UNIVERSIDADE FEDERAL DO SUL E SUDESTE DO PARÁ
INSTITUTO DE GEOCIÊNCIAS E ENGENHARIAS
Faculdade de Engenharia da Computação
Bacharelado em Engenharia da Computação

Projeto Final de Curso

DESENVOLVIMENTO DE UM JOGO EM UNITY BASEADO NOS JOGOS CLÁSSICOS SUPER MARIO BROS, BOKTAI - THE SUN IS IN YOUR HAND E STREET FIGHTER 2

Arthur Martins de Oliveira

Marabá-PA
2025

Arthur Martins de Oliveira

DESENVOLVIMENTO DE UM JOGO EM UNITY BASEADO NOS JOGOS CLÁSSICOS SUPER MARIO BROS, BOKTAI - THE SUN IS IN YOUR HAND E STREET FIGHTER 2

Projeto Final de Curso, apresentado à Faculdade de Engenharia da Computação do Instituto de Geociências e Engenharias da Universidade Federal do Sul e Sudeste do Pará, como parte dos requisitos necessários para obtenção do Título de Bacharel em Engenharia da Computação.

Orientador:
Profº. Dr. Manoel Ribeiro Filho

Marabá-PA
2025

Dados Internacionais de Catalogação na Publicação (CIP)
Universidade Federal do Sul e Sudeste do Pará
Centro de Biblioteca Universitária

O48d Oliveira, Arthur

Desenvolvimento de um Jogo em Unity Baseado nos Jogos Clássicos Super Mario Bros, Boktai - The Sun Is In Your Hand e Street Fighter 2 / Arthur Oliveira. — 2025.

Orientador(a): Manoel Ribeiro Filho.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal do Sul e Sudeste do Pará, Campus de Marabá, Instituto de Geociencias e Engenharias, Faculdade de Computação e Engenharia Elétrica, Curso de Bacharelado em Engenharia da Computação, Marabá, 2025.

1. Desenvolvimento. 2. Vídeo Games. 3. Programação. 4. Pixel-Art. I. Ribeiro Filho, Manoel, orient. II. Título.

CDD: 22. ed.: 005

Gerada automaticamente pelo módulo Ficha Fácil, conforme os dados fornecidos pelos(as) autores(as).

Arthur Martins de Oliveira

DESENVOLVIMENTO DE UM JOGO EM UNITY BASEADO NOS JOGOS CLÁSSICOS SUPER MARIO BROS, BOKTAI - THE SUN IS IN YOUR HAND E STREET FIGHTER 2

Projeto Final de Curso, apresentado à Universidade Federal do Sul e Sudeste do Pará, como parte dos requisitos necessários para obtenção do Título de Bacharel em Engenharia da Computação.

Marabá: 13 de Fevereiro de 2025

BANCA QUALIFICADORA:

Profº. Dr. Manoel Ribeiro Filho
(Orientador - UNIFESSPA)

Profº. Dr. Joao Victor Costa Carmona
(Membro da Banca - UNIFESSPA)

Profª. Dra. Cindy Stella Fernandes
(Membro da Banca - UNIFESSPA)

Marabá-PA
2025

Dedico este trabalho à Deus, minha família e meus amigos, que foram a minha base, me apoiando e enviando forças durante toda minha vida. Dedico também aos professores, que além de compartilharam seus conhecimentos de forma tão atenciosa, me auxiliaram quando precisei.

AGRADECIMENTOS

Em meio a essa estrada tortuosa que é a formação acadêmica, finalizar este trabalho não seria possível sem o apoio de pessoas especiais para mim. Algumas já conhecidas, já outras, tive o prazer de conhecer no decorrer do curso.

Primeiramente, agradeço aos meus pais, que me apoiaram financeiramente e emocionalmente durante todos esses anos de faculdade. Em especial à minha mãe, Kátia, que nunca deixou de se preocupar comigo, mesmo estando longe. Não posso deixar de mencionar minha avó, Elba, que cuidou de mim desde pequeno e me apoia com muito amor e carinho.

Sou grato também por meus amigos, que me ajudaram e motivaram nessa jornada, Lucas Antonio, Lucas Leite, Messias Silva, Thiago Eleutério, Breno Cortez, Luís Felipe e Henrique Viana. Amigos esses que me auxiliaram quando tive dúvidas e me motivaram quando eu estive desanimado. Serei eternamente grato por tudo que fizeram por mim e pelas memórias que compartilhamos.

Agradeço aos meus professores, que iluminaram o meu caminho no período de formação e não desistiram de nos ensinar mesmo em meio ao período da pandemia. Em especial, gostaria de expressar minha gratidão ao meu orientador, Manoel Ribeiro, por auxiliar e não desistir deste projeto. Também devo agradecer aos professores Diego Kasuo, Warley Muricy, João Victor, Leslye Estefania e Cindy Stella, pelo empenho que demonstraram em seu trabalho e pelo impacto positivo que causaram em nossa turma.

Aos meus grandes amigos, Gustavo Soares, Raphael Rocha e Arthur Cardoso, e minha namorada, Rebeca Karolinne, agradeço por moldarem o que sou, vocês são meu tesouro e tenho certeza que, sem vocês, eu seria uma pessoa muito diferente.

A todos, muito obrigado por fazerem parte deste capítulo da minha vida.

Uma escolha não é algo que se ganha, mas algo que se cria.

(Papillon)

RESUMO

Este trabalho faz uma análise de três jogos 2D famosos das décadas de 80, 90 e meados de 2000, que possuem a essência dos gêneros clássicos de suas respectivas épocas. O objetivo principal do trabalho é falar sobre a influência dos jogos citados e como eles impactaram os desenvolvedores indies nos anos subsequentes, seguido pelo desenvolvimento de um jogo de três níveis que busca prestar uma homenagem a estes títulos reproduzindo seus estilos de jogatina. Cada nível foi feito inspirado em um dos jogos analisados e desenvolvido na Unity Engine, um motor gráfico gratuito utilizado no desenvolvimento de jogos de alta qualidade. O processo de criação seguiu as etapas tradicionais da indústria de jogos, incluindo pré-produção, produção, testes e refinamento. O jogo desenvolvido apresenta três gêneros distintos: plataforma 2D, aventura isométrica e luta, cada um recriando mecânicas e estilos visuais característicos dos jogos que o inspiraram, buscando preservar a essência retrô dos jogos dos anos 90. O desenvolvimento deste jogo demonstra os desafios de um desenvolvedor independente ante a experiência de desenvolver um jogo do início nos dias atuais.

Palavras-chave: Desenvolvimento, Vídeo Games, Super Mario, Boktai, Street Fighter.

ABSTRACT

This work analyzes three famous 2D games from the 80s, 90s and mid-2000s, which have the essence of the classic genres of their respective eras. The main objective of the work is to talk about the influence of the aforementioned games and how they impacted indie developers in subsequent years, followed by the development of a three-level game that seeks to pay homage to these titles by reproducing their gameplay styles. Each level was inspired by one of the games analyzed and developed using the Unity Engine, a free graphics engine used to develop high-quality games. The creation process followed the traditional steps of the gaming industry, including pre-production, production, testing and refinement. The game developed features three distinct genres: 2D platform, isometric adventure and fighting, each recreating mechanics and visual styles characteristic of the games that inspired it, seeking to preserve the retro essence of games from the 90s. The development of this game demonstrates the challenges of a independent developer with the experience of developing a game from scratch today.

Keywords: 2D Game Development, Video Games, Super Mario, Boktai, Street Fighter.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

2D	*Duas Dimensões*
3D	*Três Dimensões*
C#	*C Sharp*
GDD	*Game Design Document*
SGDD	*Short Game Design Document*
SNES	*Super Nintendo Entertainment System*
IA	*Inteligência Artificial*
IDE	*Ambiente de Desenvolvimento Integrado*

SUMÁRIO

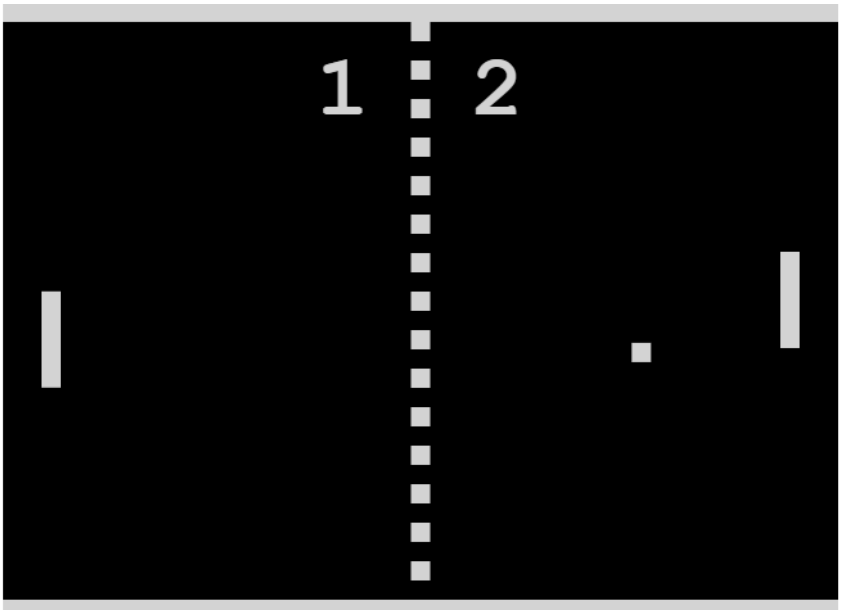
1 INTRODUÇÃO

Jogos eletrônicos, um termo que é amplamente conhecido e falado nos dias atuais, com o surgimento do primeiro destes em 1958, o jogo Tennis for Two desde sua criação já visava a diversão e entretenimento (AMORIM, 2006), porém por ter sido o primeiro, este carecia de características muito importantes que jogos que vieram após ele começariam a trabalhar e aperfeiçoar, os pilares visuais: identidade e apelo. O próprio Space Invaders, lançado em 1978, foi um dos que deram os primeiros passos em rumo à Pixel-Art com as representações dos aliens presentes no jogo, contudo ainda não era o suficiente, já tinham a identidade, mas ainda precisavam aperfeiçoar o apelo visual, e suas artes, apesar de marcantes por estarem entre os primeiros e possuírem sua identidade, não tinham a beleza e o carisma do apelo visual, que até os dias atuais, é um dos maiores chamativos no conteúdo dos jogos.

E foi então que, após alguns anos de estudo e desenvolvimento, surgia *Donkey Kong* em 1981, e pelas mãos de Shigeru Miyamoto ganhava vida aquele que seria conhecido como o primeiro jogo de plataforma. Era o jogo de arcade mais complexo da época, se diferenciando dos demais desde a gameplay – ao contrário da maioria que focava apenas no gênero de tiro e naves, no jogo de Miyamoto o jogador deveria subir vigas e escadas até o macaco Donkey Kong para salvar uma mulher chamada Lady enquanto o macaco jogava barris para tentar atrapalhar o jogador (MATTOS, 2011) – até seus gráficos que impressionavam apesar da limitação de cores e resolução das máquinas da época, que trabalhavam apenas com 8 bits(256 cores, com máximo de 25 cores presentes na tela por conta do baixo poder de processamento).

