UNIVERSIDADE DO ESTADO DO AMAZONAS

ESCOLA SUPERIOR DE TECNOLOGIA

SISTEMAS DE INFORMAÇÃO

NADINE DA CUNHA BRITO CAPISTRANO

REMASTERIZAÇÃO DO JOGO BERZERK EM UNITY

Manaus – Fevereiro – 2024

NADINE DA CUNHA BRITO CAPISTRANO

REMASTERIZAÇÃO DO JOGO BERZERK EM UNITY

Trabalho de Conclusão de Curso apresentado à banca avaliadora do Curso de Sistemas de Informação, da Escola Superior de Tecnologia, da Universidade do Estado do Amazonas, como pré-requisito para obtenção do título de Engenheiro de Computação.

Orientador: Prof. Dr. Jucimar Maia da Silva Junior

Manaus – Fevereiro – 2024

Resumo

Este trabalho confecciona uma remasterização do jogo *Berzerk* (Atari, 1982) implementada na *Unity Engine*, tendo como objetivo principal preservar a estrutura do jogo original trazendo elementos do desenvolvimento de *software* modernos, mas sem perder a essência do *design* de jogo original, buscando a fidelidade aos elementos marcantes de um dos jogos mais populares da era de ouro dos jogos de *arcade*.

Palavras-Chave: Jogo, Berzerk, Atari, Arcade, Remaster, Unity, Unity Engine.

Abstract

This project creates a remaster of the game Berzerk (Atari, 1982) implemented in the Unity Engine, with the main goal of preserving the structure of the original game while incorporating modern software development elements. However, it aims to retain the essence of the original game design, striving for fidelity to the iconic features of one of the most popular games from the golden era of arcade gaming.

Keywords: Game, Berzerk, Atari, Arcade, Remaster, Unity, Unity Engine.

Agradecimentos

Agradeço imensamente a Deus por me guiar nessa jornada na universidade, na elaboração deste trabalho, e principalmente na jornada da vida, sem sua mão poderosa agindo sobre minha vida, nada seria possível.

Agradeço meu marido maravilhoso, Ariel, por todo o suporte que me deu durante os dias difíceis que passei, sem sua parceria e bom humor tudo ficaria mais complicado.

Agradeço meu pai Edilim, minha mãe Eny e minha irmã Linda que me apoiaram e incentivaram desde o início do curso, sempre preocupados com meu bem-estar, sendo essenciais para o meu progresso na academia.

Agradeço aos meus amigos que foram compreensivos com minha ausência, e por toda ajuda que me deram, algumas foram essenciais para que eu pudesse elaborar esse trabalho.

Agradeço ao professor Jucimar por aceitar se tornar meu orientador, sua visão inovadora de tornar o estado do Amazonas um polo tecnológico me inspirou profundamente desde o início do curso.

Agradeço imensamente a professora Diana que me ajudou no processo de escrita desse trabalho, suas sugestões me guiaram a escrever um trabalho melhor do que se eu tivesse feito sem auxílio.

Agradeço ao professor Raimundo e toda a equipe do projeto Finance e a equipe do Tuti-Labs que foram compreensivos com meu foco na escrita desse trabalho.

E finalmente agradeço a Universidade do Estado do Amazonas que me proporcionou a oportunidade de me tornar profissional na área da computação, tal oportunidade mudou minha visão de vida e me tornou uma pessoa melhor tanto interpessoalmente como profissionalmente.

Sumário

Lista de Figuras

Lista de Códigos

Capítulo 1

Introdução

A palavra jogo tem por significado: “Atividade física ou mental organizada por um sistema de regras que definem a perda ou o ganho” (FERREIRA 2009). Segundo (GULARTE 2010) jogos de tabuleiro são uma das atividades recreativas mais antigas da história da humanidade, tendo registros oriundos dos anos 3500 a.C. de jogos de tabuleiro sendo jogados no Egito antigo, o autor ainda cita outros tipos de jogos, iniciando pelos jogos de cartas, que segundo o autor, são constituídos por um conjunto de símbolos com valores de combinação lógica, onde possuem regras para que as combinações neles empregadas gerem alguma pontuação, além destes é possível registrar a competitividade dos jogos atléticos, que inspiraram a criação dos famosos jogos olímpicos, um evento tão aguardado e prestigiado por pessoas de todo o mundo, e indo para o ramo educacional é possível citar os jogos lúdicos para crianças, os quais são tão importantes para o aprendizado e crescimento intelectual dos pequenos, e finalmente os jogos eletrônicos, os quais se caracterizam pelo uso de dispositivos eletrônicos a base de energia elétrica ou baterias, onde esses jogos não exigem qualidade física, todavia segundo (LUZ 2010) “desenvolvem a cognição e a percepção físico-espacial”.

De acordo com vários autores o mercado de jogos eletrônicos ascendeu muito rapidamente nos últimos anos, tanto tecnologicamente como comercialmente, ganhando cada vez mais apreciadores pelo mundo todo, mais investimentos, mais exposição midiática, o mercado está se tornando maior do que nunca, mas para se chegar a esse patamar, é preciso de um início, de uma faísca, de toda uma história por trás deste sucesso, e segundo ARANHA (2004, *apud*

(MENDONÇA 2019)) a história dos *video games* se divide em 4 fases antes de chegar a fase atual da história, onde na primeira fase, engloba os primórdios da tecnologia, onde os computadores digitais eram a novidade da época, é nessa fase que se encontra o primeiro registro de um *video game*, um jogo de tênis, que foi criado em 1958 por um físico chamado Willy Higinbotham, onde esse *video game* era processado por um computador analógico, e como na época ainda não existia a visão mercantil de jogos, o *Tennis for Two* como era chamado não chegou a ser patenteado. Na segunda fase da história dos *video games* se encontra o período de disseminação dos jogos, onde a evolução das tecnologias daquela época criaram vários produtos eletrônicos para fins de usos para jogos dentre eles os *arcades* que imediatamente fizeram sucesso com a sua disseminação em bares, restaurantes e desse sucesso se deu início ao mercado de *video games*, dentro dessa era nasceram alguns jogos marcantes como por exemplo *Pong, Space Invaders, Pac-Man e Donkey Kong*, jogos que se tornaram o símbolo dessa era e que fizeram muito sucesso comercial com os jogadores. Já a terceira fase da história dos *video games* foi um marco, pois nessa fase ocorreu um decaimento do mercado, pois as empresas de *video games* de sucesso se apegaram tanto aos seus jogos populares, que realizaram uma produção em massa de *video games* sem qualquer novidade somente deixando o mercado saturado de clones de seus jogos de sucesso, consequentemente gerando uma baixa comercial significativa e uma insatisfação dos consumidores, para contornar essa situação foi criado novos parâmetros de qualidade no mercado de jogos para impedir que situações parecidas ocorressem novamente. A quarta fase da história dos *video games* se caracteriza pela retomada do mercado, mesmo com a desconfiança dos consumidores, essa retomada foi possível com um planejamento de *marketing* bem feito, acordos entre empresas internacionais o que gerou competitividade entre elas e criou jogos com personagens carismáticos e com narrativas e jogabilidades melhoradas, tudo para atrair novamente a confiança dos consumidores, essa fase da história também e marcada pelos avanços tecnológicos que geraram a era dos *consoles*, que são estações de jogos menores que os *arcades* que cada consumidor poderia ter em sua residência trazendo a experiência do entretenimento para o seio familiar. Na fase atual da história dos *video games*, novos dispositivos tecnológicos como celulares e dispositivos de realidade virtual foram incorporados a indústria de

jogos trazendo experiências diferentes ao usuário, integrando cada vez mais os jogadores entre si principalmente por meio de redes sociais que fazem essa comunicação mais direta acontecer e a era atual também traz *consoles* cada vez mais poderosos e jogos cada vez mais diversos, essas novidades fomentam o mercado que se encontra em constante elevação.

De acordo com GULARTE (2010) dentro da era de sucesso dos *arcades* foram criados jogos extremamente importantes para a história da indústria dos *games*, o maior sucesso inicial foi do jogo *Pong*, um jogo de *ping-pong*, onde existiam duas raquetes nas extremidades esquerda e direita de uma tela que podiam ser controladas por dois jogadores diferentes, ou um jogador poderia jogar sozinho, enfrentando o próprio computador, onde uma bolinha podia ser rebatida em vários ângulos diferentes, e o objetivo do jogo era rebater a bola de modo que o adversário não conseguisse rebater de volta a bola e assim fazer mais pontos, ele foi produzido pela famosa empresa Atari no início de 1970, e pelos próximos 5 anos fez um sucesso tão grande que todas as empresas rivais da Atari tinham um clone de Pong. Com o passar do tempo o mercado já estava saturado de *Pong* e de seus clones, foi com isso que em 1978 se iniciou a segunda geração dos *arcades*, com os esforços das maiores empresas de jogos da época a Atari, Taito, Namco e a novata Nintendo, lançaram os jogos ícones dos *arcades*, que já foram citados anteriormente, o *Space Invaders*, o *Pac-Man* e o *Donkey Kong*. *Space Invaders*, um jogo criado pela empresa Taito, onde o jogador, na base da tela, se defende de uma invasão alienígena contra a terra, atirando em seus invasores e se movendo da esquerda para a direita e vice-versa pela base da tela, esse jogo inclusive inseriu o conceito de pontuação e recorde no seu jogo trazendo a competitividade no meio social dos jogadores, pois agora era possível ultrapassar a pontuação do colega. Já o *Pac-Man* foi criado pela Namco, para aqueles jogadores que não queriam mais jogar *Pong* nem *Space Invaders*, pois trazia uma jogabilidade diferente dos jogos de esporte e jogos de tiro já bastante consolidados no mercado, foi pensado para atrair o público de ambos os sexos, tentando se afastar da previsibilidade dos jogos populares da época, o jogo consistia em um labirinto, com 240 pontos que podiam ser recolhidos pelo seu personagem o *Pac-Man*, onde o jogador além disso precisava se preocupar com os fantasmas que o seguiam e podiam o matar, mas em certo momento o jogo podia virar pois existia a possibilidade do jogador comer

pílulas especiais onde ele ganhava o poder de comer os fantasmas que o perseguiam, o jogo foi imediatamente aprovado pelos consumidores pois era inovador na sua jogabilidade. E finalmente *Donkey Kong* outro título de sucesso feito pela iniciante Nintendo, onde o jogador que controlava o *Jumpman*, cujo tinha como objetivo, salvar a sua namorada Pauline da prisão de um macaco gigante e raivoso que atirava em *Jumpman* pesados barris pelas plataformas abaixo, por ser o primeiro jogo no estilo de plataforma inovou e fez bastante sucesso. Outro sucesso da década de 80 foi um jogo chamado *Berzerk* (Atari, 1982), de acordo com (RIPARDO 2018) "*Berzerk* é um jogo multi-direcional desenvolvido partir do seu título original do *arcade*, lançado em Novembro de 1980 e em 1982 para Atari 2600 [...] o objetivo geral do jogo é bastante simples, sobreviver. O jogo se passa dentro de um labirinto, onde o jogador tem que enfrentar os inimigos e ir passando entre salas para conseguir o maior número de pontos possíveis."
A era de ouro dos *arcades* foi marcante para uma geração inteira, a transformação inspirada por essa revolução nos jogos eletrônicos tem seus efeitos perceptíveis nos dias de hoje e só tendem a se manifestarem cada vez mais com a grande popularidade que os jogos eletrônicos possuem atualmente. Para que tudo isso fosse possível, a geração que viveu essa época, preservou e propagou a cultura *gamer* que se mostra tão presente na contemporaneidade. Mediante a importância histórica e intelectual, existe a necessidade de preservar os costumes e conhecimentos gerados por essa cultura para a posteridade, para que isso ocorra, é importante o estudo do conhecimento gerado e das tecnologias e inovações que foram trazidas pelos jogos de *arcade* dos anos 80. Aprender com o passado é extremamente enriquecedor no sentido intelectual em qualquer área de pesquisa, e é importante para a construção de conhecimento, o aperfeiçoamento de tecnologias antecessoras, para se ter em mente o que inicialmente foi inovador e disruptivo e ao mesmo tempo adaptar para as tecnologias das ferramentas modernas.

