Criação de Recursos 3D e Implementação de Áudio num Videojogo (The Abyss)

uma dissertação escrita por

Tiago Alexandre Ferreira dos Santos Almeida

e supervisionado por

Agostinho Gil Teixeira Lopes

Professor Auxiliar, ISMAI

Cláudia Sofia Borlido de Freitas

Professor Assistente, ISMAI

Mestrado Tecnologias de Informação, Comunicação e Multimédia

Universidade da Maia - ISMAI (ISMAI)

6 January, 2023

Resumo

Neste documento vai ser falado brevemente, sobre o que são os videojogos para ter um contexto sobre o tema geral deste projeto, que é um videojogo, desenvolvido por Bruno Monteiro, João Peneda, João Ribeiro e Tiago Almeida.

O foco deste documento vai ser em modelação 3D e animação, o uso das ferramentas do motor de jogo para a implementação de áudio nas personagens, inimigos, NPCs e mapas. Vai ser apresentado a história dos videojogos, o processo de fazer um videojogo, os vários tipos de motores de jogos, como contar histórias através de videojogos e as várias ferramentas para a criação de personagens, animações e implementação de áudio.

Vai haver um capítulo de desenvolvimento que vai apresentar o contexto narrativo do jogo e como será aplicado na prática. Irá detalhar os processos feitos para a criação de modelos 3D, as ferramentas encontradas para a criação de materiais e texturas e como criar um "esqueleto"que permite mover a figura criada e criar animações. O capitulo termina com implementação de áudio feita a partir do motor de jogo. Isto detalha as configurações feitas no motor e *scripts* criados para a implementação.

No capítulo de testes, encontram-se os testes feitos ao videojogo na parte de performance com modelos detalhados e modelos *LowPoly* e na parte de som. Também se encontra um questionário feito a um grupo de pessoas que experimentaram o videojogo e as respostas obitdas desse questionário.

O documento termina com a conclusão chegada a partir do trabalho feito e dos resultados obtidos pelos testes, com um subcapítulo que fala do trabalho futuro.

Palavras-Chaves: Definição dos videojogos. História dos videojogos. Motores de Jogos. Narrativa. Modelação. Animação. Som.

Abstract

This document will briefly talk about what videogames are, to have a context about the general theme of this project, which is a videogame, developed by Bruno Monteiro, João Peneda, João Ribeiro and Tiago Almeida.

The focus of this document will be on 3D modeling and animation, the use of game engine tools to implement audio on characters, enemies, NPCs and maps. It will be presented the history of video games, the process of making a video game, the various types of game engines, how to tell stories through video games and the various tools for creating characters, animations and audio implementation.

There will be a chapter about the development of the game, that will present the narrative context of the game and how it will be applied in practice. It will detail the processes used to create 3D models, the tools found for creating materials and textures and how to create a "skeleton" that allows moving the created figure and creating animations. The chapter ends with audio implementation made from the game engine. This details the settings made inside the engine and *scripts* created for the implementation of sound.

In the tests chapter, find the tests made to the videogame in the performance part, with detailed models and *LowPoly* models and in the sound part. There is also a questionnaire made to a group of people who tried the video game and the responses obtained from this questionnaire.

The document ends with a conclusion based on the work done and the results obtained by the tests, with a subchapter that talks about future work.

Keywords: Definition of video games. Video game history. Game Engines. Narrative. Modeling. Animation. Sound.

Agradecimentos

A realização desta dissertação de mestrado foi possível devido ao suporte de algumas pessoas.

Como orientadores, o professor Agostinho Gil Teixeira Lopes e professora Cláudia Sofia Borlido de Freitas, por acompanharem o desenvolvimento do projeto estarem sempre disponíveis para qualquer dúvida que o grupo coloca-se.

O professor Pedro Filipe Cruz Pinto pelos conselhos dados para o desenvolvimento da dissertação.

Conteúdo

Lista de Figuras

Lista de Tabelas

Siglas

AI Inteligência artificial

FPS First Person Shooter

IDE Integrated Development Environment

ISMAI Universidade da Maia - ISMAI

KB Kilobyte

MTICM Mestrado em Tecnologias de Informação Comunicação e Multimédia

NPC Non-Player Character

RPG Role Playing Game

UI User Interface

WWDC Apple's Worldwide Developers Conference

Glossário

Assets Neste caso são os ativos usados para construir o videojogo

Cross-platform Algo que existe em várias plataformas, como por exemplo algo que existe em Windows, Linux e MacOS.

Demo Demostração de algo

Ficheiro FBX Formato usado para trocar geometria 3D e dados de animação

FPS (First Person Shooter) Género de um videojogo em que tem uma perspetiva de primeira pessoa com uma jogabilidade de tiros, traduzido em português, atirador em primeira pessoa

Gameplay Jogabilidade do jogo

IDE (Integrated Development Environment) Ambiente de Desenvolvimento Integrado.

KB Kilobyte, uma unidade de medida para ficheiros de computador

LowPoly (Low Polygon) Modelos 3D com uma quantidade de polígonos baixa

Networking Criar uma rede online

NPC(Non-Player Character) Personagens em videojogos que o jogador não controla

Open Source Termo em inglês que significa código aberto, ou seja, o código fonte não tem licença e é partilhada com toda gente

PS4/5 Consola de jogos Playstation 4/5

RPG (Role Playing Game) Tipo de jogo em que os jogadores assumem papéis de personagens e criam narrativas colaborativamente

Scripting Programar / criar programas script

Start-ups Uma empresa recém-criada ainda em fase de desenvolvimento

Capítulo 1

Introdução

Neste capitulo vai ser apresentado o contexto, a motivação para o desenvolvimento deste projeto e os objetivos a serem concluídos. O desenvolvimento do videojogo foi dividido entre os membros do grupo, Bruno Monteiro, João Peneda, João Ribeiro e Tiago Almeida. Foram criadas duas equipas, uma equipa artística e uma equipa de programação, em que dentro de cada equipa cada membro tem um papel a cumprir, como se pode observar na figura 1.1.

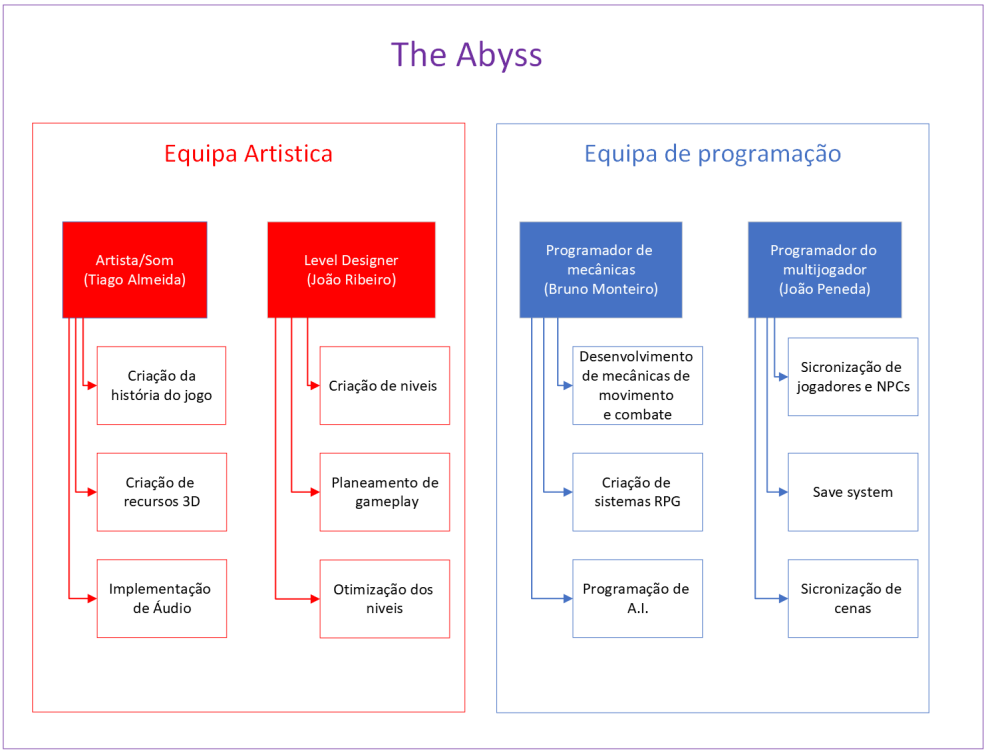


Figura 1.1: Esquema de divisão de trabalho

1.1 Contexto: Definição de videojogo

Nicolas Esposito [1] escreveu como uma possível definição, "*Um videojogo é um jogo que jogamos graças a um aparelho audiovisual e que pode ser baseado numa história*", e apesar da sua definição ser curta e simples, não deixa de ser verdade, porque antes de ser uma forma de arte ou de narrativa, um videojogo continua a ser um jogo. Apesar da sua evolução ao longo dos anos, os videojogos continuam a ter aspetos em comum, como um objetivo e obstáculos que impedem o jogador de alcançar o fim, e isto, no fundo constitui o que é um jogo. Mas depois falta a parte do vídeo, e isto varia mais devido à evolução da tecnologia, desde um osciloscópio em 1958, onde apareceu o primeiro videojogo "*Tennis for Two*"[2] que mais tarde tornou-se o "*Pong*"[3], até agora que se pode encontrar videojogos a serem jogados em óculos de realidade virtual. Depois de estabelecer o elemento de jogo no videojogo, começa-se a pensar num tipo de progresso ou narrativa que puxe o jogador a continuar a jogar até ao fim e aqui entra a história, que muitas vezes é posto como o foco do videojogo e o gameplay como uma forma de experienciá-la em primeira pessoa. Outras vezes a história só existe para dar contexto ao jogador, mas não é necessária para desfrutar do jogo, como o jogo *Doom*[4].

1.2 Problemática e Motivação

Como o objetivo final é o desenvolvimento de um videojogo, vão-se encontrar vários obstáculos com a complexidade do projeto e o tempo disponível, por isso as tarefas foram divididas em quatro, para cada membro do grupo, assim o desenvolvimento tornou-se mais eficiente. O grupo também decidiu que o projeto final seria no mínimo um demo com um nível completo em que quatro jogadores podem jogar em simultâneo contra inimigos com a sua AI, Inteligência Artificial, já programada.

Este documento foca-se mais na parte da modelação 3D, animação e áudio e também vai ser encontrado obstáculos, como saber optimizar os modelos para não consumir muito do hardware quando estão a ser renderizados dentro do jogo. Problemas que podem ser encontrados com a implementação do áudio, é conseguir fazer com que os sons sejam dinâmicos que variam dependendo da distância dos jogadores e dos materiais que estão a ser interagidos.

A motivação para ter sido escolhido este trabalho como projeto de mestrado, foi querer contar uma história de forma interactiva com as tecnologias disponíveis ao público, poder criar recursos 3D próprios, como modelos de armas, animações e poder trabalhar com áudio e usá-lo para criar um mapa de sons com o áudio nas animações dos jogadores, nos objetos e nos mapas.

1.3 Objetivos

O objetivo final do projeto é criar um videojogo com multijogador até quatro jogadores, com um nível completo em que contém inimigos com AI e elementos RPG. Os objetivos para este projeto são:

- Escrever a história para dar contexto às mecânicas do jogo.
- Aprender e usar a ferramentas de modelação para a criação de modelos 3D (armas, personagens, estruturas para mapas) e animação.
- Criar recursos 3D otimizados (*Low Poly*) para o jogo.
- Criar animações para os recursos 3D criados.
- Usar as ferramentas do motor de jogo para sincronizar os sons com as animações dos jogadores.
- Sincronizar os sons com as animações de ataque.
- Sincronizar os sons com as animações dos inimigos.
- Implementar sons no ambiente dos níveis.
- Implementar um sistema que substitui os sons usados nos passos do jogador dependendo no material interagido.

1.4 Organização

Este documento vai ser dividido em sete capítulos, Introdução, Estado da arte, Planeamento, Desenvolvimento, Testes, resultados e discussão, Conclusão e Trabalho futuro.
No capítulo de Introdução será falado sobre a definição dos videojogos, a problemática e motivação e objectivos.
No Estado da Arte será exposta a história dos jogos, o processo de fazer um jogo, os motores de jogos, e aqui será distinguido entre o Unity e o Unreal Engine, modelação e animação, a narrativa em videojogos e som nos videojogos.
No Planeamento será exposta a visão geral, métodos a serem usados, o projeto a desenvolver e a sua implementação, os testes a realizar, os recursos e um cronograma planeado.
No Desenvolvimento será falado do trabalho feito para este projeto na parte da história, criação de recursos 3D, *shading*, animação e a implementação do som.
No capítulo de Testes, resultados e discussão, vão ser colocados os testes feitos na parte de modelação e na implementação de som. Também vai ser apresentado a recolha de dados através de um questionário distribuído junto com uma versão de teste do videojogo a um grupo de voluntários.
No capítulo de Conclusão vai ser apresentada a conclusão do trabalho feito.
No capítulo de Trabalho futuro é apresentado os planos de expansão para a parte de modelação e som no videojogo a ser desenvolvido.

Capítulo 2

Estado da Arte

Neste capítulo vai ser apresentado a história dos jogos, o processo para criar um jogo, vários motores de jogo, as suas funcionalidades e uma comparação entre eles, *software* para animação e modelação, narrativa nos jogos e implementação de áudio.

2.1 História dos Jogos

A história dos videojogos começa nas décadas de 1950 e 1960, quando o primeiro videojogo foi criado, "*Tennis for Two*", e mais tarde em 1961 foi criado o jogo "*Spacewar*", considerado o primeiro jogo para computador, foi lançado em 1962 e ocupava 2KB [5].
Na década de 1970 viu-se o nascimento das máquinas de videojogos, em 1972 a Atari foi fundada e dominou a indústria dos videojogos durante esta década com o lançamento do jogo "*Pong*"[3], o primeiro jogo a ser sucesso global. Em 1977 lançou a primeira consola doméstica a Atari 2600 e em 1978 lançou o jogo "*Space Invaders*"[6] em que marcou uma era de ouro para *arcade* [5].
Na década de 1980 foi quando apareceram os clássicos como o "*Pac-man*"(1980) [7], "*Ultima*"(1980) [8], "*Mario Bros*"(1983) [9], "*Tetris*"(1984) [10] e "*SimCity*"(1989) [11] e com isto a entrada da Nintendo à competição de consolas com a sua própria consola a NES (Nintendo Entertainment System) [5].
Na década de 1990 foi quando os videojogos começaram a ser apresentados em três dimensões e foram lançadas as consolas Playstation (Sony) e Nintendo 64 (Nintendo). Também foi nesta altura que os jogos começaram a ser usadas como um meio para contar

histórias [5].

Na década de 2000 começaram a aparecer jogos online, como o "*Counter Strike*"(2000) [12] e "*World of Warcraft*"(2004) [13] [5].

Na década de 2010 a indústria dos videojogos tornou-se um negócio de biliões de dólares, ouve um crescimento enorme de estúdios independentes e pode-se jogar videojogos em qualquer lado, até nos telemóveis. Vários jogos começaram a optar por uma estética foto realística e com diálogos e narrativas dignas de um filme, até ao ponto de ter atores de cinema em videojogos, exemplo disto no jogo "*Death Stranding*"(2019) [14] [5].

2.2 O processo de fazer um jogo

O processo de criação de um videojogo pode ser dividido entre cinco a seis passos [15]:

1. Conceptualizar o jogo: Conceptualizar o mundo, as personagens, a narrativa, a sua jogabilidade, etc.

2. Recolher informação: Experimentar outros jogos do mesmo género que a ideia conceptualizada, ver palestras de outros criadores de jogos, decidir que tipo de ferramentas utilizar e em que linguagem é que o jogo será programado, quão grande será o trabalho, etc.

3. Começar a criação: Começar a programar o jogo e criar um protótipo.

4. Polir o conceito: Melhorar o protótipo e começar a criar o mundo, as suas mecânicas e narrativa.

5. Testar o jogo: Fazer testes à procura de erros e testar as mecânicas do jogo, para verificar se é necessária alguma alteração.

6. Comercializar o produto final: Criar uma página na internet e fazer publicações em redes sociais para fazer publicidade ao jogo, entrar em contacto com o máximo de plataformas relevantes para a distribuição e criar um demo para poderem experimentar o videojogo e para receber opiniões.

2.3 Motores de jogos

Provavelmente a peça mais importante para criar um videojogo é um motor de jogo que, segundo António Andrade [16] na publicação da sua pesquisa, diz que "O objetivo principal de um motor de jogo é abstrair recursos comuns do videojogo, permitindo a reutilização de códigos e ativos de jogos em diferentes jogos." Para esta finalidade os motores de jogos têm várias funcionalidades, como um motor de renderização para gráficos 2D e 3D, gestão de entradas (teclado e rato, comandos de consola, etc.), recalcular eventos a cada frame, um motor de física para colisões e a sua resposta, gestão de som, animações 2D e 3D, gestão de memória e a possibilidade de ter múltiplos processos a trabalhar em paralelo. Os motores de jogos também podem ter *scripting*, inteligência artificial, networking e publicação para várias plataformas. Teoricamente qualquer um pode escolher um motor de jogo e criar um videojogo, e apesar de existir casos disto acontecer (ex. The first tree em 2017 por David Wehle[17]), na prática isto não é verdade, pois apesar dos motores de jogo facilitarem o trabalho ao utilizador, a criação de um videojogo requer conhecimento em linguagens de programação como C++, C#, Java ou Python. Existem vários motores de jogos conhecidos, como o Unity, Unreal Engine, CryEngine, Godot, RPG Maker, GameMaker: Studio, etc. Este documento vai apresentar alguns dos motores de jogo listados acima, comparando as suas características e diferenças, para ser feita uma escolha para o desenvolvimento do jogo.

2.3.1 Unity

O motor de jogo Unity foi anunciado em 2005 na WWDC (Apple's Worldwide Developers Conference) e, desde então, produziu grandes mudanças na indústria de videojogos [16]. O Unity consegue exportar para múltiplas plataformas, ao momento em que este documento está a ser escrito, o Unity consegue exportar (com o mesmo código base) para: iOS, Android, Windows PC, Linux, PS4, PS5, Xbox One, Xbox Series S/X, Oculus Rift, Nintendo Switch, Andoid TV, tvOS, Google Stadia, Microsoft HoloLens, Magic Leap e WebGL [18]. O Unity é oferecido gratuitamente para criadores independentes ou a empresas com uma faturação anual inferior a 100.000 dólares americanos, havendo também uma versão Pro que inclui vários recursos para uso avançado. Programação em Unity

pode ser feito com UnityScript, linguagem personalizada com uma sintaxe semelhante ao JavaScript, ou C#, que é a opção recomendada. O Unity consiste de um editor visual e um IDE, o que torna o Unity um motor de jogo intuitivo e fácil de usar para fazer protótipos rápidos. É possível criar jogos simples sem escrever uma linha de código, ao momento em que este documento está a ser escrito, as versões mais recentes do Unity vêm incluído com o novo *Visual Scripting* que (baseado no *Blueprint Visual Scripting* do Unreal Engine) permite ao programador criar mecânicas e comportamentos no videojogo sem saber programação. Na prática a entidade básica de jogo é chamado de *GameObject* e uma *Scene* (cena) é um contentor para todos os *GameObjects*. Estes *GameObjects* podem ser várias coisas como, o chão do nível, personagens, objetos que o jogador pode interagir, cada um com os seus próprios componentes e mecânicas. O Unity tem incorporado uma loja de recursos e que, como o nome indica, é uma loja onde os criadores podem aceder a recursos feitos pela comunidade e estes podem ser gratuitos ou pagos.