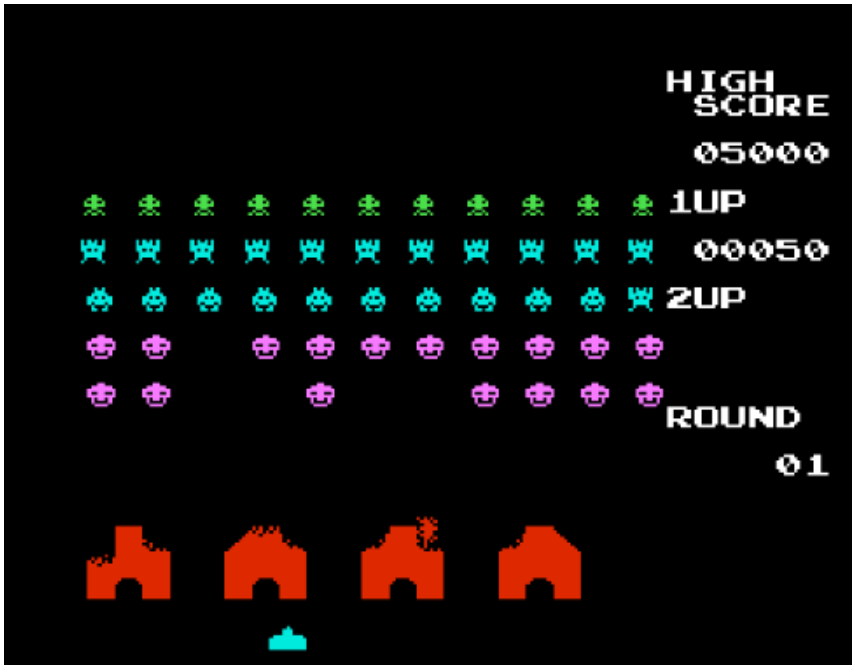
Jogos anteriores como *Space-Invaders*, Figura 2 e *Pong*, Figura 1, não conseguiam se expressar suficientemente com seus visuais simples onde os "personagens" que o jogador controlava eram compostos apenas de pequenos retângulos, porém quando *Donkey Kong* chegou, trouxe consigo um novo passo na história da Pixel-Art, visuais marcantes, expressivos e cheios de vida – era isso que estava sendo transmitido na tela dos arcades em 1981 para as crianças e adultos que viam aquele jogo pela primeira vez, eram apresentados um homem, um macaco e uma mulher que até os dias atuais podem ser facilmente identificados com apenas um olhar. Mecânicas de jogo e efeitos sonoros são partes importantíssimas para garantir o sucesso de um jogo, mas não podemos negar a surreal importância que o apelo e a identidade visuais causam nos jogadores e em quem está apenas observando aquilo, o objetivo principal destes é atrair a atenção das pessoas e esse foi o efeito que a Pixel-Art feita em *Donkey Kong* causou em diversos jogadores e desenvolvedores, tanto da nova quanto da antiga geração.

Figura 1 – Captura do jogo Pong



Fonte: Disponível em: <https://www.ponggame.org>.

Figura 2 – Captura do jogo Space Invaders



Fonte: Disponível em <https://www.retrogames.cz>.

Por conta deste efeito gerado pelo apelo visual, muitas pessoas se tornaram ligadas emocionalmente à Pixel-Art, o que fez com que esse estilo se propagasse e atualmente esteja sendo usado não mais por limitações técnicas, mas como escolha artística de muitos desenvolvedores, como uma verdadeira herança deixada por estes jogos clássicos.

Por outro lado, apesar do processo da criação e evolução da Pixel-Art ser um assunto chamativo no desenvolvimento de jogos 2D, a forma como estes eram de fato

programados era bem mais complicada, pois sua programação era feita em Assembly, uma linguagem de programação primitiva. Felizmente, com o passar dos anos foram surgindo novas linguagens mais acessíveis e que são utilizadas até hoje e junto delas as game engines. Existem diversas game engines famosas atualmente, como Game Maker Studio, RPG Maker, Godot, etc; contudo, neste trabalho a game engine escolhida foi a Unity Engine, por possuir uma interface amigável e bastante conteúdo de auxílio na internet, desde documentação oficial até vídeos ensinando os básicos da engine.

Neste trabalho será brevemente discutido como é possível notar a influência da Pixel-Art nos jogos indies e também o desenvolvimento na Unity Engine de um jogo com 3 fases que irá explorar gêneros famosos de jogos das décadas de 80 e 90, sendo feito com visuais em Pixel-Art utilizando jogos marcantes como fonte de inspiração e base para mecânicas.

1.1 Justificativa

O trabalho de desenvolver um jogo é complicado e oneroso, por isto geralmente é realizado por grupos de desenvolvedores, pois quanto mais complexa a proposta do jogo, mais atenção aos detalhes, trabalho e custo serão exigidos no desenvolvimento; afinal, um jogo possui elementos de diferentes áreas que precisam ser trabalhados individualmente, como design visual, design sonoro e programação.

O processo de desenvolvimento de jogos possui três etapas principais: Pré-produção, Produção e Pós-produção, cada uma contendo atividades de diferentes áreas como marketing, composição musical, artes conceituais, etc.; com o resultado da produção culminando nas características que definem a "alma" do jogo.

Mesmo em meio a tantos processos necessários numa produção minimamente coesa, grupos menores de desenvolvedores que ainda possuem pouca experiência de mercado, mas desejam desenvolver e publicar seus jogos se arriscam, mesmo com a possibilidade de haver prejuízo, já que geralmente acabam gastando do próprio bolso para financiar seu jogo. Se aventurar nesse mercado mesmo sem receber grandes investimentos como produtoras mais famosas, torna o trabalho de desenvolvimento muito mais difícil. Os jogos destes desenvolvedores são chamados de "jogos indies", termo que vem da abreviação de "Independent" e também é usado para se referir aos desenvolvedores, chamando-os de "indie game dev's".

Segundo o site *Coruja Informa*, os jogos indies surgiram nos computadores no início da década de 90 com ferramentas práticas e mais acessíveis aos desenvolvedores, sendo elas os sharewares e freewares - termos usados para softwares sem cobrança ao usuário - , como por exemplo o software *Adventure Game Studio(AGS)* DE 1997, que permitia a criação de jogos de aventura do gênero Point-and-Click com exemplos bem

famosos como as séries *Monkey Island*(1990-2022) e *King's Quest* (1984-2015) nas Figuras 3 e 4.

Figura 3 – Monkey Island

Fonte: Disponível em: <https://store.steampowered.com/app/32360/The$_{S}ecret_{o}f_{M}onkey_{I}sland_{S}pecial_{E}dition/$ > .*Acessoem* : 15fev.2025.

Figura 4 – King's Quest

Fonte: Disponível em: <https://www.retrogames.cz/play$_{1}$18 − *DOS.php* > .*Acessoem* : 15fev.2025.

Apesar de possuírem um começo bastante humilde e até mesmo despercebido, os jogos indies foram conquistando seu lugar "na boca do povo" e atualmente alguns deles "fazem mais barulho" que jogos de empresas consolidadas há muito tempo, como por exemplo o rogue-like *Hades*(2020), que conquistou o público com sua gameplay frenética e seus designs cativantes. Existem vários outros exemplos de jogos indies que alcançaram o sucesso e serão citados posteriormente para discutirmos suas inspirações e o efeito de perpetuar as essências visuais e de estilo de jogos clássicos. Boa parte da "leva atual" de

indie game dev's tiveram influência dos jogos das décadas de 80/90 e muitos deles se apegaram tanto aos estilos como a arte destes, como por exemplo os jogos *The Legend of Santa* - 2023(inspirado em Super Mario World), *Pocket Rumble* – 2016(inspirado no gênero de luta dos clássicos da SNK), *Kura5* - 2019(primeiras versões,é um jogo episódico) que, assim como o jogo desenvolvido neste trabalho, foram feitos se inspirando em gêneros e jogos famosos das décadas de 80/90.

Portanto, no decorrer deste trabalho as etapas citadas anteriormente serão destrinchadas, seguidas por um estudo sobre o desenvolvimento de um jogo com 3 níveis correspondentes a jogos famosos feitos com estilo visual em Pixel-Art e como é possível observar um pouco da influência que certas abordagens na execução da produção de jogos da década de 80 e 90 criaram, trazendo desde padrões que afetam diretamente o desenvolvimento de jogos atuais, como a permanência de um estilo de arte, com foco nos indies, de forma que os leitores deste trabalho possam compreender melhor o desenrolar e as nuances da produção de um jogo.

1.2 Objetivo Geral

Este trabalho tem por objetivo geral detalhar o desenvolvimento de um jogo 2D na Unity Engine baseado no visual e mecânicas de três jogos populares que também são fonte de inspiração para outros jogos indies.

1.3 Objetivos Específicos

- Comentar brevemente sobre um pouco da influência que a Pixel-Art dos jogos das décadas de 80 e 90 causa nos jogos indies atuais;
- Desenvolver um jogo 2D na Unity Engine que consiga abordar um gênero diferente de jogo em cada nível, feito com inspiração em três jogos específicos. Cada fase terá estilos visuais e objetivos diferentes relacionados com suas respectivas inspirações;
- Comparar e analizar cada uma das fases inspiradas com seus respectivos jogos, documentar as dificuldades encontradas no desenvolvimento e na animação das Pixel-Arts.

1.4 Organização do Trabalho

Após a introdução, este trabalho possui mais cinco capítulos, sendo estes:

- Capítulo 2: Neste capítulo, um breve detalhamento das fontes utilizadas para pesquisa e inspiração deste projeto será abordado. Sendo válido ressaltar que os trabalhos

escolhidos como fontes literárias, assim como este, abordam o desenvolvimento de um jogo na Unity Engine;

- Capítulo 3: Este capítulo, aborda os materiais que pavimentaram o desenvolvimento, sendo eles: os métodos de produção e organização utilizados no desenvolvimento de jogos e também os softwares que auxiliaram na criação deste jogo;
- Capítulo 4: Este capítulo entrega uma visão geral de cada um dos três níveis do projeto, além de resumir a história e o objetivo do jogador;
- Capítulo 5: Neste capítulo o processo de implementação do projeto será abordado, desenvolvendo temas como: Método empírico utilizado na produção, mecânicas e programação de cada nível e por último uma análise dos desafios e resultados obtidos;
- Capítulo 6: Este capítulo abordará os desafios e também uma análise de resultados deste projeto.
- Capítulo 7: Neste capítulo se encontra a conclusão do trabalho e tudo aquilo que foi discorrido até então, além de disponibilizar um link para o repositório deste projeto para aqueles que se interessarem em conferi-lo na íntegra.
- Apêndices: Aqui se encontram alguns scripts importantes que foram abordados neste trabalho para leitura com mais detalhes.

2 TRABALHOS RELACIONADOS

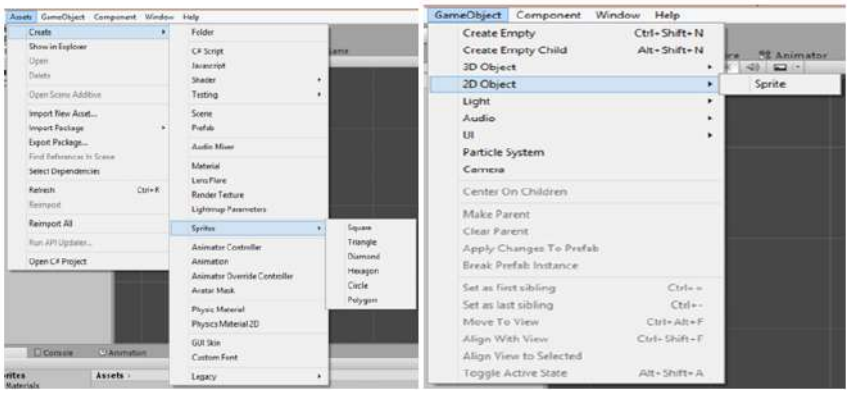
A sessão 2 irá abordar alguns trabalhos correlatos que foram desenvolvidos na Unity Engine e são focados em desenvolvimento de jogos.

2.1 Desenvolvimento de um jogo de plataforma 2D na Unity

(SOURAV, 2017) documentou em seu trabalho o desenvolvimento de um jogo de plataforma 2D destacando o aprendizado que teve com as ferramentas e funcionalidades específicas para criação de jogos 2D. Focando em implementar um jogo básico que servisse de exemplo para estudantes e desenvolvedores iniciantes.

O objetivo do autor era desenvolver um bom exemplo de jogo 2D de plataforma moderno e que seu trabalho também pudesse ser usado como um tutorial passo-a-passo para criar um jogo. Por este motivo o autor detalhou cada componente da interface de forma introdutória antes de avançar para a explicação propriamente dita de como produzir um jogo 2D, na Figura 5, é demonstrado como adicionar um sprite 2D ao Canvas da Unity(componente que mais tarde será discutido em nossos tópicos de produção) de forma que o leitor compreenda a ferramenta e como usá-la.