1.1 Justificativa

Vários jogos lançados na época dos arcades foram revolucionários de alguma maneira, como por exemplo o *Pong* sendo divertido ao mesmo tempo que era simples e prático, ou o famoso *Space Invaders* instituindo o estilo "atirador" nos jogos, ou até mesmo o icônico *Pac-*

Man inserindo o conceito de "poderes" no mundo dos *games*, mas se observarmos do ponto de vista de desenvolvimento de *software*, esses jogos eram mais simples de construir e mais despretensiosos do que os jogos eletrônicos produzidos nos dias de hoje, que exigem muito mais complexidade na produção, na jogabilidade e nas animações, deixando-as mais rebuscadas e com funcionalidades melhores, consequentemente deixando os jogos melhores, tudo isso se torna possível pela evolução das tecnologias de desenvolvimento de *software* atuais que hoje se encontram em um patamar superior em questão de *design*, eficiência, experiência do usuário, facilidade de implementação e integração entre tecnologias.

1.2 Objetivos

Seguem especificados nessa sessão os objetivos deste trabalho.

Objetivo Geral

Desenvolver um jogo *top-down* de tela fixa 2D baseado inteiramente no jogo original, *Berzerk* (Atari, 1982), tendo como ferramenta principal a *Unity*, e como ferramentas auxiliares, o editor *Rider*, o *software* de controle de versionamento *GitHub Desktop* e o FMOD como ferramenta de edição de áudio, visando a remasterização do jogo inicial para a melhoria em vários aspectos incluindo *IA*, mecânicas, gráficos e som.

Objetivos Específicos

Para atingir plenamente o objetivo geral o cumprimento das seguintes etapas são necessárias:

Etapa 1 (Sistemas Principais)

- Implementar um sistema de movimentação do jogador em 8 direções;
- Implementar o sistema de tiro do jogador;
- Implementar um sistema de geração de salas aleatórias;

- Implementar um sistema de eletrificação das paredes;
- Implementar um sistema de geração de robôs inimigos;
- Implementar um sistema de movimentação dos robôs;
- Implementar um sistema de detecção de presença dos robôs (horizontal e vertical);
- Implementar um sistema de detecção de robôs abatidos;
- Implementar um sistema de pontuação;

Etapa 2 (Sistemas Secundários)

- Implementar um sistema de surgimento do Evil Otto;
- Implementar um sistema de vidas extras;
- Implementar um *design* minimalista para combinar com o jogo original;
- Implementar efeitos de áudio, para os tiros, paredes eletrificadas, e para a aproximação do Evil Otto;
- Implementar efeitos de transição de sala;
- Testar a usabilidade do jogo implementado;

É válido mencionar que o *software* resultante desse trabalho é somente baseado no jogo original e tem apenas fins educacionais e acadêmicos, e o *software* é código aberto e ficará disponível em repositório público e de livre acesso no link a seguir: Repositório Berzerk Remaster.

1.3 Metodologia de Pesquisa

A metodologia de pesquisa usada nesse trabalho foi a pesquisa exploratória, a escolha dessa metodologia foi feita por conta de sua abordagem permissiva à exploração do que já foi realizado dentro do tema do presente trabalho. (GIL 2002) diz que a pesquisa exploratória "tem

como objetivo proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a constituir hipóteses" e com o objetivo de aprofundamento do tema estabelecido e buscando criar uma base teórica sólida para este trabalho, dentro da pesquisa exploratória foi realizada uma pesquisa qualitativa onde foram buscadas as palavras-chave: "*video game*", "*remake*" e "*game* design" em trabalhos similares a este e igualmente foi usada uma pesquisa quantitativa, onde foi aplicado um filtro de busca: o arquivo com acesso liberado, com isso totalizando nove trabalhos com temas parecidos que foram encontrados em plataformas digitais de hospedagens de trabalhos científicos, principalmente na plataforma CAFe que dá acesso ao Portal de Periódicos CAPES e também dentro da plataforma Google Scholar.

1.4 Cronograma

As atividades do presente trabalho de conclusão de curso foram divididas e organizadas em duas etapas: cronograma do TCC 1 e cronograma do TCC 2 detalhadas nas figuras abaixo:

	maio/2023				**junho/2023**					**julho/2023**				**agosto/2023**			
	semana				**semana**					**semana**				**semana**			
Atividades	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
Definição do tema e escopo		X															
Revisão bibliográfica		X	X	X	X	X	X	X	X	X	X	X	X				
Definição dos requisitos		X	X	X													
Planejamento do projeto		X	X	X													
Projeto de arquitetura		X	X	X	X												
Implementação das mecâncias principais					X	X	X	X	X	X	X	X					
Análise dos resultados do TCC 1													X	X	X		
Elaboração do TCC 1		X	X	X	X	X	X	X	X	X	X	X	X	X	X		
Entrega do TCC 1																X	
Apresentação do TCC 1																	X

Figura 1.1: Cronograma de Atividades do Trabalho de Conclusão de Curso I

	outubro/2023				novembro/2023					dezembro/2023				janeiro/2024					fevereiro/2024			
	semana				semana					semana				semana					semana			
Atividades	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**
Implementação das mecânicas secundárias			X	X	X																	
Implementação do sistema de som						X	X	X														
Refatoração do design do jogo									X	X	X											
Testes												X										
Depuração e correção de erros													X	X								
Avaliação e validação														X	X							
Análise dos resultados do TCC 2														X	X	X						
Elaboração do TCC 2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X					
Entrega do TCC 2																		X				
Apresentação do TCC 2																			X			

Figura 1.2: Cronograma de Atividades do Trabalho de Conclusão de Curso II

Capítulo 2

Referencial Teórico

No capítulo que se segue, será apresentado os detalhes sobre todo o material conceitual e as ferramentas utilizadas para a elaboração deste trabalho, para que se tenha um pleno entendimento do referencial teórico empregado na elaboração do *software*.

2.1 Remasterização de Jogos

Para (AUSTIN e SLOAN 2022) "O *remake* ou *remaster* é uma reconstrução literal de um jogo existente, na maioria das vezes com o objetivo de atualizar a fidelidade audiovisual (e ocasionalmente também a jogabilidade) para uma nova geração de *hardware* de jogos." essa definição ajuda o entender como remasterizações são importantes pois com o passar do tempo e conforme a tecnologia vai evoluindo fica cada vez mais difícil acessar certos tipos de jogos, pois suas tecnologias antigas dificultam a jogabilidade, a experiência do jogador e também dificultam a sua propagação, com isso remasterizações tem cada vez mais importância na preservação dos jogos originais (SALTER 2017). Para contraponto o estudo de (DHARMA e SETIADI 2023) sobre prós e contras de jogos *remake* e *remasters* e concluíram que quando um jogo passa por um processo de remasterização geralmente perde pontos de classificação com jogadores que tiveram acesso ao jogo original, pois como eles já jogaram o jogo original não sentem a necessidade de jogar novamente, além disso os jogadores se mostraram insatisfeitos com os preços, pois demonstram que não vale mais a pena empregar recursos, por outro lado também foram

detectados prós das remasterizações de jogos que são: melhorias na qualidade dos gráficos e animações, desempenho melhorado, áudio aperfeiçoado e adição de minijogos foram pontos positivos na experiência dos jogadores que foram levantados nessa pesquisa, além disso, a possibilidade de correções de *bugs* e uma sensação nostálgica que os fãs podem sentir, trazem uma nota positiva para jogos remasterizados. Se tratando de tecnologia, na década de 70, se tornou comum o desenvolvimento de jogos de *arcade* baseados em microprocessadores (KENT 2001), os desenvolvedores de jogos precisavam estudar o *hardware* para desenvolver jogos que extraíssem melhor desempenho daquele *hardware*, pois a memória dos dispositivos daquela época eram bastante limitadas.

Dentro deste trabalho, a remasterização do *Berzerk* foi feita em duas etapas, na primeira etapa a remasterização foi feita de modo a se parecer esteticamente e mecanicamente idêntica ao jogo original usando um *software* chamado *Unity* em vez de tomar como base um *hardware*, já que atualmente não existe necessidade de se preocupar com memória, pois as tecnologias atuais já suprem essa necessidade, na segunda etapa o jogo passou por uma transformação estética afim de deixar os gráficos e áudios do jogo modernos e para a realização dessa remasterização e novamente foi utilizado a ferramenta *Unity* uma das ferramentas mais completas e modernas disponíveis para desenvolvimento de jogos.

2.2 Design de Jogos

Segundo (FULLERTON 2008) o *game designer* é responsável por criar um conjunto de regras dentro das quais há meios e motivação para jogar, por isso essa atribuição possui grande importância para uma experiência positiva do usuário com o jogo, se existir algum objeto, botão, imagem em um lugar errado, pode mudar drasticamente a experiência do usuário com aquele jogo, e o processo de *design* de jogos é encarregado disso, por deixar bem definido a forma que o jogo vai tomar, tanto em questões visuais, quanto em questões de estilo de jogo, regras, enredo, sonorização, construção das fases, objetivos do jogo dentre outras, tudo isso é necessário pois a maior preocupação do *design* de jogos é a experiência do jogo e que tipos de sensações, emoções e reações que serão planejadas para serem causadas nos jogadores.

O *design* do jogo *Berzerk* é baseado em mecânicas simples, basicamente andar pelas salas e atirar nos robôs inimigos, porém o jogo se torna desafiador por conta do nível de dificuldade que o jogo vai adquirindo conforme os inimigos vão ganhando velocidade de movimento e de tiro ao longo dos níveis e também por conta das possibilidades de estratégias possíveis apenas usando essas simples mecânicas, como por exemplo a possibilidade do jogador atirar nas diagonais, pois os robôs apenas detectam a presença do jogador nas verticais e horizontais, essas características fizeram com que o jogo fizesse sucesso com o público e o tornando-o um jogo marcante. O *design* de *Berzerk* foi inovador para a época, oferecendo uma combinação de ação frenética, estratégia e desafio, e contribuiu para estabelecer muitos dos conceitos e mecânicas que seriam posteriormente usados em outros jogos de *arcade* e gêneros de *videogame*. Na remasterização realizada nessa trabalho, foi utilizado a ferramenta FMOD para o *design* de som com objetivo de dar mais imersão ao jogo por meio dos sons característicos do jogo original.

2.3 Inteligência Artificial

De acordo com MCCARTHY (1955 *apud* (MANNING 2020)), a Inteligência Artificial seria a "ciência e a engenharia dedicada a tornar as máquinas inteligentes", no entanto segundo (KISHIMOTO 2004) existem diferenças entre os objetivos da inteligência artificial acadêmica e da inteligencia artificial empregada nos jogos, esses objetivos diferentes seriam que na *IA* acadêmica o objetivo é solucionar problemas difíceis se baseando nas habilidades humanas de reconhecimento e percepção, já o objetivo da *IA* de jogos seria a diversão do jogador, não importando muito como essa *IA* chegaria ao cumprimento desse objetivo. Ainda segundo (KISHIMOTO 2004) a *IA* aplicada nos jogos da época de 1980 eram em sua maioria padrões de movimento, o que se aplica ao jogo *Berzerk* 1982, onde o movimento dos robôs inimigos era baseado na perseguição da posição do jogador. O autor (KISHIMOTO 2004) ainda afirma que o principal benefício do emprego das *IAs* nos jogos é proporcionar ao jogador divertimento trazendo erros humanos feitos pelos personagens o que traria uma personalidade ao jogo, esse é outro fator presente no jogo *Berzerk* de 1982 pois os robôs cometem erros bobos como atirarem nos robôs aliados e irem em direção as paredes eletrificadas que os destroem.

Dentro do jogo *Berzerk* de 1982, a inteligência artificial foi utilizada de maneira bem principiante para os dias atuais, mas para a época era bastante inovadora. A *IA* do *Berzerk* não tem um algoritmo de disparo sofisticado, sempre que o meio do corpo do jogador se alinhar com uma das oito linhas de tiro dos robôs inimigos (cima, baixo, esquerda e direita), eles atiram no jogador não importa a distância entre eles, nem a existência de uma parede no meio do caminho impede que eles atirem. Essa *IA* se mostra mais efetiva nos movimentos dos robôs inimigos, assim que eles detectam o jogador em algum lado dele, seja superior, inferior, lado esquerdo e lado direito, os robôs param tudo o que estão fazendo, detectam a presença do jogador andam e atiram nele iniciando uma perseguição contra o jogador, mas como a inteligência dos robôs não é muito desenvolvida, com essa perseguição desenfreada ao jogador os robôs podem acabar atirando em seus aliados, e também podem esbarrar nas paredes que são eletrificadas e mortais. Na remasterização feita nesse trabalho, procurou-se adaptar fielmente essa *IA* do jogo de 1982, utilizando um algoritmo simples de perseguição composto por funções de rastreamento do jogador, acionamento de movimentação dos robôs e acionamento de ataque dos robôs.

Capítulo 3

Metodologia de Desenvolvimento

O objetivo deste capítulo é detalhar os meios utilizados para o desenvolvimento da remasterização do jogo no quesito de desenvolvimento de *software* propriamente dito.

