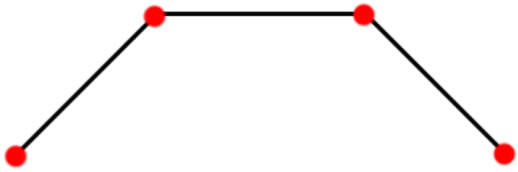

Figura 4.19: Linha original

Na Figura 4.19 está a representação de uma linha à qual vai ser aplicada tesselação. Não existe nenhum controlo de como é que os vértices serão reposicionados. No UE4, se nada for dito em contrário, os vértices criados serão colocados linearmente entre os vértices originais, como demostrado na Figura 4.20:

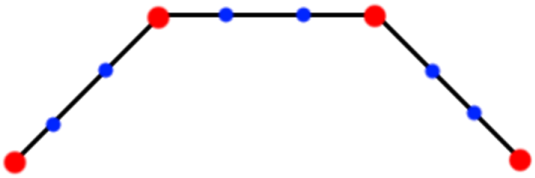

Figura 4.20: Tesselação sem controlo

Idealmente, a posição dos vértices será determinada por uma função apropriada para o objectivo desejado. Por exemplo, utilizando o exemplo figurado, pode-se querer que os novos pontos provenientes da tesselação sejam ajustados de forma a respeitarem uma certa curva, como se mostra na Figura 4.21:

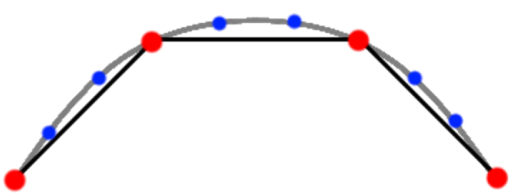

Figura 4.21: Tesselação com controlo

A grande vantagem da tesselação é a construção de modelos em baixa definição aos quais se podem adicionar detalhe de qualidade sem isso implicar uma necessidade de modelos 3D maiores em termos de memória ocupada. Outrossim, nem toda a malha 3D é igualmente afectada pela tesselação, pois tipicamente será tida em conta a distância de cada ponto da malha à câmara, sendo que a técnica pode ser só aplicada se se esperar que o resultado seja perceptível.

Retornando ao problema da onda, é prático que se possa reduzir a complexidade da animação da malha, pelas limitações que a utilização de texturas para transportar dados implicam. É um caso conveniente de utilização de tesselação, pois com um número relativamente reduzido de pontos pode-se definir a curvatura da onda com a resolução que se pretender. Contudo, mantém-se o problema de determinar qual a função que o mecanismo de tesselação deverá usar para posicionar os vértices novos.

Não foi encontrada uma função única que permitisse uma continuidade fluída entre os vários pontos de uma só trama da animação, pois a curvatura necessária depende totalmente da posição desses pontos. Uma solução possível encontrada é a utilização de curvas paramétricas com recurso a polinómios cúbicos na forma:

$$x(t) = At^3 + Bt^2 + Ct + D$$
$$y(t) = Et^3 + Ft^2 + Gt + H$$

A, B, C, D, E, F, G e H são parâmetros que podem ser passados para o *shader* de forma igual aos restantes valores, sendo assim que cada segmento entre dois pontos da animação terá um conjunto de parâmetros diferentes. De notar mais uma vez que são precisas duas funções porque cada ponto da animação tem duas coordenadas. Considerando explicações anteriores, isto implica que cada uma destas funções irá adicionar uma textura de parâmetro ao *shader* (cada um daqueles parâmetros será passado num canal de cor da textura; no total, para todos os parâmetros serão necessárias duas texturas).

O cálculo destes valores também é feito em pré-processamento.

Na altura da escrita deste relatório ainda não existia um protótipo funcional que utilizasse estas curvas para realizar tesselação, apesar dos valores já serem transportados para o *shader*.

5 Protótipo Independente

A experiência adquirida no decorrer do estágio por parte do autor deste documento levou a uma nova proposta de trabalho relevante para o tema geral dos jogos. Aproveitando algumas competências adquiridas, gerou-se a ideia de criar um jogo de forma independente do estágio. Este jogo teria uma componente mais académica, e poderia ser, de certa forma, considerado um protótipo, pois o desafio foi, numa frase:

"Um jogo de luta em que o adversário aprendesse com as acções do jogador."

Este desafio apresentou a oportunidade perfeita para se aplicar as metodologias típicas usadas no desenvolvimento de um videojogo, nomeadamente as várias fases existentes no tempo de vida do projecto.

Apesar da equipa de desenvolvimento ser reduzida, contando apenas com uma pessoa, tal não implica que exista alguma fase que possa/deva ser descartada. Antes pelo contrário, a aplicação de um método testado e comprovado só poderá beneficiar toda a concepção do jogo.

Viu-se nas partes iniciais deste documento quais as fases principais no desenvolvimento de videojogos. Relembrando:

- Pré-produção;
- Produção;
- Pós-produção.

5.1 Pré-produção

Nesta fase pretendeu-se delimitar mais concretamente todos os aspectos do jogo de modo a suportar convenientemente a fase de produção.

A primeira grande tarefa foi a obtenção de inspiração, ou seja, observação e estudo de jogos de luta já existentes. Isto permite definir mais claramente quais os mecanismos de jogo que são apropriados para corresponder ao desafio inicial de uma inteligência artificial que se possa

dizer auto-suficiente. Também permite descartar ideias demasiado ambiciosas para os recursos (humanos, temporais e computacionais) disponíveis.