Figura 5 – Demonstração de como adicionar um sprite 2D no Canvas.



Fonte: Sourav, 2017.

O trabalho de Sourav é uma explicação, quase como uma receita de como começar na produção de jogos 2D, por exemplo, seu tópico 4.4.2 ”Attack and Health System”, é uma descrição do funcionamento da IA dos inimigos e os motivos de suas ações, além disso, o autor utiliza de imagens do código e o explica de forma didática. Sourav também explica que por conta dos sprites dos personagens serem obtidos por meio da Asset Store da Unity, alguns personagens não possuem animações de ataques físicos, como o arqueiro, por exemplo, o que o colocou em processo de adaptação para o desenvolvimento, de forma que este precisou adaptar o design do projeto.

O ponto alto do trabalho proporcionado por Sourav é sua capacidade de explicar detalhadamente os processos do desenvolvimento de seu jogo, desde a escrita de scripts até explicações das funções presentes nas mecânicas dos personagens, funções da Unity engine, etc.

2.2 Desenvolvimento de um jogo de puzzle isométrico na Unity

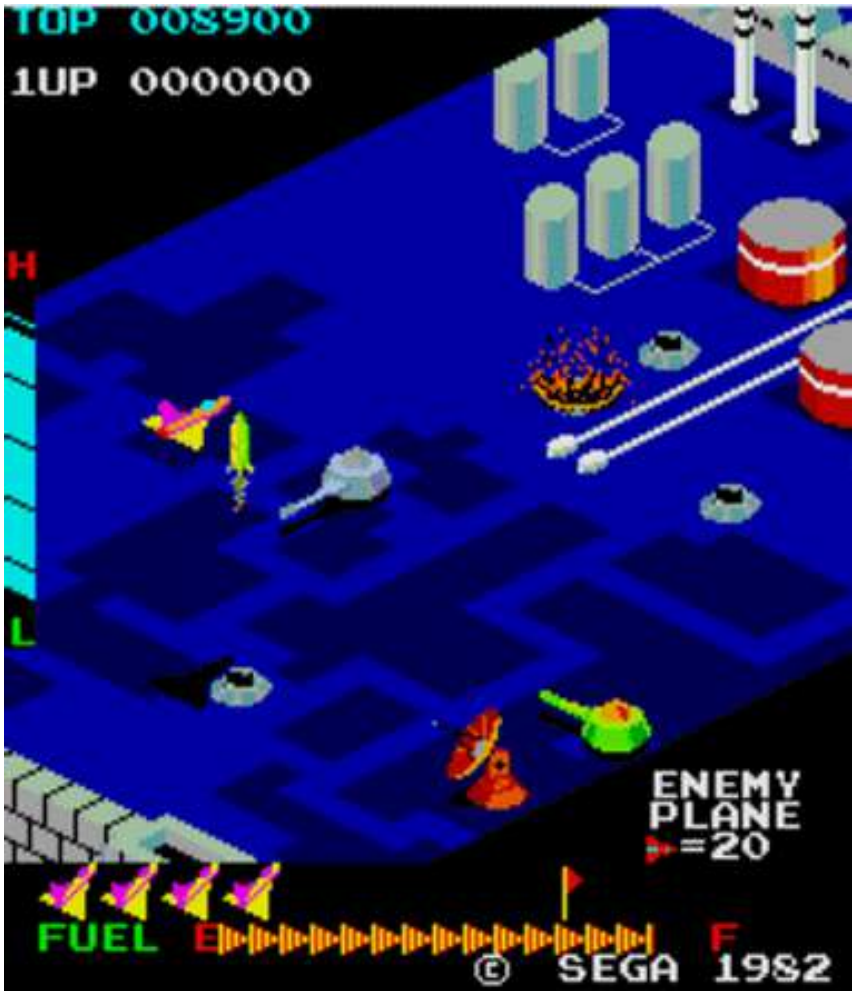
O segundo trabalho escolhido foi uma tese de mestrado(MARCUSSEN SOFIE SCHMIDT, 2023) concebida por três autoras, sobre o desenvolvimento de um jogo em perspectiva isométrica focado na resolução de quebra-cabeças. Sarah, Sofie e Eleni desejavam fazer um jogo com a temática referente ao livro "Inferno de Dante" de Dante Alighieri, onde a protagonista havia sido mandada ao inferno por conta de sua vida pecaminosa e no decorrer do jogo passaria por uma jornada para encarar suas ações. A ideia principal das autoras era de que o jogo conseguisse contar sua história por meio dos designs de ambiente, sendo feito em 3 níveis, cada um abordando um pecado capital, de forma que a narrativa estivesse sempre se comunicando visualmente com o jogador. Veja a tela inicial do jogo na Figura 6.

Figura 6 – Tela inicial do jogo # INFERNO.

Fonte: # Inferno Master Thesis por Sarah Marcussen, Sofie Kajaer Schmidt e Eleni Zakynthinou(2023).

Em seu artigo "O que é um jogo isométrico e Como fazê-lo", Süleyman diz que a técnica da perspectiva isométrica em jogos se baseia em permitir que objetos 2D planos pareçam ter 3 dimensões. Isto é feito desenhando os sprites (ou posicionando os modelos em casos de jogos com assets tridimensionais) seguindo um formato de losango, de forma que cada lado se assemelhe à visão de uma câmera com ângulo de 30 graus para cada lado (CENGIZ, 2021). O primeiro jogo a utilizar desta técnica foi *Zaxxon*(1982), da SEGA, que ficou tão famoso por seus gráficos que simulavam profundidade 3D que popularizou o gênero isométrico, encadeando num boom de jogos do mesmo gênero. A Figura 7 ilustra um momento do jogo, destacando seu pioneirismo no uso da perspectiva isométrica.

Figura 7 – Zaxxon(1998).



Fonte: Disponível na revista *JOGOS 80* Ano 88 - Nº10.

Após o impacto visual de Zaxxon, que impressionou com seus gráficos inovadores para a época, diversos outros jogos passaram a adotar a perspectiva isométrica. Entre os diversos exemplos que podem ser citados, foram escolhidos os dois presentes nas Figuras 8 e 9, sendo estes as franquias icônicas Diablo (1996) e Baldur's Gate (1998), que se tornaram referências atemporais do gênero, sendo amplamente reconhecidos e respeitados até os dias atuais.

Figura 8 – *Diablo* demo(1996).

Fonte: Disponível na revista *COMPUTER GAMES STRATEGY PLUS* Nº72

Figura 9 – *Baldur's Gate*.

Fonte: Disponível em: Interplay E3 Asset Disc(1998).

A parte do trabalho que chama mais atenção e passa a ideia de adaptação do desenvolvedor, é o momento em que as autoras compartilham que, no decorrer da produção do jogo não puderam contar com a funcionalidade de projeção de sombras embutida na Unity, pois a mesma não é adaptada para projeção isométrica e que observando *Hades*(Supergiant Games, 2020) e *Bastion*(Supergiant Games, 2011), jogos tomados como referência por elas, perceberam que os sprites deveriam ser desenhados de forma que suas

luzes e sombras fossem embutidos, sendo aparentes e coerentes à projeção de luz dos cenários. Na Figura 10, é possível observar exemplos de como os sprites são feitos para se encaixar perfeitamente nos cenários.

Figura 10 – Capturas de *Bastion*(Supergiant Games, 2011) e *Hades*(Supergiant Games, 2020).

Fonte: # INFERNO Master Thesis Report(2023).

2.3 Tese sobre o Design dos Jogos Indies

A tese de Enrique Alejandro Pérez Domínguez, "The Design of Indie Games, a Different Paradigm"(DOMíNGUEZ, 2019), aborda o design de jogos independentes em sua concepção demonstrando que estes em sua maioria, são feitos de forma diferente da tradicional utilizada na indústria de jogos. Enrique também comenta sobre como desenvolver jogos é algo relativamente lógico e intuitivo para ele, por conta do mesmo ter desenvolvido e re-inventado jogos a vida toda contudo, certa vez um de seus alunos lhe disse que o processo de criar jogos não era tão claro e este foi um dos motivos para a concepção de sua tese.

O autor inicia contextualizando o movimento indie, com suas raízes culturais e históricas, destacando que um dos pontos altos destes é que possuem a chamada "independência criativa", característica que os permite arriscar em inovação, ao contrário de seus concorrentes na grande indústria que geralmente focam em seguir um caminho seguro seguindo fórmulas que já possuem sucesso consolidado, como por exemplo as franquias *Call Of Duty* e *FIFA*(atualmente conhecido apenas como FC) que tem seguido o mesmo modelo desde seus títulos iniciais *Call Of Duty* (2003) e *FIFA International Soccer* (1993) contando com mudanças mínimas em cada sequência, geralmente focadas apenas em melhoria gráfica e balanceamentos.

O formato de construção do trabalho combina revisão teórica e estudo empírico. Para o estudo empírico, o autor entrevistou 30 designers indies premiados nos festivais Indiecade e Independent Games Festival, utilizando a teoria fundamentada construtivista para analisar os dados identificando padrões e filosofias nos processos de design dos entrevistados, também abordando teorias de design em disciplinas como engenharia, arquitetura e design de produtos, aplicando-as ao contexto dos jogos para compreender

melhor os métodos usados pelos criadores indie.

Um dos destaques da tese de Enrique é que o mesmo conclui que o design de jogos indie se aproxima mais de uma prática artística do que o convencional comercial, onde os criadores tem como objetivo principal se expressar e fazer jogos da forma que atenda aos seus desejos pessoais ao invés de comerciais. Enquanto o design tradicional é marcado por metas comerciais e restrições rígidas, o design indie é reconhecido por sua liberdade, ocasional inovação e expressão pessoal. A tese também aborda como frequentemente o desenvolvedor indie assume diversos papéis no desenvolvimento por conta da falta de orçamento, onde muitas vezes o mesmo enfrenta desafios significativos por ser encarregado não só de programar o jogo como também da parte artística, de game design, marketing, etc. Apesar de tudo, em sua tese Enrique sugere que o sucesso dos jogos indie não depende apenas de técnicas de design, mas também da capacidade de criar narrativas e experiências que ressoem com os jogadores, citando em sua tese exemplos de jogos indies que fizeram sucesso comercial como *Cuphead*(Entertaiment, 2017) e *Overcooked*(Ghost Town Games, 2016), ilustrados na Figura 11:

Figura 11 – À esquerda *Cuphead*(Entertainment, 2017) e *Overcooked*(Ghost Town Games, 2016).

Fonte: The Design of Indie Games - A Different Paradigm(2019).

Por fim, o autor argumenta que os jogos indie são uma força transformadora dentro dos estudos de jogos e da própria indústria. Ao desafiar paradigmas tradicionais, os criadores indie mostram que o design de jogos é mais do que uma prática comercial: é também uma forma de expressão cultural e artística. Essa perspectiva amplia a compreensão do design como atividade humana, oferecendo novas possibilidades para pesquisa e prática no campo dos jogos.

3 MATERIAIS E MÉTODOS

Este capítulo será responsável por descrever e detalhar as ferramentas e metodologias utilizadas no desenvolvimento do nosso jogo. Serão abordados: SGDD, desenho, animação e a Game Engine utilizada, de forma que o leitor possa absorver o conteúdo e entender o processo de desenvolvimento de um jogo 2D.

3.1 SGDD(Short Game Design Document)

Uma das principais ferramentas na produção de um jogo, o GDD (Game Design Document) serve como uma bíblia para guiar os desenvolvedores na produção de um jogo, sendo disponibilizado pelo game designer da equipe, o GDD contempla a descrição de diversos elementos como proposta do jogo, world building, mecânicas, história, como será o fluxo de gameplay, menus e diversas outras informações, dependendo de quão grande o escopo do projeto for, podendo chegar a ter dezenas de páginas para detalhar minuciosamente cada parte do projeto (RYAN, 2009). Para que tudo ocorra de forma coesa e sólida, o GDD tem de ser grande e detalhado, o que já se tornou um padrão consolidado da indústria (MOTTA, 2013).