2.3.2 Unreal Engine

Acabou de ser anunciado a nova versão do Unreal Engine, o Unreal Engine 5 e apesar de já estar disponível gratuitamente ao publico, ainda é uma versão de testes por isso este documento vai-se focar na versão estável, o Unreal Engine 4. O Unreal Engine 4, como o Unity, tem suporte de publicação em múltiplas plataformas, como: Windows PC, PlayStation 5, PlayStation 4, Xbox Series S/X, Xbox One, Nintendo Switch, Google Stadia, macOS, iOS, Android, AR, VR, Linux, SteamOS e HTML5 [19]. A linguagem usada em programação no Unreal Engine é C++, mas o Unreal Engine 4 introduziu um recurso novo que torna possível a criação de videojogos sem a necessidade de saber linguagens de programação, que é chamado *Blueprint Visual Scripting*. Estas *Blueprints* (plantas/esquemas) consistem de uma interface à base de blocos, em que dentro destes blocos já estão configuradas e programadas mecânicas de jogo. O objetivo deste recurso é para ajudar *Level* e *Game Designers* a visualizar uma parte do videojogo ser ter que estar dependente das mecânicas estarem prontas pelos programadores, e também ajudam criadores iniciantes que não têm muito conhecimento e experiência em linguagens de programação [16]. O Unreal Engine, como Unity, também tem uma loja de recursos onde os criadores podem aceder e trabalhar com recursos feitos pela comunidade. A maior diferença entre o

Unreal Engine e o Unity é que o código fonte do Unreal é aberto para subscritores, o que possibilita melhorias feitas pela comunidade e aprendizagem para criadores de motores de jogos.

2.3.3 Godot

Segundo a documentação oficial do Godot [20], o Godot Engine é um motor de jogo multiplataforma com recursos para criar jogos 2D e 3D a partir de uma interface unificada. Ele fornece um conjunto de ferramentas comuns, para que os utilizadores possam-se concentrar em criar jogos sem precisar reinventar a roda. "*Os jogos podem ser exportados para várias plataformas, incluindo as principais plataformas (Linux, macOS, Windows), bem como plataformas móveis (Android, iOS) e baseadas na web (HTML5). Godot é totalmente gratuito e de código aberto sob a licença permissiva do MIT.*"[20] O desenvolvimento do Godot é totalmente independente e orientado pela comunidade, capacitando os utilizadores a ajudar a moldar o seu mecanismo para atender às suas expectativas. É suportado pelo *Software Freedom Conservancy* sem fins lucrativos. O Godot usa como linguagens de programação o C# (como o Unity) em que o suporte ainda é relativamente novo [21], C++ (como o Unreal), *Visual Scripting* e uma linguagem própria chamada GDScript, em que este foi criado de raíz para maximizar o potencial do Godot sem precisar de muitas linhas de código. O Godot também tem uma loja de recursos, como os motores antes referidos.

2.3.4 CryEngine

O CryEngine é um motor de jogo desenvolvido pela Crytek para suportar todos os seus jogos, começando com o FarCry [22] em 2004 e Crysis [23] em 2007 e é mais conhecido por conseguir desenvolver jogos ultra realísticos. Como o Unreal Engine, o CryEngine é gratuito com um sistema de *royalties*, que foi adotado em 2018. Isto consiste em que os primeiros 5.000 dólares americanos anuais por projeto são gratuitos e não é necessário pagar *royalties*, mas depois desse valor os utilizadores terão que pagar cinco porcento em *royalties* para a crytek. Os utilizadores também podem optar por uma licença que dá mais suporte e acesso a outras ferramentas [24]. As linguagens de programação usadas são o C++ e Lua, e têm uma loja de recursos que é disponível a qualquer utilizador,

independente se possui uma licença ou não.

2.3.5 GameMaker Studio 2

O GameMaker Studio é um motor de jogo criado com utilizadores que não são programadores em mente, por isso usa um editor *Drag and Drop*, que é um tipo de *Visual Scripting* parecido com os que podem ser encontrados nos motores mencionados. O GameMaker também usa uma linguagem própria chamade de GML (*Game Maker Language*), esta é uma linguagem feita para ser orientada a objetos, fácil de aprender e de usar.
O motor de jogo é mais focado em jogos 2D, mas também tem capacidade para 3D e os jogos produzidos podem ser exportados para plataformas *desktop* como Windows, Linux e MacOs, plataformas móveis como Android e também pode ser exportado para consolas, como a *Playstation*, Xbox e Nintendo. O Gamemaker Studio tem um plano gratuito, mas tem poucas opções de exportação em relação aos vários tipos de subscrições pagas disponíveis.

2.3.6 Motores a considerar

De acordo com várias estatísticas, incluído a publicada na página *Game Developer*, escrita por Marcus Toftedahl [25] e publicada em 2019, os motores de jogos mais usados são o Unity e o Unreal Engine, tal como se pode verificar na tabela 2.1:

Tabela 2.1: Lista de motores usados em jogos publicados na Steam [25]

Motor de Jogo	Número de Projetos	% total de jogos identificados
Unreal Engine	1726	25.6
Unity	889	13.2
Source	270	4.0
CryEngine	238	3.5
Gamebryo	215	3.2
IW (Infinite Ward Engine)	192	2.9
Anvil	166	2.5
Id Tech	113	1.7
Essence	73	1.1
Clausewitz	68	1.0
Jogos identificados com diferentes motores	3266	48.4

A tabela 2.2 mostra os motores de jogo mais usados na plataforma Itch.io e como se pode observar, ao fazer a soma dos dados, mesmo contando com jogos repetidos, o Unity é o motor de jogo com mais projetos criados, o que torna este o motor mais popular :

Tabela 2.2: Lista de motores usados em jogos publicados na Itch.io [25]

Motor de Jogo	Número de projetos	% total de jogos
Unity	24200	47.3
Construct	6275	12.3
GameMaker	5643	11.0
Twine	3184	6.2
RPG Maker	1982	3.9
Bitsy	1683	3.3
Pico-8	1479	2.9
Unreal	1458	2.8
Godot	1274	2.5
Ren'Py	1008	2.0
Jogos com outros motores	2993	5.9
Número total de jogos	51179	

Na tabela 2.3 pode-se observar uma comparação dos pontos principais dos diferentes motores de jogo:

Tabela 2.3: Comparação dos diferentes motores

Motor de Jogo	Linguagem	Plataformas	Versão gratuita	Custo	Loja de Recursos
Unreal Engine	C++	Windows PC, PlayStation 5, PlayStation 4, Xbox Series S/X, Xbox One, Nintendo Switch, Google Stadia, macOS, iOS, Android, AR, VR, Linux, SteamOS e HTML5	Sim	5% em royalties quando publicado numa loja que não seja a Epic	Sim
Unity	C#/JavaScript/Boo	iOS, Android, Windows PC, Linux, PS4, PS5, Xbox One, Xbox Series S/X, Oculus Rift, Nintendo Switch, Andoid TV, tvOS, Google Stadia, Microsoft HoloLens, Magic Leap e WebGL	Sim	Necessita de uma licença depois de passar os 100.000 dólares americanos na faturação anual	Sim
Godot	C#/C++/GDScript	iOS, Android, Windows PC, Linux, PS4, Xbox One, Oculus Rift, Nintendo Switch, HTML5	Sim	Gratuito	Sim
CryEngine	C++/Lua	Windows, Linux, Playstation 4, Xbox One, Oculus Rift, OSVR, PSVR, HTC Vive	Sim	5% em royalties depois de passar os 5.000 dólares por projeto	Sim
GameMaker Studio 2	GML (Game Maker Language)	Windows, Linux, macOs, Playstation4, Playstation 5, Xbox One, Xbox Series X/S, Nintendo Switch, iOS, Android, Amazon Fire, Android TV, tvOS, HTML5	Sim	Tem múltiplas licenças para poder ter várias funções e opções de exportação	Sim

Depois de analisar estes dados sobre os diferentes motores de jogo, foi decidido mover o foco para estes dois motores (Unity e Unreal Engine) e como o tema principal deste documento é sobre a implementação de áudio no jogo, teve que ser analisada a linguagem de programação e as ferramentas disponíveis dentro dos respetivos IDEs. Isto é apresentado no capítulo 2.6.2 Implementação nos motores de jogo.

2.4 Narrativa em videojogos

Usar videojogos para contar uma história é uma coisa relativamente nova comparado a livros e filmes, por isso maior parte das vezes as pessoas dizem que jogos só servem para distrair e nada mais. Mas se os últimos anos mostraram alguma coisa foi que às vezes

um videojogo é o melhor meio para transmitir uma emoção uma mensagem do criador para o público, porque os videojogos conseguem colocar o observador na história a ser contada. Existem várias formas de contar histórias em videojogos. Existem jogos como "*Dark Souls*"[26] e "*Doom*"[4], que a história não é o ponto principal do videojogo, mas ela está presente nas personagens, no diálogo, no mundo e nos pedaços de informação dispersos pelos níveis do jogo, em que o jogador pode simplesmente ignorar e continuar a avançar no jogo até o completar. Depois existem jogos como o "*Heavy Rain*"[27] e "*Disco Elysium*"[28] em que tudo no jogo anda à volta da história e, como estes dois exemplos, dão a oportunidade ao jogador de fazer as suas próprias escolhas e mudar história de acordo com elas. Anthony Jondreau no seu artigo entitulado de "*The Unique Storytelling Power of Video Games*" [29] refere que o jogo "*Kingdom Come: Deliverance*"[30] é um dos seus jogos favoritos porque a história do jogo está diretamente ligada às mecânicas do jogo. "*O jogo conta a história de Henry de Skalitz, filho de um ferreiro que se encontra no meio de um grande conflito geopolítico. Para sobreviver, a personagem Henry deve aprender a lutar, ler, andar a cavalo, disparar um arco, falar com a nobreza e fazer poções. E para que Henry aprenda a fazer tudo isso, o jogador deve aprender a fazer tudo isso ao mesmo tempo.*"[29] Isto é um exemplo de um jogo RPG em que o progresso da personagem e do jogador estão ligados, ou seja, a personagem para concluir a sua história tem de aprender como é que o mundo funciona e aprender novas técnicas. Ao mesmo tempo o jogador tem de aprender como é que as mecânicas do jogo funcionam para poder avançar e terminar o jogo e à medida que isto acontece o jogo começa a ficar mais fácil para o jogador e a personagem controlada consegue ser mais eficiente no que está a fazer. No fim o jogador sente um sentimento de realização e quando for lembrar-se dos eventos do jogo, em vez de dizer "A personagem fez X", provavelmente vai dizer "Eu fiz X", porque experienciou a história do videojogo como ator em vez de observador.

2.5 Modelação e Animação

Apesar de vários motores de jogos terem já implementados recursos para criar animação e modelação 3D, vários estúdios optam por usar *software* especializado para este tipo de trabalho e nesta área existem vários produtos com licença paga e gratuita. *Software* de

modelação e animação inclui o Autodesk Maya, Houdini, Cinema 4D, Autodesk 3ds Max, Modo, Lightwave 3D, ZBrush, como opções pagas e com opções gratuitas temos o Blender, Daz Studio, SketchUp, Hexagon, Wings3D, Rocket 3F [31]. Desta lista de *software* os dois principais são o Autodesk Maya e o Blender, sendo Maya um dos *software* mais usados em grandes empresas de produção e o Blender a melhor solução gratuita mais encaixada para start-ups, mas com o potencial para projetos maiores e complexos.

2.5.1 Blender

O Blender é um *software* de criação 3D gratuito e de código aberto. Permite fazer todo o tipo de trabalhos em 3D - modelação, montagem, animação, simulação, renderização, composição e edição de vídeo. Os utilizadores avançados empregam a API do Blender para *scripts* Python para personalizar a aplicação e escrever ferramentas especializadas, frequentemente estes são incluídos nos lançamentos futuros do Blender. O Blender é adequado para uso pessoal e pequenos estúdios que beneficiam da sua *pipeline* unificada e processo de desenvolvimento responsivo. O Blender também é multiplataforma e funciona igualmente bem em Windows, Linux e Mac [32].

2.5.2 Autodesk Maya

O Autodesk Maya, normalmente chamado de Maya, é um *software* de modelação e animação 3D, preferido por grandes estúdios de animação e desenvolvimento de jogos. Alguns exemplos do seu uso em cinema inclui o "*O Senhor dos Anéis: As Duas Torres*"[33] e "*Star Wars: Episódio II - O Ataque dos Clones*"[34], na indústria dos videojogos existe como exemplos o "*Warframe*"[35] e "*Hades*"[36].

O Maya tem um teste gratuito, mas este tem múltiplas limitações [37] como, não poder exportar projetos, marcas de água e várias funcionalidades desativadas, para poder usar o Maya com todas as suas funcionalidades é necessário pagar uma licença de 1.700 dólares americanos anualmente ou 125 dólares americanos mensalmente [38], mas estudantes podem concorrer para uma licença de até três anos gratuitamente.

2.5.3 3ds Max

O 3ds Max é um *software* de modelação 3D da Autodesk e com isto tem muitas similaridades com o Maya. É um programa de modelação 3D, com um teste gratuito e também uma licença com o custo de 1.700 dólares americanos [39]. O 3ds Max é mais focado para modelação mais realista e desenhos de arquitetura, engenharia e construção e também usado no por vários estúdios de desenvolvimento de videjogos na construção das suas personagens. Exemplos incluem o jogos "*For Honor*"[40], "*Watch Dogs 2*"[41] e "*Dead by Daylight*"[42].

O teste gratuito do 3ds Max dura 30 dias, mas estudantes podem concorrer para uma licença de até três anos gratuitamente.

2.5.4 Software a considerar

Na tabela 2.4 fez-se uma comparação entre os *software* acima descritos para ser mais facilitada a escolha para o desenvolvimento do projeto:

Tabela 2.4: Comparação de *software* de modelação [43]

	Blender	Maya	3ds Max
Tem teste gratuito	—	Sim	Sim
Custo	Gratuito	Licença de 1700 dólares americanos anualmente ou 215 dólares mensalmente (estudantes e educadores têm acesso a uma licença de três anos)	Licença de 1700 dólares americanos anualmente ou 215 dólares mensalmente (estudantes e educadores têm acesso a uma licença de três anos)
Código Aberto	Sim	Não	Não
Definição	Foco em animação e efeitos visuais	Foco em texturas e animação	Foco em modelação, desenhos de arquitetura, engenharia e construção
Principal Uso	Escolha preferida em estúdios com projeto de grande escala	Desenvolvimento de jogos e arquitetura	Preferido para start-ups, animação 3D e efeito visuais

2.6 Implementação de Áudio em videojogos

O tipo de áudio num videojogo varia desde sons incorporados na personagem até música de orquestra a ser tocada para aumentar a o contexto narrativo e imersão do jogador. Para existir uma implementação de áudio é necessário compreender as ferramentas disponíveis, a narrativa no jogo e conhecimento do motor em que está a ser desenvolvido.

2.6.1 Áudio e imersão

Um bom exemplo de um jogo em que o som é a parte mais importante é o jogo de terror "*Perception*"[44]. Neste jogo o jogador é uma mulher cega e o jogo só apresenta o que a protagonista interpreta do mundo a partir dos sons, por isso o jogador navega pelo nível através de uma habilidade do género sonar. Isto torna-se extremamente importante a meio da história quando o jogador encontra um monstro que também navega através de som e para o jogador não perder, ele terá que parar de fazer som o que o torna efetivamente cego e surdo. O jogo funciona de forma a que o som é o recurso mais importante a ter que ser gerido para o jogador conseguir ganhar.

Um exemplo de um jogo onde o som age de uma forma mais passiva e ao mesmo tempo ter um papel ativo é o "*Journey*"[45]. Este jogo não tem muita ação. Por isso, maior parte do som está na música ambiente, que às vezes nem é muito aparente, excepto nos momentos mais altos do jogo, o que ajuda a transmitir a sensação de uma grande aventura ao jogador. Mas a parte mais interessante na sua implementação de áudio é o facto de que o jogo não ter vozes, nem narradores e não avisa que tem uma componente de multijogador. Por isso quando dois jogadores se encontram ficam surpresos e provavelmente vão querer comunicar, mas o jogo não tem um sistema de troca de mensagens. A única ferramenta disponível aos jogadores durante o jogo todo é a habilidade de fazer sons básicos com o premir de um botão. Assim os jogadores podem comunicar, mas de uma forma mais primitiva. Isto foi propositado para que o jogador se sinta isolado só com os sons presentes no jogo, incluindo quando encontra outros jogadores.

Noutra parte oposta temos jogos, como o mencionado no capítulo anterior ("*Doom*") [4], em que apesar de ter uma historia e diálogo, o ponto principal do jogo é ação. Por isso o som está mais focado no som das armas, movimento da personagem, inimigos e música.

Também existem jogos como o "*Disco Elysium*"[28] em que coloca o diálogo acima de tudo e assim o foco do som será colocado nos atores que têm de conseguir transmitir várias emoções através da sua atuação e os *designers* de som têm de produzir sons e música que combine com essas atuações para que o jogo tenha o máximo de impacto.

2.6.2 Implementação nos motores de jogo

Dependendo do motor de jogo, existem várias maneiras para implementar áudio num videojogo. Serão apresentadas de seguida as formas mais comuns de implementação no Unity e no Unreal Engine.

Unity

No Unity existem dois objetos chamados de "*AudioSource*"e "*AudioListener*", em que o primeiro é a fonte de onde vem o som que o programador está a implementar no jogo e normalmente é colocado dentro do objeto que vai reproduzir o som, por exemplo para uma personagem fazer um som, ele precisa de um *AudioSource* para poder reproduzir o som. O "*AudioSource*"tem múltiplas opções disponíveis dentro do Unity (como colocar o som em *loop*, mudar a área onde o som pode ser ouvido, etc.), mas o programador pode, num *script*, alterar o comportamento do *AudioSource* de outras formas para além das disponíveis. O *AudioListener* só pode existir um por cena e normalmente é colocado na câmara da personagem, assim o jogador ouve o som.
Para gerir a quantidade dos sons é pela criação de um *AudioManager*, que permite ao programador conseguir ditar o comportamento dos sons presentes na cena. Para sincronizar sons com as animações, o programador tem que encontrar na animação o momento em que deve ser reproduzido o som e colocar uma ordem, normalmente através de *script*, para o som ser reproduzido sempre que animação chegar a esse ponto.
Um método para reproduzir sons diferentes, dependendo no tipo de material interagido pelo jogador, é criado um *script* que guarda os dados associados em cada textura usada. Criar uma função que permite associar um som ao código da textura, assim o jogador se pisar, por exemplo, areia, será reproduzido o som que estará associado à textura de areia.

Unreal Engine

Para implementar áudio no Unreal Engine é preciso ficheiros de som do tipo *Wav* a serem colocados dentro do motor. Depois é necessário tornar esses sons em *Cues*. Isto é semelhante a tornar os sons em *AudioSources* no Unity, premitindo ao programador controlar vários aspetos dos sons. Para sincronizar os sons com a animação, o programador

precisar de escolher um ponto da animação e selecionar a *Cue* que quer reproduzir.

Para reproduzir sons diferentes, dependendo do tipo de material interagido pelo jogador, o Unreal Engine permite colocar uma identificação nos tipos de materiais usados e associar sons a esses materiais. Assim, por exemplo, se o jogador interagir com um chão identificado como madeira, é reproduzido o som de madeira.

Capítulo 3

Planeamento

Neste capitulo vai ser apresentado a visão geral do trabalho, os métodos utilizados, a implementação, os testes a realizar, os recursos utilizados e um cronograma do trabalho planeado.

3.1 Visão geral e métodos utilizados

Como já foi referido nos objetivos do capítulo 1, a meta final é criar um videojogo multijogador *online* com até quatro jogadores ao mesmo tempo e com pelo menos um nível completo. E para conseguir alcançar este objetivo num período de um ano foi necessário dividir as partes do projeto para os quatro membros do grupo. O método usado para esta dissertação sobre o desenvolvimento do videojogo foi o método de *practice-led research*, que é um tipo investigação realizada a fim de obter novos conhecimentos, por meio da prática e dos resultados dessa prática.

3.2 Projeto a desenvolvido e implementação

Para ter um videojogo funcional em pouco tempo e só com quatro pessoas, dividiu-se as tarefas em duas partes: programação e artística. A parte da programação foi trabalhada por dois membros, em que um focou-se na parte online, enquanto o outro membro focou-se na programação das mecânicas do videojogo e inteligência artificial. A parte artística foi trabalhada por outros dois membros em que um irá focou-se na parte de modelação 3D, criação de assets, animação e criação de história, o outro membro irá focou-se na

criação dos níveis do videojogo. Isto implica a criação do ambiente em que o jogador irá andar e decidir como é que estes níveis podem ser interagidos pelo jogador. Ao longo do desenvolvimento do projeto, foram feitas reuniões para juntar as duas componentes desenvolvidas, testar e avaliar o estado do videojogo e decidir o que abordar a seguir. No estado final do projeto, os membros do grupo focaram-se em polir o videojogo e depois distribuir o demo para um público de teste para receber feedback e fazer as alterações necessárias antes da entrega do produto final.