Poder-se-ia colocar aqui uma lista considerável de jogos de luta (e suas sequelas) que serviram de inspiração para este projecto. Indubitavelmente, alguns dos mais conhecidos serão nomes como *Mortal Kombat*, *Street Fighter*, *Tekken*, entre muitos outros. Apesar de serem todos muitos parecidos em termos de objectivo (derrotar o adversário), cada um tinha alguma particularidade que o tornava único.

Figura 5.1: Da esquerda para a direita: *Mortal Kombat II*; *Street Fighter II*; *Tekken*

A grande parte dos jogos mencionados, especialmente os mais antigos, é em 2D, apesar de versões mais recentes já serem implementadas em 3D. Contudo, esta característica em pouco afecta a jogabilidade, sendo que a quantidade de acções disponíveis não muda consideravelmente, excepção feita para o movimento dos personagens, que passa a ser mais livre. Assim, a primeira grande decisão conceptual do jogo é a sua implementação num ambiente 3D.

Um ambiente 3D traz inerentemente algumas complicações ao desenvolvimento, não obstante o benefício consequente quase inteiramente visual. Em primeiro, no lado da arte, é necessária a modelação de personagens 3D, a sua texturização, armação e animação dos vários movimentos e acções disponíveis; também será necessário a construção do nível de jogo (o ambiente onde os jogadores ir-se-ão inserir). No lado do *game design* a dimensão adicional implica preocupações especialmente no movimento dos personagens, pois terá de ser definido de que forma é que o utilizador irá interagir com o jogo para o controlo deste parecer natural.

Todas as decisões de *game design* devem ser escritas num documento de modo a que seja o mais claro possível quais os comportamentos requeridos da parte de programação e quais os

conteúdos que a arte tem de criar. Este documento é o *Game Design Document*, já referido (ver página 8).

O primeiro ponto criado no GDD foi o que se assumiu que iria ser o mais simples e objectivamente o menos importante: a arte. A lista dos objectos a ser criados deve ser curta devido aos recursos limitados com que se estão a desenvolver o jogo. Como tal, a componente da arte é composta apenas pelo personagem e por um nível de teste. O personagem será a maior parte deste trabalho artístico, sendo que é necessário concretizar vários passos para se obter um personagem funcional, como será discutido. No GDD explicitam-se as animações requeridas para os movimentos e acções do personagem. Ao nível mais básico pretende-se que existam animações de locomoção, para movimentos em geral, animações de luta, tais como para o soco, pontapé e defesa, e uma animação de pose inactiva.

O nível (ou ambiente de jogo) pode ser de construção mais simples, bastando que respeite apenas algumas regras básicas de *game design*. Particularmente, visto que o jogo será em 3D, considera-se útil que o nível seja na forma de um ringue, permitindo a mecânica de derrota por *"Ring Out"*, ou seja, ao sair do ringue, o jogador é imediatamente derrotado.

As regras do jogo são simples, e listam-se de seguida:

- os personagens devem estar sempre virados na direcção do adversário;
- cada jogador tem o objectivo de reduzir a vida do adversário até zero, estando à sua disposição três tipos de ataques (o soco, o pontapé e o míssil) para o efeito;
- cada jogador pode defender-se, e nesse caso recebe danos reduzidos dos vários ataques;
- cada acção do jogador, com a excepção do movimento, custa energia, que se regenera lentamente ao longo do tempo;
- a vida do jogador não se regenera;
- um jogador que se encontre fora do ringue será automaticamente derrotado;
- algumas acções ofensivas têm um efeito de afastamento do adversário, podendo esta mecânica ser usada para empurrar o inimigo para fora do ringue;
- o soco só tem efeito em distâncias curtas; o pontapé tem um alcance maior, mas causa menos danos; o míssil só pode ser lançado se o adversário estiver afastado o suficiente, sendo também que é o ataque mais fraco, apesar de ter o maior alcance;

- o míssil só tem efeito se o adversário for realmente atingido pelo projéctil, ou seja, o efeito não é imediato;
- o adversário, se for controlado por inteligência artificial, deverá inicialmente não ter conhecimento de nenhum tipo de ataque; aprenderá a realizar acções consoante as que o adversário realizar, tendo em conta a circunstância em que a acção foi tomada.

Por "jogador" entenda-se que pode tanto ser um humano como o computador, sendo que em termos de mecânica de jogo a única diferença que existirá entre os dois é o mecanismo de escolha de acção, como se verá.

Com esta informação, implementaram-se alguns protótipos como prova de conceito para alguns dos mecanismos listados, de modo a antecipadamente compreender a sua exequibilidade. Por exemplo, um dos mecanismos testados foi o lançamento do míssil, garantido que se conseguia criar um míssil, definir o seu comportamento de movimento e detectar uma colisão com o adversário. Este passo é importante de modo a assegurar que as regras de jogo propostas no GDD possam ser cumpridas na fase de produção.

A interface proposta deve ser minimalista, devendo mostrar pouco mais que uma barra representante da quantidade de vida restante para cada jogador, com a identificação do mesmo. Analogamente deverá haver uma barra de energia para cada jogador. Adicionalmente, deverá colocar-se um menu simples que coloque o jogo em pausa e permita reiniciar ou retomar a luta, com a opção também de limpar o que a inteligência artificial já sabe, efectivamente recomeçando do zero.

5.2 Produção

Esta é a fase principal do desenvolvimento, onde os conteúdos são criados e todos os comportamentos desejados são implementados.

5.2.1 Arte

Sendo que a arte não é o foco deste projecto, decidiu-se simplificar o processo de criação do personagem, dando-lhe apenas uma forma humanóide básica que satisfizesse as condições de aplicação de ossos típicos de deformação da malha.

A grande maioria do trabalho necessário para obter uma personagem funcional foi feito no ambiente de desenvolvimento *Blender*. Este fornece as ferramentas propícias (e gratuitas) para modelação, texturização, *rigging*, *skinning* e animação de objectos 3D.