Contudo, o GDD é um documento muito robusto por natureza e em situações onde o jogo produzido não é de grande porte, a quantidade de detalhes necessária neste se torna um contratempo e foi pensando nisso que o SGDD foi criado, com o "S" representando a palavra "Short" que significa "curto".

A primeira fase do desenvolvimento - a pré-produção - é iniciada no momento em que o SGDD começa a ser concebido. Documentos de design não possuírem um formato único e definido é um fato, a forma como serão construídos varia de projeto para projeto e de empresa para empresa (SCHELL, 2008), apesar disso o propósito do SGDD permanece em servir como um guia para a produção do jogo de forma resumida e linear, é nele que estarão descritas as ideias de mecânicas e conceitos iniciais, tornando-se o farol que guiará os desenvolvedores e em casos de advergames (jogos que anunciam uma marca, produto ou serviço) como *PEPSIMAN*(1991), será responsável por convencer os investidores a depositar seu capital na produção. De forma resumida, sua função é assegurar que todos os membros da equipe de desenvolvimento estejam alinhados e que o projeto siga uma direção definida.

O SGDD possui o objetivo de fazer em menos tempo e de forma resumida aquilo que seu irmão mais velho tem feito em projetos maiores, mas poupando tempo. Como este projeto se trata de um jogo de pequeno porte, o SGDD foi utilizado.

3.2 Metodologia no Desenvolvimento

No desenvolvimento, um método empírico foi adotado, focado em iterações do projeto após reuniões mensais para mostrar o que já foi feito e o que deveria ser entregue na próxima reunião, de forma a adaptar e fazer as modificações necessárias no decorrer do projeto, fornecendo um processo conveniente e flexível no desenvolvimento.

3.3 Ferramentas de desenvolvimento

- Unity: É uma plataforma de desenvolvimento de jogos, sendo uma das mais famosas no ramo, a Unity foi lançada em 2005 e, desde então, tem crescido e sido utilizada tanto por desenvolvedores indies como grandes estúdios. Versátil e possuindo uma interface amigável, a Unity Engine se consolidou no mercado como referência pois, além de possuir diversas documentações e vídeos de auxílio em seu próprio site, a mesma permite desenvolvimento para multiplataforma oferecendo suporte para gráficos 2D e 3D. Utilizando a linguagem C#, a Unity permite que os desenvolvedores modifiquem as funções disponibilizadas pela engine via código, tornando-se uma game engine adaptável. Além das vantagens citadas anteriormente, a Unity também conta com um marketplace de assets embutido, onde os desenvolvedores podem disponibilizar e adquirir modelos 3D, sons, scripts, texturas e muitas outras coisas de forma gratuita. Com todas essas vantagens os desenvolvedores iniciantes podem focar em estudar o desenvolvimento pois a Unity dispõe de um ambiente amigável àqueles que estão começando e alta usabilidade para aqueles que são mais experientes, se tornando altamente atrativa no mercado;

- Visual Studio(2022): Atualmente uma das IDE's mais utilizadas mundialmente, o Visual Studio possui diversas razões para ter sido utilizado neste trabalho, sendo essas: suporte a C# e diversas outras linguagens, ferramentas de depuração e diagnóstico que facilitam a correção de erros de lógica no código, integração nativa com o Git incluindo desde commits até controle de versões, IntelliSense e Code Completion – recurso utilizado para facilitar e evitar erros no momento de digitar os códigos, pois enquanto estes são digitados, essa função sugere o restante do código de forma a facilitar a programação e economizar tempo -, entre outras vantagens porém o que determinou sua escolha foi sua integração nativa com a Unity, possuindo ferramentas feitas especialmente para a depuração de jogos;

- Aseprite: Lançado em 2014, o Aseprite é um software de criação de Pixel-Art e animações em 2D, utilizado tanto por Pixel-Artistas como por desenvolvedores de jogos. O Aseprite possui diversas ferramentas que justificaram sua escolha, valendo citar algumas que se destacam: linha do tempo no canvas para auxílio na animação,

sobreposição de camadas, personalização de paletas de cores, ferramenta de prévia da animação (é possível ver como a animação vai ficar enquanto o usuário está desenhando frame a frame individualmente) e uma interface intuitiva, salvamento dos arquivos em formato de Sprite Sheet(sequência de frames numa imagem só, serve tanto para animações quanto para criação de "paletas de sprites"),além de ser um software leve e eficiente com preço acessível;

- GitHub: Atualmente conhecido tanto por quem trabalha na área de desenvolvimento quanto por quem não trabaha, o GitHub é uma plataforma de versionamento, utilizada em sua maioria para versionamento e armazenamento online de código, fundado em 2008 e baseado no Git, sistema de controle que detecta alterações em arquivos, o GitHub vem sendo uma "mão amiga" para os desenvolvedores e se tornou um dos pilares do desenvolvimento de software sendo utilizado tanto por desenvolvedores independentes como startups e grandes empresas.

3.4 Desenho e Animação

Neste capítulo o processo de desenho e animação deste projeto será abordado em conjunto com uma breve observação de como alguns jogos das décadas de 80 e 90 têm tido influência na cena indie atual, abordando desde: escolha de determinados gêneros famosos nessas épocas até escolhas de estilo visual.

3.4.1 Animação no Aseprite

O processo de animação foi feito totalmente no Aseprite e conforme o projeto avançava, foi possível observar uma melhoria no design e animação das Pixel-Arts do Relp e dos personagens presentes no jogo.

Abordando primeiramente sobre o design inicial do protagonista do jogo. No início uma visão mais abrangente era o foco, com um sprite maior, sendo possível observar isto na Figura 12.

Figura 12 – Primeira Pixel-Art do Relp, Fonte: O Autor.

Contudo, para um “animador de primeira viagem”, animar um corpo maior se mostrou uma tarefa difícil, pois as animações de movimentação acabavam ficando “tortas”; por isso, procurar uma forma mais simples de trazer o Relp do papel para o jogo foi um dos objetivos principais no início da produção. Com base nesse pensamento, optar por desenhar e animar em “Mega Man Style”(Figura 13) era a melhor escolha.

Figura 13 – Sprite Mega man, Fonte: (SPRITERSRESOURCE, 2024b)

Vindo de outro clássico dos jogos de plataforma, o design inspirado em Mega Man auxilia bastante Pixel-Artistas iniciantes, pois é um design expressivo e que dá foco nas características principais do protagonista, mesmo que este possua poucos pixels. No caso deste projeto, só foi preciso fazer com que a jaqueta do Relp fosse bem adaptada ao design, já que ela é o chamariz de seu design. É possível ver o sprite “Iddle” do Relp para o nível 01 em Mega Man Style na Figura 14.

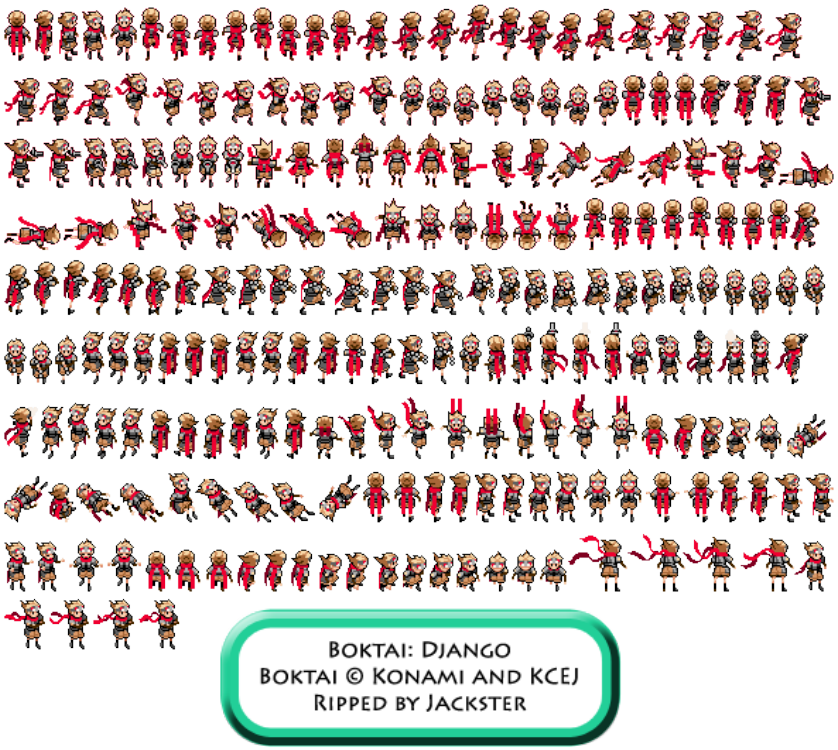
Figura 14 – Sprite do Relp no primeiro nível, Fonte: O autor.

Apesar de ser bastante utilizado por Pixel-Artistas iniciantes, o Mega Man Style também é utilizado por desenvolvedores experientes em jogos com uma peculiaridade que vem crescendo nos dias atuais, o chamado “Demake”, onde desenvolvedores re-lançam jogos famosos, só que com o diferencial de serem feitos numa visão “mais antiga” do jogo, como se, por exemplo, um jogo que lançou para o console Playstation 4, fosse refeito para o Playstation 1. Um ótimo exemplo está na Figura 15, onde temos o lançamento de Mega Man Legends 3 em sua versão Demake feita por um fã da série.

Figura 15 – Mega Man Legends Demake Fangame, Fonte: (WHITEHEAD, 2014)

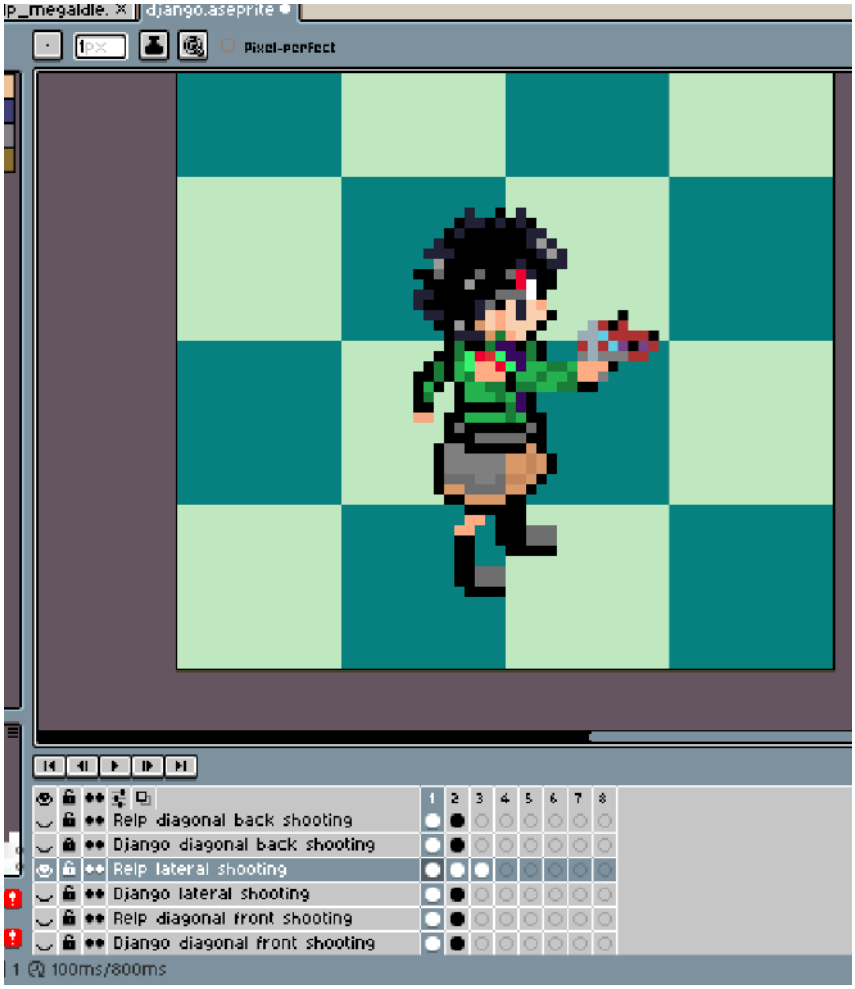
Passando para o nível 02, a animação do protagonista se tornou bem mais complexa, pois o gênero escolhido possuía perspectiva isométrica. Então, como as animações ainda eram uma tarefa difícil, a melhor forma de chegar próximo do desejado era transcrever os sprites do jogo tomado como inspiração, Figura 16, e desenhar o Relp por cima de cada sprite, para que assim a animação ficasse mais consistente.