3.3 Testes a Realizar

Vai ser testado o videojogo para verificar se existem erros nas mecânicas, ambiente, jogabilidade e a parte online. Também é testada a otimização gráfica, ou seja, testar os requisitos do videojogo em computadores mais fracos e tentar otimizar os recursos, de modo, a que computadores mais fracos consigam executar o jogo sem muitos problemas. A experiência do utilizador também vai ser testada, durante o desenvolvimento do videojogo, para que o videojogo tenha uma experiência suave e para que o utilizador não se sinta perdido durante a sua sessão de jogo.

3.4 Recursos

Para o desenvolvimento deste projeto foi necessário alguns recursos básicos:

- GitHub - Plataforma usada para hospedar o projeto.
- GitHub Desktop - Aplicação para sincronizar todas versões e atualizações do projeto.
- Visual Studio 2019 - Software para programar os scripts usados no projeto.
- Blender - *Software* utilizado para a criação de assets.
- Unity - Motor de jogo usado para o desenvolvimento do videojogo.

3.5 Cronograma

Na figura 3.1 observa-se um cronograma que apresenta linhas com duas cores diferentes. Laranja e roxo, em que a cor laranja representa o desenvolvimento do projeto em geral e a cor roxa representa o trabalho em que este documento tem como foco.

Figura 3.1: Cronograma

No cronograma pode-se observar que no o projeto começa no dia 15/09/2021 com o planeamento e depois vem as várias etapas de desenvolvimento de um projeto, com estudos, recolha de dados, desenhos de um protótipo e inicio do desenvolvimento. Depois começa o trabalho que está presente neste documento que é a criação de assets, implementação do áudio e criação da história. Os passos finais são testes ao protótipo, ajustes dependendo dos resultados, distribuição de uma versão *beta*, análise do *feedback* dos *beta testers*, ajustes finais ao projeto e interpretação dos resultados para conclusões.

3.6 Requisitos

Nesta secção são apresentados os requisitos para o projeto final:

- R1: Sincronizar os sons com as animações de ataque.
- R2: Sincronizar os sons com as animações dos inimigos.
- R3: Implementar sons no ambiente dos níveis.
- R4: Implementar um sistema que substitui os sons usados nos passos do jogador dependendo no material interagido.

Capítulo 4

Desenvolvimento

Neste capitulo vai ser apresentado o que foi feito durante desenvolvimento do projeto, começando pela história do jogo, a criação dos recursos 3D e a implementação do som.

4.1 História

Esta secção vai ser divida entre o contexto narrativo, que irá contar a premissa do jogo, e a narrativa em prática que irá mostrar como é que a história é traduzida para jogabilidade.

4.1.1 Contexto narrativo

O contexto narrativo para este jogo é o seguinte: Depois de uma grande guerra que quase levou a humanidade à sua extinção foi encontrada uma cratera numa ilha desabitada. Esta cratera ocupava maior parte da ilha e o mais irregular é que dentro dela, o espaço e o tempo encontram-se distorcidos o que faz com que existam múltiplos tipos de biomas, desde zonas primitivas até cidades futuristas. Depois de um pouco de exploração foram feitas duas descobertas: a cratera tem vários recursos que podem ser usados para sobrevivência do que resta da humanidade e que, apesar da ilha estar desabitada, a cratera não. Foram vistos vários tipos de criaturas que mostram hostilidade contra quem tentar entrar na cratera, o que tornou a exploração dos seus recursos arriscada.
Os exploradores construiram uma aldeia, que no inicio foi evoluindo com os recursos trazidos pelos sacrifícios feitos, mas com o passar dos anos a aldeia começou a ficar estável e

as pessoas deixaram de explorar a cratera, porque achavam que os recursos não valiam as vidas perdidas.

No presente as únicas pessoas que ainda exploram a cratera são pessoas em busca de glória ou prisioneiros que têm a opção de trabalhar na cratera para descontar anos da pena em vez de desperdiçar o resto da vida na prisão.

Este é caso do jogador, um prisioneiro que recebeu uma pena de 100 anos pelo simples crime de roubar uma maçã e agora não tem outra hipótese a não ser trabalhar na cratera.

4.1.2 Narrativa em prática

A história vai começar na prisão onde lhe é ensinado os aspetos básicos para sobreviver no mundo e ao sair vai encontrar a aldeia, que vai servir para o jogador descansar e comprar equipamento e mantimentos para a viagem até à cratera.

Quando o jogador estiver pronto, ele pode encontrar um portal à porta da aldeia que o leva para a cratera em que depois o jogador vai atravessar os vários biomas para conseguir recolher os vários recursos, enquanto tenta sobreviver contra as criaturas que habitam a cratera. Para combater as criaturas o jogador vai poder escolher entre duas classes de personagem: o *Mage* e o *Warrior*. Cada uma destas classes têm acesso a um conjunto de habilidades únicas para essa classe. O objetivo final do jogador é descer até ao nível mais baixo da cratera enquanto derrota os *bosses* de cada nível e acabar com a pena imposta na personagem do jogador. Depois o jogador pode continuar a ir para a cratera, mas em contexto da história, já não é forçado.

Durante o percurso do jogo o jogador vai encontrar personagens que o vão ajudar, como um ferreiro que lhe repara o equipamento e um vendedor lhe pode vender novas armas poções para tornar a jornada mais fácil. No inicio do jogo estas personagens não vão mostrar empatia ao jogador apesar de oferecerem estes serviços, mas à medida que o jogador avança na cratera, as personagens vão começar a gostar mais do jogador e certos itens podem ser vendidos a preços reduzidos.

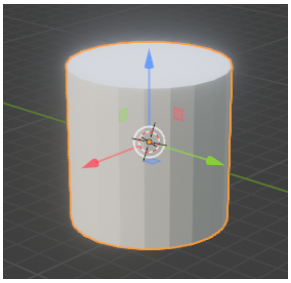
O jogador paga por estes serviços através de um tipo de moeda que pode ser apanhada dentro dos biomas na cratera.

4.2 Criação de recursos 3D

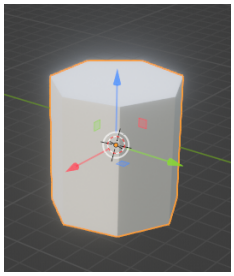
Nesta secção vai ser apresentado o trabalho feito na criação de recursos 3D e os problemas encontrados. Esta parte vai ser dividida em três partes: modelação, criação de materiais e texturas, *rigging* e animação. O trabalho descrito nestas partes vai ser apresentado em forma de tutorial para facilitar a explicação dos fundamentais usados nos processos na criação de recursos 3D.

4.2.1 Modelação

Na parte de modelação foram feitos vários modelos como: armas para os jogadores e inimigos usarem, peças para a construção dos níveis e uma personagem para prática de *rigging* e animação. Para a construção destes modelos tinha-se em mente um estilo *Low Poly*, para a otimização do jogo para o estilo do jogo em si. Para isto foram usadas duas técnicas. Uma técnica foi, ao adicionar peças ao modelo, essas peças eram criadas com o número de vértices reduzido, que a torna menos complexa. A outra técnica foi usada quando certos modelos já estavam completos na sua construção e continham mais vértices do que necessário, então usou-se um modificador de decimação (*decimate*), que permite reduzir o número de vértices e faces, com alterações mínimas ao objeto. Exemplo destas técnicas é demonstrada com as Figuras 4.1a e 4.1b.



(a) Cilindro sem alterações



(b) cilindro com vértices reduzidos

Figura 4.1: Exemplo de uma redução de vértices

Os primeiros modelos a serem criados foram peças para o nível da prisão, como as portas das celas e correntes. A primeira parte da porta foi feita a partir de um cilindro que teve as medidas da escala alterados até ter uma forma que parece-se com uma coluna e depois multiplicado várias vezes. A seguir foram adicionados dois cubos que foram modificados e colocados em cima e em baixo dos cilindros como apresentado na Figura

4.2.

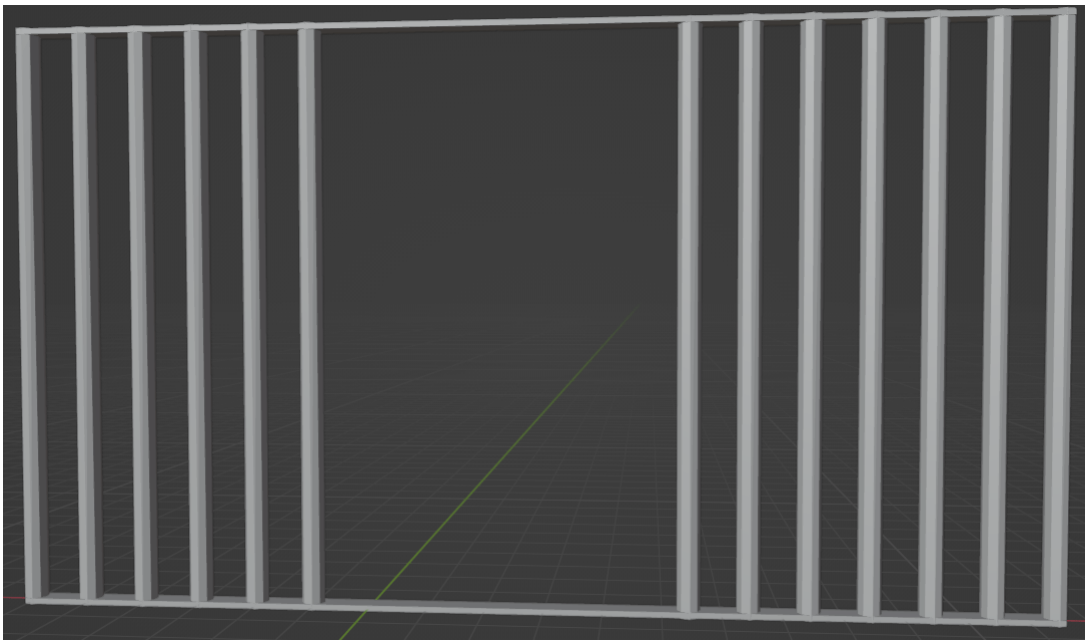
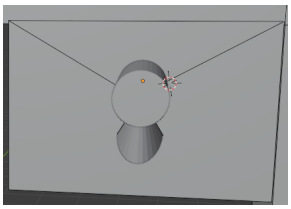


Figura 4.2: Inicio da porta da cela

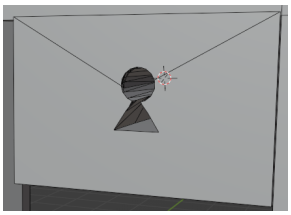
Ao repetir o processo anterior, mas mudando os tamanhos das Figuras, foi feita a porta em si que se observa na Figura 4.3.

Figura 4.3: Inicio da porta da cela

A fechadura da porta foi conseguida ao criar um cubo e de seguida colocar um pequeno cilindro e cone da forma apresentada na Figura 4.4a e usar um *modifier* chamado *bollean* que permite selecionar modelos que se intersetam e ao eliminá-los deixar a impressão deles atrás, como na Figura 4.4b.



(a) Cilindro e cone no cubo



(b) Fechadura terminada

Figura 4.4: *Modifier bollean*

Com a porta terminada, foi usada a opção *join* para juntar todas as Figuras em duas peças, a parte exterior da porta e a porta em si, a razão porque foi feita a divisão desta forma em vez de tornar tudo numa só peça foi como propósito de dar a opção de poder manipular a porta sem alterar o exterior, como abrir e fechar a porta, demonstrado na Figura 4.5.

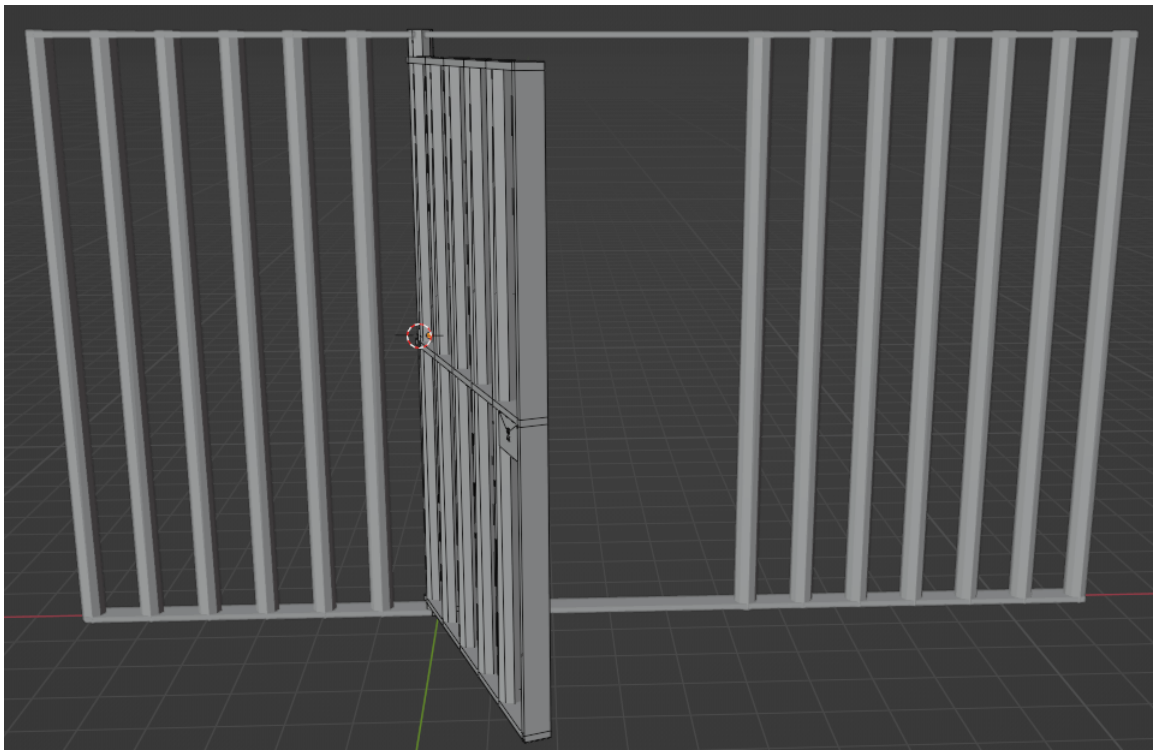
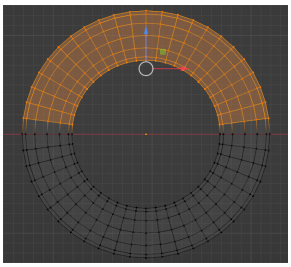
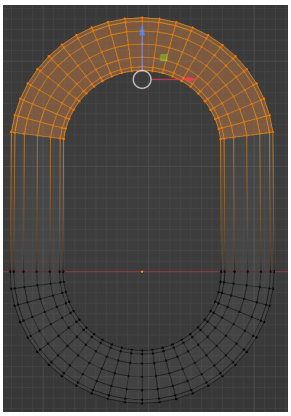


Figura 4.5: Cela com a porta aberta

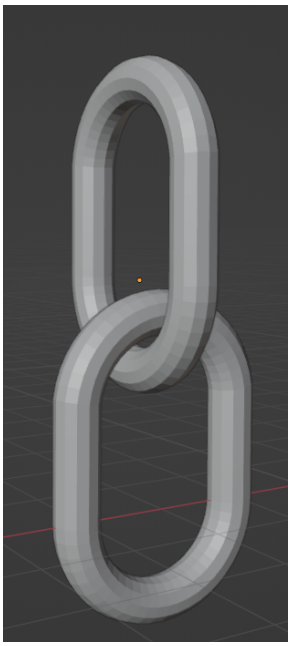
Para criar as correntes foi criado um toro e no modo de edição foi selecionado metade dos pontos foram esticados até ter a forma da ligação de uma corrente como demonstrado nas Figuras 4.6a e 4.6b. Depois essa figura foi duplicada, alterou-se a rotação e colocou-se na parte de cima da original como na figura 4.6c. Este processo foi repetido até ter um corrente suficientemente grande e no fim foi ligada a um toro inalterado, isto observa-se na Figura 4.6d.



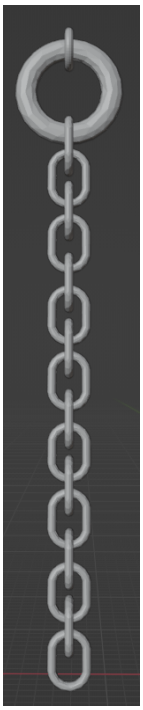
(a) Toro inicial



(b) Toro editado



(c) Primeira ligação



(d) Corrente completa

Figura 4.6: Construção de uma corrente

Depois foi feita uma cópia deste modelo e partir desta foi criado outro modelo a partir dos mesmos processos usados para a primeira corrente para ter o modelo apresentado na Figura 4.7.

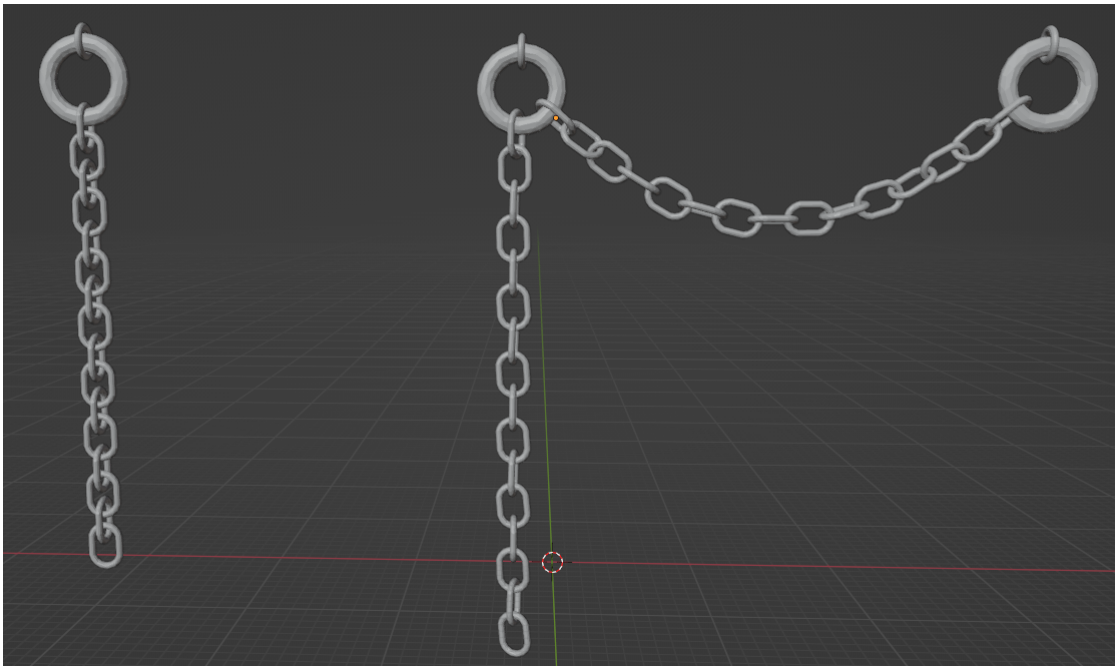
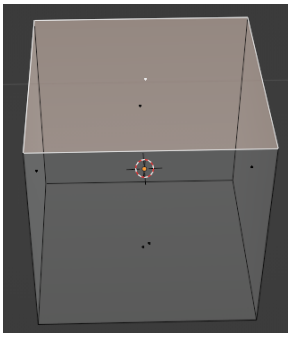
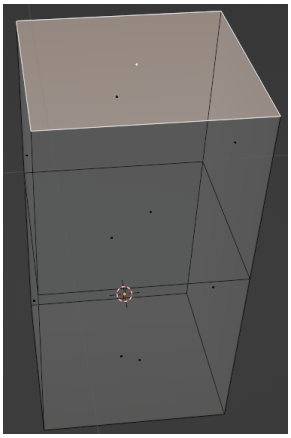


Figura 4.7: Correntes de prisão

Outras ferramentas usadas múltiplas vezes foram *Extrude Region* e *Loop Cut*. A ferramenta *Extrude Region* permite duplicar os vértices, mantendo a nova geometria conectada aos vértices originais. Os vértices são transformados em arestas e as arestas formarão faces, um exemplo deste processo é demonstrado nas Figuras 4.8a e 4.8b.