O personagem foi criado usando as formas geométricas mais básicas possíveis, como cilindros ou esferas, reduzindo na medida possível o número de vértices utilizados por personagem. Não foi realizada texturização na malha dos personagens, pois foi decidido que lhe seria aplicada simplesmente um conjunto de cores fixas a determinadas partes da malha. Utilizou-se um *addon* do *Blender*, chamado *Rigify*, que já permite adicionar um esqueleto completo com vários ossos de controlo. Alguns ossos necessitaram de alguns ajustamentos para melhor se adequarem à malha. O processo seguinte, de *skinning*, implica associar cada vértice da malha ao osso mais apropriado. Apesar do *Blender* ter uma funcionalidade para automatizar este processo, o resultado nem sempre é o mais desejado, pois alguns vértices podem ficar erradamente associados. Considerando que a malha é composta de formas geométricas básicas, a associação de vértices aos ossos torna-se algo directa, sendo então um trabalho manual não muito dispendioso. Por fim, utilizando os ossos de controlo providenciados pelo *Rigify*, pode-se activar o modo de animação da malha, e definir as posições dos ossos ao longo de um período de tempo. Neste ponto realça-se a importância das animações terem os valores de transformação dos ossos iguais na trama de início e de fim. Se tal não acontecesse, os ciclos de animação poderiam apresentar erros derivados da interpolação incorrectamente calculada entre tramas. Uma visualização do processo pode ser resumidamente visto na Figura 5.2.

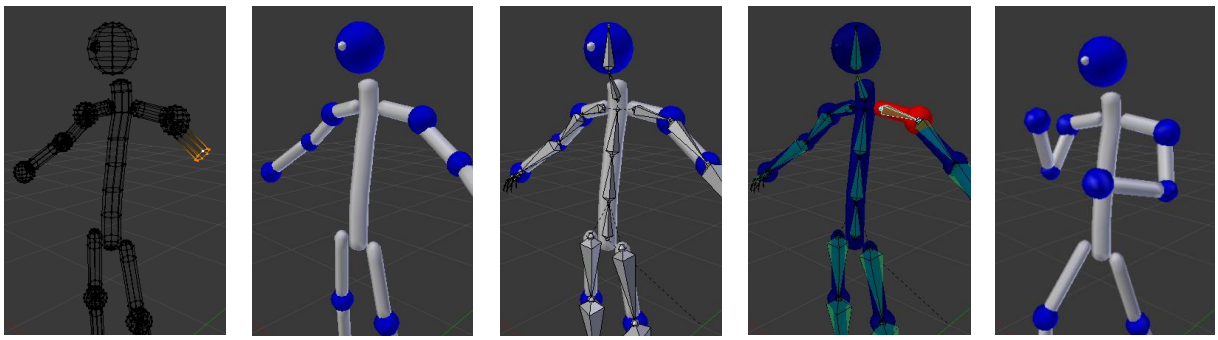

Figura 5.2: Etapas da criação do personagem: modelação; coloração; *rigging*; *skinning*; animação

Com os conteúdos criados, resta exportá-los para um ficheiro de formato FBX (**F**ilm**b**o**x**) e importá-los para o contexto de um projecto na plataforma *Unity*, que será a utilizada para este projecto.

Já no contexto da plataforma *Unity*, começou-se por criar o nível que seria usado como ambiente de jogo, utilizando as formas básicas presentes na plataforma. Não se fez isto em *Blender* pois o plano era este nível ser composto de formas geométricas simples, particularmente cubos, permitindo definir de imediato as caixas de colisão, também cúbicas e ajustadas às paredes e chão do nível. Tendo-se obtido algumas texturas de um repositório *online*, bastou importá-las convenientemente para o *Unity* e associá-las directamente aos objectos através da interface facilitadora da plataforma. De seguida, na Figura 5.3, mostra-se um exemplo de um nível possível.

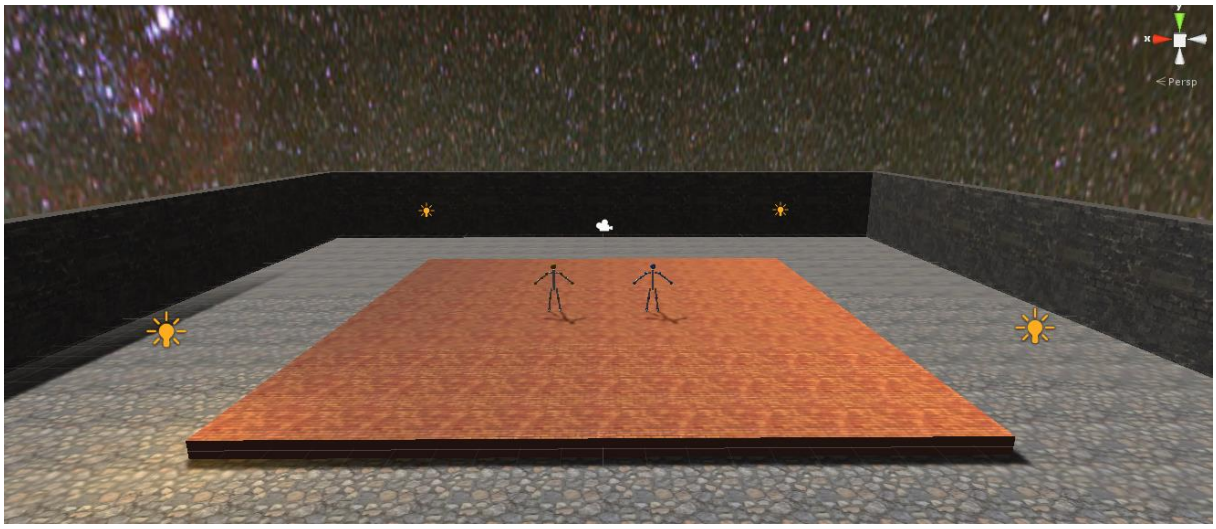

Figura 5.3: Nível de teste: um ringue com os dois personagens; notem-se as texturas repetidas em mosaico