Figura 16 – Django Sprite Sheet, Fonte: (SPRITERSRESOURCE, 2024a)



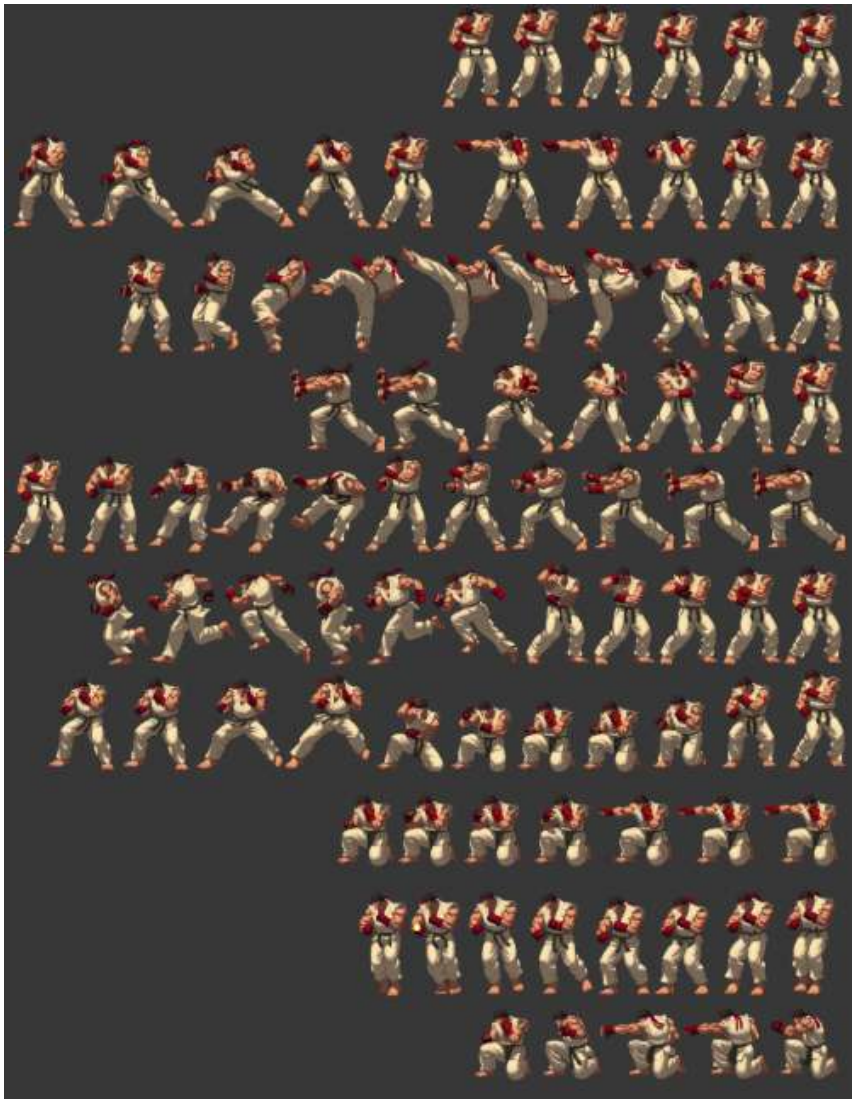
É possível observar na Figura 17, que existia uma camada do Relp e do Django para cada animação, onde, após finalizada uma camada do Django, os frames do Relp eram desenhados por cima.

Figura 17 – Camadas do Relp e do Django, Fonte: O Autor.



Por último o nível 03, nele bastou apenas a inspiração pois, por falta de tempo, não seria possível reproduzir o estilo de arte com fidelidade. Contudo, apesar disso uma sprite sheet feita por um fã foi utilizada, Figura 18, para ter uma ideia de como os movimentos e golpes deveriam ser animados.

Figura 18 – Sprite sheet do Ryu feita por um fã, Fonte: (METHIOU, 2016)



Street Fighter 02 impressionou com seus sprites na época em que foi lançado, e para destacar a influência deste é que foi decidido tomar uma sprite sheet feita por um fã para desenvolver a animação do Relp neste projeto. Apesar de ser evidente a disparidade em técnica, os sprites do Relp também contemplam uma sequência de golpes, estado de animação parado e também de defesa, sendo possível observar alguns dos frames das animações dele na Figura 19, assim como as tags em cima dos frames detalhando as animações que foram possíveis de desenvolver, como a de correr para frente, defesa, pulo, etc..

Figura 19 – Animação do Relp no nível 03, Fonte: O Autor

A animação tomou grande parte do tempo de produção deste trabalho e por conta disso, foi possível notar outra das dificuldades presentes no desenvolvimento indie de jogos 2D em Pixel-Art: a Animação. Geralmente, jogos indies com boa Pixel-Art, possuem um integrante do grupo que trabalha mais ou somente naquela área, pois, para manter um estilo de animação e sprites consistentes, é preciso ter técnica e prática frequente, tarefa que um desenvolvedor solo, geralmente, não pode contemplar por falta de tempo. E apesar de todas essas dificuldades, é possível desenvolver se utilizando de pequenos auxílios como o uso de estilos mais simplificados de arte, sendo o Mega Man Style um deles.

3.4.2 Um pouco da Influência da Pixel-Art dos anos 80 e 90 nos jogos Indies atuais

A nostalgia é um sentimento muito forte, e é por conta dela que muitos desenvolvedores indies atuais têm preservado a estética retrô nos jogos, já que, além da nostalgia "vender como água" (DESCONHECIDO, 2019), muitas pessoas que experienciaram os jogos das fascinantes décadas de 80 e/ou 90, desejam reviver esse sentimento através de novos jogos que lembrem aqueles momentos tão especiais de suas vidas, assim como os desenvolvedores destes jogos indies. Exemplos claros disso são jogos que capturam a essência da estética e gameplay retrô, como 99 vidas, Iconoclasts e Shovel Knight, presentes na Figura 20, que contemplam, respectivamente, três dos gêneros mais famosos daquela época: Beat'Em Up, Metroidvania e Plataforma.

Figura 20 – Jogos Indies que capturam a essência retrô, Fonte: Steam

Indo além dos indie game dev's, podemos falar de gigantes da indústria que também são influenciados pelos trabalhos em Pixel-Art dessas épocas, com lançamentos famosos como Octopath Traveller, presente na Figura 21, que trouxe ao público o avanço gráfico tecnológico ao mesmo tempo em que preservou a estética da Pixel-Art dos RPG's do início das décadas de 80 e 90, como o famigerado Chrono Trigger, Figura 22, alvo de diversos elogios por sua inovação em narrativas e design de personagens pelo consagrado Akira Toriyama (TASAKA, 2016).

Figura 21 – Octopath Traveller, Fonte: (MCAULEY, 2018)

Figura 22 – Chrono Trigger, Fonte: (TASAKA, 2016)

Com base nisto, é possível observar que a Pixel-Art das décadas de 80 e 90 causa influência clara no mundo dos jogos até os dias atuais e com a recente ascensão dos jogos indies, essa influência tende a perdurar por longos e bons anos.

3.5 Unity Engine

Como a Unity foi a principal ferramenta no desenvolvimento deste jogo, é justo abordar seu ambiente de trabalho de forma individual ao projeto. Nesta seção, com o auxílio da documentação disponibilizada pelo site da Unity(UNITY, 2025), explicaremos um pouco sobre a interface de desenvolvimento na Figura 23, onde cada letra representa um ponto da interface. Vale ressaltar que a organização da interface na Figura 23 é diferente da nossa, mas isso se deve ao fato de que a interface é manipulável, onde o desenvolvedor tem a opção de mover as abas de lugar, ou mesmo adicionar novas abas, como foi o nosso caso.

Figura 23 – Interface da Unity.

Fonte: Disponível em: <https://docs.unity3d.com/Manual/UsingTheEditor.html>. Acesso em 02 de maio. 2025

- (A) Barra de Ferramentas: Provê acesso à sua conta da Unity e serviços de nuvem. Também contém os controles para o modo de jogo, ação de desfazer, pesquisar ferramentas na Unity, um menu de visibilidade de camadas e o meno do editor de layout;
- (B) Janela de Hierarquia: É uma representação textual de cada GameObject na cena. Cada item na cena tem um nome e ícone na hierarquia, então as duas janelas são ligadas. A hierarquia também revela a estrutura de como os GameObjects se relacionam um com o outro;
- (C) Visualização do Jogo: Simula o que seria uma visão do seu jogo renderizado de acordo com as câmeras posicionadas. Se o botão Play for acionado, a simulação é iniciada e é possível jogar seu jogo;
- (D) Visualização de Cena: Permite visualizar e editar a "cena" do seu jogo; A visualização de cena pode mostrar tanto uma perspectiva 3D como 2D, dependendo do tipo do seu projeto;
- (E) Overlays: Contém as ferramentas básicas para manipular a visualização de cena e os GameObjects presentes nela. Também é possível adicionar Overlays customizadas para melhorar o ciclo de trabalho;
- (F) Inspector: Permite ver e editar todas as propriedades do GameObject selecionado. Por conta da possibilidade de existirem GameObjects com diferentes propriedades internas, o layout e conteúdos da janela Inspector são alterados cada vez que um GameObject diferente é selecionado;
- (G) Janela de Projeto: Mostra a livraria de Assets disponível no projeto. Quando assets novos são importados, eles aparecem lá;

- (H) Barra de Status: Envia notificações de vários processos da Unity, geralmente avisa sobre erros de código;

3.5.1 Unity Engine: Componentes

Cada GameObject adicionado ao projeto geralmente contém algum componente disponibilizado pela Unity, então para esclarecer o que cada um deles é, uma lista foi feita, pois alguns desses termos serão amplamente utilizados no decorrer do trabalho.

- Colliders: São componentes que simulam a colisão e geralmente são colocados em volta dos GameObjects, neste projeto os colisores utilizados foram: Box Collider 2D, Circle Collider 2D e Composite Collider 2D(este foi utilizado apenas na colisão dos ambientes do nível 01).
- Rigidbody: É um componente que simula a gravidade no GameObject em que ele é adicionado, no caso deste projeto foram utilizados componentes do tipo Rigidbody 2D;
- Scripts: São códigos de programação que permitem aos desenvolvedores criar a lógica de interação e a mecânica de seus jogos. Foram utilizamos para controlar a maioria das coisas em no jogo;
- Canvas: O canvas é um GameObject focado na implementação de interfaces visuais. Funciona como uma segunda tela sobre a câmera e é onde posicionamos elementos visuais como por exemplo: botões de menu, contagem de vidas, barras para identificação de dados, etc;
- Animator: Componente que é associado a GameObjects para que estes passem a ter animações. Trabalha em conjunto com um componente chamado Animator Controller, para alterar as animações caso aquele GameObject tenha mais de uma;
- Transform: Responsável pela localização, movimentação, rotação e escala, todos os GameObjects possuem um componente Transform desde sua criação. O Transform permite a manipulação desses atributos de forma individual em cada objeto;

3.6 Estágios do Desenvolvimento de um Jogo

A produção de jogos é um processo demorado conhecido como pipeline, pois a produção precisa ser concluída em uma ordem específica para garantir que seja eficiente e bem-sucedida no final do processo" – assim diz (DENISYUK, 2022), chefe do departamento de produção do Pingle Studio (empresa ucraniana que atua no mercado de desenvolvimento de jogos desde 2007 oferecendo serviços de desenvolvimento, porting, co-desenvolvimento,

arte e animação e teste de jogos), em seu artigo "*What are the stages of game development?*", onde os estágios de produção são descritos e que usaremos para explicar os mesmos nos próximos tópicos começando pela Pré-Produção.