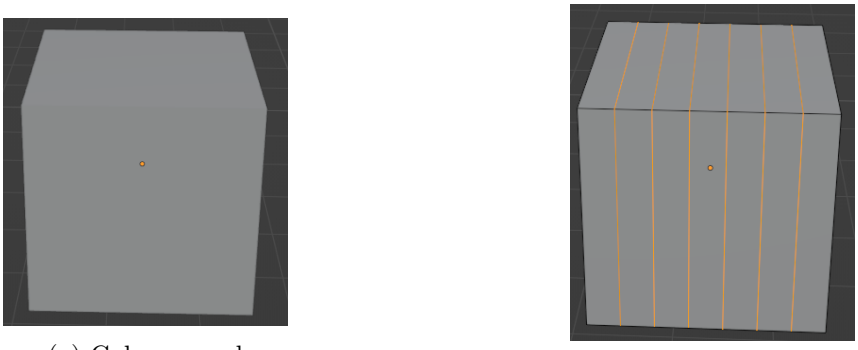
(a) Face selecionada



(b) Face duplicada

Figura 4.8: Exemplo de utilização da ferramenta *extrude*

A ferramenta *Loop Cut* permite dividir um loop de faces inserindo novos loops de aresta que cruzam a aresta escolhida, isto é demonstrado nas Figuras 4.9a e 4.9b.



(a) Cubo normal

(b) Cubo com seis loop cuts

Figura 4.9: Exemplo de utilização da ferramenta *loop cut*

Estas ferramentas e os processos anteriormente referidos foram usadas para fazer modelos mais complexos como armas para o jogador (Figura 4.10) e uma personagem (Figura 4.11).

Figura 4.10: Armaria de espadas

Figura 4.11: Primeira personagem

4.2.2 Criação de Materiais e Texturas

Antes de exportar os modelos para o Unity, primeiro têm que ter texturas e para isso é necessário criar materiais para as suas diferentes partes. Para criar materiais selecionou-se uma parte do modelo (ou o modelo em si se o material é para ser aplicado no modelo todo) e depois abriu-se as propriedades de material no editor. Nas propriedades criou-se uma material o que depois é apresentado múltiplas opções de personalização do material como alterar a superfície, a cor e *sliders* que alteram as propriedades metálicas, rugosas, transparência. Depois de ter escolhido os valores desejados, encontramos um resultado como o que se observa na Figura 4.12.

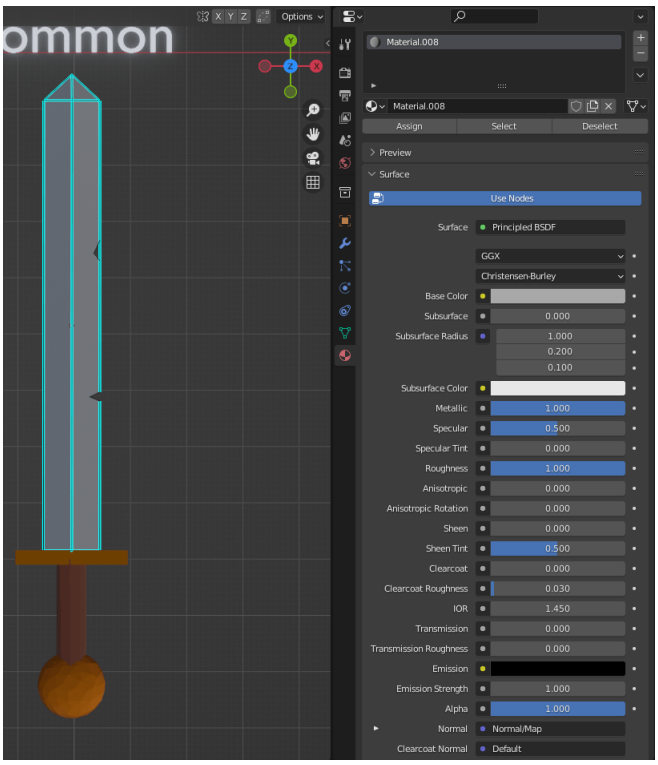


Figura 4.12: Material de uma lâmina

Depois de repetir este processo para todas as partes do modelo, este podia ser exportado para o jogo, mas isto não seria otimizado porque existem vários materiais "soltos" que depois no Unity teríamos que ligá-los a cada parte do modelo, por isso foi criado um material só com uma textura que incorpora todos os materiais usados. Para isso foi necessário tornar todas as partes num só modelo, por isso selecionou-se todas as partes, abriu-se o *Object Context Menu* e selecionou-se a opção *join*. Agora na janela de materiais

são apresentados todos os materiais usados no modelo, em vez de um material individual por parte. No painel de propriedades do objeto é encontrado um *"UV Map"* criado automaticamente. Um *UV Map* é um mapa de uma textura 2D para um modelo 3D, como apresentado na Figura 4.13.

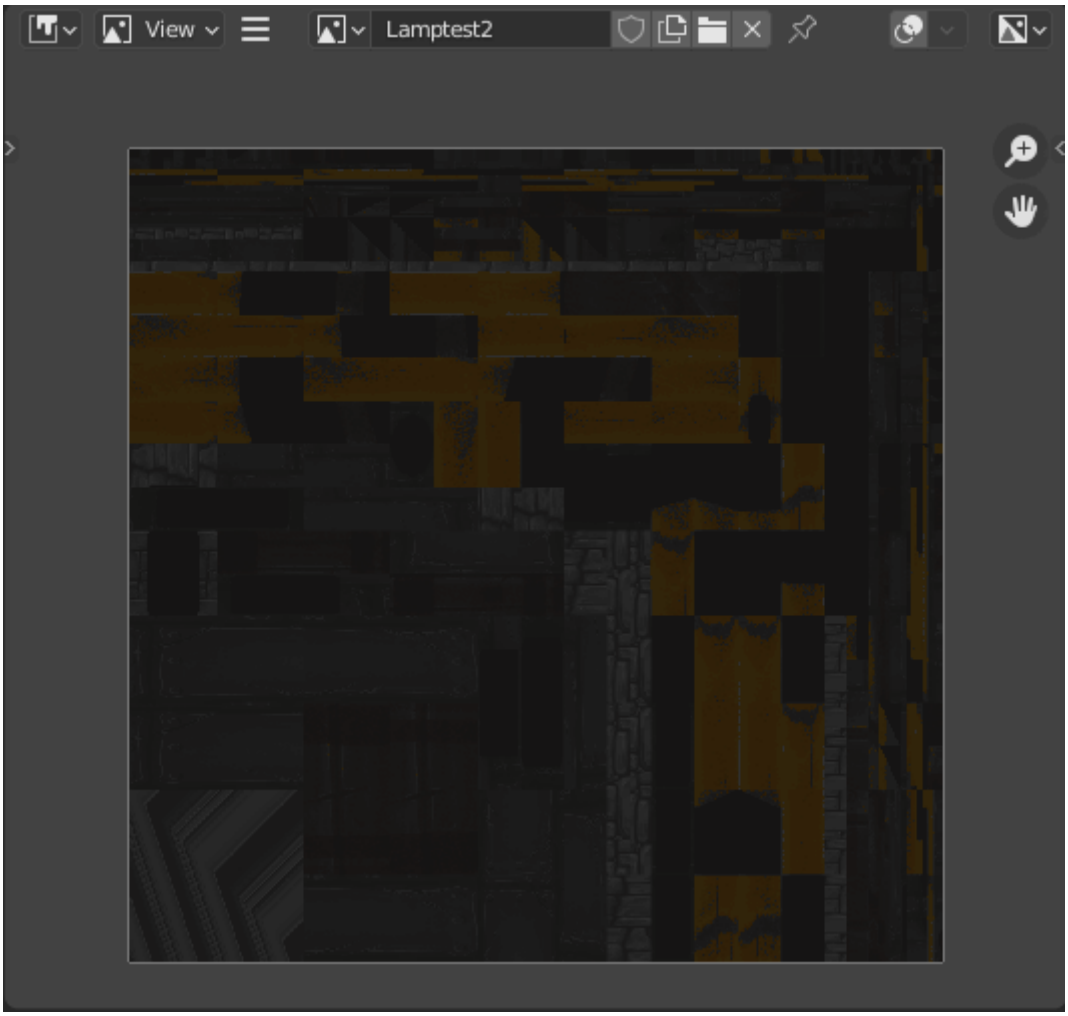


Figura 4.13: UV Map com textura

Esta Figura tem os seus níveis de brilho aumentados para poder mostrar melhor a textura, mas em troca a qualidade da cor foi diminuída.

Com o modelo todo selecionado, foi criado um novo *UV Map* e selecionou-se a opção *Smart UV Project* no menu de *UV Mapping*, o resultado observa-se na Figura 4.14.

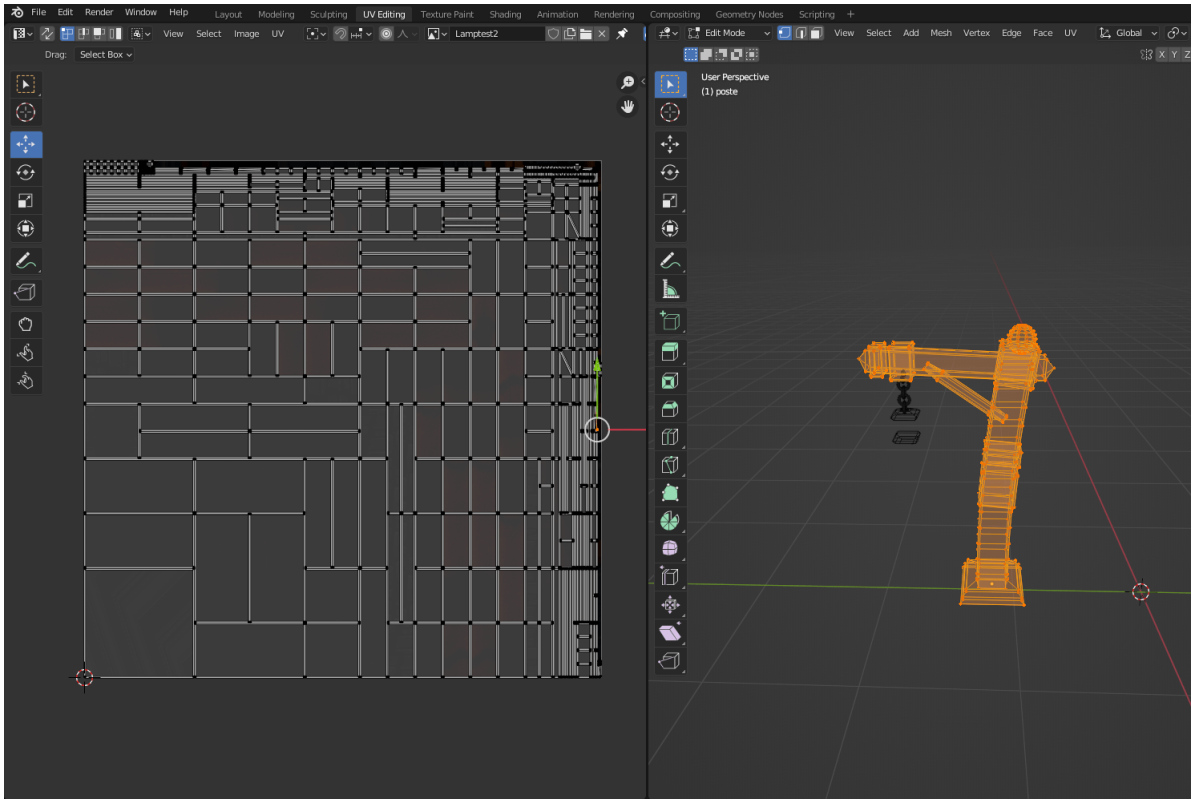


Figura 4.14: UV Map de um candeeiro

A seguir mudou-se a janela de trabalho para a janela de *Shading* e aí foi criada uma nova *Texture Image*, que vai ser a textura final para este modelo, e depois no painel de módulos foi feito o seguinte (Figura 4.15):

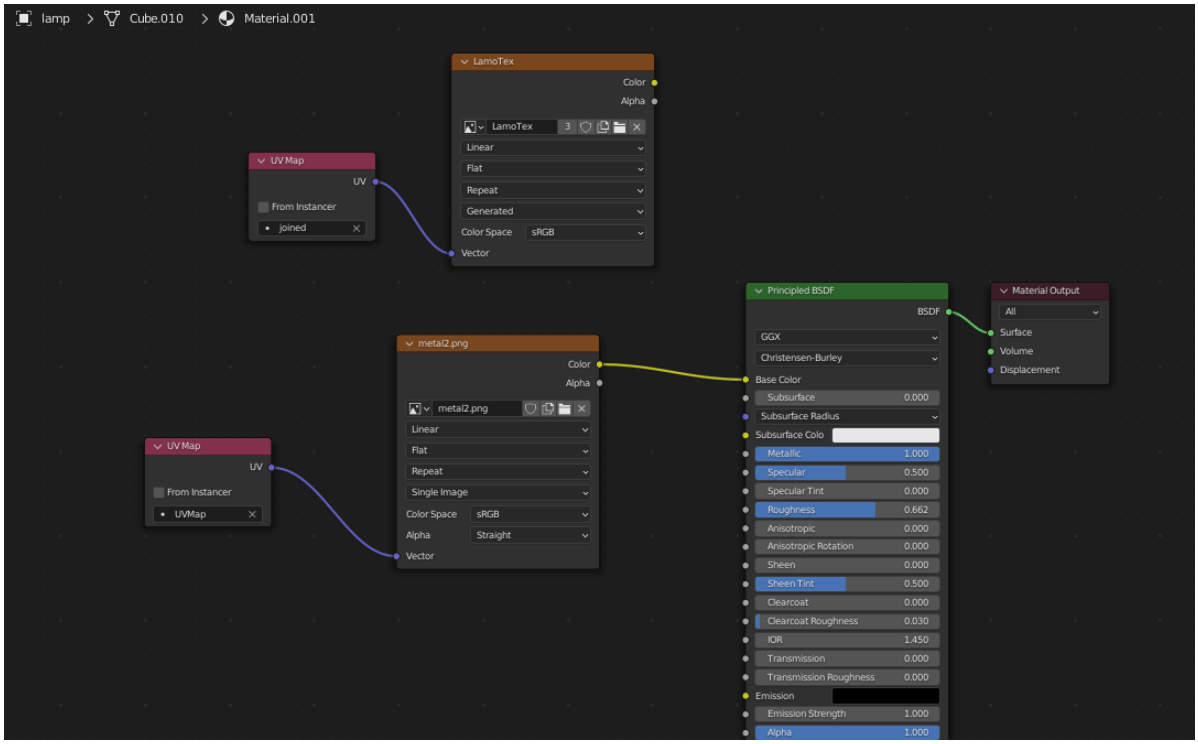


Figura 4.15: Módulos de texturas

Primeiro foi colocado *Image Texture* criada ligada ao novo *UV Map*. Isto observa-se nos módulos da Figura 4.15, e foi ligado o *UV Map* criado automaticamente às propriedades definidas do material. Isto faz com que a única textura no novo *UV Map* é a que foi criada. Este passo foi repetido em todos os materiais utilizados e depois com a nova textura selecionada em todos os materiais, foi-se ao painel de *Render Properties*, e no painel de *Bake* foi selecionado o *Bake Type Diffuse* e a única contribuição selecionada foi a cor porque não foi preciso as alterações causadas pela luz. Com isto foi selecionada a opção *"Bake"*. Depois do *Bake* estar terminado foi guardado a textura final, eliminou-se todos os materiais usados anteriormente criou-se um novo material com a textura final e o resultado final observa-se na Figura 4.16.

Figura 4.16: Candeeiro com a textura final

4.2.3 *Rigging* e Animação

Depois de ter criado os modelos e as texturas, o último trabalho feito em certos modelos foi animação, mas para isso primeiro foi necessário fazer o *rig* do modelo. *Rigging* é o processo de criar um "esqueleto" para que um modelo 3D se possa mover. Durante esta secção vai ser descrito o trabalho de *rigging* e animação no arco criado. Primeiro foi adicionado um *armature* e isto criou o primeiro "osso" do esqueleto, a partir daqui foi ajustado o osso para que esteja dentro do modelo e usou-se a ferramenta *extrude* a criar mais ossos a partir da ponto do anterior até ter um esqueleto inteiro dentro do modelo 3D, como se observa na Figura 4.17.

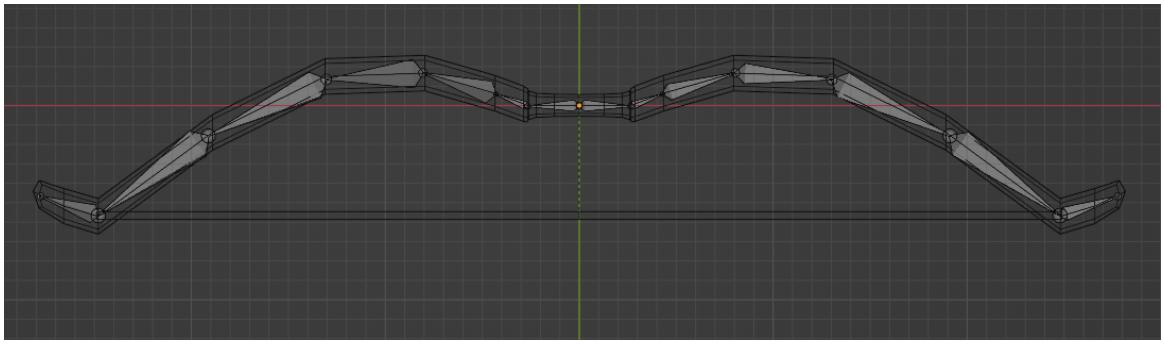


Figura 4.17: Esqueleto básico do arco

Foi adicionado um osso no centro do fio do arco, isto serve para depois manipular o fio e a estrutura do arco quando o fio for puxado na animação, mas para isso ainda faltam passos adicionais. Depois foi selecionado o esqueleto e o modelo, abriu-se o menu *"Set parent to"* e foi selecionado a opção *"Armature Deform With Automatic Weights"*. Esta função calcula a influência que cada osso terá nos vértices com base na distância destes vértices a um determinado osso [46]. Este método é o mais fácil em ter termos de configuração, mas pode causar problemas em que a *armature* não deforme o modelo como esperado, o que leva depois a ter que alterar manualmente a influência calculada anteriormente. Isto não aconteceu durante o *rigging* do arco. Adicionou-se um osso no centro do punho do arco e tornou-se esse osso como pai dos outros ossos criados para ter um osso principal para facilitar o controlo do esqueleto. A seguir selecionou-se o penúltimo osso situados em cada ponta do arco e adicionou-se um *Bone Constraint* a cada. Estes *Bone Constraints* selecionados chamam-se *Inverse Kinematics*, isto pode ser visualizado na figura 4.18.