De seguida importou-se a malha do personagem criada. No entanto, como o *Unity* não é compatível com todos os ossos criados pelo *addon Rigify* no *Blender*, existe um processo manual a realizar de modo a compatibilizar o objecto 3D com o motor. O *Rigify* cria um número de ossos desnecessários, sendo também que as convenções de nomes dados aos ossos são muito específicas e têm de ser garantidas. De notar também que ao personagem foi adicionado um objecto de colisão em cápsula (em forma de elipsóide), típico para personagens em ambientes de jogo, de modo a haver interacção física com os vários objectos em cena.

As animações podem ser incluídas no mesmo ficheiro FBX onde está incluído o personagem. Dito isso, as animações têm várias opções de importação, sendo que uma particularmente importante para animações baseadas em ciclos (como o movimento) é a opção "*Loop Time*" que permite automaticamente repetir a animação quando esta chega ao seu final. O *Unity* mostra inclusivamente se a animação em causa tem um ciclo fluido definido ou não e permite fazer ajustes à sua duração para corrigir eventuais erros.

O *Unity* tem um mecanismo de controlo de animações útil para combinar as várias animações no decorrer do jogo, denominado *Animation Controller*. Este controlador é implementado na forma semelhante a uma árvore de estados, em que, segundo certos eventos ou regras, existem transições entre esses estados. Cada um desses estados representa uma animação. Por exemplo, um objecto pode transitar da pose de inactividade para a de movimento consoante a sua velocidade obedecer a certas condições. O controlador tem também o conceito de camadas de animação, que podem ser usadas para substituir ou complementar outras camadas de animação em ossos específicos (utilizando uma *Avatar Mask*). Assumindo uma camada base com a animação do movimento, poderá existir uma camada superior que substitua apenas as partes superiores do corpo do personagem, o que seria útil no caso do soco quando este é activado, por exemplo, sendo que o resto do corpo continuaria a animação normal. Os eventos que despoletam as transições entre estados podem ter variáveis que são usadas nas condições de transição. Estas variáveis podem ter os seus valores alterados programaticamente (numa afirmação anterior falou-se de velocidade, por exemplo).

É de alguma importância que este trabalho anterior seja feito em primeiro lugar. Muitos destes mecanismos serão utilizados na parte da programação, nomeadamente as colisões entre objectos e as variáveis criadas no *Animation Controller*.

5.2.2 Programação

O foco deste projecto está na parte de programação. É nesta fase que várias decisões de *game design* terão de ser feitas, pois alguns factores só surgem ao longo do desenvolvimento, tais como limitações da linguagem ou da plataforma, independentemente de terem ou não sido feitos protótipos iniciais.

É verdade que o grande objectivo é implementar um mecanismo de inteligência artificial, mas é necessário que exista uma base sólida de jogabilidade sobre a qual esta se vai assentar.

Conceptualmente, como já foi referido, a única diferença entre um jogador humano e um artificial é o método com que os personagens recebem as acções a tomar (Figura 5.4). No caso do humano seria através da interface por dispositivos, particularmente o teclado numa fase inicial. Contudo, para a classe inicial do jogador, abstraiu-se imediatamente o fornecedor de acções, de modo a ser fácil definir outros controladores para os jogadores.

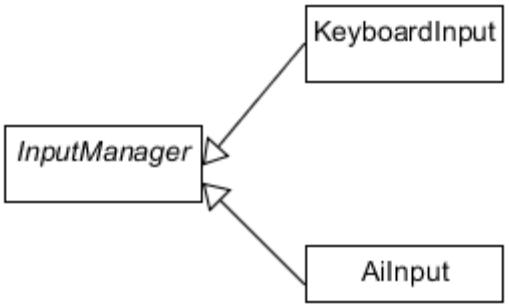

Figura 5.4: Tipos de input de acções e movimentos

Apesar do código estar implementado em inglês, utilizar-se-ão os termos respectivos em português para manter a fluidez da leitura do texto.

Uma das classes centrais em termos de jogo é a classe *Player*, ou Jogador, que possui um conjunto de características e propriedades relativas ao mesmo, tais como a quantidade de vida e de energia ou a sua velocidade de movimento actual. A plataforma *Unity* permite a associação directa de classes com objectos de jogo através da sua interface, o que abstrai o utilizador de muitos processos intermédios e possibilita a concepção imediata de código relevante ao jogo. Para além disso, certas palavras-chave da sintaxe de programação têm significado para a plataforma (exemplo: a palavra-chave "*public*" aplicada a um atributo é interpretada pela plataforma e permite a edição directa do valor através da interface, Figura 5.5).