3.1 Berzerk

Com o objetivo de contextualizar o leitor sobre a história de criação do jogo *Berzerk* e a narrativa do jogo, foi realizada uma síntese histórica, para relatar os acontecimentos que fizeram o jogo o que ele é, e a sua relevância na atualidade.

Descrição do Jogo

Berzerk é um jogo de tiro multidirecional e sobrevivência, criado e lançado por Alan McNeil em 1980 para *arcade* pela Stern Electronics e portado para Atari em 1982. No jogo o *player* navega por um labirinto de salas enquanto luta contra robôs armados e evita paredes eletrificadas.

História da Criação do Jogo

Segundo a revista (RetroGamer 2021) Alan McNeil sempre gostou de desenvolvimentos de jogos, e procurou empresas que incentivassem a sua vontade de desenvolver jogos, até que encontrou a Stern Electronics que permitiu que Alan colocasse em prática as suas idéias. Segundo o próprio Alan após várias sessões noturnas de *Robots* (Um jogo primitivo que Alan estava desenvolvendo) ele sonhou com um *videogame* em preto e branco com um homem-palito

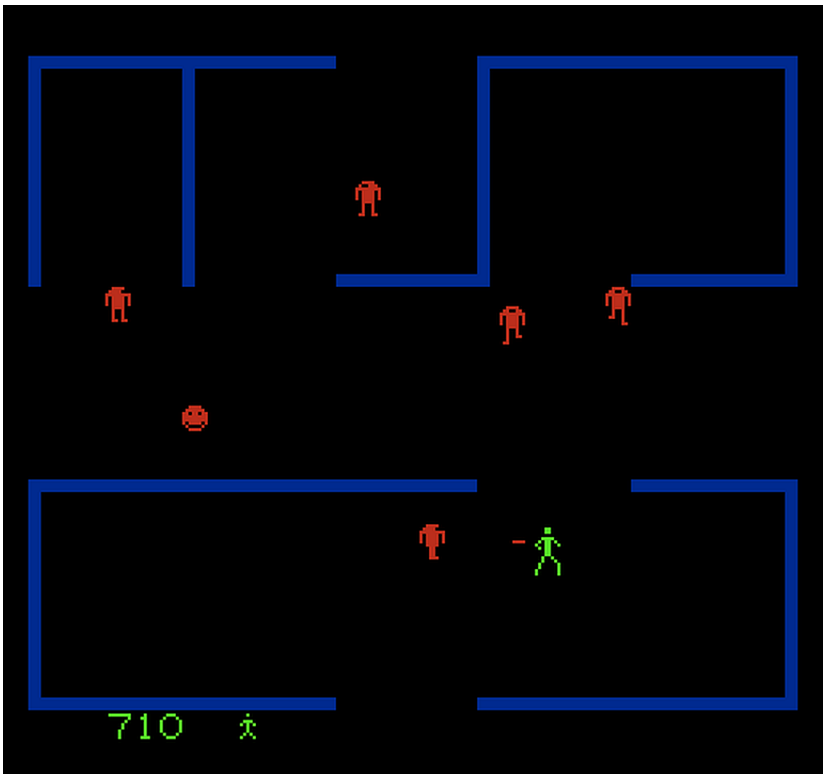

Figura 3.1: Primeira versão do jogo *Berzerk* de *arcade* da Stern Eletronics

e vários robôs se aproximando dele, e assim nasceu o *Berzerk* que mais tarde teve sua versão convertida para a plataforma Atari 2600.

Narrativa

De acordo com a revista de instruções do jogo *Berzerk* da Atari de 1982 (ATARI 1982), o jogador é o último sobrevivente de um grupo de pessoas que foram explorar o planeta Mazeon, no ano de 3200, onde achavam que era inabitado, porém, ao chegarem no planeta, a nave em que eles estavam foi destruída por automazeons, matando praticamente todos a bordo, esses robôs eram controlados por Evil Otto, um ser louco e implacável que observa a batalha e aparece sempre para perseguir e matar.

Mecânicas dos Personagens

- Humanoid (Jogador): O jogador precisa atirar nos inimigos, precisa desviar das balas inimigas, não pode encostar nas paredes eletrificadas e precisa passar entre as salas para conseguir o maior número de pontos possíveis.

- Automazeons: Os robôs não são muito inteligentes e também podem ser destruídos sempre

que se chocam, tocam as paredes mortais, são atingidos por seus próprios *lasers* ou são atropelados por Evil Otto. Mas não importa como eles são destruídos, o jogador recebe o crédito e marca pontos. Eles só enxergam em linha reta, seja vertical ou horizontal, o que facilita a eliminação deles pelas diagonais.

- Evil Otto: Um rosto sorridente e aterrorizante segundo vários jogadores da época, se Evil Otto aparecer quando ainda houver robôs na sala, ele saltará muito lentamente. Mas depois que todos os robôs forem destruídos, Evil Otto estará saltando ainda mais rápido. O humanóide deve escapar antes que Evil Otto o pegue ou ele estará perdido porque Evil Otto não pode ser morto. Ele aparece se você passar muito tempo em uma sala, vindo direto para destruir você.

Jogabilidade

- O jogo é *Single Player*;
- Quando o jogador usa o *Joystick* ele escolhe a direção que deseja andar;
- Quando o jogador usa o *Button*(ou *Fire*) ele atira.

Pontuação

- Cada robô abatido vale 50 pontos;
- Cada sala que o jogador esvazia vale mais 10 pontos extras por robô abatido;
- Se não esvaziar a sala não ganha pontos extras.

Variações do Jogo

Ainda de acordo com o manual de instruções do jogo (ATARI 1982), o *Berzerk* (1982) possui 12 variações de jogo que o jogador pode escolher selecionando com o *joystick* algum número entre 1 e 12, cada variação tem sua característica própria.

- Variação 1: Nessa variação o jogador recebe uma vida extra a cada 1000 pontos acumulados, e não há a aparição do Evill Otto, mas os robôs irão atirar;

Figura 3.2: Matriz de dificuldade de cada variante de jogo presente no *Berzerk*(1982)

- Variação 2: Variação que oferece uma vida bônus a cada 1000 pontos, os robôs atiram e o Evil Otto rebate o tiro que o jogador der nele;
- Variação 3: Variação que oferece uma vida bônus a cada 1000 pontos, os robôs atiram e existe a invencibilidade do vilão Evil Otto;
- Variação 4: Nessa variação o jogador só ganha vidas extras a partir de 2000 pontos acumulados, não existe Evil Otto mas os robôs irão atirar;
- Variação 5: A variação 5 é idêntica a variação 2 mudando só o fato de o jogador precisará acumular 2000 pontos para ganhar uma vida extra;
- Variação 6: Nessa variação o Evil Otto é invencível, jogador ganha uma vida extra a cada 2000 pontos e os robôs atiram;
- Variação 7: Na varição 7 o jogador não ganha vidas extras porém o Evil Otto não aparece, o jogador precisa se preocupar apenas com os robôs;
- Variação 8: Essa variação é idêntica a variação 7, tendo como diferença que o Evil Otto aparece, mas ele pode ser atingido por tiros do jogador e sumindo por um tempo.
- Variação 9: A variação 9 é um desafio, nassa variação o Evil Otto é invencível, os robôs atiram e o jogador não ganha vidas extras;

- Variação 10: Na variação 10, o jogador ganha uma vida extra a cada 1000 pontos, os robôs não atiram, todavia o Evil Otto é invencível;

- Variação 11: Essa variação foi feita para acumular pontos pois os robôs não atiram, o Evil Otto pode ser atingido por tiros do jogador e o jogador acumula vidas extras a cada 1000 pontos;

- Variação 12: Essa variação foi feita para crianças e jogadores iniciante pois é a mais fácil de todas as variações, nela os robôs não atiram, o Evil Otto não aparece e o jogador ganha uma vida a cada 1000 pontos acumulados.

Para a elaboração do presente trabalho foi escolhida a variação 3 para basear a remasterização, por conta de abranger toda a jogabilidade dificultosa disponível onde os robôs atiram, o vilão Evil Otto é invencível porém o jogador consegue uma vida extra mais facilmente, com o acúmulo de 1000 pontos.

Níveis de dificuldade

Tendo como referência o manual de instruções do jogo *Berzerk*, nas fases que os robôs atiram, a primeira fase do jogo nenhum robô irá atirar no jogador, todavia ele ainda pode destruir o jogador se tocá-lo, isso ocorre com o objetivo de fazer o jogador se acostumar com o ambiente do jogo, a partir da segunda sala que o jogador entrar os robôs irão atirar, com a velocidade de movimento e de tiro aumentando consecutivamente a cada nova sala, até a 16° sala, onde os robôs voltam a se mover lentamente, aumentando novamente a cada nova sala explorada pelo jogador, fazendo um ciclo de aumento de velocidade consecutiva, mas a velocidade de tiro permanece a mesma do personagem do jogador até o final do jogo. É importante mencionar que o jogador inicia o jogo com três vidas disponíveis podendo ganhar e/ou perder vidas conforme o andamento do jogo e a variação de jogo escolhida.

Áudio

De acordo com (COLECO 2023) *Berzerk* foi um dos primeiros exemplos de síntese de fala em jogos de fliperama, apresentando robôs falantes que ambientavam o jogador de maneira fantástica. Em 1980, a compressão de voz por computador era extremamente cara, estimando-

se que custasse ao fabricante 1.000 dólares por palavra, a versão em inglês tem um vocabulário de trinta palavras.

- “Frango! Lute como um robô!” (se o humanóide sair do labirinto antes que todos os robôs sejam destruídos);
- “Alerta de Intruso! Alerta de Intruso!” (sempre que Evil Otto aparece);
- “O humanóide não deve escapar” (quando o humanóide sai do labirinto depois que todos os robôs são destruídos).

Inteligência Artificial

Outra área em que o *Berzerk* foi inovador foi no setor da inteligência artificial que era avançada para a sua época de lançamento, essa inteligência artificial consistia na programação que os robôs possuíam para perseguir o personagem do jogador, o que podia ocasionar “fogo-amigo” entre os robôs, que seria quando um robô atira em outro robô aliado dele e o elimina, além disso os robôs podiam andar até as paredes eletrificadas, onde também eram eliminados, dando assim uma possibilidade de estratégia ao jogador, que podia usar isso ao seu favor.

3.2 Engenharia de Software

É notável que hoje em dia é indispensável o uso de técnicas de desenvolvimento de *software*, pois é cada vez mais necessário se ter gerenciamento eficaz, produtividade, eficiência no desenvolvimento, redução de custos, inovação e outros benefícios significativos para se chegar a um produto que satisfaça todas as necessidades do cliente, e para se chegar nesses tão almejados benefícios, é imperativo que haja uma adoção de técnicas de desenvolvimento de *software*, feita de maneira correta, obedecendo as suas diretrizes. Explicaremos a seguir o modelo de desenvolvimento de *software* usado nesse trabalho, e entraremos em detalhes sobre sua definição, justificativas de uso, validade, confiabilidade, e resultados esperados.

Modelo Cascata

A técnica escolhida para o desenvolvimento deste jogo foi o modelo cascata, que de acordo com (DIAS 2019) esse modelo foi criado em 1970 por Winston W. Royce e é recomendado

a projetos pequenos por conta da facilidade de implementação e sua estrutura sequencial. A escolha dessa metodologia para o presente trabalho, está ligada as suas principais características que são: abordagem sequencial, fases bem definidas, modelo rígido, impassível a erros e testes rigorosos. A abordagem sequencial exige que se siga um modelo sequencial de desenvolvimento, ou seja, cada fase do desenvolvimento de *software* ocorre em uma ordem linear e rígida, onde uma fase deve ser concluída antes que a próxima possa ser iniciada, o fluxo de desenvolvimento flui de cima para baixo, como uma cascata, por isso o seu nome é assim. Outra característica importante do modelo em cascata é o seu planejamento, pois todo o desenvolvimento se baseia nele então é de extrema importância que o planejamento esteja bem próximo á perfeição, que cada fase esteja muito bem descrita e minuciosamente detalhada, para que todas as engrenagens do desenvolvimento se acoplem perfeitamente, gerando um produto totalmente funcional e com o mínimo de imprevistos possível durante a fase da produção do *software*.