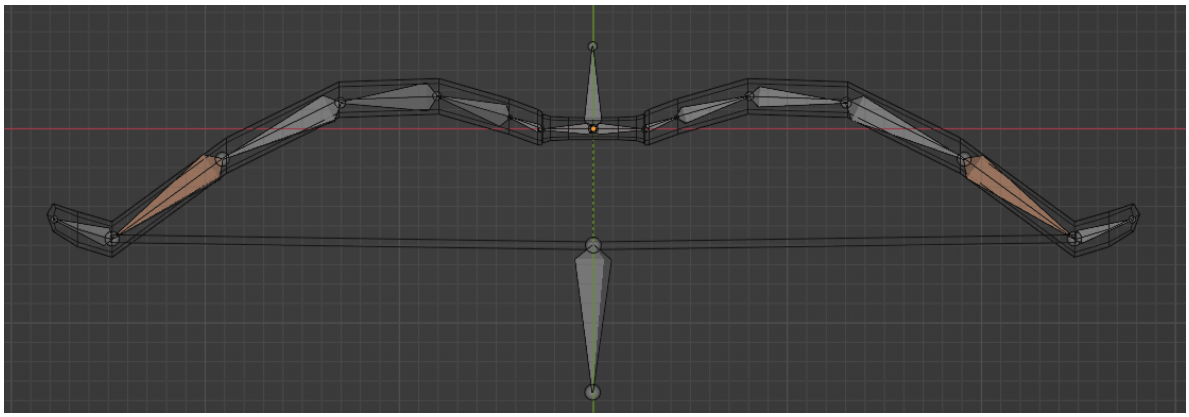


Figura 4.18: Esqueleto com *Inverse Kinematics*

Nas opções do *Esqueleto com Inverse Kinematics* alterou-se o valor de *"chain length"* para cinco. Isto dita quantos ossos é que vão ser afetados, tal como se observa na Figura 4.19.

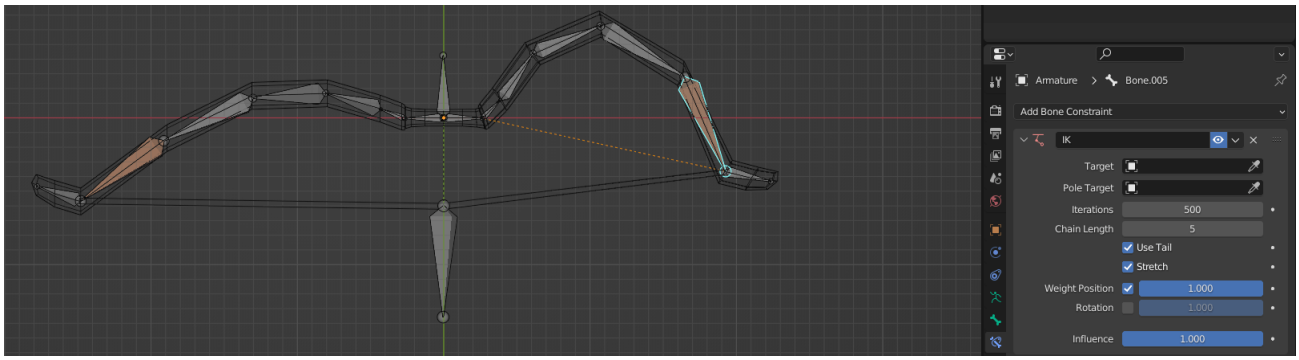


Figura 4.19: Demonstração de *Inverse Kinematics*

Depois, a partir de cada um destes ossos, usou-se o *extrude* para criar mais dois ossos. Estes não são filhos do osso principal, ao contrário do resto do esqueleto, pois foram adicionados *Bone Constraints Limit Distance*. Nestes *Bone Constraints* o *target* é o *armature* criado e o *bone* é o osso no fio do arco, como se observa na Figura 4.20.

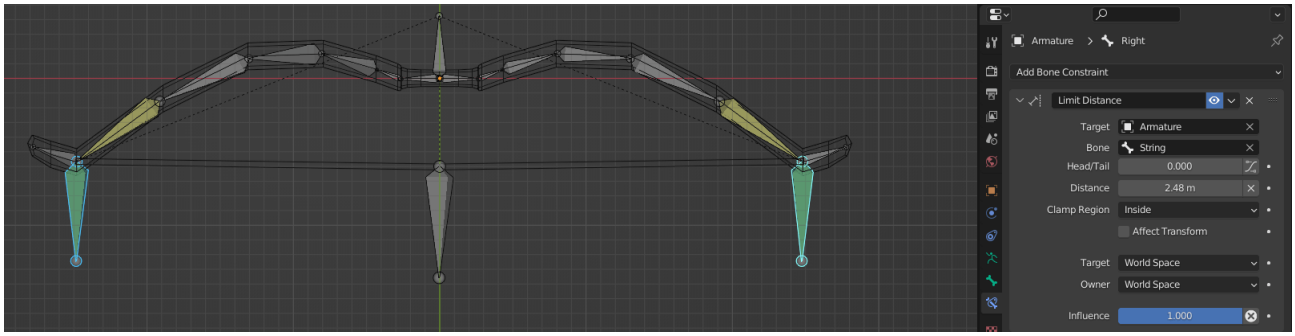


Figura 4.20: *Bone Constraints Limit Distance*

Com estes dois ossos criados, voltou-se ao ossos com *Inverse Kinematics* e alterou-se os valores do *target* para *armature* e o *bone* para o osso criado do respetivo lado. O osso esquerdo com *Inverse Kinematics* está ligado ao osso esquerdo com *Limit Distance* e o mesmo se passa com o osso direito. Com isto o *Rigging* do arco está terminado como se pode ver na Figura 4.21.

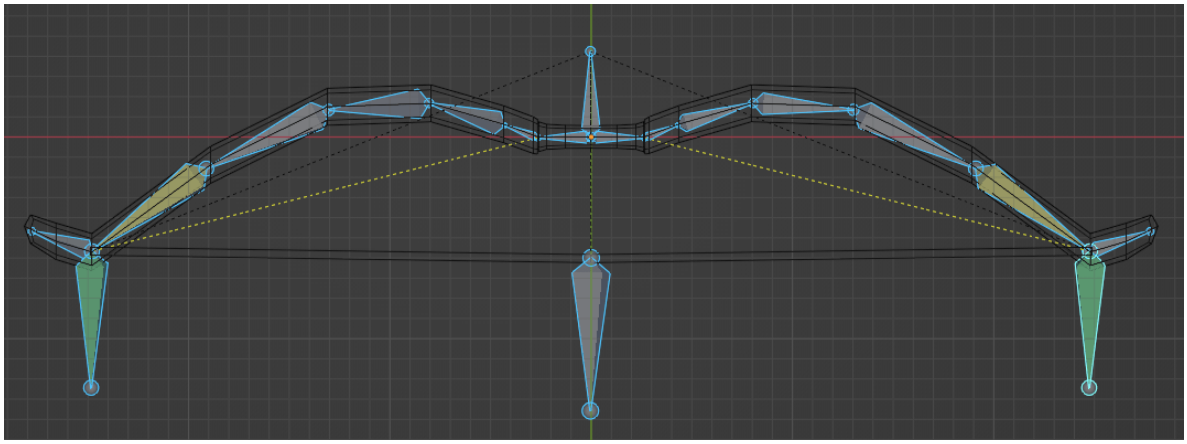


Figura 4.21: Arco com *Rigging* terminado

Com o *Rigging* terminado, passa-se à animação e para isso abriu-se a janela de animação. Nesta janela é apresentado o modelo 3D criado com o esqueleto e em baixo uma *timeline* dos *Keyframes*. Para animar, seleciona-se um osso e move-se para uma posição. Isto cria um *Keyframe*. De seguida, num ponto mais tarde na *timeline*, move-se o mesmo osso para uma posição diferente. Isto cria outro *KeyFrame*. Depois o Blender (ou outro software de animação) cria a animação entre os dois *KeyFrames*. Podemos ver este processo na Figura 4.22.

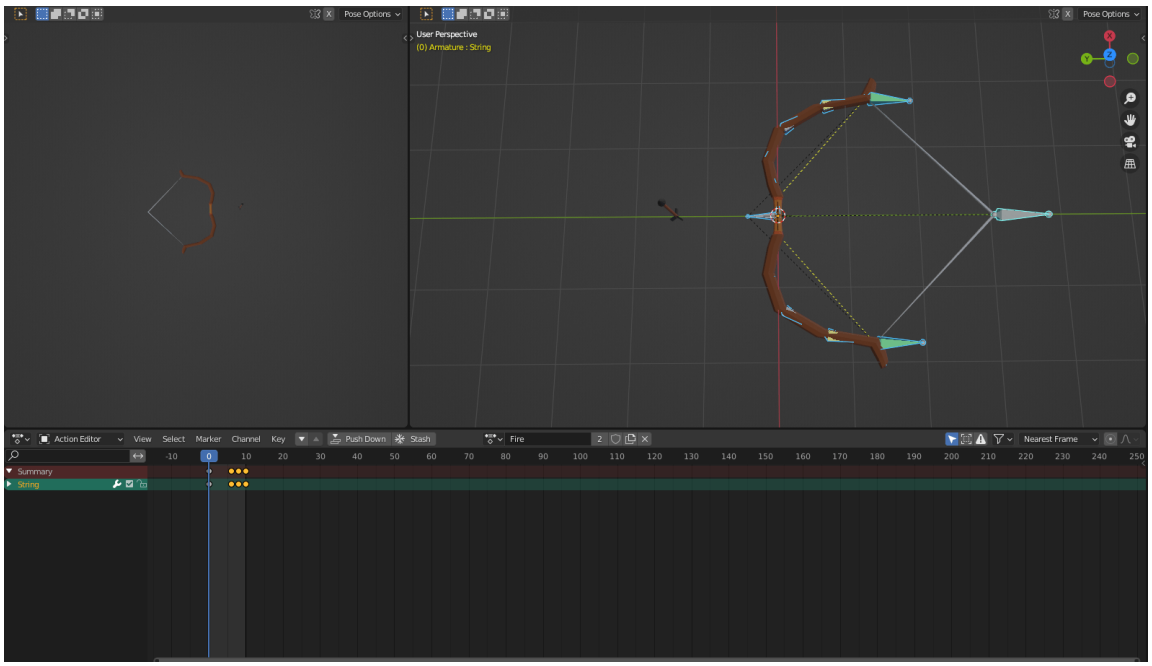


Figura 4.22: Animação do arco

4.2.4 Problemas encontrados

Um problema que foi encontrado encontra-se na exportação de modelos com texturas para Unity. Apesar do modelo ter material e textura, no Unity o modelo e o material vinham separados, no sentido em que quando o modelo era arrastado para a cena, o modelo vinha em branco sem os materiais e as texturas aplicadas. Por isso o material tinha que ser adicionado manualmente. Foram tentadas várias abordagens para resolver este problema, como exportar os modelos com o *Path mode* em *copy* e com a opção de texturas embutidas ligada. Nenhuma delas deu resultado. Por isso, e para não atrasar o desenvolvimento do projeto, foi decidido adicionar os materiais manualmente no Unity.

4.3 Implementação de som

Nesta secção vai ser apresentado o trabalho na implementação de áudio do jogo.

4.3.1 PlayerAudioManager

No desenvolvimento do áudio, começou-se por criar um "PlayerAudioManager"(J). Ele fica responsável pelos todos os sons que vêm da personagem do jogador. Primeiro criou-se um *script* chamado "Sound"(M). Este *script* contém todos elementos que constitui um som. Estes elementos podem ser adicionados ou retirados conforme o utilizador, mas no mínimo é recomendado um nome, um clip, volume e pitch. Estas quatro variáveis são os elementos básicos para o controlo desejado no PlayerAudioManager a ser desenvolvido. Na Figura 4.23 observam-se as variáveis para a criação de um som.

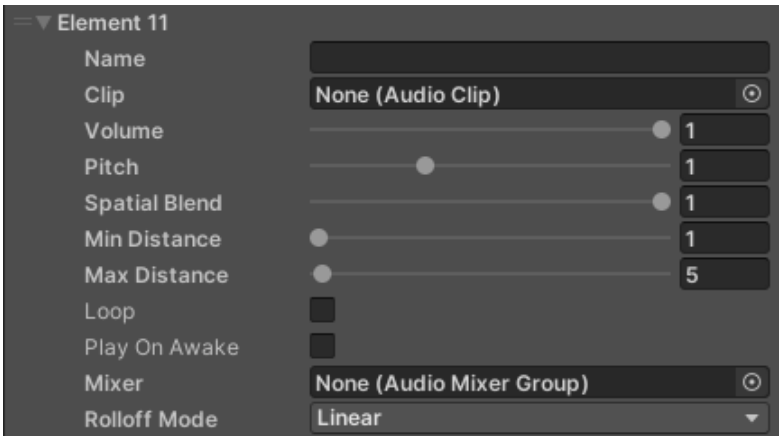


Figura 4.23: Som no PlayerAudioManager

Agora vai ser explicado qual a função de cada uma das variáveis presentes na figura acima (Figura 4.23).

- A variável "Name" serve para identificar o som, que vai ser útil para a função de busca e para gerir os sons usados dentro do PlayerAudioManager.
- A variável "Clip" é o ficheiro .mp3 que vai ser usado neste som.
- A variável "Volume" controla o volume do clip a ser utilizado, em que o valor mínimo é zero e o máximo 1.
- A variável "Pitch" controla quão alto ou baixo o som está ser ouvido, em que um será o valor normal e se elevar-mos esse valor o som parece estar mais alto do que deveria.
- A variável "Spatial Blend" define se o som é suposto ser ouvido num ambiente 2D ou 3D, em que zero é 2D e um é 3D. Como o jogo está a ser desenvolvido para um ambiente 3D o valor é um.
- As variáveis "Min Distance" e "Max Distance" controlam a distância mínima e máxima em que o AudioListener tem que estar para ouvir o som.
- A variável "Loop" define se o som está em repetição ou não.
- A variável "Play on Awake" define se o som começa ao iniciar a cena.
- A variável "Mixer" é o equivalente à opção output de um AudioSource, em que permite o utilizador associar o som a uma faixa de som no Mixer.
- A variável "Rolloff Mode" permite ao utilizador escolher o tipo de função a ser usada para dissipar o som. Existem duas opções dadas pelo Unity (Linear e Logarítmica), mas o utilizador pode personalizar como é que o som é dissipado.

Depois do "Sound" configurado criou-se o *script* para o PlayerAudioManager. Neste *script* existe um array de sons, da classe Sound. Estes sons podem ser adicionados ou removidos consoante ao que o utilizador precisa e que permite aceder às variáveis dos sons publicamente dentro do Unity. Quando a cena é iniciada, são criados AudioSources com os valores definidos pelo o utilizador no PlayerAudioManager. Este é colocado dentro da

personagem controlada pelo jogador e como este jogo é multijogador, cada personagem controlada por cada jogador também tem um PlayerAudioManager. O resultado desta configuração pode ser observado Figura 4.24.

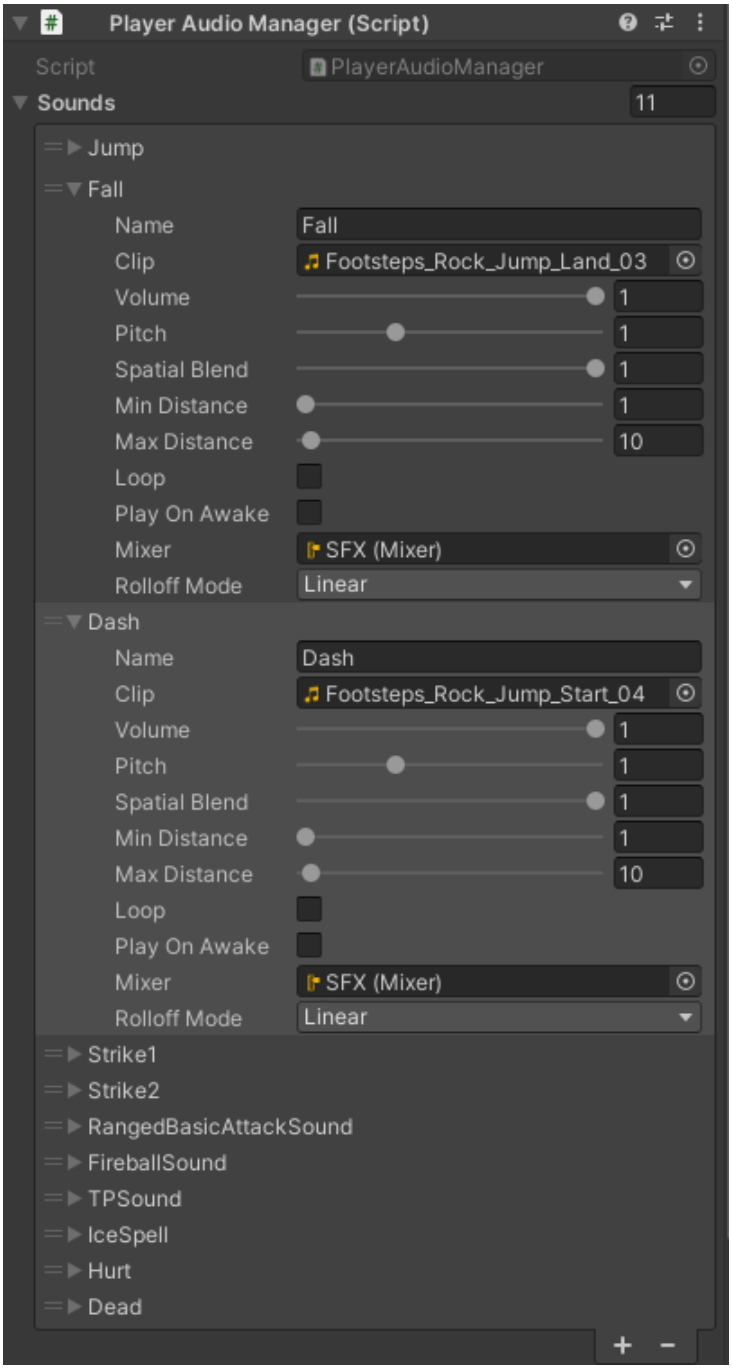


Figura 4.24: PlayerAudioManager

Muitos destes sons, como o de salto, têm que ser sincronizados com animações para fazerem efeito.

4.3.2 Sincronização de áudio em animações

Primeiro criou-se um *script* chamado "AnimationSounds"(G) onde contém uma variável pública chamada "AM", que representa o AudioManager a ser usado, e várias chamadas ao "AM" para começar o som pedido, por exemplo:
AM.gameObject.GetComponent<PlayerAudioManager>().Play("Jump").
Este *script* é colocado nos objetos que têm um AudioManager e sons que precisam de ser sincronizados com animações, como o jogador e inimigos no jogo. Para sincronizar os sons às animações, basta ir abrir a animação que se pretende usar e adicionar um evento no momento em que se espera ouvir um som. Nesse evento é colocado uma função em que, neste caso, é colocado uma das chamadas no *script* "AnimationSounds". Utilizando o exemplo anterior, seria abrir a animação de salto, adicionar um evento quando a personagem começa a levantar o pés do chão e nesse evento selecionar a função AM.gameObject.GetComponent<PlayerAudioManager>().Play("Jump"), como se pode observar abaixo na Figura 4.25.