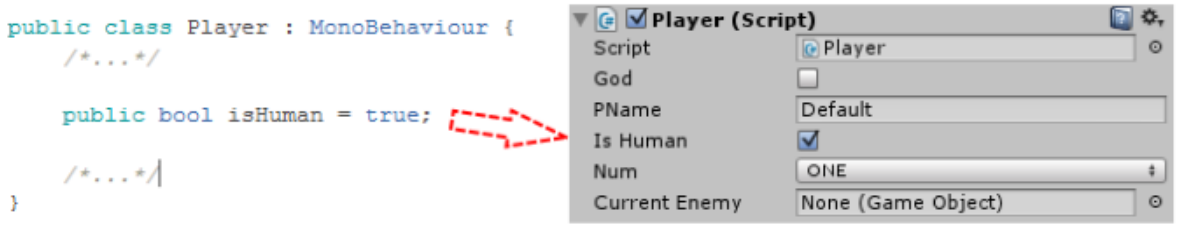

Figura 5.5: Exemplo de um atributo *public*

Uma propriedade existente no Jogador é a definição binária do controlo ser humano ou artificial. Isto permitirá logo no início do jogo instanciar um método de controlo ou outro. Este método de controlo, denominado *InputManager*, também é um atributo do Jogador, e é usado em todos os ciclos do jogo para obter um vector de movimento e uma acção, se existirem. A classe Jogador, tendo sido criado através do *Unity*, deriva por predefinição da classe *MonoBehaviour*, integrante do motor de jogo. É esta classe que disponibiliza métodos fundamentais, tais como o método *Update()* ou métodos de detecção de colisão.

O método *Update()* referido é particularmente relevante pois nele estará implementado o comportamento que o objecto deverá ter a cada ciclo de jogo. No caso do Jogador, isto apenas implica dois passos principais: Mover e Agir.

Para se mover, o Jogador obtém um vector de movimento do seu método de controlo. Se este vector for nulo (de magnitude 0), então o personagem não se move. Caso contrário, o personagem aumenta a sua velocidade e prossegue na direcção obtida. A velocidade aumenta consoante uma aceleração até uma velocidade máxima pré-estabelecida.

Antes de se prosseguir para o método como o Jogador age, considera-se necessária uma explicação da estrutura que define as acções possíveis: a classe *Action* (Acção).

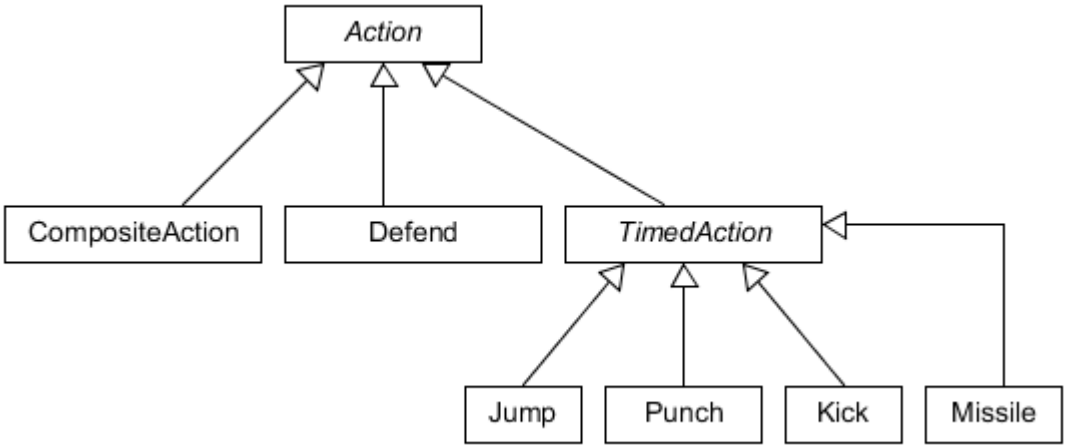

Figura 5.6: Diagrama da classe Action e suas derivadas

Deve ser possível obter do controlo objectos do tipo abstracto Acção, como se ilustra na Figura 5.6. Cada objecto derivado deste tipo terá a sua própria implementação de como agir e de que transformação causará no jogo. Deste modo diferencia-se a acção Soco da acção Salto utilizando no entanto a mesma estrutura. Considerando que é possível que numa só ocasião seja necessário executar mais que uma acção, criou-se uma *CompositeAction* (Acção Composta), que encapsula mais do que uma acção numa só. A execução desta acção consiste na execução de todas as acções encapsuladas. A Acção *TimedAction* (Acção Temporizada) provém da necessidade de definir um intervalo de tempo entre as mesmas acções para impedir um número irrealista de acções por segundo.

Qualquer Acção realizada pelo Jogador tem primeiramente de passar por um gestor de acções, de nome original *ActionManager*. Este gestor é responsável por garantir uma lista de requisitos:

- se a acção for composta, gere-se cada uma das acções encapsuladas individualmente;
- se a acção for temporizada, então é adicionada a uma lista auxiliar que mantém registo de todos os tipos de Acções temporizadas que foram recentemente usadas; esta lista é actualizada ao longo do tempo quando um certo tipo de acção já poder voltar a ser executado;
- realiza-se uma verificação final de algumas condições específicas à execução de cada acção (por exemplo, para se saltar é necessário que o Jogador esteja no chão).

Um último ponto relevante sobre o Jogador é a sua componente de colisões. Esta impede os jogadores de se intersectarem ou trespassarem, permite a colisão com mísseis inimigos, e possibilita a detecção de uma ocasião de *ring out* (quando o Jogador colide com o chão fora do ringue, devidamente identificado).

A classe *GameController*, ou controlador de jogo, pode ser considerada o centro de toda a jogabilidade, pois guarda informação sobre os Jogadores, permitindo que muitas outras classes consigam aceder a esta informação estaticamente para verificarem condições ou alterarem valores dos jogadores.

Finalmente, depois de uma visão geral sobre a estrutura e outros pormenores sobre a jogabilidade básica do jogo, entra-se agora na componente fundamental da inteligência artificial, particularmente focada na aprendizagem. Contudo, existe também uma componente mais simples de IA baseada em agentes reactivos, com mecanismos de reacção a estímulos.