3.3 Arquitetura do Software

O autor (TEIXEIRA 2022) explica que na década de 80 ao se projetar um jogo, existia a necessidde de se basear no *hardware* que as empresas fabricantes de jogos forneciam, devido as limitações tecnológicas da época existia a necessidade de adaptação por parte dos desenvolvedores, no caso específico do *Berzerk*, inicialmente iria ser usado um processador Motorola 6809E mas por conta de problemas com esse componente ele foi substituído pelo Zilog Z80 um microprocessador de 8 bits com 208 bits de memória (ZILOG 2016). Outra prática comum em desenvolvimento de jogos dessa época era o uso da linguagem de programação *Assembly*, por conta do baixo nível dessa linguagem, eles usavam essa linguagem para obter mais precisão e controle sobre o *hardware* e com isso alcançar melhor desempenho e eficiência. Com todas essas dificuldades mencionadas era necessário que a arquitetura que os desenvolvedores utilizavam se baseassem exclusivamente nos *hardwares* que eram utilizados, hoje em dia com as melhorias significativas das tecnologias e das linguagens de programação, memória não é mais considerado um problema, permitindo que os programadores levem outras características em conta na hora de projetar a arquitetura de um *software*, por exemplo o tipo de paradigma de programação

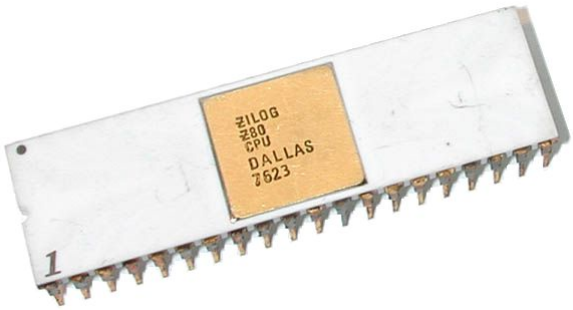

Figura 3.3: Um Z80 fabricado em junho de 1976

mais adequada para aquele fim.

Como não existe mais a necessidade de se desenvolver um *software* em baixo nível, nesse trabalho, será empregada uma arquitetura mais atualizada para a presente época, a arquitetura baseada em componentes a qual será explicada e detalhada a seguir.

3.3.1 Arquitetura Baseada em Componentes

Conforme afirma (FEIJÓ 2007) a arquitetura baseada em componentes é "um paradigma de desenvolvimento de *software* caracterizado pela composição de partes já existentes, ou pela composição de partes desenvolvidas independentemente e que são integradas para atingir um objetivo final." o autor também afirma que o uso desse paradigma aumenta a qualidade do *software* e aumenta a velocidade de desenvolvimento, benefício bastante procurado por desenvolvedores atualmente. A segunda parte da definição de Feijó se enquadrou muito bem no desenvolvimento da remasterização do *Berzerk*, pois bastante código pôde ser reaproveitado. Será demonstrado na modelagem a seguir como o jogo foi projetado.

3.3.2 Modelagem do Sistema

A modelagem do sistema, foi feita usando um diagrama de classes utilizando a ferramenta Lucidchart, feramenta bastante popular de elaboração de UML (*Unified Modeling Language* ou Linguagem de Modelagem Unificada) e também foi utilizado o modelo C4 que consiste em um conjunto hierárquico de diagramas de arquitetura de *software* para contexto, containers, componentes e código.

Diagrama de Classe

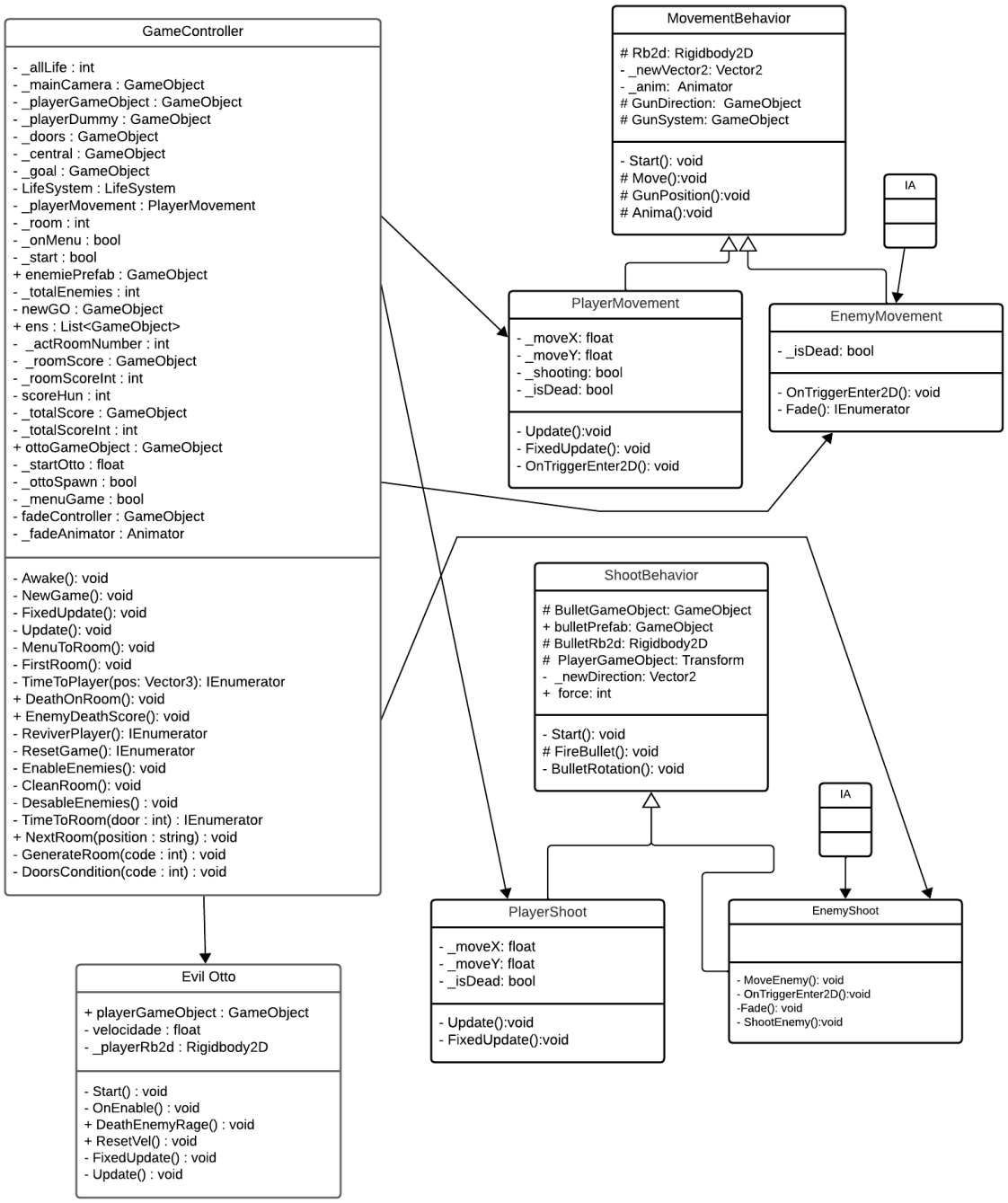

Figura 3.4: Diagrama de Classes do Jogo Berzerk

O diagrama da figura acima mostra a estrutura completa do jogo tendo como superclasses os comportamentos comuns entre o jogador e o inimigo, tratando os comportamentos como componentes que podem ser reaproveitados por ambos os personagens do jogo. Além disso o GameController é responsável por chamar os comportamentos específicos de cada personagem

e chamar o vilão Evil Otto, que tem um comportamento único, por esse motivo não herda nenhum comportamento.

Modelo C4

O Modelo C4 é uma modelagem de fácil entendimento para os vários profissionais da área do desenvolvimento de *software*, esse modelo busca ser claro e prático na demonstração da estrutura de um *software* (Brown 2018; Cipriano 2021) e é para este mesmo fim que o usaremos neste presente trabalho.

Contexto

No 1° nível desse modelo temos o nível do contexto, que tem como objetivo dar uma visão geral do software e de onde ele está inserido, seus relacionamentos e limites, nesse primeiro momento os detalhes não são muito importantes, na imagem a seguir será exposto o nível de contexto do *Berzerk*.

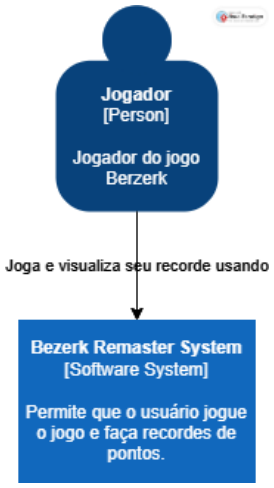

Figura 3.5: Modelo C4 - 1° nível: Contexto

Na figura acima de forma bem resumida, está o contexto do *Berzerk*, onde um jogador pode interagir com o jogo de duas maneiras: jogando e visualizando os recordes de pontuação.

Container

Já no 2° nível, o nível de container tem como função expor a arquitetura do sistema da forma mais simples possível, separando em containers onde cada tecnologia é usada e como elas se relacionam entre si.

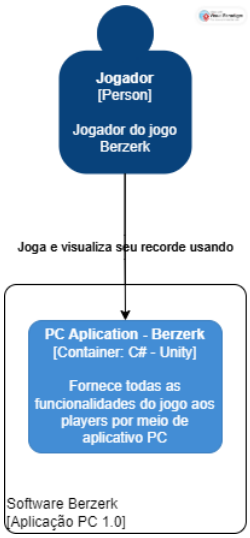

Figura 3.6: Modelo C4 - 2° nível: Container

Na figura acima vemos o container de aplicação do *Berzerk*, como só foi utilizado uma tecnologia então esse modelo está bastante resumido.

Componentes

No 3° nível já se inicia o detalhamento do *software*, onde os containers são expandidos e se tornam componentes com funcionalidades, responsabilidades e tecnologias detalhadas, e é nesse nível que eles são expostos ao leitor.

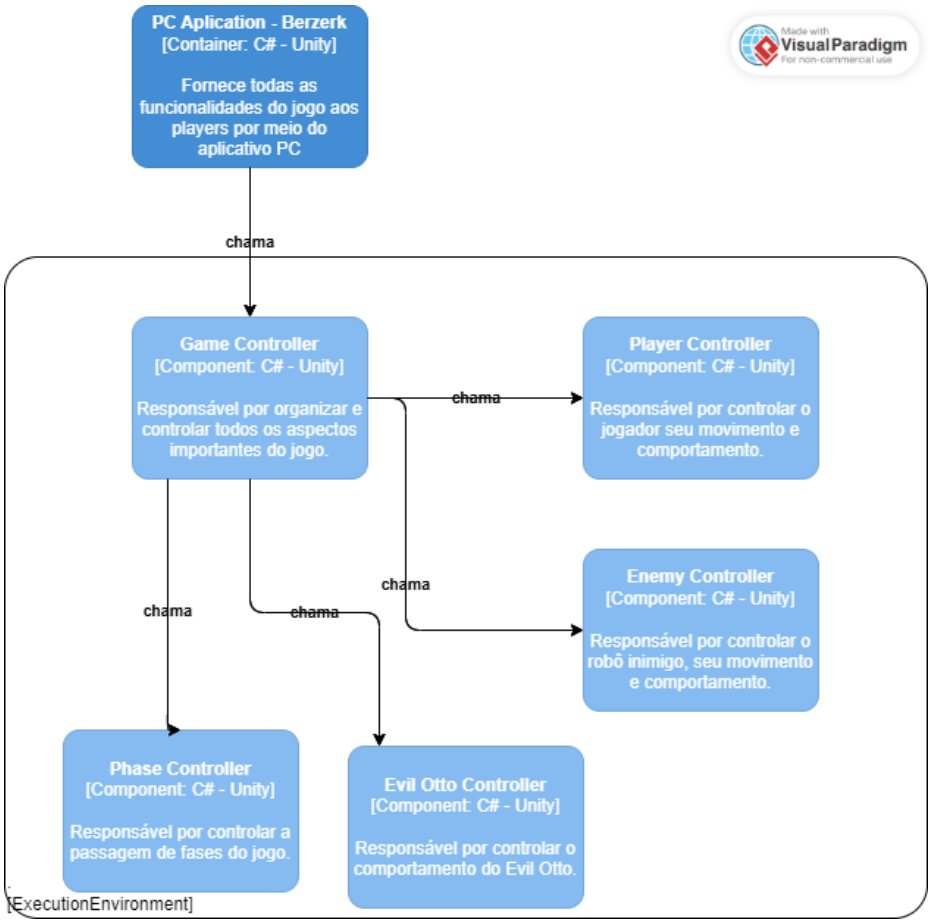

Figura 3.7: Modelo C4 - 3° nível: Componentes

Nesta última figura podemos ver mais profundamente como funciona a aplicação do jogo com os seus componentes sendo especificados e seus relacionamentos expostos, por exemplo na figura acima fica claro que um único componente, o *Game Controller* chama todos os outros componentes importantes do jogo, controlando assim todo o jogo, seu início, os personagens as salas e a pontuação também.