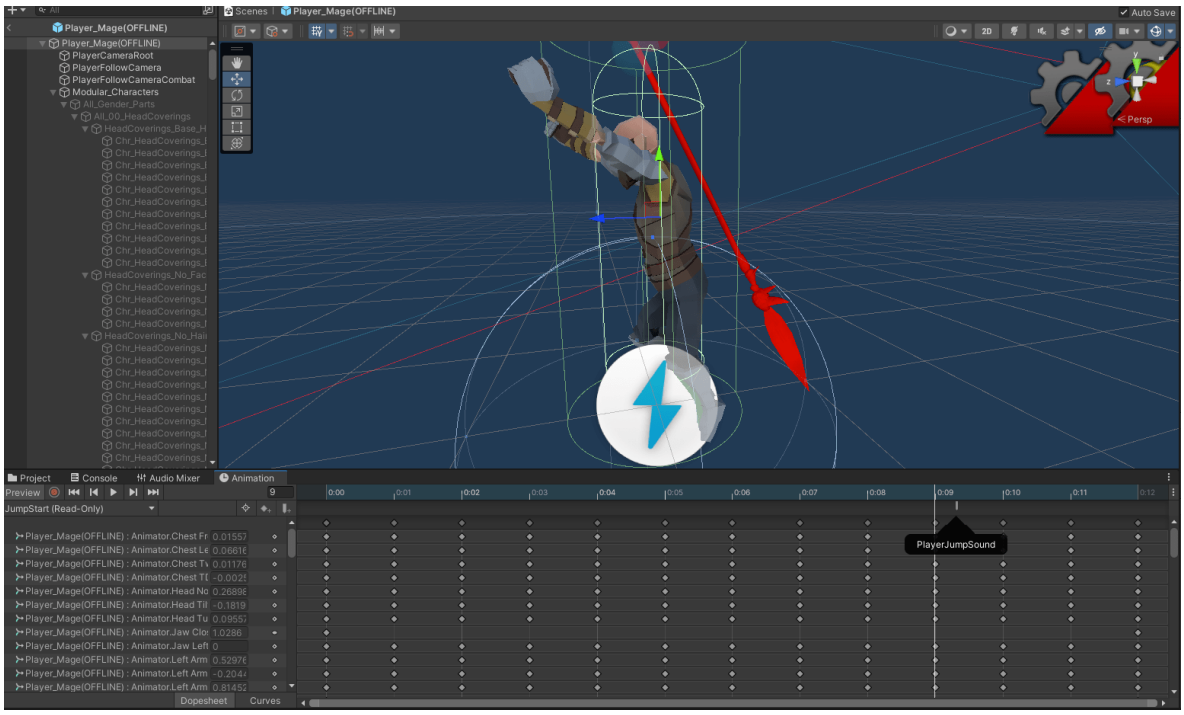


Figura 4.25: Evento na animação de salto

4.3.3 FootSteps

Um dos objetivos propostos foi implementar um sistema que substitui os sons usados nos passos do jogador, dependendo do material interagido. Para isso foram criados dois *scripts*: "FootStep"(I) e "TerrainDetector"(N). O *script* "TerrainDetector" é uma classe que analisa os materiais usados para fazer o terreno no nível e guarda o seu index. Depois no *script* "FootStep", são criados tantos *arrays* de clips de som quanto o número de materiais usados. Cada um destes *arrays* tem um função *Random*. Assim o mesmo som não é repetido em cada passo. Depois criou-se uma função chamada "Step" que associa o array a ser usado ao index do material equivalente, por exemplo usar array com sons de terra no material terra. O *script* "FootStep" é depois colocado nas personagens dos jogadores e nos NPCs que circulam pelo nível. O resultado observa-se abaixo na Figura 4.26.

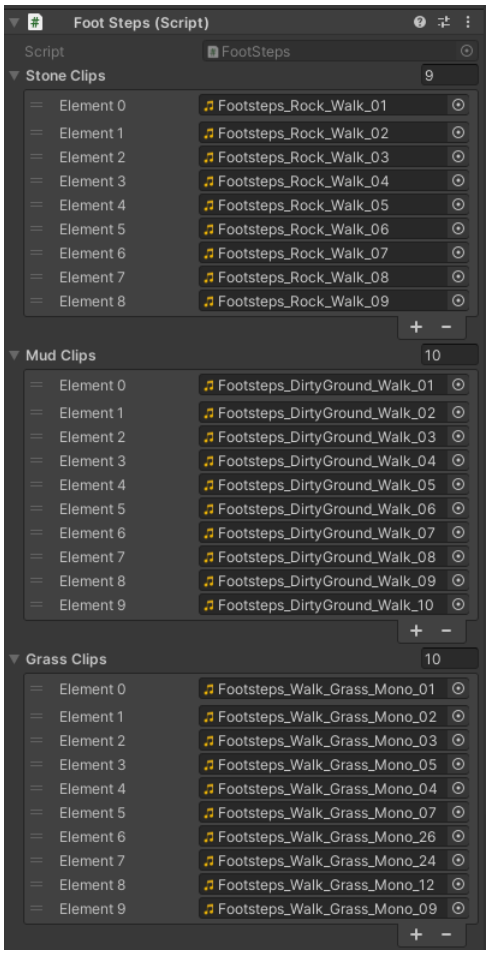


Figura 4.26: FootSteps

Para terminar abriram-se as animações usadas para movimento, como a de andar e correr, colocaram-se eventos onde os pés das personagens tocam no terreno do nível e nesses eventos adicionaram-se a função "Step".

4.3.4 LevelAudioManager

O "LevelAudioManager"(4.27) é um AudioManager que é responsável pelos sons que estão a ser usados no ambiente da cena, como pessoas a falar e música. Observa-se um exemplo disto na Figura 4.27.

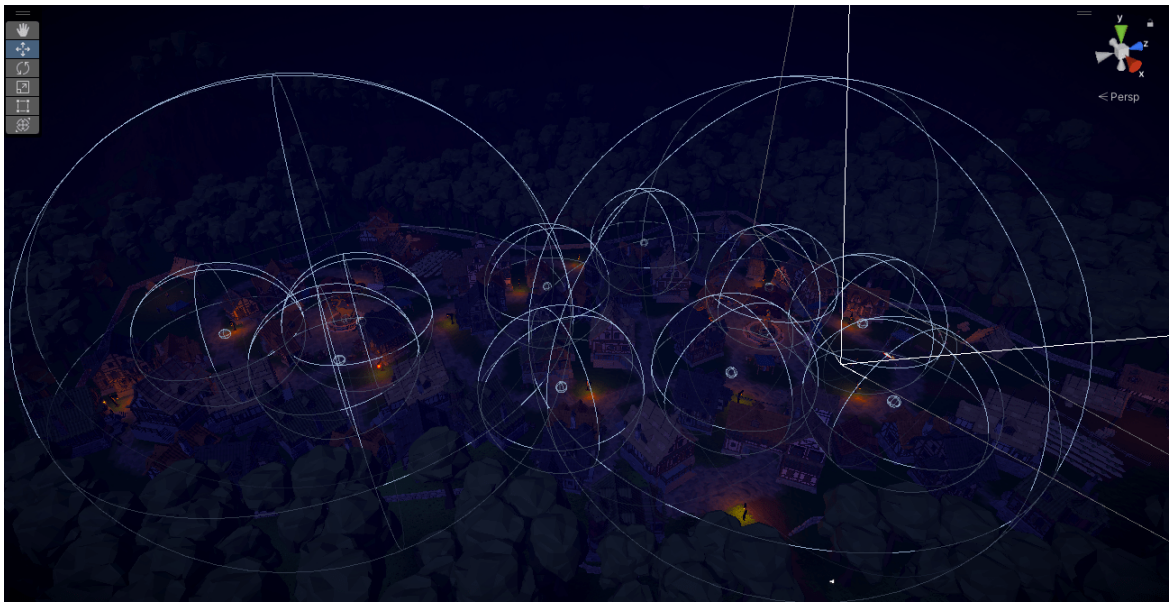


Figura 4.27: Sons presentes num nível do jogo

Em certas partes do jogo, a música do nível pode transitar para outra música tal como quando os jogadores entram em combate com os inimigos do nível ou quando o jogador passa de uma floresta para uma caverna. Para isso, foi criado um *script* chamado "AudioSwap"(H) que faz com que quando o jogador entra nestas zonas especificas, o volume da música ambiente é reduzido a zero e ao mesmo tempo o volume da outra música escolhida passa a um.

4.3.5 Inimigos e NPCs

Para adicionar sons nos inimigos do jogo, foi reutilizado o código do *script* PlayerAudioManager e fez-se um EnemyAudioManager, em que este tem as mesmas funções

que o PlayerAudioManager para guardar sons e reproduzi-los quando necessário. Depois foram adicionados aos inimigos os *scripts* AnimationSounds, para os sons sincronizados em animações, e FootStep para os sons de movimento. Para os sons quando os inimigos levam dano do jogador, foi criado um *script* chamado "PropSync"(L) que faz com que quando o ataque do jogador colide com o *collider* do inimigo, é reproduzido um som de dano. Depois foi reutilizado para os NPCs da cidade que estão a fazer atividades, como por exemplo um ferreiro a martelar ferro ou um lenhador a cortar árvores.

4.3.6 Espacialidade do Som

A espacialidade de som 3D foi obtida a partir de um conjunto de configurações feitas no Unity. Primeiro, cada som (*sound*) que não fosse música, teve que ter a opção de *spatial blend* posta a um. Assim o *AudioSource* criado tem o seu *spatial blend* na opção 3D (este ponto sobre a variável *spatial blend* foi brevemente falado no subcapítulo 4.3.1 PlayerAudioManager). Com esta configuração o som é 3D, mas é também preciso configurar a dissipação do som à medida que o jogador se afasta da fonte do som.

Como já foi referido no subcapítulo 4.3.1 (PlayerAudioManager) existe uma variável chamada de *Rolloff Mode*, em que permite ao utilizador escolher o tipo de função a ser usada na dissipação do som, em que existem duas opções disponíveis pelo o Unity, Linear e Logarítmica, com uma terceira opção sendo o utilizador poder personalizar a curva de dissipação.

Entre as funções disponíveis foi escolhida a função Linear, porque durante os testes feitos, encontrou-se que com a função Logarítmica selecionada o som nunca se dissipava por completo, mesmo depois de o jogador estar num canto completamente diferente do mapa de onde se encontrava a fonte do som, e este com a distância do configurada para o seu mínimo.

Com a função Linear selecionada, foi configurada a distância mínima e máxima, em que a mínima é com perto é que o jogador tem que estar da fonte do som para poder ouvir-lo perfeitamente e a máxima é o com longe é que o jogador pode estar para ainda conseguir ouvir o som. Fora desta área configurada o jogador deixa de ouvir completamente.

4.3.7 Mixers

O Unity tem uma ferramenta chamado de Mixer que permite criar várias faixas de som que depois facilita o controlo de volume no projeto sem ter que alterar os valores dos AudioSources individualmente. Quando se cria um mixer é automaticamente criada uma faixa de som chamada "Master". Esta faixa vai servir como um pai para as faixas depois criadas. Foram criadas duas faixas, "Music" e "SFX", onde a primeira vai controlar a música ambiente e a segunda vai controlar os efeitos sonoros presentes na cena, como por exemplo sons de ataque, pessoas a falar, o jogador a andar, etc. Para associar os vários AudioSources no projeto às respetivas faixas, em cada AudioSource existe uma opção chamada output, onde se escolhe a faixa correta a esse AudioSource. Depois de associadas, todos os AudioSources com a faixa "Music" no output, o seu volume será controlado por essa faixa e os AudioSources com a faixa "SFX" serão controlados por essa faixa. Se for necessário alterar o volume dos dois grupos em simultâneo, basta alterar os valores da faixa "Master". Com o Mixer configurado, podemos associar o controlo de cada faixa criada ao UI e assim o jogador pode controlar livremente os níveis de som no jogo.

Capítulo 5

Testes, resultados e discussão

Foram feitos testes na parte da modelação e na implementação do som.

O teste feito na parte da modelação foi a comparação entre um modelo *LowPoly* e um modelo com mais detalhe e observar a diferença na performance do jogo. Os modelos trocados em cena, foram as portas de prisão apresentadas nas Figuras 5.1 e 5.2. Os valores apresentados nas Figuras 5.1 e 5.2 foram recolhidos através da janela de estatísticas, que foi ligada enquanto o jogo está a ser corrido em tempo real.

Figura 5.1: Modelo *LowPoly*

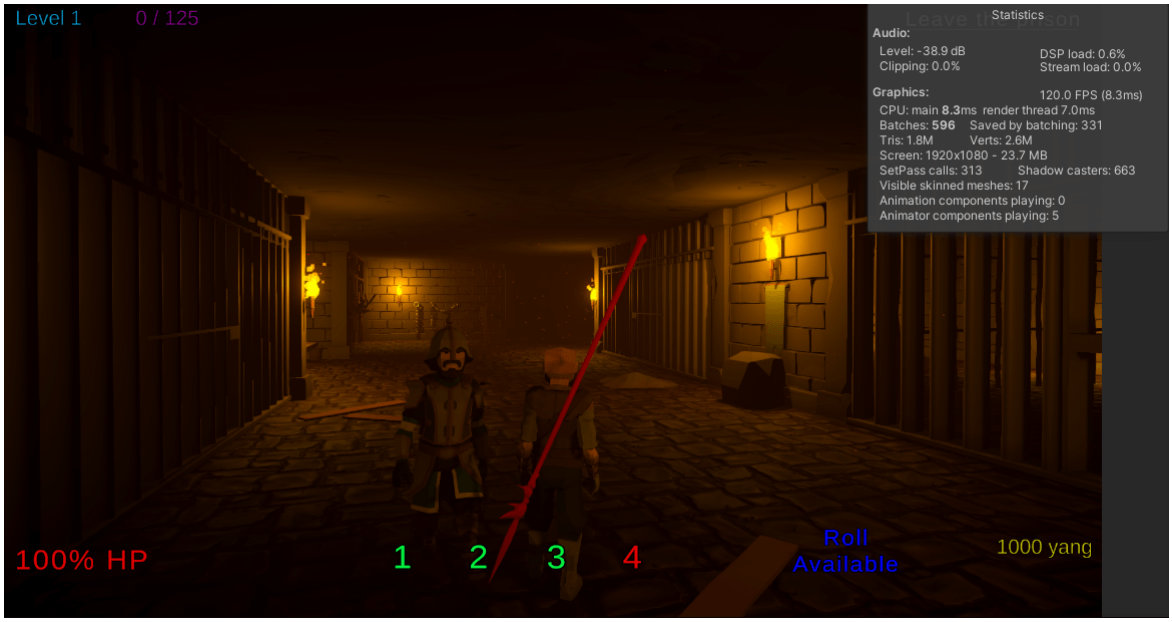


Figura 5.2: Modelo mais detalhado

Ao fazer a comparação das Figuras 5.1 e 5.2, pode-se observar que com os modelos *LowPoly* o utilizador tem mais *frames per second* (127.2 *frames per second*) o que torna a experiência do jogo melhor. Como já era esperado, com os modelos mais detalhados o jogo está a processar mais triângulos e vértices, quase o dobro, o que resulta em menos *frames per second* (120.0 *frames per second*) e uma pior experiência para o jogador.

O teste feito na implementação do som foi entre os níveis individuais de cada som e o *mixer*, porque como cada som já tem um volume diferente de quando foi criado, existiam sons com o volume mais alto que outro, apesar dos dois estarem ao mesmo nível de volume no Unity. Isto resultava em alguns sons deixarem de ser ouvidos antes de outros, ao reduzir o nível de volume do mixer que os controlava. Para resolver esta situação foram ajustados os níveis de volume de cada som introduzido para que o volume do jogo em geral ficasse normalizado.

Foi entregue uma versão de teste do videojogo a um grupo constituído de 12 pessoas que estão acostumados a jogar videojogos do mesmo género que o videojogo que está a ser desenvolvido. Depois dos membros deste grupo experimentarem foi entregue um questionário com perguntas sobre cada área desenvolvida do videojogo. Aqui vão ser apresentadas as perguntas e as respostas obtidas na área da implementação de som (Figuras 5.3, 5.4, 5.5, 5.6 e 5.7).

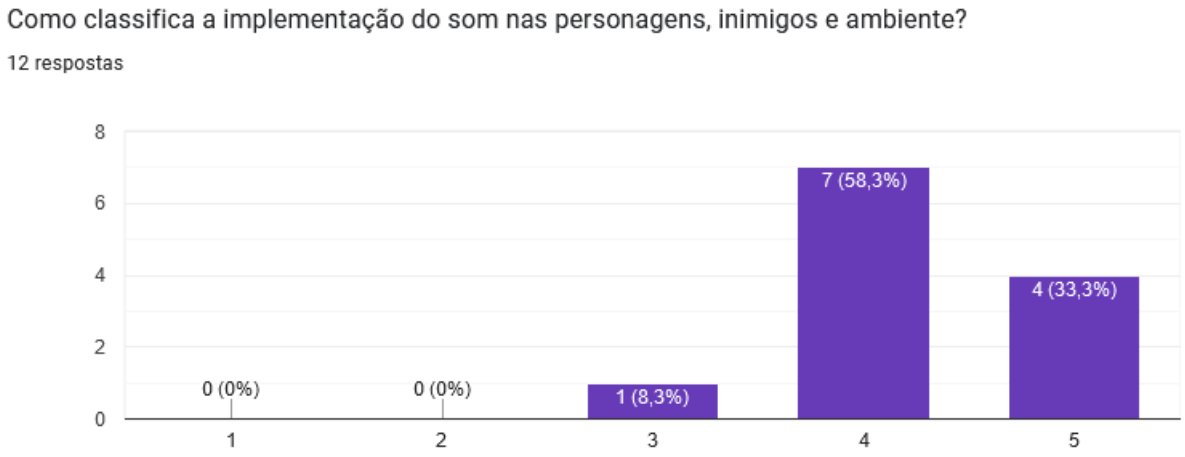


Figura 5.3: Pergunta número 1

Dos resultados na Figura 5.3, observa-se que das 12 respostas obtidas, numa escala de um a cinco, uma pessoa (8,3%) classificou a implementação do som no videojogo medíocre (valor 3), sete pessoas (58,3%) classificaram a implementação do som boa (valor 4) e quatro pessoas (33,3%) classificaram a implementação do som muito boa (valor 5).

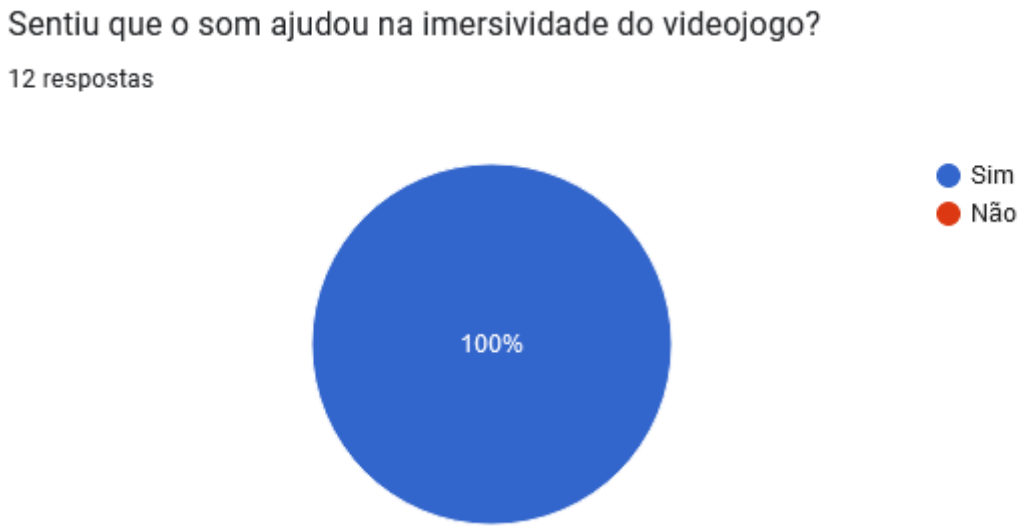


Figura 5.4: Pergunta número 2

Dos resultados obtidos na Figura 5.4, observa-se que todos os 12 membros da amostra responderam que sentiram que o som ajudou na imersividade do videojogo.

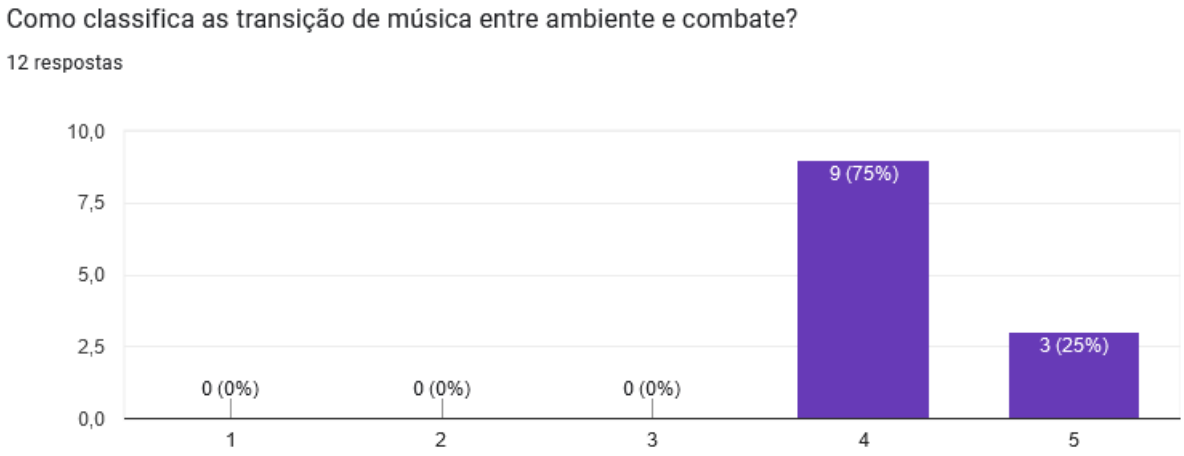


Figura 5.5: Pergunta número 3

Dos resultados na Figura 5.5, observa-se que das 12 respostas obtidas, numa escala de um a cinco, nove pessoas (75%) classificaram a transição de música entre ambiente e combate boa (valor 4) e três pessoas (25%) classificaram a transição de música muito boa (valor 5).