Repete-se que o objectivo do agente, como doravante se denominará o jogador artificial, é partindo de um estado de desconhecimento total das acções disponíveis conseguir observar, incorporar e tomar decisões com a evolução das acções realizadas pelo adversário. Para tal utilizar-se-á uma variante dos vários mecanismos de aprendizagem existentes baseados em recompensas associadas a estados-acção.

Seguem-se as descrições de alguns conceitos chave do processo de inteligência artificial criado:

- Evento (*Event*): um evento pode ser activado em várias situações, sendo que é um dos responsáveis pela aprendizagem; todos os eventos são observados pelo agente, e através deles obtém um estado, uma acção e uma recompensa;
- Estado (*State*): uma representação do estado do ambiente de jogo num dado momento; é composto de um conjunto de características;
- Característica (*Feature*): uma descrição de alguma propriedade do ambiente;
- Acção (*Action*): o método de interacção do agente com o ambiente;
- Recompensa (*Reward*): o valor positivo ou negativo associado a um evento.

O mecanismo de inteligência artificial é o centro do processo, sendo apenas implementado numa classe (*AIMecanism*). O processo de aprendizagem molda-se resumidamente nos seguintes passos:

1. Um evento é despoletado. Isto pode ocorrer tanto por parte do adversário como do próprio agente, permitindo que este também tenha em conta o resultado das próprias acções. O evento não ocorre necessariamente apenas como consequência de uma acção, pois é provável que ocorram situações onde isto fosse indesejado (exemplo: um míssil falha o adversário, batendo na parede; aí pode ser despoletado um Evento sem intervenção directa de uma acção);
2. O evento gerado é adicionado a uma lista de eventos por processar (no caso de num só ciclo de jogo ser detectado mais do que um evento);
3. O mecanismo de IA adquire os eventos, se existirem, da lista mencionada, e assimila-os para uma estrutura que relaciona um par Estado-Acção com um valor de utilidade;
4. O agente pode utilizar o mecanismo de IA para obter uma acção a tomar através de uma função de selecção de acção, fornecendo para isso apenas o seu estado actual;
5. A função de selecção de acção combina o estado fornecido com todas as acções possíveis conhecidas e determina qual o par Estado-Acção que apresenta maior utilidade;
6. É devolvida a acção correspondente à maior utilidade, e o agente apenas terá de a reencaminhar para o seu gestor de acções.

Programaticamente, faça-se notar que a lista que transporta os eventos do ambiente de jogo para o mecanismo de IA foi feita com recurso a uma variável estática, que poderá não parecer imediatamente a solução mais modular, mas tendo em conta a simplicidade relativa do jogo não se achou necessário adicionar mais complexidade com pouco benefício.

A estrutura de dados usada para relacionar os pares Estado-Acção com o seu valor de utilidade foi um Dicionário. Com esta estrutura a indexação é mais directa e abstraída para o utilizador do dicionário. No entanto, existem alguns pontos a considerar antes de se poder utilizar livremente esta estrutura.

Na linguagem C# qualquer classe pode ser utilizada como chave de um dicionário, bastando para isso que tenha uma implementação sua do método base *GetHashCode*(), substituindo-o.

Este método é utilizado para representar um objecto na forma de um inteiro, podendo este ser utilizado para uma indexação interna no dicionário.

Ora, por omissão, uma classe que não tenha a sua própria implementação do método referido irá utilizar a versão base que se baseia na representação binária do objecto. Isto poderá não ser útil, pois poderá ocorrer que um objecto que represente a mesma informação tenha um código *hash* diferente. É essencial que os pares Estado-Acção tenham um código *hash* inequívoco para diferentes instâncias do mesmo par.

O par Estado-Acção está abstraído numa classe auxiliar (*StateAction*) que no geral apenas serve como contentor dos dois objectos. O seu método de *hash* (que é a função relevante para a indexação no dicionário) calcula um código baseado nos métodos de *hash* do Estado e da Acção. Assumindo um inteiro de 32 bits, os primeiros e últimos 16 bits estão reservados aos códigos de *hash* do Estado e da Acção, respectivamente. Esta solução não é final, pois não está assegurado que os códigos de *hash* de ambos os objectos tenham no máximo 16 bits relevantes. A Figura 5.7 mostra uma representação do mapeamento dos bits.

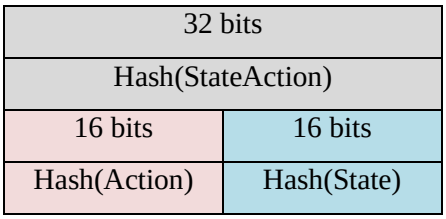

Figura 5.7: Mapeamento do código hash de um Estado-Acção

É agora responsabilidade de cada um dos objectos de definir o seu método de *hash*. Cada Acção tem um identificador único definido no GDD. Sendo único, é um candidato a ser usado como código *hash*, visto também que o número de acções feitas não é grande. Isto implica algum trabalho manual de manutenção dos códigos das acções, podendo inclusivamente acontecer um lapso na implementação deste método numa das funções.

Um estado é composto de várias características ambientais. O seu método de *hash* será baseado numa conjunção destas características num valor, ou seja, mais uma vez delega-se a determinação do valor de *hash* para os atributos do objecto. Cada característica terá de determinar o seu valor de *hash* através de condições específicas, que variam com o contexto

da característica. Por exemplo, a distância entre os jogadores é uma característica, que é ponderada e amostrada para um de quatro valores possíveis. Cada um desses valores representa a distância de soco, pontapé, míssil e um meio-termo (onde nenhuma acção tem efeito). Dito isto, com apenas dois bits consegue-se ter informação bastante relevante em termos de jogabilidade e em termos de decisão para a IA (Figura 5.8). De modo a economizar o máximo espaço num inteiro, as outras características calculam valores semelhantes, de modo que se pode mapear o espaço ocupado por cada característica no código *hash*.