3.3.3 Mecânicas do Jogo

Segundo (MATOS 2020) mecânicas de jogos "são os conjuntos de regras do jogo que permitem determinar as ações possíveis do jogador" e com esse conceito em mãos nessa subseção irá abordar minuciosamente todas as mecânicas que serão implementadas nessa remasterização do *Berzerk*, como já explicado anteriormente, a implementação dessas mecânicas serão realizadas em duas etapas as quais serão descritas a seguir:

Etapa 1: Mecânicas Principais

- Se o jogador pressionar as teclas do teclado “W”, “S”, “A”, “D”, ou as teclas de seta do teclado, superior, inferior, esquerda e direita, ele será capaz de andar em oito direções, para cima, para baixo, para a esquerda e para a direita, essa ação corresponde ao sistema de movimentação do jogador;

- Se o jogador pressionar a tecla “espaço” do teclado, ele será capaz de atirar para a direção que foi o seu último movimento, essa ação corresponde ao sistema de tiro do jogador;

- Se o jogador estiver na tela inicial do jogo e pressionar a tecla “espaço” irá acionar o sistema de geração de salas aleatórias, surgindo na tela a primeira sala;

- Se o jogador tiver acionado o sistema de geração de salas aleatórias, ele acionará ao mesmo tempo o sistema de eletrificação das paredes, que causará a destruição de qualquer personagem que tocar nela exceto o Evil Otto;

- Se o jogador tiver acionado o sistema de geração de salas aleatórias, ele acionará juntamente o sistema de geração de robôs inimigos e o sistema de movimentação será ativado como consequência;

- Se o jogador tiver acionado o sistema de geração de robôs, ele acionará também o sistema de detecção de presença dos robôs inimigos, fazendo os robôs perseguirem e detectarem a presença do jogador caso este esteja no campo de visão do robô que é vertical e horizontal, jamais o robô terá visão na diagonal;

- Caso o jogador ative o sistema de tiro e acerte um robô inimigo, o sistema de detecção de abates será ativado e uma contagem será realizada e irá atualizar em tempo real o sistema de pontuação que será explicitado a seguir;

- Caso o jogador acerte um robô, ou passe de sala eliminando todos os robôs presentes naquela sala, será ativado o sistema de pontuação que será atualizado na tela instantaneamente caso alguma dessas condições mencionadas seja satisfeita;

Etapa 2: Mecânicas Secundárias

- Se o jogador permanecer em uma mesma sala por 30 segundos, será ativado o sistema de surgimento do Evil Otto, vilão que não pode ser derrotado, sendo a única alternativa do jogador fugir da sala em que está;
- A cada 1000 pontos que o jogador acumular será ativado o sistema de vidas extras onde o jogador ganha uma vida extra;
- Caso o jogador, atire, morra, passe de sala, ganhe uma vida e caso um robô atire ou seja destruído será ativado o sistema de som, reproduzindo o som correspondente a algum dos eventos citados;
- Caso o jogador seja eliminado e perca todas as suas vidas extras, o sistema de pontuação será ativado, salvando a melhor pontuação do jogador e irá exibi-lo na tela de *stand-by*;
- Caso o jogador passe de sala, será acionado o sistema de transição de sala.

Capítulo 4

Implementação

Neste capítulo será aprofundado a questão da implementação de *software*, apresentando códigos principais usados na construção do jogo e explicações sobre cada parte deles.

4.1 Algoritmos

Nesta sessão serão detalhados os algoritmos utilizados como base para a estruturação da remasterização do *Berzerk*, como a ferramenta usada para a construção do jogo foi a *Engine Unity*, a linguagem de programação usada para a escrita dos códigos a seguir, foi o *C Sharp*, pois é a linguagem padrão da *Unity*.

4.1.1 Comportamentos Gerais

Para a implementação dos personagens do jogador e dos robôs inimigos, foi constatado que eles tinham comportamentos em comum, os movimentos e os tiros, e visando a componentização e consequentemente o reaproveitamento de código foram implementados comportamentos gerais, que foram chamados dentro das instâncias de cada personagem.

Listing 4.1: Algoritmo Geral de Movimento

```
1   public class MovementBehavior : MonoBehaviour
2   {
3       [...]
4       protected void Move(float moveX, float moveY, int speed, bool
            isShooting, bool isDead)
5       {
6
7           if (!isDead)
8           {
9               if (!isShooting)
10              {
11                  _newVector2 = new Vector2(moveX,moveY).normalized;
12                  Rb2d.velocity = _newVector2 * speed;
13              }
14              else
15              {
16                  _newVector2 = Vector2.zero;
17                  Rb2d.velocity = _newVector2 * speed;
18              }
19          }
20      }
21      [...]
22  }
```

Esse trecho de código é responsável por receber um *game object* (no caso o jogador ou o robô inimigo) e dar a esse *game object* generalista, velocidade de movimento e posição na cena do jogo. Esse código (*MovementBehavior*) também verifica se esse personagem está vivo, caso esteja vivo ele também verifica se o personagem está atirando ou não, caso esteja atirando a velocidade do personagem é zerada, pois os personagens não podem atirar se movendo e se o personagem não tiver atirando ele tem a possibilidade de se mover. E por último dentro do código de *MovementBehavior* fica a função *GunPosition*, onde nessa função é verificado para que direção o personagem está olhando ou se movendo, com a informação para qual direção o personagem está direcionado, a bala sairá caso o personagem atire.

Listing 4.2: Algoritmo Geral de Tiro

```
1   public class ShootBehavior : MonoBehaviour
2   {
3       [...]
4       protected void FireBullet(float moveX, float moveY, bool
            shooting)
5       {
6           if (!shooting || BulletGameObject.activeSelf) return;
7           [...]
8           BulletRotation(moveX,moveY);
9
10          if (moveX == 0 && moveY == 0)
11          {
12              _newDirection = PlayerGameObject.rotation.y == 0 ? new
                    Vector2(force, 0) : new Vector2(-force, 0);
13          }
14          else
15          {
16              _newDirection = new Vector2(force * moveX, force *
                    moveY);
17          }
18
19          BulletGameObject.transform.position = gameObject.transform
                .position;
20          BulletGameObject.SetActive(true);
21          BulletRb2d.AddForce(_newDirection, ForceMode2D.Impulse);
22      }
23      [...]
24  }
```

Esse trecho de código é responsável pelo sistema de instanciamento de bala, ele instancia um *prefab* de bala dependendo do tipo de personagem que vai usufruir do sistema de tiro, seja ele o *player* ou o robô. O interessante desse sistema de tiro é que as balas são sempre "recicladas" pois dentro do jogo original um personagem nunca atira antes de sua bala anterior acertar uma superfície, seja ela uma parede, o jogador ou outro robô, uma bala por vez para cada personagem é garantido pela função *FireBullet*, essa função é importante pois ela ajuda a evitar desperdício de memória.

4.1.2 Humanoide (Jogador)

Nessa sub-seção será explicada as diferenças entre os personagens na questão de seus respectivos comportamentos e o primeiro a ser exposto será o personagem do jogador.

Listing 4.3: Algoritmo de Movimento do Jogador

```
public class PlayerMovement : MovementBehavior
{
    [...]
    private void Update()
    {
        _moveX = Input.GetAxisRaw("Horizontal");
        _moveY = Input.GetAxisRaw("Vertical");
        _shooting = Input.GetButton("Shoot");
        Anima(_moveX,_moveY,_shooting,true,_isDead);
        GunPosition(_moveX,_moveY,_shooting);
    }
    private void FixedUpdate()
    {
        Move(_moveX,_moveY,speed,_shooting,_isDead);
    }
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Enemy") && !_isDead)
        {
            GunSystem.SetActive(false);
            Rb2d.bodyType = RigidbodyType2D.Static;
            _isDead = true;
        }
    }
}
```

Esse trecho de código é encarregado da movimentação do jogador, é possível notar que ele herda o código de movimento geral, mas acrescenta a característica exclusiva do personagem do jogador, que pode escolher quando e pra onde andar, com a captura de direção obtida por meio dos botões do teclado. A função *OnTriggerEnter2D* é responsável por detectar se o jogador colidir com algum objeto que tenha a marcação (*tag*) de inimigo, seja um robô inimigo ou uma parede, se ele tocar algum destes o jogador morre.

Listing 4.4: Algoritmo de Tiro do Jogador

```
public class PlayerShoot : ShootBehavior
{
    [...]
    private void Update()
    {
        _moveX = Input.GetAxisRaw("Horizontal");
        _moveY = Input.GetAxisRaw("Vertical");
        _shooting = Input.GetButton("Shoot");
    }
    private void FixedUpdate()
    {
        FireBullet(_moveX,_moveY,_shooting);
    }
}
```

A classe de tiro do jogador herda o comportamento geral de tiro e adiciona a singularidade do jogador, que é a detecção das teclas do teclado de movimentação para saber pra qual direção o jogador está olhando e atirar.

4.1.3 Automazeons (Robôs Inimigos)

Agora será explicado mais detalhadamente o funcionamento do código dos robôs inimigos.

Listing 4.5: Algoritmo de Movimento dos Robôs Inimigos

```
public class EnemyMovement : MovementBehavior
{[...]
    private void Update(){
        Anima(0,0,false,true,_isDead);}
    private void OnTriggerEnter2D(Collider2D other) {
        if ((other.CompareTag("Player") && !_isDead) || (other.
           CompareTag("Bullet") && !_isDead) || (other.CompareTag(
           "Enemy") && !_isDead)){
            Rb2d.bodyType = RigidbodyType2D.Static;
            _isDead = true;
            StartCoroutine(Fade());
        }}
    private IEnumerator Fade(){
        yield return new WaitForSeconds(0.3f);
        gameObject.SetActive(false);
    }}
```

O código acima é responsável por garantir que se o robô inimigo estiver vivo e encoste em outro robô, em uma bala qualquer, no próprio *player* ou qualquer objeto que tenha a *tag* de inimigo, o robô fica estático, recebe a classificação de "morto" e aciona a animação de explosão e após a animação se encerrar o *game object* do inimigo é desativado.

Listing 4.6: Algoritmo de Tiro dos Robôs Inimigos

```
1  public class EnemyShoot : ShootBehavior
2  {
3      public void ShootEnemy(float moveX, float moveY,bool shoot,
           int diff)
4      {
5          FireBullet(moveX,moveY,shoot,diff,false);
6      }
7  }
```

O código acima é responsável pela ignição dos tiros dos robôs, o código está bastante reduzido e limpo pois é invocada a herança do comportamento geral de tiro, essa função só faz o robô atirar, a decisão de quando e em que direção o robô deve atirar, está na classe que será aprofundada a seguir.

Listing 4.7: Algoritmo de IA dos Robôs Inimigos

```
public class IaSystem : MonoBehaviour
    {[...]
        private void Start(){
            _enemyMovement = gameObject.GetComponent<EnemyMovement>();
            _enemyShoot = gameObject.transform.GetChild(0).
               GetComponent<EnemyShoot>();}

        private void OnEnable(){
            _playerGameObject = GameObject.Find("Player");}

        private void Update(){
            if (!_action){
                _enemyMovement.AnimaEnemy(0, 0, false);}
            if (!_iaCooldown){
                StartCoroutine(Cooldown()); }
            Ia();}}
        private void Ia(){
        [...]
        }
        private IEnumerator Cooldown(){
        [...]
        }
```

O código exposto acima é o encarregado de acionar a "vida" dos robôs inimigos, onde na função *Start()* aciona os sistemas de movimento e tiro, após isso na função *OnEnable()* é onde o robô marca o seu alvo, o *Player*. Na função *Update()* ocorre uma verificação: se existe movimentação do robô na direção do jogador ou se é necessário que a função *Cooldown* seja ativa, essa função *Cooldown* serve para determinar o tempo de reação do robô, para que o jogador tente escapar, esse tempo de reação diminui ao longo das fases o que adiciona a dificuldade crescente do jogo, após essa verificação é invocada a função *IA()* que é responsável pela decisão do robô em questão de qual direção ele irá tomar, qual rota ele usará para chegar até seu objetivo de eliminar o *player*. É importante informar que tanto a função de *IA()* como a função de *Cooldown()* possuem bastante linhas de código, e para a simplificação do entendimento geral da implementação essas funções foram omitidas da demonstração acima.

4.1.4 Evil Otto

Já o vilão Evil Otto tem comportamento e animações um pouco diferente dos outros personagens então se fez necessário um algoritmo geral para o seu comportamento e ele será mostrado a seguir.