Figura 5.6: Pergunta número 4

Dos resultados obtidos na Figura 5.6, observa-se que 10 pessoas (83,3%) não entraram nenhum problema com a implementação do som e duas pessoas (16,7%) encontram pelo menos um problema na implementação do som.

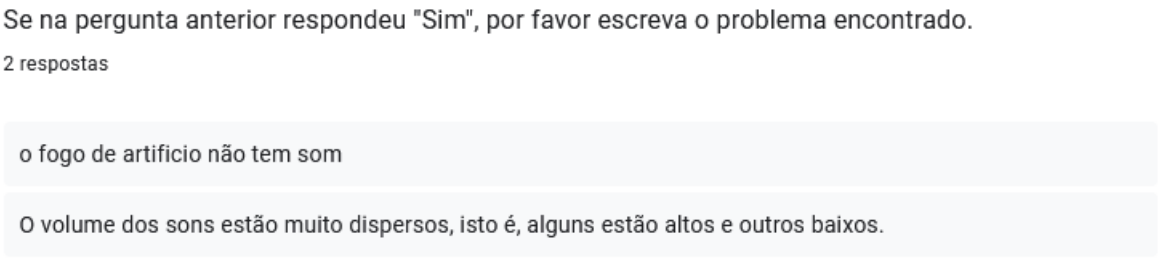


Figura 5.7: Pergunta número 5

Em seguimento com a pergunta anterior (Figura 5.6) a pergunta na Figura 5.7 pergunta às pessoas que responderam afirmativo na pergunta anterior, qual o problema encontrado. As respostas obtidas foram que o fogo de artificio encontrado na cidade não tinha som e que o volume dos sons estão dispersos.

No fim do questionário foram colocadas duas perguntas sobre o videojogo em geral. Estas duas perguntas e as suas respostas, podem ser observadas nas Figuras 5.8 e 5.9.

Figura 5.8: Pergunta geral número 1

Dos resultados na Figura 5.8, observa-se que das 12 respostas obtidas, numa escala de um a cinco, uma pessoa (8,3%) classificou a sua experiência geral do videojogo medíocre (valor 3), 10 pessoas (83,3%) classificaram a sua experiência geral do videojogo boa (valor 4) e uma pessoa (8,3%) classificou a a sua experiência geral do videojogo muito boa (valor 5).

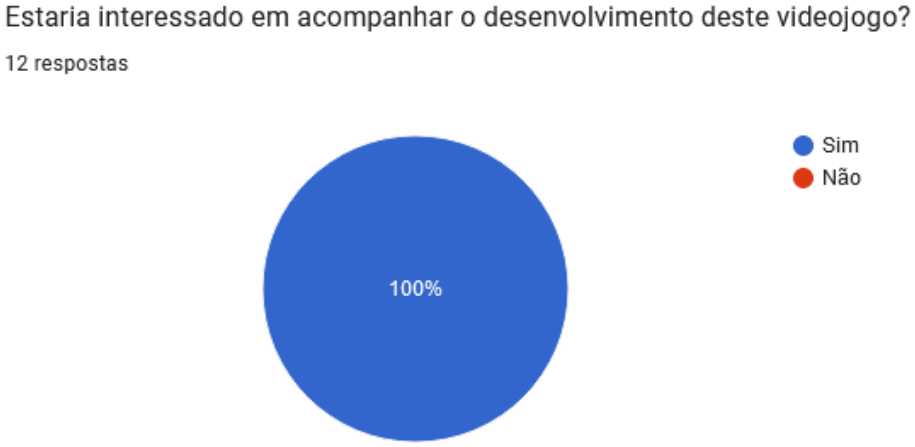


Figura 5.9: Pergunta geral número 2

Na última pergunta, que se encontra na Figura 5.9, todas as pessoas do grupo de amostra responderam que estariam interessados em acompanhar o desenvolvimento deste jogo.

A partir dos resultados obtidos das respostas do questionário, foram tiradas algumas sugestões para a melhoria do videojogo, como colocar som no fogo de artificio e normalizar o volume dos sons. A normalização do som foi feita, como foi referido anteriormente no documento, na parte de testes feitos na implementação de som.

Capítulo 6

Conclusão

Com o trabalho realizado, pode-se dizer que os objetivos propostos foram cumpridos. No planeamento do projeto tinha-se uma ideia que dava contexto ao videojogo e durante o seu desenvolvimento essa ideia foi expandida para a história que está presente nesta dissertação e durante a jornada do jogador.

Apesar de não existir experiência na criação de recursos 3D, animações e na ferramenta Blender, foram conseguidos vários modelos que se encontram nos níveis do videojogo e na mão do jogador. Estes modelos com materiais e texturas criadas usando processos e ferramentas encontradas no Blender, foram aprendidas durante o planeamento e o desenvolvimento, e concluindo-se dizer que a criação dos modelos foi bem sucedida, visto que não sacrificam a performance do jogo.

A implementação do som feita em Untity foi o maior foco deste trabalho, porque os sistemas criados ao longo do desenvolvimento do videojogo, tinham que ser compatíveis como as outras partes do projeto, em especifico a parte de rede e mecânicas, visto que os sons tinham que ser sincronizados nas animações das personagens e as suas ações.

Graças à recolha de dados obtida através do questionário feito, conseguiu-se ter uma ideia dos problemas presentes no projeto e no que melhorar numa expansão futura.

Para o futuro, ainda é necessário mais prática nas áreas trabalhadas, como em *shading* e animação, mas com o que foi já referido e para os objetivos propostos, pode-se dizer que este projeto foi um sucesso.

6.1 Trabalho Futuro

Com a melhoria e expansão do videojogo, planeia-se para a área de implementação de som, novos sistemas, como diferentes sons para cada arma quando esta bate em superfícies, como paredes e inimigos. Isto pode ser interpretado como uma variação do sistema usado na interação do movimento da personagem do jogador com o material do terreno. Com a história já planeada e escrita, podem ser implementadas vozes, como um narrador e para personagens que podem transmitir melhor a história do videojogo para o jogador. Para a criação de recursos 3D, criar modelos com mais detalhe, mas ainda manter a arte do jogo e sem sacrificar a performance do videojogo. Desenvolver mais a prática com materiais, texturas e animação.

Referências

[1] Nicolas Esposito. "A short and simple definition of what a videogame is". Em: *Proceedings of DiGRA 2005 Conference: Changing Views - Worlds in Play* (2005).

[2] TENNIS. *Tennis for Two – Wikipédia, a enciclopédia livre.* URL: `https://pt.wikipedia.org/wiki/Tennis_for_Two` (acedido em 14/01/2022).

[3] PONG. *Pong – Wikipédia, a enciclopédia livre.* URL: `https://pt.wikipedia.org/wiki/Pong` (acedido em 14/01/2022).

[4] DOOM. *DOOM no Steam.* URL: `https://store.steampowered.com/app/379720/DOOM/` (acedido em 14/01/2022).

[5] Alexander Rechsteiner. *The history of video games.* 2020. URL: `https://blog.nationalmuseum.ch/en/2020/01/the-history-of-video-games/` (acedido em 02/06/2021).

[6] SPACEINV. *Space Invaders – Wikipédia, a enciclopédia livre.* URL: `https://pt.wikipedia.org/wiki/Space_Invaders` (acedido em 14/01/2022).

[7] PACMAN. *Pac-Man – Wikipédia, a enciclopédia livre.* URL: `https://pt.wikipedia.org/wiki/Pac-Man` (acedido em 14/01/2022).

[8] ULTIMA. *Ultima I: The First Age of Darkness – Wikipédia, a enciclopédia livre.* URL: `https://pt.wikipedia.org/wiki/Ultima_I:_The_First_Age_of_Darkness` (acedido em 14/01/2022).

[9] SMB. *Super Mario Bros. — NES — Jogos — Nintendo.* URL: `https://www.nintendo.pt/Jogos/NES/Super-Mario-Bros--803853.html` (acedido em 14/01/2022).

[10] TETRIS. *Tetris – Wikipédia, a enciclopédia livre.* URL: `https://pt.wikipedia.org/wiki/Tetris` (acedido em 14/01/2022).

[11] SIMCITY. *SimCity™*. URL: https://www.ea.com/games/simcity/simcity (acedido em 14/01/2022).

[12] CS. *Counter-Strike no Steam*. URL: https://store.steampowered.com/app/10/CounterStrike/ (acedido em 14/01/2022).

[13] WOW. *World of Warcraft*. URL: https://worldofwarcraft.com/pt-br/ (acedido em 14/01/2022).

[14] DEATHS. *DEATH STRANDING no Steam*. URL: https://store.steampowered.com/app/1190460/DEATH_STRANDING/ (acedido em 14/01/2022).

[15] MasterClass. *How to Make a Video Game: 6 Steps to Develop Your Game - 2021 - MasterClass*. 2021. URL: https://www.masterclass.com/articles/how-to-make-a-video-game#how-to-develop-a-video-game (acedido em 02/06/2021).

[16] A. Andrade. "Game engines: a survey". Em: *EAI Endorsed Transactions on Game-Based Learning* 2.6 (2015), p. 150615. DOI: 10.4108/eai.5-11-2015.150615.

[17] Game Dev Unlocked. *How Making Indie Games Changed My Life*. URL: https://www.youtube.com/watch?v=Y3Rs1z7it5M (acedido em 05/09/2022).

[18] Unity. *Multiplatform — Unity*. URL: https://unity.com/features/multiplatform (acedido em 02/06/2021).

[19] Unreal Engine. *Frequently Asked Questions - Unreal Engine - Unreal Engine*. URL: https://www.unrealengine.com/en-US/faq?sessionInvalidated=true (acedido em 02/06/2021).

[20] Godot. *Introduction — Godot Engine (stable) documentation in English*. URL: https://docs.godotengine.org/en/stable/about/introduction.html (acedido em 13/01/2022).

[21] Godot$_{Q}$*uestions*. *Frequently asked questions — Godot Engine (stable) documentation in English*. URL: https://docs.godotengine.org/en/stable/about/faq.html (acedido em 13/01/2022).

[22] FAR. *Far Cry® no Steam*. URL: https://store.steampowered.com/app/13520/Far_Cry/ (acedido em 14/01/2022).

[23] CRYSIS. *Crysis Remastered no Steam*. URL: https://store.steampowered.com/app/1715130/Crysis_Remastered/ (acedido em 14/01/2022).

[24] Marie Dealessandri. *What is the best game engine: is CryEngine right for you? — GamesIndustry.biz*. URL: https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-cryengine-the-right-game-engine-for-you (acedido em 13/01/2022).

[25] Marcus Toftedahl. *Which are the most commonly used Game Engines?* URL: https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines- (acedido em 13/01/2022).

[26] DS. *DARK SOULS™: REMASTERED no Steam*. URL: https://store.steampowered.com/app/570940/DARK_SOULS_REMASTERED/ (acedido em 14/01/2022).

[27] HR. *Heavy Rain no Steam*. URL: https://store.steampowered.com/app/960910/Heavy_Rain/ (acedido em 14/01/2022).

[28] DISCO. *Disco Elysium - The Final Cut no Steam*. URL: https://store.steampowered.com/app/632470/Disco_Elysium__The_Final_Cut/ (acedido em 14/01/2022).

[29] Anthony Jondreau. *The Unique Storytelling Power of Video Games — by Anthony Jondreau — Medium*. 2020. URL: https://anthonyjondreau.medium.com/the-unique-storytelling-power-of-video-games-e68f36f74145 (acedido em 02/06/2021).

[30] KCD. *Kingdom Come: Deliverance no Steam*. URL: https://store.steampowered.com/app/379430/Kingdom_Come_Deliverance/ (acedido em 14/01/2022).

[31] Creative Bloq. *The best 3D modelling software in 2021 — Creative Bloq*. URL: https://www.creativebloq.com/features/best-3d-modelling-software (acedido em 03/06/2021).

[32] Blender. *About — blender.org*. URL: https://www.blender.org/about/ (acedido em 02/06/2021).

[33] IMDB$_L$R. *O Senhor dos Anéis - As Duas Torres (2002) - IMDb*. URL: https://www.imdb.com/title/tt0167261/ (acedido em 15/01/2022).

[34] IMDB$_SW$. *Star Wars: Episódio II - O Ataque dos Clones (2002) - IMDb*. URL: https://www.imdb.com/title/tt0121765/?ref_=nv_sr_srsg_0 (acedido em 15/01/2022).

[35] Warframe. *Warframe no Steam*. URL: https://store.steampowered.com/app/230410/Warframe/ (acedido em 15/01/2022).

[36] Hades. *Hades no Steam*. URL: https://store.steampowered.com/app/1145360/Hades/ (acedido em 15/01/2022).

[37] AutoDesk$_F$T$_Q$A. *Solved: Limitations of the Trial Version - Autodesk Community - Autodesk Media and Entertainment*. URL: https://forums.autodesk.com/t5/flame-training-edition/limitations-of-the-trial-version/td-p/6794944 (acedido em 15/01/2022).

[38] Autodesk Maya. *Maya Software — Get Prices and Buy Maya 2022 — Autodesk*. URL: https://www.autodesk.com/products/maya/overview?term=1-YEAR&tab=subscription (acedido em 15/01/2022).

[39] Autodesk 3ds Max. *3ds Max Software — Get Prices Buy Official 3ds Max 2022*. URL: https://www.autodesk.com/products/3ds-max/overview?term=1-YEAR&tab=subscription (acedido em 15/01/2022).

[40] For Honor. *FOR HONOR™ no Steam*. URL: https://store.steampowered.com/app/304390/FOR_HONOR/ (acedido em 15/01/2022).

[41] WD2. *Watch$_D$ogs2noSteam*. URL: https://store.steampowered.com/app/447040/Watch_Dogs_2/ (acedido em 15/01/2022).

[42] Dead by Daylight. *Dead by Daylight no Steam*. URL: https://store.steampowered.com/app/381210/Dead_by_Daylight/ (acedido em 15/01/2022).

[43] EDUCBA. *Maya vs 3Ds Max vs Blender — Which 3D Software is better?* URL: https://www.educba.com/maya-vs-3ds-max-vs-blender/ (acedido em 15/01/2022).

[44] PER. *Perception no Steam*. URL: https://store.steampowered.com/app/426310/Perception/ (acedido em 14/01/2022).

[45] JORN. *Journey no Steam*. URL: https://store.steampowered.com/app/638230/Journey/ (acedido em 14/01/2022).

[46] Blender. *Armature Deform Parent*. URL: https://docs.blender.org/manual/en/latest/animation/armatures/skinning/parenting.html (acedido em 24/08/2022).