3.6.1 Pré-Produção

O primeiro e mais importante estágio, onde a concepção das ideias é feita. Aqui é onde são estabelecidas as bases de qualidade para que o jogo seja desenvolvido sem problemas, onde são firmados os pilares do projeto e sobre o quê aquele jogo será, a audiência que visam alcançar, onde será publicado, como será distribuído, quanto tempo será gasto desenvolvendo e os recursos necessários para sua produção.

Na pré-produção os artistas conceituais também iniciam os primeiros esboços de como serão os elementos que irão compor o jogo, sendo eles coisas como personagens, ambientes, paletas de cores, etc; Elementos responsáveis pela construção do mundo que os desenvolvedores desejam para aquele jogo, é também nesta etapa que o Documento de Design do jogo(GDD) é produzido, o qual iremos tratar com mais detalhes a frente no trabalho.

Assim que os elementos principais como mecânicas e estilo de gameplay estiverem decididos, se inicia a concepção do protótipo inicial, momento em que o time de desenvolvimento tem a chance de testar o funcionamento daquilo que idealizaram, se os conceitos são mesmo interessantes e se haverá alguma adição ou mudança nas ideias. Dependendo do escopo do projeto, essa etapa pode durar semanas ou até meses.

3.6.2 Produção

Após estabelecer os pilares, metas e ter o protótipo inicial em mãos, a produção do jogo se inicia, este é o processo de maior duração na pipeline e o mais trabalhoso também, pois é diretamente relacionado com o escopo do projeto, quanto maior o escopo, maior o tempo de produção.

Durante a produção ocorre o "refinamento" de vários aspectos do jogo, desde história e personagens até ambientação e efeitos sonoros, etc. Do início da produção até o final da mesma serão desenvolvidos vários protótipos para acompanhamento e revisão das equipes de testes e do produtor para garantir que a gameplay continue funcional e atraente.

Além do refinamento, o processo de produção é dividido em várias sub-etapas que iremos abordar nos próximos tópicos.

3.6.3 Vertical Slice

Uma amostra do projeto com o intuito de ser utilizada para marketing do jogo, geralmente é uma pequena parte pronta para mostrar funcionalidades e a qualidade do produto final, como se fosse uma prévia. A ideia é criar uma seção do jogo que seja totalmente representativa do que os jogadores experimentarão na versão completa, incluindo todos os aspectos principais, como mecânicas de jogo, gráficos, áudio e interface.

3.6.4 Pré-Alfa

Na pré-alfa, são realizados testes pela equipe de desenvolvimento para refinamento e tomadas de decisão (adicionar, manter ou remover aspectos e/ou funcionalidades).Geralmente na pré-alfa o projeto não possui muitos assets terminados, então alguns place-holders são utilizados para que os desenvolvedores consigam seguir no desenvolvimento enquanto os assets do jogo não estão prontos, contudo, apesar disso, é nessa etapa que a "cara" do jogo está sendo definida e com o avançar das etapas do projeto os designs vão sendo aprimorados e refinados, de forma que no final deste, tudo esteja caracterizado de acordo com as ideias iniciais.

3.6.5 Alfa

Este estágio é onde os elementos principais do jogo estão todos prontos e jogáveis: gameplay, física, sons, interface, IA, etc. É possível que alguns assets e efeitos ainda estejam em desenvolvimento, mas geralmente é nessa etapa que o jogo já está pronto para testes com usuários internos, o projeto está quase pronto.

3.6.6 Beta

A última etapa ostensiva do projeto, na Beta os desenvolvedores tem o trabalho de polir e otimizar os conteúdos do jogo: funções principais, menus, mecânicas, IA's, compressão dos arquivos, redução de código, tudo o que for necessário para que o produto final fique o mais "limpo" possível.

3.6.7 Finalização

O jogo finalmente está pronto para comercialização e a partir daqui só são feitas atualizações em casos de bugs específicos, problemas de compatibilidade ou no caso de ser um jogo-serviço, atualizações de conteúdo para manter o público ativo.

4 PROJETO RETROGAME

Este capítulo visa abordar a idealização e características do jogo de forma objetiva, contendo suas mecânicas de jogabilidade, níveis, visuais, história e etc.

4.1 Resumo

Este jogo tem por foco passar a experiência de um jogo 2D com aspectos de alguns jogos 2D das décadas de 80, 90 e meados de 2000, sendo desenvolvido na Unity. O jogo contém 3 níveis que serão jogados de formas diferentes, cada um inspirado em um jogo de gênero e visual diferentes dos anteriores, de forma que os comandos, mecânicas e visuais se alteram no decorrer da jogatina.

4.2 Objetivo

O jogador deve vencer todos os desafios das três fases impostas pelo vilão principal do jogo, enfrentando o mesmo na última fase e terminando o jogo.

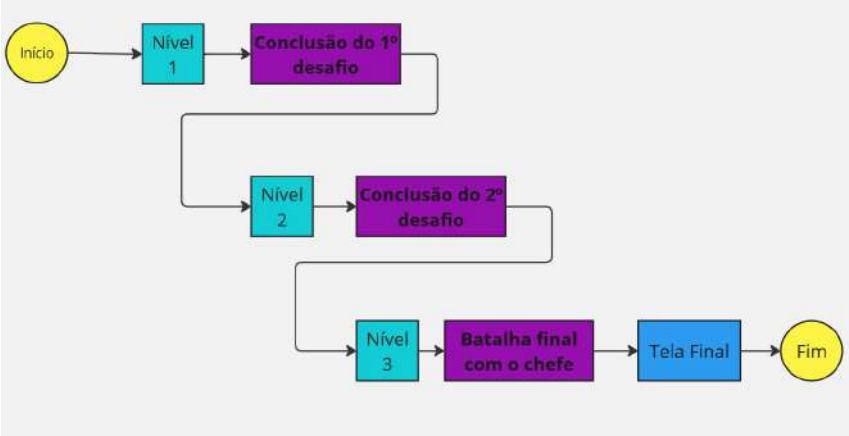
4.3 História

Relp, o protagonista, foi transportado para um mundo de jogos por um fantasma que tem por objetivo prender as pessoas e brincar com elas nesse mundo, portanto o mesmo tem de escapar passando pelos desafios impostos pelo fantasma e enfrentá-lo no final.

4.4 Visão Geral

Esta seção tem por objetivo detalhar aspectos importantes de cada nível do jogo, abordando desde escolhas visuais até mecânicas individuais planejadas para cada nível. Além disso, discorreremos sobre as inspirações em cada um dos níveis e como foram executadas. Na Figura 24, é apresentado um diagrama, no qual é possível vizualizar o fluxo planejado para a jogatina.

Figura 24 – Fluxo de progressão do jogo.



Fonte: O Autor

4.4.1 Nível 01

Para que o jogador pudesse se acostumar melhor com os controles, o primeiro nível deveria ser mais fácil e simples, então a inspiração escolhida foi o jogo Super Mario Bros de 1985, lançado para o NES (Nintendo Entertainment System), mostrado na Figura 25. O objetivo do primeiro nível segue com a mesma proposta, avançar até o final da fase até completá-la, coletando itens e desviando dos obstáculos e inimigos no caminho. Apesar da escolha de desenhá-lo em ”Mega Man Style”, o protagonista é representado em 8 bits assim como Mario.

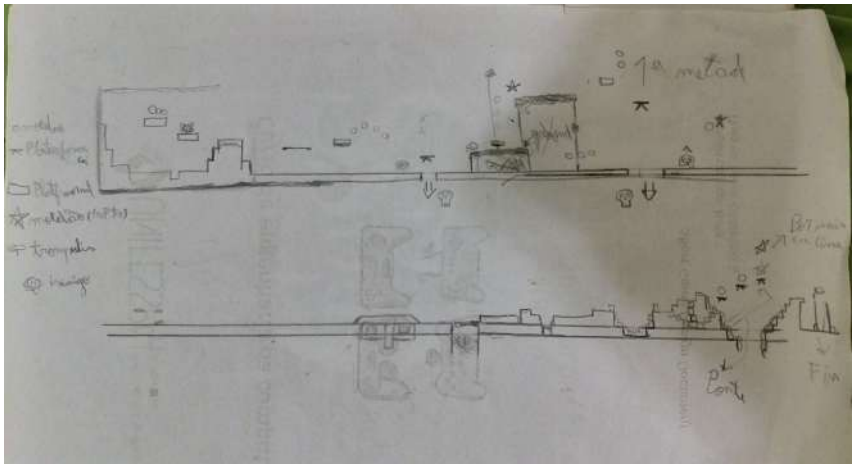
Figura 25 – Super Mario Bros (1985)

Fonte: (SUPERMARIOWIKI, 2025)

Como dito anteriormente, o objetivo era que fosse simples de jogar inicialmente, então o gênero de plataforma foi escolhido. Neste gênero, o jogador só precisa andar para frente ou para trás com os direcionais e pular com o botão de pulo, uma jogabilidade relativamente simples. É possível finalizar o nível sem utilizar as outras mecânicas(moedas e pontos por derrota de inimigo). Contudo, houve a necessidade de introduzir tais mecânicas para auxiliar o jogador a finalizar o nível com mais facilidade.

Após definir o gênero do primeiro nível, o próximo passo é idealizar o mesmo, onde a ambientação escolhida foi uma floresta repleta de obstáculos e inimigos. Na Figura 26 é possível observar a arte conceitual do nível 01.

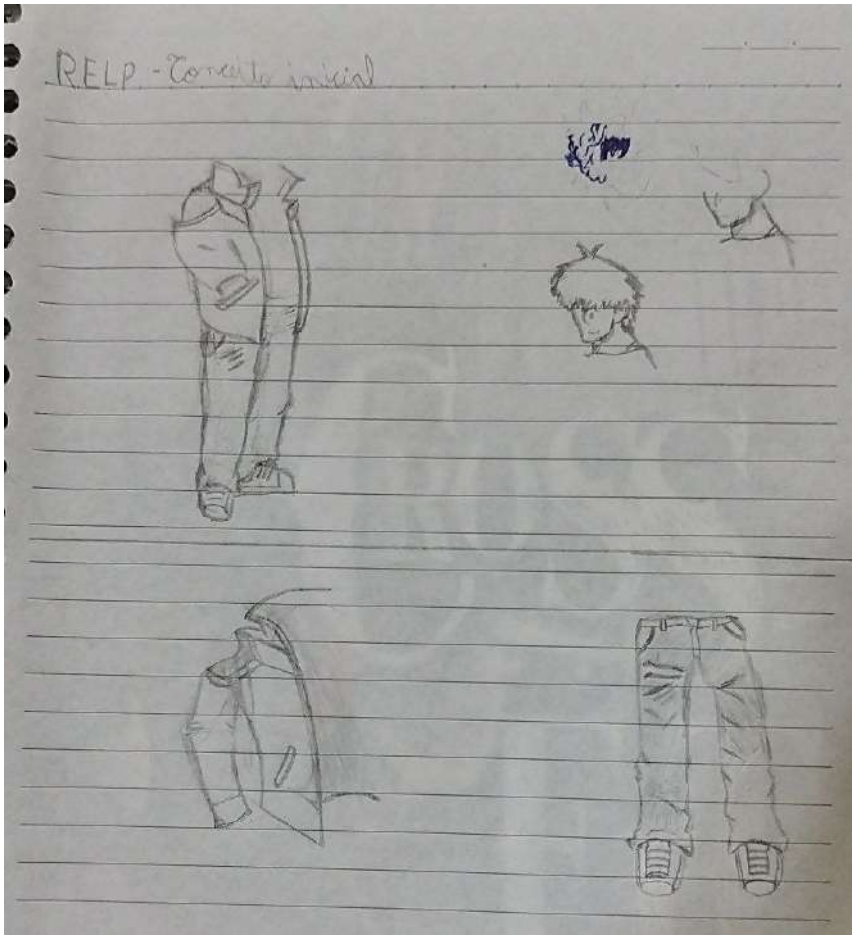
Figura 26 – Esboço do Nível 01.