Listing 4.8: Algoritmo de comportamento do Evil Otto

```
public class OttoBehavior : MonoBehaviour
    {[...]
        private void Start(){
            _playerRb2d = playerGameObject.GetComponent<Rigidbody2D>()
               ;
        }

        private void OnEnable(){
            gameObject.tag = "Otto";

        }

        public void DeathEnemyRage(){
            velocidade += 0.5f;
        }

        public void ResetVel(){
            velocidade = 0.5f;
        }

        private void FixedUpdate(){
            Vector3 direcao = playerGameObject.transform.position -
               transform.position;
            direcao.Normalize();
            transform.Translate(direcao * (velocidade * Time.deltaTime
               ));
        }

        private void Update(){
            if(_playerRb2d.bodyType == RigidbodyType2D.Static &&
               gameObject.CompareTag("Otto") )
            {
                gameObject.tag = "Untagged";
            }
        }
    }
```

De início o Otto pega a posição *game object* do *player* para saber qual direção ele deve ir

com a função *Start()*, já a função *DeathEnemyRage()* aumenta a velocidade do vilão conforme for chamado no *GameController*, no *ResetVel()*, quando o *player* consegue passar por uma porta essa função é responsável por parar o ganho de velocidade do Evil Otto, que ele ganhou na função *DeathEnemyRage()*, já na função *FixedUpdate()*, o *boss* se direciona para a posição do *Game Object* do *player*, e finalmente na função *Update()* é verificado se o Otto alcançou seu objetivo de eliminar o *player*, caso consiga realizar seu objetivo essa função para o movimento do Evil Otto.

4.1.5 Game Controller

O *Game Controller* é a principal classe desse sistema pois nele se encontra a lógica de invocação de todos os outros códigos que já foram citados anteriormente, ou seja no *Game Controller* começa tudo, e como ele também é um código extenso ele será detalhado por partes.

Listing 4.9: Função NewGame() do Game Controller

```
public class GameController : MonoBehaviour
    {[...]
        private void NewGame()
        {
            _actRoomNumber = 1;
            _mainCamera.GetComponent<Camera>().backgroundColor = Color
               .blue;
            _playerMovement.Revive();
            _allLife = 3;
            _onMenu = true;
            DesableEnemies();
            _playerGameObject.SetActive(false);
            _playerDummy.SetActive(true);
            _playerGameObject.transform.position = new Vector3(-7.5f,
               0.5f,0f);
            _playerDummy.transform.position = new Vector3(-7.5f, 0.5f
               ,0f);
            _room = Random.Range(0,4);
            DoorsCondition(0);
            GenerateRoom(_room);
        }
        [...]
    }
```

A função *NewGame()* é responsável pelas configurações iniciais do jogo na seguinte ordem: assim que o jogo é aberto, é gerado uma tela de *stand-by* (uma tela de espera) que aguarda uma interação do jogador para iniciar o jogo nessa tela de espera o movimento do jogador é restringido, a quantidade de vidas é especificada em três, para serem usadas quando o jogo for iniciado, o tipo de mapa também é definido nessa função, o tipo definido como primeiro mapa do jogo é um mapa que tenha portas abertas na parte superior e inferior e fechadas nos lados esquerdo e direito, essa configuração é definida nessa função, e por fim após definido os tipos de portas abertas e fechadas o tipo de labirinto da parte de dentro do mapa também é definido de maneira aleatória entre cinco tipos de labirintos pré-definidos.

Listing 4.10: Funções FixedUpdate() Update() e MenuToRoom() do Game Controller

```
public class GameController : MonoBehaviour
    {[...]
        private void FixedUpdate(){
            if (!_menuGame && !_ottoSpawn){
                if (Time.fixedTime > (_startOtto+30f))
                {
                ottoGameObject.SetActive(true);
                    _ottoSpawn = true;
                }
            }
        }

        private void Update(){
            if (_onMenu){
                _start = Input.GetButtonDown("Shoot");
            }
            if (_onMenu && _start){
                _fadeAnimator.Play("Fade1");
                MenuToRoom();
            }

        }

        private void MenuToRoom(){
            if (_onMenu){
                _menuGame = false;
                _roomScoreInt = 0;
                _totalScoreInt = 0;
                _roomScore.GetComponent<TMPro.TextMeshProUGUI>().text
                   = Convert.ToString(_roomScoreInt);
                _totalScore.GetComponent<TMPro.TextMeshProUGUI>().text
                    = Convert.ToString(_totalScoreInt);
                _onMenu = false;
                FirstRoom();
            }
        }
        [...]
    }
```

FixedUpdated() é a função que conta 30 segundos a partir do momento que o jogador entra em uma sala para que caso o *player* permaneça na sala por esse tempo o Evil Otto seja *spawnado* na sala para perseguir o *player*.

Já na função *Update()* fica aguardando a interação do usuário especificamente com a tecla

"espaço" do teclado e se a tecla for pressionada a função aciona a animação (*fade*) de mudança de tela e chama a função *MenuToRoom()* que será explicada a seguir.

A função *MenuToRoom()* verifica as pontuações da tela e zera essas pontuações para garantir que o jogo se inicie do zero, após isso desativa a tela de *stand-by* e chama o primeiro labirinto.

Listing 4.11: Funções FirstRoom() TimeToRoom() e TimeToPlayer() do Game Controller

```
1         private void FirstRoom()
2         {
3             [...]
4             _startOtto = Time.fixedTime;
5             _playerMovement.Revive();
6             StartCoroutine(TimeToPlayer(new Vector3(-7.5f, 0.5f, 0f)))
                  ;
7             ottoGameObject.transform.position = (new Vector3(-7.5f,
                  0.5f, 0f));
8             StartCoroutine(TimeToRoom());
9             _mainCamera.GetComponent<Camera>().backgroundColor = Color
                  .black;
10        }
11        private IEnumerator TimeToRoom()
12        {
13            yield return new WaitForSeconds(0.9f);
14            _room = Random.Range(0,4);
15            DoorsCondition(0);
16            GenerateRoom(_room);
17        }
18        private IEnumerator TimeToPlayer(Vector3 pos)
19        {
20            _playerDummy.transform.position = _playerGameObject.
                  transform.position;
21            _playerGameObject.SetActive(false);
22            _playerDummy.SetActive(true);
23            yield return new WaitForSeconds(0.5f);
24
25            DesableEnemies();
26            yield return new WaitForSeconds(0.5f);
27            _playerDummy.SetActive(false);
28            _playerGameObject.transform.position = pos;
29            _playerGameObject.SetActive(true);
30            EnableEnemies();
31        }
```

No final da função *MenuToRoom()* é chamada a função *FirstRoom()* que configura a

primeira sala, iniciando com a contagem para o Evil Otto surgir, liberar o jogador para se movimentar, chamar a função *TimeToPlayer()* que prepara todo o labirinto para poder ser jogado, preparação que consiste em desativar os inimigos enquanto a transição de sala estiver ocorrendo, posicionar o *player* na posição inicial correta dentro do labirinto e após isso ativar os inimigos, depois de sair da função *TimeToPlayer()* a função *FirstRoom()* chama a função *TimeToRoom()* que gera a sala aleatoriamente após um curto período tempo em que aguarda a animação da morte do *player*, pois é chamada só quando ocorre esse evento.

Listing 4.12: Funções DeathOnRoom() e EnemyDeathScore do Game Controller

```
1         public void DeathOnRoom(){
2             if (_allLife > 0)
3             {
4                 _fadeAnimator.Play("Fade2");
5                 _allLife = _allLife - 1;
6                 _actRoomNumber = _actRoomNumber+1;
7
8                 LifeSystem.LossLife(_allLife);
9
10                StartCoroutine(ReviverPlayer());
11                return;
12            }
13            if (_allLife == 0)
14            {
15                _menuGame = true;
16                StopAllCoroutines();
17                StartCoroutine(ResetGame());
18            }
19
20        }
21
22        public void EnemyDeathScore(){
23            ottoGameObject.GetComponent<OttoBehavior>().DeathEnemyRage
                  ();
24            _roomScoreInt = _roomScoreInt + 50;
25            _roomScore.GetComponent<TextMeshProUGUI>().text = Convert.
                  ToString(_roomScoreInt);
26
27            if (scoreHun < (_roomScoreInt))
28            {
29                scoreHun += 1000;
30                _allLife += 1;
31                LifeSystem.GainLife(_allLife);
32            }
33            _roomScore.SetActive(true);
34        }
```

Na função *DeathOnRoom()* é chamada quando o *player* perde uma vida ele verifica se o jogador ainda possui vidas, se ele ainda possuir vidas ele chama o *fade*, decrementa uma vida do total de vidas disponíveis, chama a próxima fase, chama a função de perda de vidas e ressuscita o *player* para a próxima fase, caso o jogador não possua mais vidas disponíveis essa função chama o menu, que seria a tela de *stand-by*, desativa todos os inimigos e reseta o jogo.

Já a função *EnemyDeathScore()* é chamada sempre que um robô é abatido, quando isso ocorre, a velocidade do Evil Otto é aumentada mesmo que o vilão ainda não esteja aparecendo no jogo, pois a velocidade do *boss* aumenta a medida que os robôs morrem, após isso o jogador ganha os 50 pontos e esse ganho já aparece na *hud* do jogador, e por último ocorre a verificação se esse ganho de pontos já alcançou a meta de 1000 pontos, caso tenha alcançado o teto da meta é aumentada em 1000 e o jogador ganha uma vida com o acionamento do sistema de ganho de vida.

Listing 4.13: Funções ReviverPlayer() ResetGame() e EnableEnemies() do Game Controller

```
1
2         private IEnumerator ReviverPlayer()
3         {
4             yield return new WaitForSeconds(2f);
5             FirstRoom();
6         }
7
8         private IEnumerator ResetGame()
9         {
10            yield return new WaitForSeconds(2f);
11            _fadeAnimator.Play("Fade1");
12            yield return new WaitForSeconds(1f);
13
14            NewGame();
15
16        }
17
18        private void EnableEnemies()
19        {
20            _startOtto = Time.fixedTime;
21
22            _totalEnemies = Random.Range(4, 10);
23            enemiePrefab.GetComponent<IaSystem>().difficult =
                  _actRoomNumber;
24
25            for (int i = 0; i < _totalEnemies; i++)
26            {
27                ens.Add(Instantiate(enemiePrefab, new Vector3(Random.
                      Range(-7.5f, 7.5f), Random.Range(-3.4f, 3.4f),0),
                      Quaternion.identity));
28            }
29
30        }
```

A função *ReviverPlayer()* quando chamada aguarda um tempo para aguardar a animação de morte e após isso chama a função *FirstRoom()* que já foi detalhada anteriormente.

ResetGame() é chamada quando a vida do *player acaba*, quando isso ocorre é aguardado 2 segundos para o *fade* acontecer e ir para a tela de *stand-by* e por fim aguarda mais 1 segundo para chamar a função de *NewGame()* que já foi detalhada no início dessa sessão.

A função *EnableEnemies()* quando chamada no início de cada nova sala, inicia a contagem de tempo de 30 segundos para o vilão Evil Otto surgir, após isso é escolhido o número de robôs daquela sala aleatoriamente, entre 4 e 10 robôs inimigos e suas respectivas *IAs* serão instanciadas para aquela nova sala e as posições desses robôs dentro da sala também são escolhidas aleatoriamente.

Listing 4.14: Funções CleanRoom() e DesableEnemies() do Game Controller

```
1         private void CleanRoom()
2         {
3             var Enemy = GameObject.FindGameObjectsWithTag("Enemy");
4             if (Enemy.Length == 0)
5             {
6                 var saveScore = _roomScoreInt;
7                 _roomScoreInt += (_totalEnemies*10);
8                 _roomScore.GetComponent<TextMeshProUGUI>().text =
                      Convert.ToString(_roomScoreInt);
9
10                if (scoreHun < _roomScoreInt)
11                {
12                    scoreHun += 1000;
13                    _allLife += 1;
14
15                    LifeSystem.GainLife(_allLife);
16                }
17            }
18        }
19
20        private void DesableEnemies()
21        {
22            _ottoSpawn = false;
23            ottoGameObject.GetComponent<OttoBehavior>().ResetVel();
24            ottoGameObject.SetActive(false);
25            foreach (var t in ens)
26            {
27                Destroy(t);
28            }
29
30            var bullet = GameObject.FindGameObjectsWithTag("
                  BulletEnemy");
31            if (bullet == null) return;
32            else
33            {
34                for (int i = 0; i < bullet.Length; i++)
35                {
36                    Destroy(bullet[i]);
37                }
38            }
39        }
```

A função *CleanRoom()* pega todos os inimigos ativos na fase pela *tag enemy* (inimigo) e verifica se o valor é igual a zero, se essa verificação der verdadeiro significa que *player* matou

todos os robôs inimigos da sala gerando assim pontuação extra, caso essa verificação gerar valor falso, ou seja o *player* não matou todos os robôs inimigos, então a função para.