Apêndices

Apêndice A

CharRig

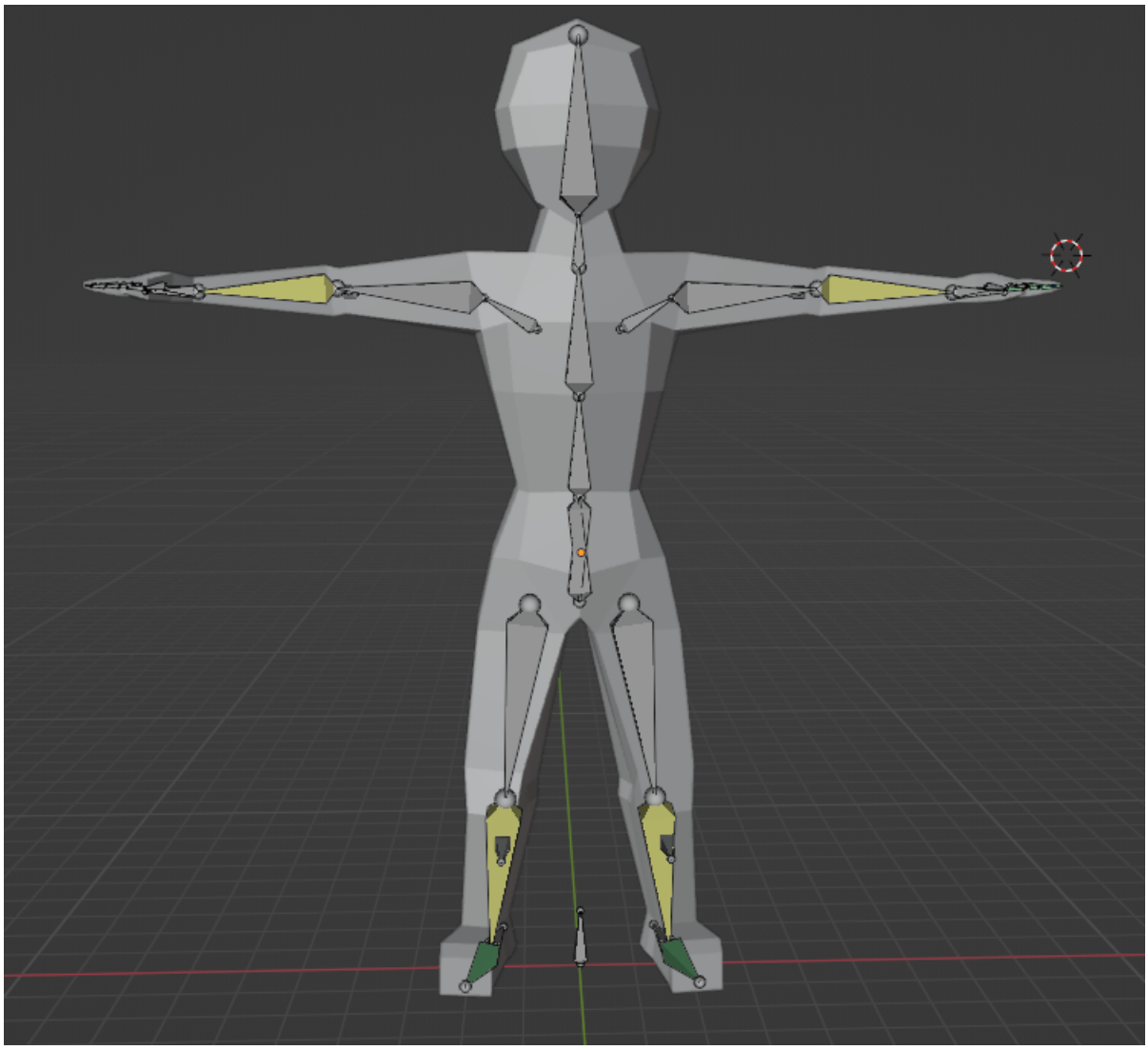


Figura A.1: Personagem com Rig

Apêndice B

GreatSwords sem Shading

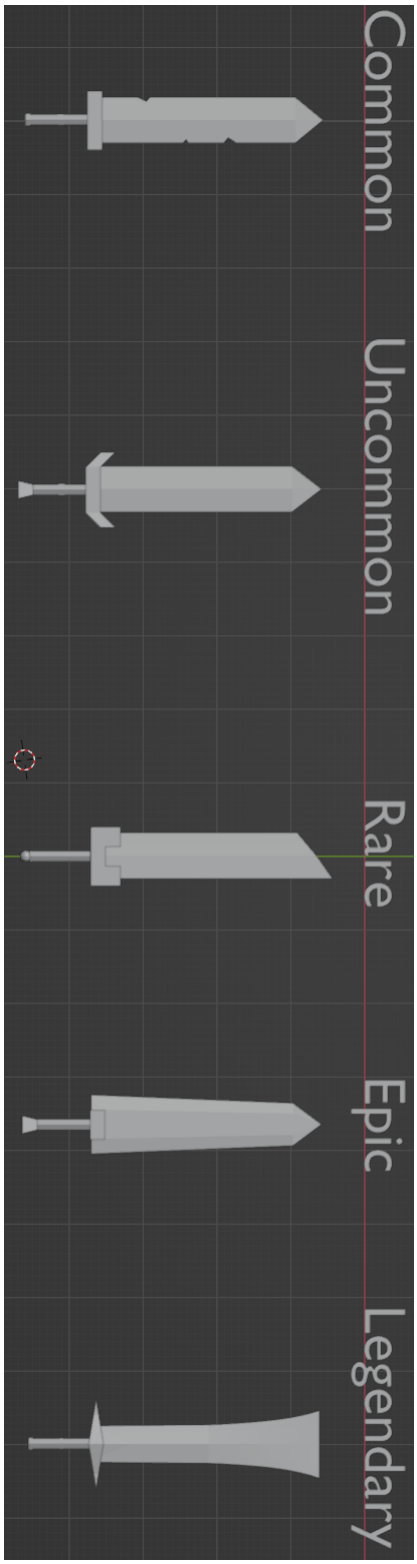


Figura B.1: GreatSwords sem Shading

Apêndice C

GreatSwords com Shading

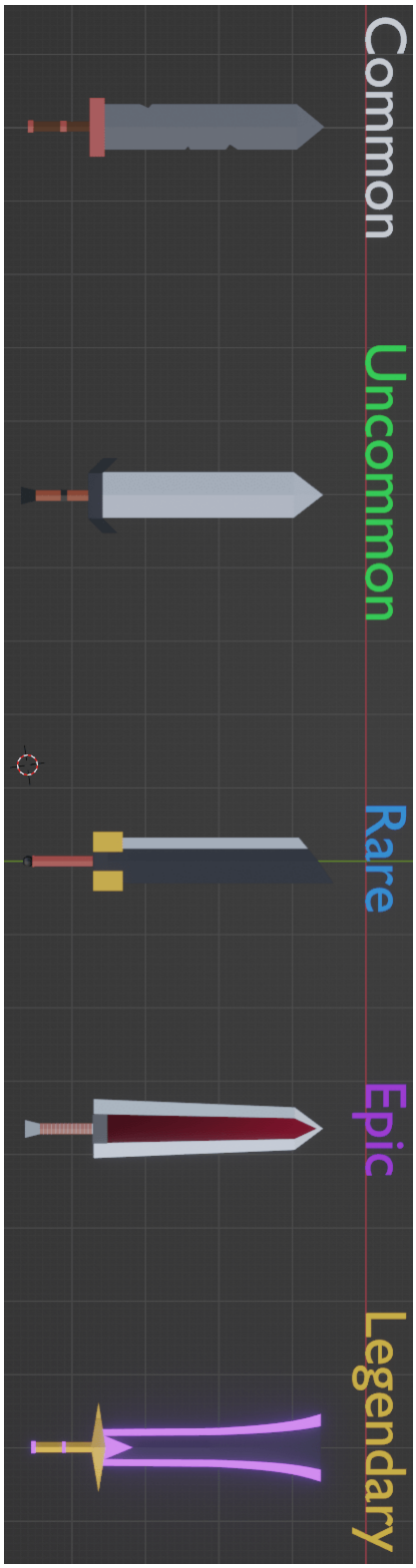


Figura C.1: GreatSwords com Shading

Apêndice D

Candeeiro sem Shading

Figura D.1: Candeeiro sem Shading

Apêndice E

Escudo

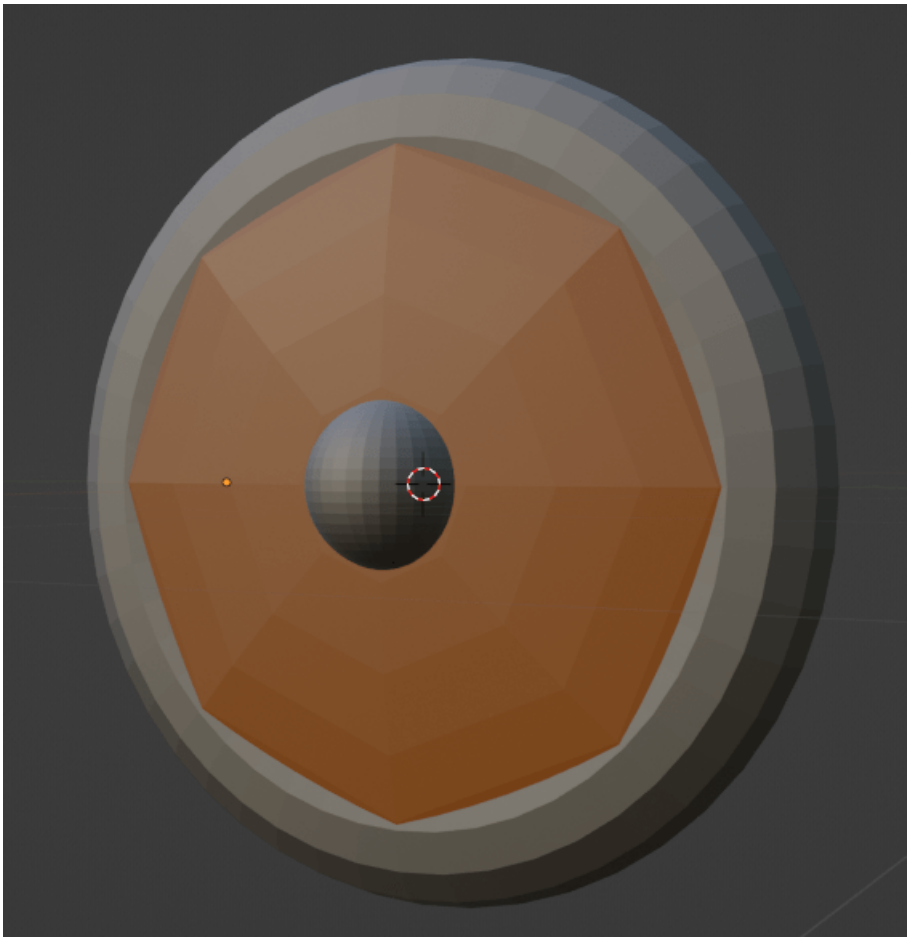


Figura E.1: Escudo

Apêndice F

Espadas sem shading

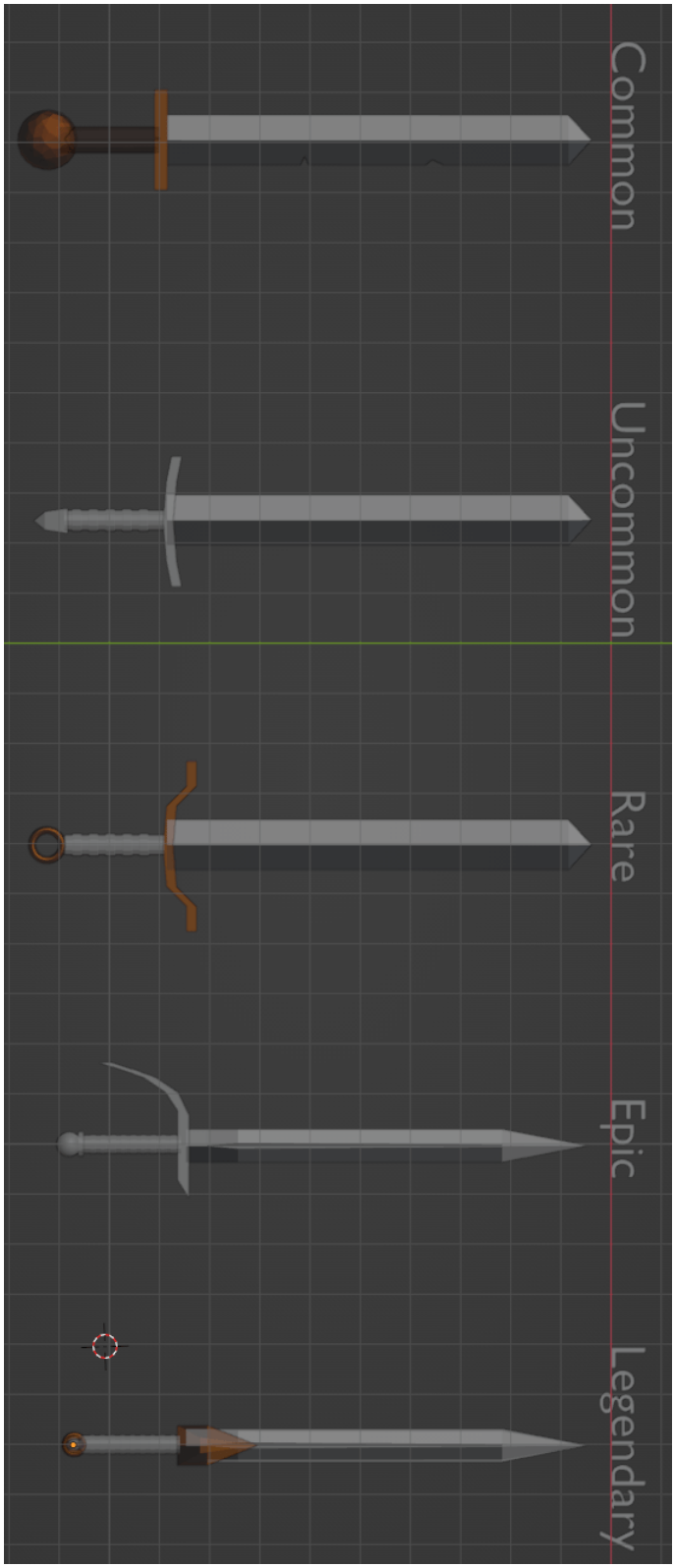


Figura F.1: Espadas sem shading

Apêndice G

AnimationSounds

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Animation_Sounds : MonoBehaviour
{
    public GameObject AM;

    //-------------------------------------------------------Basic
        Sounds-------------------------------------------------------------
    public void PlayerJumpSound()
    {
        AM.gameObject.GetComponent<PlayerAudioManager>().Play("Jump");
    }

    public void PlayerFallSound()
    {
        AM.gameObject.GetComponent<PlayerAudioManager>().Play("Fall");
    }

    public void PlayerDashSound()
    {
        AM.gameObject.GetComponent<PlayerAudioManager>().Play("Dash");
```

```
}

//-----------------------------------------------------Player
    Mage-------------------------------------------------------------
public void PlayerFireball()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("FireballSound");
}
    public void PlayerTP()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("TPSound");
}
    public void PlayerBasicMage()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("RangedBasicAttackS
}

public void PlayerIceSpell()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("IceSpell");
}

public void PlayerStrike1()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("Strike1");
}

public void PlayerStrike2()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("Strike2");
}

//-------------------------------------------------Enemy
    Skeleton---------------------------------------------------------------
```

```
public void SkeletonAttack()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("SkeletonAttack");
}

public void SkeletonHurt()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("SkeletonHurt");
}

public void SkeletonDeath()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("SkeletonDeath");
}

//-----------------------------------------Enemy
   Ranged-----------------------------------------------------------------
public void RangedFire()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("RangedFire");
}

public void RangedHurt()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("RangedHurt");
}

public void RangedDeath()
{
    AM.gameObject.GetComponent<PlayerAudioManager>().Play("RangedDeath");
}

//-----------------------------------------Enemy
   Bug--------------------------------------------------------------------
public void BugAttack()
```

```
    {
        AM.gameObject.GetComponent<PlayerAudioManager>().Play("BugAttack");
    }

    public void BugHurt()
    {
        AM.gameObject.GetComponent<PlayerAudioManager>().Play("BugHurt");
    }

    public void BugDeath()
    {
        AM.gameObject.GetComponent<PlayerAudioManager>().Play("BugDeath");
    }
}
```

Apêndice H

AudioSwap

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Audio;

public class AudioSwap : MonoBehaviour
{
    public AudioClip newTrack;
    public AudioMixerGroup mixer;
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            LevelAudioManager.instance.SwapTrack(newTrack);
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            LevelAudioManager.instance.ReturnToDefault();
        }
```

```
    }
}
```

Apêndice I

FootSteps

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class FootSteps : MonoBehaviour
{
    [SerializeField]
    private AudioClip[] stoneClips;
    [SerializeField]
    private AudioClip[] mudClips;
    [SerializeField]
    private AudioClip[] grassClips;

    private AudioSource audioSource;
    private TerrainDetector terrainDetector;

    private void Awake()
    {
        audioSource = GetComponent<AudioSource>();
        terrainDetector = new TerrainDetector();
    }
```

```
private void Step()
{
    AudioClip clip = GetRandomClip();
    audioSource.PlayOneShot(clip);
}

private AudioClip GetRandomClip()
{

    if(SceneManager.GetActiveScene().name ==
       "PrisonLevel(Prologue)")
    {
        int terrainTextureIndex =
           terrainDetector.GetActiveTerrainTextureIdx(transform.position);
        return stoneClips[UnityEngine.Random.Range(0,
           stoneClips.Length)];
    }

    else if (SceneManager.GetActiveScene().name == "CityLevel")
    {
        int terrainTextureIndex =
           terrainDetector.GetActiveTerrainTextureIdx(transform.position);
        switch (terrainTextureIndex)
        {
            case 0:
            default:
                return mudClips[UnityEngine.Random.Range(0,
                   mudClips.Length)];
            case 1:
                return grassClips[UnityEngine.Random.Range(0,
                   grassClips.Length)];
            case 2:
                return stoneClips[UnityEngine.Random.Range(0,
                   stoneClips.Length)];
        }
```

```
        }
        else if (SceneManager.GetActiveScene().name ==
            "FirstDungeon")
        {
            int terrainTextureIndex =
                terrainDetector.GetActiveTerrainTextureIdx(transform.position);
            switch (terrainTextureIndex)
            {
                case 0:
                default:
                    return grassClips[UnityEngine.Random.Range(0,
                        grassClips.Length)];
                case 1:
                    return stoneClips[UnityEngine.Random.Range(0,
                        stoneClips.Length)];
                case 2:
                    return mudClips[UnityEngine.Random.Range(0,
                        mudClips.Length)];
            }
        }
        return null;
    }
}
```

Apêndice J

LevelAudioManager

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Audio;

public class LevelAudioManager : MonoBehaviour
{
    public AudioClip defaultAmbience;
    private AudioSource track01, track02;
    private bool isPlayingTrack01;
    public AudioMixerGroup mixer;

    public static LevelAudioManager instance;

    private void Awake()
    {
        if (instance == null)
            instance = this;
    }

    private void Start()
    {
        track01 = gameObject.AddComponent<AudioSource>();
```

```
        track01.outputAudioMixerGroup = mixer;
        track01.loop = true;
        track02 = gameObject.AddComponent<AudioSource>();
        track02.outputAudioMixerGroup = mixer;
        track02.loop = true;

        isPlayingTrack01 = true;

        SwapTrack(defaultAmbience);
    }

    public void SwapTrack(AudioClip newClip)
    {
        StopAllCoroutines();

        StartCoroutine(FadeTrack(newClip));

        isPlayingTrack01 = !isPlayingTrack01;
    }

    public void ReturnToDefault()
    {
        SwapTrack(defaultAmbience);
    }

    private IEnumerator FadeTrack(AudioClip newClip)
    {
        float timeToFade = 2f;
        float TimeElapsed = 0f;

        if (isPlayingTrack01)
        {
            track02.clip = newClip;
            track02.Play();
```

```
            while (TimeElapsed < timeToFade)
            {
                track02.volume = Mathf.Lerp(0, 0.5f, TimeElapsed /
                    timeToFade);
                track01.volume = Mathf.Lerp(0.5f, 0, TimeElapsed /
                    timeToFade);
                TimeElapsed += Time.deltaTime;
                yield return null;
            }

            track01.Stop();
        }
        else
        {
            track01.clip = newClip;
            track01.Play();

            while (TimeElapsed < timeToFade)
            {
                track01.volume = Mathf.Lerp(0, 0.5f, TimeElapsed /
                    timeToFade);
                track02.volume = Mathf.Lerp(0.5f, 0, TimeElapsed /
                    timeToFade);
                TimeElapsed += Time.deltaTime;
                yield return null;
            }

            track02.Stop();
        }
    }
}
```

Apêndice K

PlayerAudioManager

```
using UnityEngine.Audio;
using System;
using UnityEngine;
//using Unity.Netcode;

public class PlayerAudioManager : MonoBehaviour
{
    public Sound[] sounds;

    public static PlayerAudioManager instance;

        foreach (Sound s in sounds)
        {
            s.rolloffMode = AudioRolloffMode.Linear;
            s.source = gameObject.AddComponent<AudioSource>();
            s.source.outputAudioMixerGroup = s.mixer;
            s.source.clip = s.clip;
            s.source.volume = s.volume;
            s.source.pitch = s.pitch;
            s.source.spatialBlend = s.spatialBlend;
            s.source.minDistance = s.MinDistance;
            s.MinDistance = 1;
            s.source.maxDistance = s.MaxDistance;
```

```
            s.MaxDistance = 5;
            s.source.loop = s.loop;
            s.source.playOnAwake = s.playOnAwake;
        }
    }

    public void Play (string name)
    {
        Sound s = Array.Find(sounds, sound => sound.name == name);
        if (s == null)
        {
            Debug.LogWarning("Sound: " + name + " not found!");
            return;
        }

        s.source.Play();
    }
        public void Stop (string name)
    {
        Sound s = Array.Find(sounds, sound => sound.name == name);
        s.source.Stop();
    }
}
```

Apêndice L

PropSync

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Audio;

public class PropSync : MonoBehaviour
{
    public AudioSource prop;
    public GameObject particle;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("AudioProp"))
        {
            prop.Play();
            StartCoroutine(ActivateParticle());
        }
    }

    private IEnumerator ActivateParticle()
    {
        particle.SetActive(true);
        yield return new WaitForSeconds(1f);
```

```
            particle.SetActive(false);
        }
}
```

Apêndice M

Sound

```
using UnityEngine.Audio;
using UnityEngine;


[System.Serializable]
public class Sound
{
    public string name;

    public AudioClip clip;

    [Range(0f, 1f)]
    public float volume;
    [Range(.1f, 3f)]
    public float pitch;
    [Range(0f, 1f)]
    public float spatialBlend;
    [Range(1f, 500f)]
    public float MinDistance;
    [Range(1f, 500f)]
    public float MaxDistance;

    public bool loop;
```

```
    public bool playOnAwake;

    public AudioMixerGroup mixer;

    public AudioRolloffMode rolloffMode;
    [HideInInspector]
    public AudioSource source;
}
```

Apêndice N

TerrainDetector

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TerrainDetector
{
    private TerrainData terrainData;
    private int alphamapWidth;
    private int alphamapHeight;
    private float[,,] splatmapData;
    private int numTextures;

    public TerrainDetector()
    {
        terrainData = Terrain.activeTerrain.terrainData;
        alphamapWidth = terrainData.alphamapWidth;
        alphamapHeight = terrainData.alphamapHeight;

        splatmapData = terrainData.GetAlphamaps(0, 0,
           alphamapWidth, alphamapHeight);
        numTextures = splatmapData.Length / (alphamapWidth *
           alphamapHeight);
    }
```

```
private Vector3 ConvertToSplatMapCoordinate(Vector3
    worldPosition)
{
     Vector3 splatPosition = new Vector3();
     Terrain ter = Terrain.activeTerrain;
     Vector3 terPosition = ter.transform.position;
     splatPosition.x = ((worldPosition.x - terPosition.x) /
        ter.terrainData.size.x) * ter.terrainData.alphamapWidth;
     splatPosition.z = ((worldPosition.z - terPosition.z) /
        ter.terrainData.size.z) *
        ter.terrainData.alphamapHeight;
     return splatPosition;
}

public int GetActiveTerrainTextureIdx(Vector3 position)
{
     Vector3 terrainCord =
        ConvertToSplatMapCoordinate(position);
     int activeTerrainIndex = 0;
     float largestOpacity = 0f;

     for (int i = 0; i < numTextures; i++)
     {
          if (largestOpacity < splatmapData[(int)terrainCord.z,
             (int)terrainCord.x, i])
          {
               activeTerrainIndex = i;
               largestOpacity = splatmapData[(int)terrainCord.z,
                  (int)terrainCord.x, i];
          }
     }

     return activeTerrainIndex;
}
```

```
}
```