Fonte: O Autor

No início do desenvolvimento do design do protagonista, suas roupas e aparência foram idealizadas, também foi decidido que Relp seria um rapaz de altura média, cabelos pretos cheios e que usaria como roupas uma camisa verde com aberturas nos ombros e uma calça preta. O objetivo principal era de que o protagonista tivesse características fáceis de identificar na sua animação, então suas características corporais eram poucas, em compensação houve mais destaque nas suas vestimentas. Como é mostrado na Figura 27.

Figura 27 – Design inicial do Relp.



Fonte: O Autor

Inicialmente, Relp teria um cabelo que cobrisse os olhos, porém, posteriormente foi decidido que seu cabelo apenas seria cheio, por conta dos olhos de um personagem se comunicarem muito bem com o jogador. E assim se deu o processo de pré-produção, discutido anteriormente no capítulo 3.5.1, para o primeiro nível.

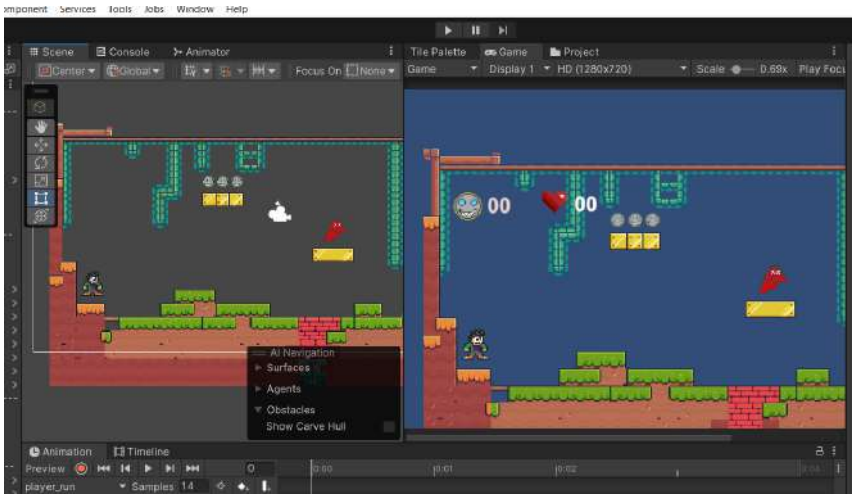
Depois do processo de concepção inicial, prototipação se iniciava e os seguintes detalhes foram decididos:

- Teclas que seriam usadas para o jogador controlar o personagem, sendo estas WASD e/ou setas direcionais para a movimentação e a barra de espaço para que o jogador pule;
- Construção do início do nível, para testar as colisões e a movimentação;
- Criação dos inimigos;
- Sistemas de vidas e de moedas.

Além disso, é válido citar que o primeiro nível passou por várias alterações no decorrer do projeto, já que foi recebendo atualizações para balanceamento da dificuldade de alguns desafios impostos ao jogador, como a sequência de armadilhas colocadas no final que geralmente, ocasionava em fim de jogo. Então, para evitar frustrações desnecessárias, a função de ganhar mais uma vida ao coletar cem pontos(conseguidos coletando moedas e derrotando inimigos) foi adicionada e também foram colocados Checkpoints em locais específicos da fase, para que, apesar da dificuldade da sequência final, o jogador não tivesse que voltar o nível inteiro quando perdesse uma vida e também tivesse mais chances de errar até completar o desafio.

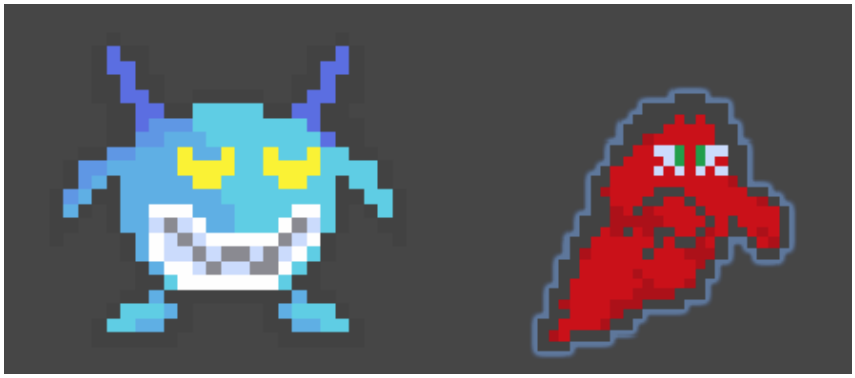
E para finalizar, nas Figuras 28 e 29 estão o protótipo do primeiro nível e a descrição das ações dos inimigos presentes nele:

Figura 28 – Protótipo Nível 01.



Fonte: O Autor

Figura 29 – Inimigos Sympathy e Misery.



Fonte: O Autor

- Misery: Se movimenta indo de um ponto A até um ponto B e repete esse ciclo até ser morto pelo jogador ou encostar nele e matá-lo.

- Sympathy: Se movimenta da mesma forma que o inimigo Misery, porém tem um ataque invencível onde o mesmo pula e nesse momento em qualquer lugar que ele encostar do jogador, vai matá-lo. Colocamos o Sympathy como um inimigo mais forte pois queríamos que o jogador também aprendesse a evitar ameaças mais fortes e fugisse quando sentisse perigo.

4.4.2 Nível 02

Para o segundo nível, seria abordada a perspectiva isométrica e o jogo escolhido como inspiração foi Boktai: The Sun Is In Your Hand (2003), Figura 30, um Dungeon Crawler onde o jogador pode combater inimigos em tempo real e precisa resolver quebra-cabeças para obter recompensas e terminar os calabouços.

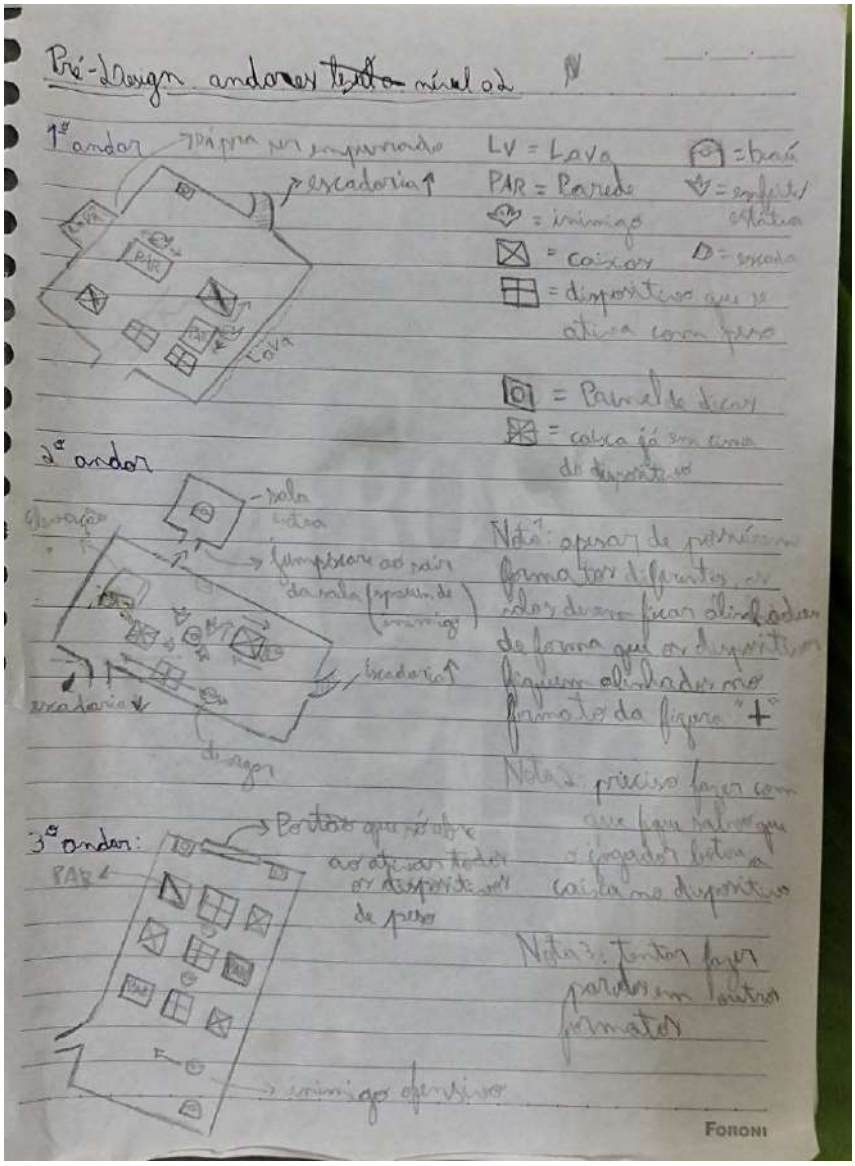
Figura 30 – Captura de um trecho de Boktai - The Sun Is In Your Hand.

Fonte: Disponível em (PIXELHUNTED, 2021)

Para a metade do jogo, o jogador deverá se acostumar com uma jogabilidade completamente diferente, onde agora ao invés de se mover apenas em duas direções(para frente e para trás), o mesmo poderá se mover em 8 direções e terá uma barra de vida maior, sobrevivendo a mais ataques, assim como no jogo tomado como inspiração. Contudo, o jogador perde a opção de pular, mas em seu lugar, utilizando o mesmo botão, o jogador ganha a função de disparar uma arma de energia para destruir seus inimigos e combater o chefe presente neste nível.

Além dos aspectos citados anteriormente, o jogador deverá colocar as caixas presentes no nível em seus respectivos mecanismos para que seja possível passar pela porta da sala do chefe. No processo de pré-produção do segundo nível, é possível observar na Figura 31 o esboço inicial de planejamento do nível e suas mecânicas:

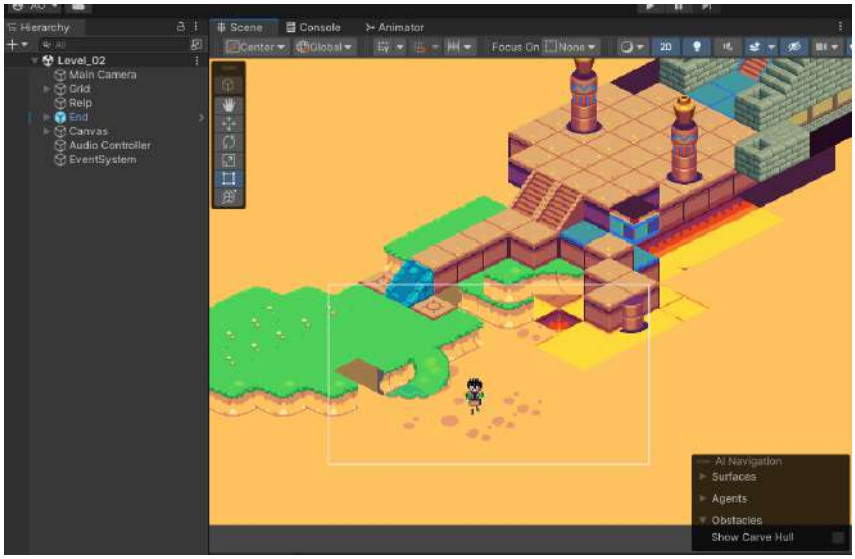
Figura 31 – Esboço do nível 02.



Fonte: O Autor

Como é possível observar no esboço, foram colocadas descrições para cada elemento que deveria estar presente nos andares do castelo, para que no decorrer da construção do nível, tais elementos fossem incluídos ou adaptados. Os inimigos desse nível são apenas Misery's do primeiro nível, contudo agora eles executam duas funções diferentes: alguns ficam à espreita esperando o jogador entrar em seu campo de visão para avançar contra ele e atacá-lo e outros executam o comportamento presente no nível 01, porém agora podendo andar em mais direções por conta da perspectiva isométrica. Na Figura 32 está o protótipo inicial do nível 02. A área que podem observar na figura foi um espaço de testes para estudo da movimentação isométrica e do posicionamento dos sprites, porém posteriormente foi decidido usá-la antes do jogador ser introduzido ao calabouço, para que o mesmo possa se acostumar com os controles.