A função *DesableEnemies()* desativa o vilão Evil Otto, reinicia a velocidade do Otto e desativa-o. Após isso destrói todos os inimigos e as suas balas.

Listing 4.15: Funções TimeToRoom() e NextRoom() do Game Controller

```
1
2  private IEnumerator TimeToRoom ( int door )
3      {
4          yield return new WaitForSeconds (1f);
5          _room = Random . Range (0,4) ;
6          GenerateRoom ( _room );
7          DoorsCondition ( door );
8      }
9  public void NextRoom ( string position )
10     {
11         CleanRoom () ;
12         _startOtto = Time.fixedTime ;
13         _actRoomNumber = _actRoomNumber + 1;
14         DesableEnemies () ;
15         if ( position == "Up" ){
16             _fadeAnimator.Play ("Fade4") ;
17             StartCoroutine(TimeToRoom(1));
18             StartCoroutine(TimeToPlayer( new Vector3 (0f, -2.5f, 0)));
19             ottoGameObject.transform.position = (new Vector3 (0f ,-2.5
                   f , 0));
20         }
21         if ( position == "Down"){
22             _fadeAnimator.Play ("Fade3") ;
23             StartCoroutine(TimeToRoom (2));
24             StartCoroutine(TimeToPlayer( new Vector3 (0f, 3.5 f, 0)));
25             ottoGameObject.transform.position = (new Vector3 (0f,3.5f
                   ,0));
26         }
27         if ( position == "Left"){
28             _fadeAnimator.Play ("Fade1") ;
29             StartCoroutine(TimeToRoom(3));
30             StartCoroutine(TimeToPlayer( new Vector3 (7.75 f , 0.5 f
                   ,0) ) );
31             ottoGameObject.transform.position = (new Vector3 (7.75f,
                   0.5f , 0));
32             }
33         if ( position == "Right"){
34             _fadeAnimator.Play ("Fade1") ;
35             StartCoroutine(TimeToRoom (4));
36             StartCoroutine(TimeToPlayer ( new Vector3 (-7.75f, 0.5f,0)
                   ));
37             ottoGameObject.transform.position = ( new Vector3 (-7.75f,
                    0.5f, 0));
38          }
39 }
```

TimeToRoom() é responsável por escolher todas as salas que o jogador irá jogar, escolhendo aleatoriamente a próxima sala por meio de cinco tipos de salas pré-especificadas e por meio da definição das portas abertas e fechadas.

Sempre que o jogador passa de sala a função *NextRoom()* é chamada, onde limpa a sala, dá largada no tempo de 30 segundos para o Evil Otto aparecer naquela sala, aumenta o nível de dificuldade da sala e desabilita os próximos inimigos até acabar o *fade* de transição de sala, após isso é feita uma verificação de qual porta o jogador passou (de cima, de baixo, da esquerda ou da direita), dependendo de qual porta o jogador passar é acionado um tipo *fade* diferente, as portas abertas e fechadas mudam e a posição inicial do jogador na próxima fase também muda e consequentemente muda a posição de *spawn* do Evil Otto, pois o Evil Otto sempre surge em uma sala da posição que o jogador também surgiu.

Listing 4.16: Função GenerateRoom() do Game Controller

```
1
2  private void GenerateRoom(int code){
3
4      if (code == 0){
5          _central.transform.GetChild(0).gameObject.SetActive(false);
6          _central.transform.GetChild(1).gameObject.SetActive(false);
7          _central.transform.GetChild(2).gameObject.SetActive(false);
8          _central.transform.GetChild(3).gameObject.SetActive(false);
9          _central.transform.GetChild(4).gameObject.SetActive(false);
10         _central.transform.GetChild(5).gameObject.SetActive(false);
11         _central.transform.GetChild(6).gameObject.SetActive(false);
12         _central.transform.GetChild(7).gameObject.SetActive(true);
13         _central.transform.GetChild(8).gameObject.SetActive(true);
14         _central.transform.GetChild(9).gameObject.SetActive(true);
15         _central.transform.GetChild(10).gameObject.SetActive(true);
16         }
17         [...]
18     }
```

A função *GenerateRoom(code)* cuida da construção dos labirintos, dependendo do código aleatório que ele recebe de parâmetro das funções *NewGame()* e *TimeToRoom()* é gerado um tipo de labirinto diferente.

Listing 4.17: Função DoorsCondition() do Game Controller

```
1
2     private void DoorsCondition(int code){
3         if (code == 0) //start
4         {
5             _doors.transform.GetChild(0).gameObject.SetActive(false);
6             _goal.transform.GetChild(0).gameObject.SetActive(true);
7             _doors.transform.GetChild(1).gameObject.SetActive(false);
8              _goal.transform.GetChild(1).gameObject.SetActive(true);
9             _doors.transform.GetChild(2).gameObject.SetActive(true);
10            _goal.transform.GetChild(2).gameObject.SetActive(false);
11
12            _doors.transform.GetChild(3).gameObject.SetActive(true);
13            _goal.transform.GetChild(3).gameObject.SetActive(false);
14        }
15        [...]
16    }
```

A função *GererateRoom()* citada anteriormente cuida da parte interna da sala, os labirintos, já a função *DoorsCondition()* cuida das portas e define quais serão abertas e fechadas, baseado no parâmetro específico que recebe, ou seja não é aleatório, pois é preciso garantir que a porta que o jogador atravessou esteja fechada na próxima fase.

4.2 Sistema de Som

O sistema de som foi construído usando a ferramenta FMOD, motivado pela necessidade de ajustar certos aspectos dos sons originais, foi necessário passar por ajustes dentro da ferramenta como pico de som em efeitos de som utilizados no jogo e os passos para essa construção do sistema de som será detalhada a seguir.

4.2.1 Configuração no FMOD

Foram catalogados cinco tipos diferentes de efeitos de som no jogo original.

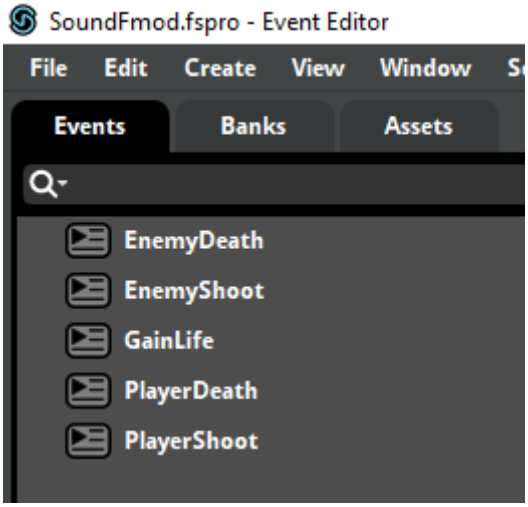

Figura 4.1: Efeitos sonoros utilizados no Berzerk

- *EnemyDeath* (Morte do Inimigo): Efeito de som que surge após o abate de um robô inimigo;
- *EnemyShoot* (Tiro do Inimigo): Efeito de som que surge após o robô inimigo atirar;
- *GainLife* (Ganho de Vida): Efeito de som que surge após o jogador ganhar uma vida extra;
- *PlayerDeath* (Morte do Jogador): Efeito de som que surge após o abate jogador;
- *PlayerShoot* (Tiro do Jogador): Efeito de som que surge após o jogador atirar;

O *SFX* de morte do inimigo também é usado em algumas outras situações:

- Quando o jogador passa de sala;
- Quando o jogador perde todas as vidas e vai pra tela de *stand-by*;
- Assim que o jogo é iniciado também é acionado o *SFX* de *EnemyDeath*;
- Quando o jogador está na tela de *stand-by* e aperta para iniciar o jogo esse *SFX* também é reproduzido.

Esse reaproveitamento do efeito de som de morte do inimigo também se encontra no jogo original, sendo apenas replicado neste trabalho.

Após a inserção dos *assets* de áudio se fez necessário a alteração do equalizador de um *SFX* específico: o *PlayerDeath*, pois o mesmo possuía um pico, o que trazia um certo desconforto auditivo, a mudança que foi realizada no equalizador será exibida a seguir.

Figura 4.2: Configuração Final do Equalizador do SFX de PlayerDeath

A definição de equalizador é "um equalizador permite que o som em determinadas faixas de frequência seja amplificado ou reduzido, a fim de ajustar a qualidade e caráter do som" (YAMAHA ?). A frequência sonora quanto menos ecoa mais baixas são as frequências e quanto mais ecoam, mais altas e com isso vem o conceito de frequências baixas (*low*), médias (*mid*) e agudas (*high*) (Borba 2012). Por conta da necessidade de melhorar a qualidade de som foi utilizado o equalizador do FMOD dentro das configurações do efeito sonoro PlayerDeath, para diminuir a frequência estrondosa do som original.

O volume do efeito de som foi reduzido de 0.00 dB para -0.50 db, o que fez uma sutil melhoria no choque inicial que esse som causava, mas o que realmente aumentou a qualidade do som foi a redução do médio e do agudo do som, mais especificamente a redução foi de 0.00 dB para -1.00 dB no baixo, de 0.00 dB para -7.50 db no médio e 0.00 dB para -8.00 dB no agudo, com essa redução o som ficou menos tumultuoso e se estabeleceu num volume similar aos outros efeitos sonoros utilizados no jogo, para que todos os *SFX* ficassem harmoniosos quando executados juntos.

4.2.2 Implementação do Sistema de Som

A ferramenta FMOD facilitou bastante a implementação propriamente dita dos efeitos de som dentro da Unity, bastando chamar o *SFX* a escolha dentro de partes específicas do código e o efeito sonoro já estará funcionando conforme foi configurado.

Os cinco efeitos sonoros foram implementados da seguinte maneira:

Listing 4.18: Implementação do SFX de Morte do Inimigo pt.1

```
1  [...]
2      private void OnTriggerEnter2D(Collider2D other)
3          {
4              if (_isDead) return;
5              if (other.CompareTag("Player") || other.CompareTag("
                   BulletPlayer") || other.CompareTag("BulletEnemy")||
                   other.CompareTag("Enemy") ||other.CompareTag("Wall")||
                   other.CompareTag("Otto"))
6              {                FMODUnity.RuntimeManager.PlayOneShot("
                   event:/EnemyDeath");
7                  Rb2d.bodyType = RigidbodyType2D.Static;
8                  _isDead = true;
9                  GameObject.Find("GameController").GetComponent<
                       GameController>().EnemyDeathScore();
10                 StartCoroutine(Fade());
11             }
12         }
13 [...]
```

O efeito de som da morte do inimigo, é usada para mais de um fim, e o código acima se refere a função *OnTriggerEnter2D()* da classe de *EnemyMovement* que detecta a colisão do robô inimigo com qualquer outro corpo se for detectado colisão com algum objeto que esteja com as *tags* de jogador, de robô, bala do jogador, bala de robô inimigo, de parede ou *tag* do Evil Otto, se houver colisão, aquele robô é abatido e o efeito de som é reproduzido por meio do método *PlayOneShot* que é usado para reproduzir um som somente uma vez, sem repetições, esse método vem da classe *FMODUnity.RuntimeManager* essa classe disponibiliza métodos para controlar e reproduzir eventos de áudio no jogo.

Listing 4.19: Implementação do SFX de Morte do Inimigo pt.2

```
[...]
        private void NewGame()
        {
            FMODUnity.RuntimeManager.PlayOneShot("event:/EnemyDeath");
            [...]
        }
        [...]
         private void FirstRoom()
        {
            FMODUnity.RuntimeManager.PlayOneShot("event:/EnemyDeath");
            [...]
        }
        [...]
         public void NextRoom(string position)
        {
            FMODUnity.RuntimeManager.PlayOneShot("event:/EnemyDeath");
            [...]
        }
[...]
```

Além de sua principal motivação o *SFX* de morte do inimigo também é utilizado logo quando se inicia o jogo e no inicio das fases e por essa razão esse efeito sonoro é reproduzido logo no início das funções de *NewGame()*, *FirstRoom()* e *NextRoom()*.