<table>
<tr><td colspan="4">16 bits</td></tr>
<tr><td colspan="4">Hash(State)</td></tr>
<tr><td>…</td><td>1 bit</td><td>1 bit</td><td>2 bits</td></tr>
<tr><td>…</td><td>Hash(EnemyMissile)</td><td>Hash(EnemyDefense)</td><td>Hash(Distance)</td></tr>
</table>

Figura 5.8: Mapeamento do código hash de um Estado

Figurativamente, e concluindo, mostra-se na Figura 5.9 uma representação final do par Estado-Acção:

<table>
<tr><td colspan="5">32 bits</td></tr>
<tr><td colspan="5">Hash(StateAction)</td></tr>
<tr><td>16 bits</td><td colspan="4">16 bits</td></tr>
<tr><td rowspan="3">Hash(Action)</td><td colspan="4">Hash(State)</td></tr>
<tr><td>…</td><td>1 bit</td><td>1 bit</td><td>2 bits</td></tr>
<tr><td>…</td><td>Hash(EnemyMissile)</td><td>Hash(EnemyDefense)</td><td>Hash(Distance)</td></tr>
</table>

Figura 5.9: Mapeamento final do código hash de um Estado-Acção

O último conceito principal a falar no mecanismo de aprendizagem é o reforço. Cada evento terá associado um reforço, que será constantemente positivo ou negativo para cada tipo de evento. Cada acção tem tipicamente um conjunto de eventos associados, que normalmente são um evento para informar do sucesso ou insucesso da acção, apesar de existir a possibilidade de se adicionar tipos de eventos mais variados. Os eventos de sucesso e insucesso têm, respectivamente, um valor de reforço positivo e negativo.

O mecanismo de IA, recebendo o evento que contém o estado do ambiente, acção executada e reforço associado, irá incorporar estas informações no dicionário. Essa incorporação baseia-se apenas na soma do reforço ao já existente no indexado pelo par Estado-Acção. Esta implementação diverge enormemente da que é típica em agentes aprendizes, em que é comum utilizar os processos de aprendizagem SARSA e Q-Learning. Periodicamente, o agente irá decidir uma nova acção a tomar. Para isto terá de passar por um processo de selecção de acção. Este processo consiste na análise dos valores de utilidade das acções conhecidas pelo agente para um certo estado. O estado é a única informação dada ao selector de acção. A partir deste, o processo verifica todos os pares Estado-Acção existentes no dicionário, verificando quais é que realmente já existem e desses qual o que tem o maior valor positivo de reforço.

Definiu-se que apenas a utilidade positiva é realmente relevante para a selecção de acção. Caso contrário, existiria a hipótese de o agente conscientemente executar uma acção que sabe que o prejudica. Normalizam-se os valores de utilidade das acções que são consideradas relevantes para cada estado, através da divisão pelo somatório desses valores. Uma particularidade deste processo dá-se agora. Os valores normalizados, cuja soma totalizará 1, não serão usados para determinar o valor máximo correspondente à melhor acção. Os valores serão, na verdade, usados como probabilidade dessa acção ser executada. A razão poderá não ser clara, e daí requer-se uma explicação suplementar: apesar de objectivamente ser mais lógico escolher sempre a melhor acção possível para um certo estado, a jogabilidade do humano contra o agente pode tornar-se monótona, pois o agente estará sempre a repetir a mesma acção para um mesmo estado. Se a escolha de acção for baseada numa probabilidade de execução das acções que conhece, o agente tomará a melhor acção a maioria das vezes, sendo, todavia, que poderá também com menor probabilidade escolher uma outra acção, também ela com reforço positivo, caso aplicável.

Adicionalmente, o selector de acção tem uma probabilidade de simplesmente escolher uma acção aleatória do leque de acção disponíveis, sem ter em conta factor algum. É possível que o agente seja prejudicado por uma acção indevida, mas o facto também possibilita que o agente possa aprender, a nível muito básico, algumas situações convenientes de combate, como por exemplo na eventualidade feliz de aleatoriamente escolher dar um soco quando está próximo do adversário, apesar da decisão não ter tido em conta essa característica ambiental. Este tipo de comportamento é comummente designado de *ε-greedy*.

O movimento do agente também é aprendido através de observação do jogador adversário. Existem 4 acções que definem o movimento em quatro direcções respectivas. Com a estrutura baseada em eventos já implementada, bastou definir-se quando é que os eventos de aprendizagem de movimento são gerados. Neste caso, cada vez que o adversário se desloca uma certa distância parametrizável numa certa direcção, é gerado um evento de aprendizagem para essa direcção, com a condição de que essa deslocação tenha sido ininterrupta. Sendo assim, o agente irá mover-se se as circunstâncias de execução forem as mesmas das que aprendeu.

Os componentes principais do jogo estão explicados. Existe um conjunto de pormenores adicionais que são menos relevantes, alguns muito específicos à plataforma *Unity* em uso, e outros que são apenas características menores de jogabilidade.

Por exemplo, alguns ataques, quando bem-sucedidos, têm um efeito de afastamento do adversário. Esta táctica pode ser usada para empurrar o adversário para fora do ringue, garantido a vitória. Utilizando o motor de forças interno da plataforma, para concretizar este efeito basta adicionar uma força na direcção contrária à que o adversário está virado.

Como retoque final embelezou-se o nível de jogo de modo a retirar-lhe o aspecto banal e simples dos materiais predefinidos do *Unity*. Adicionaram-se luzes, texturas e alguns objectos decorativos. Os personagens em si não sofreram alterações estéticas, no entanto.

Toda a fase de produção foi marcada pela evolução constante do GDD, pois algumas decisões ou características só foram pensadas e adicionadas no decurso desta fase. Um outro aspecto fundamental foram os testes que se aplicavam ao jogo à medida que novas funcionalidades eram implementadas, o que muitas vezes provou que alguma mecânica de jogo não estava bem implementada em termos de programação ou de design inicial.

5.3 Pós produção

Esta fase do desenvolvimento é pouco relevante pois é uma fase maioritariamente relacionada com a manutenção do jogo depois de acabado e lançado. Ora, essa situação não se aplica ao contexto presente.

6 Conclusão

O foco do trabalho descrito neste documento era, inicialmente, no estágio realizado na *Biodroid Entertainment*, e na descrição das tarefas do autor do mesmo enquanto membro da equipa de programação. Entretanto, esse foco foi repartido para um protótipo de jogo desenvolvimento de forma paralela e independente do estágio referido.

Enquanto estagiário, o objectivo era adquirir um conjunto de competências básicas e gerais na indústria dos videojogos, na área da programação. Todavia, sendo uma indústria que combina vários mundos profissionais diferentes (informática, gestão, arte, etc.), o estagiário teria de aprender a lidar com várias particularidades dessas áreas diferentes.

No caso do protótipo independente, o propósito era criar um jogo de luta em que um agente controlado pelo computador conseguisse aprender a lutar por observação das acções do adversário.

Apesar de realizar as duas componentes do trabalho ao mesmo tempo poder parecer uma tarefa ambiciosa, pode-se dizer que em ambas o objectivo foi cumprido. No contexto do estágio, o estagiário adquiriu conhecimentos profundos de ferramentas técnicas como o *Git* e os motores de jogo *Unity* e *Unreal Engine 4*, ao mesmo tempo que desenvolveu capacidades de trabalho de equipa, especialmente na interacção que teve com colegas de outras áreas.

O protótipo cumpriu o requisito mínimo de se mostrar que o agente consegue aprender as acções a realizar consoante o valor dos reforços gerados por essas acções. O que ficou a faltar neste protótipo foi um número maior de acções disponíveis para serem aprendidas, para o agente aparentar ser mais complexo do que realmente é; igualmente faltou uma quantidade aceitável de testes ao jogo para a descoberta de eventuais erros.

Com toda a experiência adquirida de todo o trabalho, pode-se concluir que a indústria dos videojogos é extremamente gratificante em termos académicos pela quantidade enorme de conceitos diferentes envolvidos. Nesse sentido, verificou-se que a preparação dada ao estagiário no decorrer da sua licenciatura e mestrado deu-lhe um excelente ponto de partida para ser produtivo especificamente nesta área.

A última afirmação é tão verdade que o estagiário foi convidado a permanecer na *Biodroid* após o seu estágio, para continuar o seu trabalho e crescimento pessoal. Igualmente, o desenvolvimento do protótipo paralelo aguçou a vontade de se continuar a criar novos jogos com novos desafios interessante para resolver.

7 Bibliografia

[1] https://unity3d.com/pt/unity/multiplatform Unity - Multiplatform [Consult. 29 Set. 2014];

[2] http://docs.unity3d.com/Manual/Navmeshes.html, Unity - Manual: Navigation Meshes [Consult. 29 Set. 2014];

[3] http://docs.unity3d.com/Manual/EnablingCharToNavigate.html, Unity - Manual: Enabling a Character to Navigate [Consult. 29 Set. 2014];

[4] http://docs.unity3d.com/Manual/SL-Reference.html, Unity - Manual: Shader Reference [Consult. 30 Set. 2014];

[5] http://docs.unity3d.com/Manual/BlenderAndRigify.html, Unity – Manual: Using Blender and Rigify [Consult. 30 Set. 2014];

[6] http://docs.unity3d.com/Manual/LoopingAnimationClips.html, Unity – Manual: Looping Animation Clips [Consult. 30 Set. 2014];

[7] http://docs.unity3d.com/Manual/Animator.html, Unity – Manual: Animator and Animator Controller [Consult. 30 Set. 2014];

[8] http://docs.unity3d.com/Manual/class-AvatarBodyMask.html, Unity – Manual: Avatar Mask [Consult. 30 Set. 2014];

[9] https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/Editor/index.html, Unreal Engine | Material Editor Reference [Consult. 30 Dez. 2014];

[10] https://docs.unrealengine.com/latest/INT/Engine/Blueprints/index.html Unreal Engine | Blueprints Visual Scripting [Consult. 29 Set. 2014];

[11] http://www.tutorialspoint.com/cplusplus/cpp_templates.htm, C++ Templates [Consult. 30 Dez. 2014];

[12] http://msdn.microsoft.com/en-us/library/windows/desktop/bb172357(v=vs.85).aspx, Bilinear Texture Filtering (Direct3D 9) (Windows) [Consult. 30 Dez. 2014];

[13] http://img2.wikia.nocookie.net/__cb20130129215136/tekken/en/images/9/99/Tekken_1_Fiji.png, Imagem exemplo do jogo *Tekken* 1 [Consult. 30 Set. 2014];

[14] http://upload.wikimedia.org/wikipedia/en/d/de/Street_Fighter_II.png, Imagem exemplo do jogo *Street Fighter II* [Consult. 30 Set. 2014];

[15] http://www.pcgamesabandonware.com/mod/upload/pc_pt/images/ba/9a/56/ce/0a/9b/fa/26/e8/ed/9e/10/b2/cc/8f/46/mkombat2.jpg, Imagem exemplo do jogo *Mortal Kombat 2* [Consult. 30 Set. 2014];

[16] http://www.cgtextures.com/, [CG Textures] – Textures for 3D, graphic design and Photoshop! [Consult. 30 Set. 2014];