Listing 4.20: Implementação do SFX de Tiro do Inimigo e Tiro do Jogador

```
[...]
    protected void FireBullet(float moveX, float moveY, bool shooting,
        int force , bool player){
        [...]
        if (player){
            FMODUnity.RuntimeManager.PlayOneShot("event:/PlayerShoot")
               ;
        }

        if (!player){
            FMODUnity.RuntimeManager.PlayOneShot("event:/EnemyShoot");
        }
        [...]
    }
[...]
```

Por conta do uso do Comportamento do Tiro ser compartilhado pelos personagens jogador e robô inimigo a implementação do efeito de som de tiro de ambos foi feita em um só lugar, na classe *ShootBehavior* e na função *FireBullet()* onde é feita a verificação se é o jogador que está atirando ou se é o robô inimigo, qualquer que seja o personagem atirando, seu respectivo efeito de som será reproduzido.

Listing 4.21: Implementação do SFX de Morte do Jogador

```
1  [...]
2      private void OnTriggerEnter2D(Collider2D other)
3          {
4              if ((other.CompareTag("BulletEnemy") && !_isDead) || (
                   other.CompareTag("Enemy")&& !_isDead) || ( other.
                   CompareTag("Wall")&& !_isDead) || ( other.CompareTag("
                   Otto")&& !_isDead))
5              {
6                  FMODUnity.RuntimeManager.PlayOneShot("event:/
                       PlayerDeath");
7                  [...]
8              }
9          }
10 [...]
```

Já para o efeito de som de morte do jogador é realizada uma verificação parecida com a feita na função de morte do robô inimigo, onde é verificado se o jogador teve contato com algo que pode lhe abater como a bala dos robôs inimigos, o corpo de um robô inimigo, uma parede ou o Evil Otto, caso o jogador tenha tido contato, é reproduzido o som de morte do *player*.

Listing 4.22: Implementação do SFX de Ganho de Vida

```
[...]
    public void EnemyDeathScore(){
            [...]
            if (scoreHun < (_roomScoreInt))
            {
                FMODUnity.RuntimeManager.PlayOneShot("event:/GainLife");
                [...]
            }
            [...]
        }
        [...]
    private void CleanRoom()
        {
            [...]
            if (Enemy.Length == 0)
            {
                var saveScore = _roomScoreInt;
                _roomScoreInt += (_totalEnemies*10);

                _roomScore.GetComponent<TextMeshProUGUI>().text = Convert.ToString(_roomScoreInt);

                if (scoreHun <= _roomScoreInt)
                {
                    scoreHun += 1000;
                    _allLife += 1;
                    FMODUnity.RuntimeManager.PlayOneShot("event:/GainLife");
                    LifeSystem.GainLife(_allLife);
                }
            }
        }
[...]
```

O efeito de som de ganho de vida pode ser chamado em dois momentos do jogo: na função *EnemyDeathScore()* quando alcança 1000 pontos por abater robôs, e na função *CleanRoom()* quando a pontuação atual do jogador somada com a pontuação extra concedida ao passar de sala eliminando todos os robôs inimigos também soma 1000 pontos ou mais, por isso esse *SFX* é chamado em duas funções diferentes.

4.3 Testes

Para verificar a qualidade do jogo desenvolvido foi realizado testes de jogabilidade durante toda a fase de implementação, afim de verificar se todas as funcionalidade disponibilizadas no jogo estariam operacionais e livres de erros, esse teste foi de muita importância pois com ele foi detectado o desbalanceamento da velocidade de movimento dos robôs inimigos gerados pela *IA* que os controlava como também a velocidade do tiro dos robôs que era alta, o que dificultava a jogabilidade, com essa descoberta foi realizado os devidos ajustes o que deixou o jogo mais próximo do seu objetivo de ser parecido com o *Berzerk* original.

Capítulo 5

Resultados Finais

Como já argumentado anteriormente, as remasterizações tem grande importância para a preservação cultural de jogos clássicos para a posteridade e tendo isso em vista, todo o presente trabalho se voltou em desenvolver uma versão atualizada do jogo *Berzerk* em questões de tecnologias empregadas em seu desenvolvimento, para que ao final do processo de implementação fosse obtido um *software* de qualidade técnica superior ao *software* original, o que felizmente foi alcançado com sucesso ao final da elaboração deste trabalho.

Como resultado final da implementação, foi gerado um jogo que atendeu satisfatoriamente os resultados desejados que foram especificados na sessão de objetivos do presente trabalho, sendo eles: mecânicas principais e secundárias implementadas, testadas e funcionais, efeitos sonoros implementados, design e animações da época, que juntos formaram uma fiel adaptação do jogo primordial.

O mais desafiador na elaboração deste trabalho com certeza foi a parte de implementação das mecânicas do jogo, em duas partes específicas, primeiro na parte de balanceamento de dificuldade crescente do jogo, mais especificamente foi dificultoso encontrar o equilíbrio da velocidade que os robôs inimigos ganham progressivamente com o passar das fases, e a segunda dificuldade se encontrou na parte de adaptação de algumas mecânicas do jogo, como por exemplo o tempo de surgimento do vilão Evil Otto, por mais que o jogo original tenha manual de instruções que foi muito útil para o entendimento de vários aspectos do jogo, o manual não foi muito específico como funcionava o surgimento desse *boss* o que dificultou essa implementação.

A modernização e inovação que estão presentes nesse trabalho, vieram por meio do uso de ferramentas modernas e robustas como as ferramentas *Unity* e FMOD, ambas facilitaram a implementação com suas funcionalidades dedicadas a simplificação de todo o processo de desenvolvimento de jogos, a presença delas foi essencial para o sucesso dessa missão de remasterização.

E por fim os trabalhos futuros podem aprimorar a parte de *assets* visuais do jogo, pois os *sprites* do jogo do *Berzerk* da Atari não foram encontrados de forma usual, então foram utilizadas *sprites* do jogo *Berzerk* da sua versão do *arcade*.

Referências Bibliográficas

ATARI. *Atari Game Program Instructions*. 1982. <https://www.gamesdatabase.org/Media/SYSTEM/Atari_2600/Manual/formated/Berzerk_-_1982_-_Atari.pdf>.

AUSTIN, H. J.; SLOAN, R. Through the shanzhai lens: Reframing the transmedial copying and remaking of games. *British Journal of Chinese Studies*, v. 12, n. 2, p. 133–153, 2022.

BORBA, A. *O que é um equalizador?* 2012. <https://www.supergospel.com.br/dicas-de-audio-uma-palavra-sobre-equalizacao-e-equalizadores-parte-1_3313.html>.

BROWN, S. *O modelo C4 de documentação para Arquitetura de Software*. 2018. <https://www.infoq.com/br/articles/C4-architecture-model/>.

CIPRIANO, L. *C4 Model*. 2021. <https://medium.com/pravaler-digital-team/c4-model-9b6e56705496>.

COLECO. *Revivendo Os Clássicos: Os 30 Melhores Jogos Do Atari Para Os Nostálgicos De Plantão*. 2023. <https://blog.coleco.com.br/retrogames/2023/05/17/revivendo-os-classicos-os-30-melhores-jogos-do-atari-para-os-nostalgicos-de-plantao/>.

DHARMA, H.; SETIADI, C. Adaptation in game: Consumer rating result for remade and remastered games (1997-2022). *Conference on Business and Industrial Research (ICBIR)*, 2023.

DIAS, R. *O Modelo em Cascata*. 2019. <https://medium.com/contexto-delimitado/o-modelo-em-cascata-f2418addaf36>.

FEIJÓ, R. H. B. *Uma arquitetura de software baseada em componentes para visualização de informações industriais*. Dissertação de Pós-Graduação, 2007.

FERREIRA, A. *Novo dicionário Aurélio da língua portuguesa*. Curitiba, Brasil: Editora Positivo, 2009.

FULLERTON, T. *Game design workshop : a playcentric approach to creating innovative games*. Burlington, U.S.A: Morgan Kaufmann, 2008.

GIL, A. *Como elaborar projetos de pesquisa*. São Paulo, Brasil: Editora Atlas, 2002.

GULARTE, D. *Jogos eletônicos: 50 anos de interação e diversão*. Teresópolis, Brasil: Editora Novas Idéias, 2010.

KENT, S. L. *The Ultimate History of Video Games: Volume Two: from Pong to Pokemon and beyond...the story behind the craze that touched our lives and changed the world*. Roseville, U.S.A: Prima Publishing, 2001.

KISHIMOTO, A. *Inteligência Artificial em Jogos Eletrônicos*. 2004. <https://www.academia.edu/en/2231411/Intelig%C3%AAncia_Artificial_em_Jogos_Eletr%C3%B4nicos>.

LUZ, A. *Vídeo game: história, linguagem e expressão gráfica*. São Paulo, Brasil: Editora Edgard Blücher Ltda., 2010.

MANNING, C. *Artificial Intelligence Definitions*. Texto de Definição, 2020.

MATOS, D. A. *Projetando mecânicas de jogos com base em uma abordagem iterativa*. Dissertação de Graduação, 2020.

MENDONÇA, R. S. *Videogames, Memória e Preservação de seu registro histórico-cultural no Brasil*. Dissertação de Mestrado, 2019.

RETROGAMER, T. *THE MAKING OF BERZERK*. 2021. <https://www.retrogamer.net/retro_games80/the-making-of-berzerk/>.

RIPARDO, B. *Berzerk (Atari, 1982)*. 2018. <https://bojoga.com.br/artigos/retroplay/atari-2600/berzerk-atari-1982/>.

SALTER, A. Taking over the world, again? examining procedural remakes of adventure games. *WELL PLAYED*, v. 6, n. 3, p. 68–90, 2017.

TEIXEIRA, M. *Como jogos antigos eram programados?* 2022. <https://www.dio.me/articles/como-jogos-antigos-eram-programados>.

YAMAHA. *O que é um equalizador?* ? <https://br.yamaha.com/pt/products/contents/proaudio/musicianspa/effects/equlizer.html#:~:text=Os%20equalizadores%20paramÃl'tricos%20de%20alguns,e%20HIGH%20(faixa%20alta).>.

ZILOG, I. *Z80 CPU User Manual*. 2016. <http://www.z80.info/zip/z80cpu_um.pdf>.

Glossário de Acrônimos

dB Representa decibéis e é uma unidade de medida de som. 1

IA Inteligência Artificial. 1

SFX Sound Effects, Efeitos Sonoros. 1

Glossário

Arcade É um aparelho de jogo eletrônico profissional instalado em estabelecimentos de entretenimento. 1

Assets Recursos utilizados no jogo. 1

Berzerk Jogo clássico do Atari. 1

Boss Chefão. 1

Bugs Defeito, falha ou erro no código de um programa que provoca seu mau funcionamento. 1

Console Terminal, ou conjunto de programas e equipamentos (p.ex., teclado e monitor) por meio do qual se pode operar o computador, permitindo a inserção e visualização de dados e, principalmente, o controle das operações executadas. 1

Design Conjunto de conceitos estéticos, técnicas e processos usados na criação e desenvolvimento de representações visuais de ideias e produtos. 1

Fade Efeito visual de surgimento gradual, usado para suavizar transições. 1

Game Design é a criação e planejamento dos elementos, regras e dinâmicas de um jogo. 1

Game Object Qualquer objeto em um jogo. 1

Gamer Palavra usada para se referir a algo ou alguém que tem relação com jogos eletrônicos. 1

Games Jogos. 1

Hardware Parte física de um computador. 1

Hud Elemento gráfico exibido na tela para transmitir informações ao jogador. 1

Joystick Dispositivo manual ligado a um computador, constituído por uma alavanca que se move sobre uma base permitindo o controle de um cursor. 1

Marketing Conjunto de técnicas de comercialização de produtos ou serviços, envolvendo pesquisas de mercado, adequação e promoção junto aos consumidores etc. 1

Player Jogador. 1

Prefab Game bject pré-configurado. 1

Remake Jogos remake são jogos refeitos. Portanto, diferente dos remasters, o alicerce do game original não continua. 1

Remaster Jogos que passaram por processos de melhorias em sua qualidade, sem alterar a base do jogos. 1

Single Player É um jogo eletrônico que possibilita a participação de apenas um jogador por partida. 1

Software Um programa de computador que segue instruções específicas e predeterminadas por um desenvolvedor. 1

Spawnado Criado, gerado. 1

Sprites Representação gráfica de um objeto na cena de um jogo. 1

Stand-by Em espera. 1

Tag Marcação. 1

Top-down Jogos que se tem a visão de cima. 1

Unity Ferramenta de desenvolvimento de jogos eletrônicos. 1

Unity Engine Ferramenta de desenvolvimento de jogos eletrônicos. 1

Video Game Programa eletrônico interativo que consiste num jogo em que imagens numa tela de computador ou televisão são manipuladas por meio de teclado ou console. 1