Figura 32 – Protótipo nível 02.



Fonte: O Autor

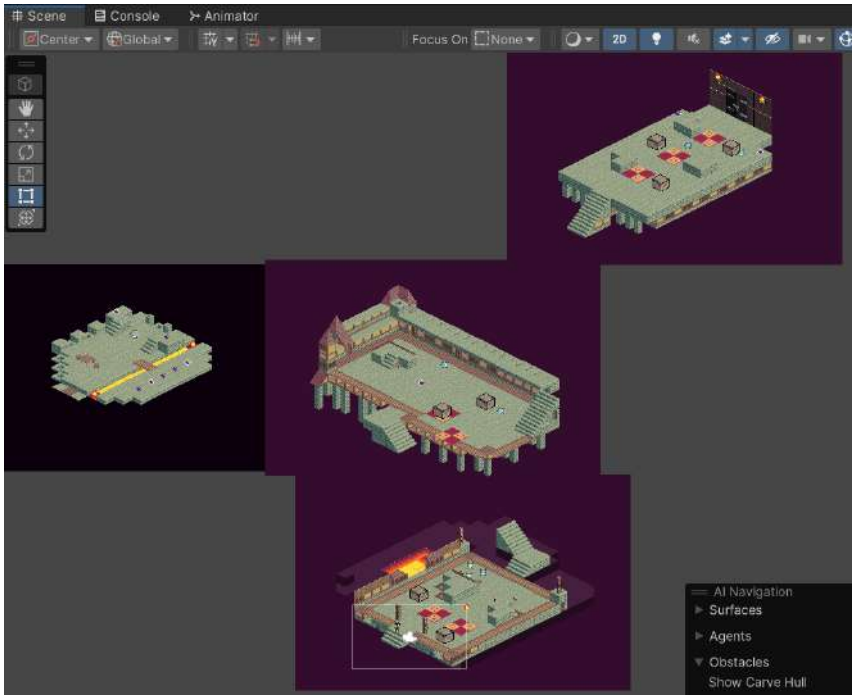
No início de cada nível, há um tutorial e na Figura 33, é dito ao jogador que ele pode encontrar algo interessante caso explore o nível, sendo este segredo a sala secreta, onde há um item que, se o Relp encostar, receberá uma melhoria de recuperação automática dos pontos de vida perdidos.

Figura 33 – Dica para incentivar a exploração do jogador.

Fonte: O Autor

Na Figura 34, está a construção do nível 02 por inteiro, de uma visão mais distante em que é possível observar cada andar do castelo, sendo importante ressaltar a sala secreta à esquerda que só é possível descobrir se o jogador for um pouco curioso.

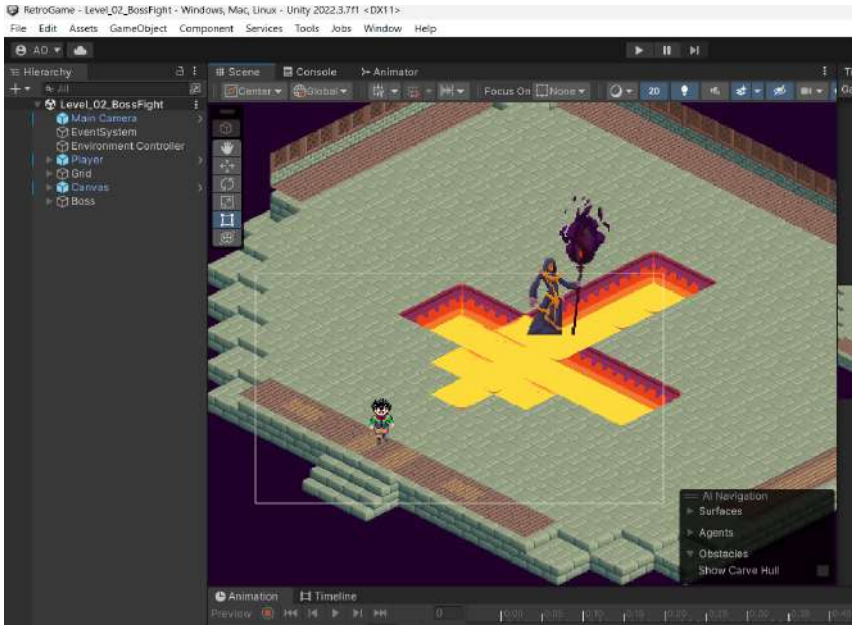
Figura 34 – Visão geral do nível 02.



Fonte: O Autor

Assim como em Boktai, ao resolver os quebra-cabeças e avançar para o final do nível, o jogador terá de enfrentar um chefe que defende aquele calabouço. A construção da sala do chefe foi mais simples, pois a mesma possui somente o piso e o chefe, evidenciados na Figura 35.

Figura 35 – Sala do chefe do nível 02.



Fonte: O Autor

Ao terminar a batalha contra o chefe, o jogador será mandado para uma tela de

vitória e poderá seguir para o terceiro nível.

4.4.3 Nível 03

Para o terceiro e último nível, finalmente teremos o confronto do Fantasma com o protagonista, onde os dois se enfrentarão em uma luta pela libertação ou condenação. Sendo relativamente mais simples que seus antecessores, mas somente em sua proposta, sua execução foi um pouco mais trabalhosa que os demais por conter elementos que contam para os dois oponentes como, por exemplo, a contagem de vitórias que é diretamente ligada à contagem de Rounds e também as barras de ataque especial que iremos discutir adiante.

O jogo tomado como inspiração dessa vez foi Street Fighter 2 (1991), Figura 36, segundo título de uma série de jogos de luta com um nome de peso e fama que perduram até os dias atuais (GOGONI, 2020). Com mais de 30 anos de bagagem, a franquia foi porta de entrada de muitas pessoas para o mundo dos jogos de luta, tanto para os que apenas jogam como aqueles que passaram a criar jogos desse sub-gênero, Street Fighter 2 foi pioneiro de seu gênero e causou euforia nas casas de Arcade tanto quanto os anteriormente citados Pong e Space Invaders, estabelecendo de vez o sub-gênero de jogos de luta competitivos.

Figura 36 – Captura de Street Fighter 2 (1991).

Fonte: Obtida pelo autor no site retrogames.sz

Apesar do gênero ”luta-2D”ser bastante nichado no mundo dos jogos, este costuma ter uma base fiel de fãs para cada título e/ou série, um ótimo exemplo é a série Mortal Kombat que atualmente é a série de jogos de luta mais vendida mundialmente, com um récorde de mais de 83 milhões de unidades vendidas desde seu primeiro lançamento até o ano de 2024 (GUINNESSWORLDRECORDS, 2024).

Outro ponto a destacar em jogos do sub-gênero, principalmente a série Street

Fighter, são seus designs marcantes e seu estilo de arte que vem evoluindo no decorrer dos anos, tornando os personagens cada vez mais atrativos ao público. Quando um personagem é idealizado num jogo de luta, a primeira coisa que se deve pensar é: ”Que características devemos inserir para chamar a atenção do jogador para este personagem apenas pelos detalhes e deixar fácil de identificar a forma como ele luta?”, o desenvolvedor precisa conquistar o jogador nos primeiros segundos em que o personagem é apresentado a ele.

Com base nessa premissa, os personagens presentes no nível 03 deveriam apresentar um visual mais detalhado em comparação com os níveis anteriores. Conforme ilustrado nas Figuras 37 e 38, o personagem Relp passa a contar com todos os elementos visuais de seu esboço incorporados aos sprites, na medida do possível. De maneira semelhante, o Fantasma, antagonista da história, embora não possua uma versão em esboço, foi concebido como uma versão mais poderosa e aterrorizante do inimigo Misery, também um fantasma.

Figura 37 – Sprite Sheet da animação de soco fraco para o personagem Relp.

Fonte: O Autor

É possível observar todos os detalhes no design do personagem, desde a abertura nas mangas da roupa como os cristais vermelhos e as golas altas da jaqueta.

Figura 38 – Nostalghia - O Fantasma.

Fonte: O Autor

A escolha de Street Fighter 2 se deve à sua grande influência no mundo dos jogos como um todo e por ser um jogo que marcou um período importante dos jogos 2D.

Assim, para implementar o máximo possível do que foi estudado sobre o gênero dentro do curto prazo de produção disponível, a jogabilidade deveria, mais uma vez, ser modificada, desta vez da seguinte forma:

- Movimentação: O jogador volta a pular e se movimentar em apenas duas direções(ainda utilizando as teclas WASD), contudo agora o pulo é no direcional para cima ao invés da barra de espaço;
- Combate: Neste nível o jogador enfrentará o fantasma num jogo de luta-2D, então é lógico que o mesmo possua botões para os 3 tipos de golpe: fraco(tecla J), médio(tecla I) e forte(tecla O);
- Barras de Vida e Ataque Especial: o jogador agora possui uma barra de vida que é reduzida a cada golpe sofrido e que ao receber dano, converte uma parte desse dano para a barra de Ataque Especial que serve como uma mecânica de reviravolta;
- Ações específicas: Agora o jogador pode se defender de ataques segurando o botão de defesa(tecla L) e caso sua barra de Ataque Especial esteja cheia, pode apertar o

botão de Ataque Especial(tecla R) para desencadear um ataque que causa bastante dano no oponente.

- Rounds e Contador de tempo: para finalizar o nível 3, o jogador ou o fantasma tem de vencer 2 Rounds, estes são contabilizados por nocaute ou por fim da contagem de 60 segundos(onde o personagem com mais vida no final vai receber a vitória do Round).

É possível conferir os detalhes da lista na Figura 39:

Figura 39 – Jogatina mostrando os elementos do nível 03.

Fonte: O Autor

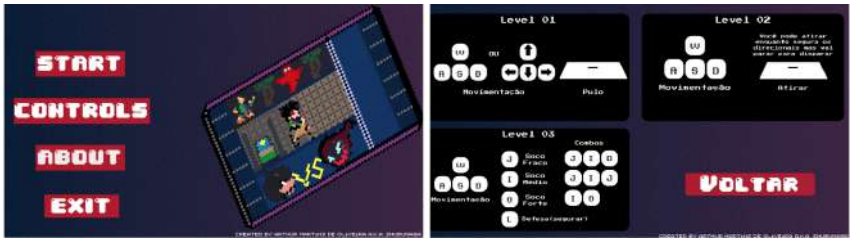
Ao nocautear o chefe duas vezes, o jogador é mandado para a tela de vitória e finaliza o jogo; caso contrário, a tela de fim de jogo é apresentada e o jogador pode tentar novamente até vencer o chefe.

4.4.4 Menus

Assim como a maioria dos jogos convencionais, este jogo precisa de menus para navegação antes e durante a jogatina, então foram implementadas telas de menu para a tela inicial(Figura 40) e para o menu de pausa enquanto o jogador está jogando os níveis(Figura 41). As seguintes opções estão disponíveis nessas telas:

- Iniciar jogo: Presente apenas na tela inicial, esta opção dá início ao jogo;
- Controles: tela que mostra os controles, se o jogador clicar nessa opção na tela inicial ele verá os controles para cada nível, contudo se abrir o menu de pausa no meio da jogatina verá apenas os controles do nível que ele está jogando;
- Sair: Na tela inicial esta opção fecha o jogo, já nas telas de pausa essa opção leva o jogador de volta à tela inicial.

Figura 40 – Tela inicial e de controles no menu principal.



Fonte: O Autor

Figura 41 – Tela de pausa.

Fonte: O Autor

5 IMPLEMENTAÇÃO

Neste capítulo, estão parte das etapas de implementação do jogo, indo desde Pixel-Art até programação e os sistemas do nosso jogo. Cada seção a seguir abordará esses processos de acordo com cada nível, de 01 a 03.

Conforme dito no capítulo 3.2, o desenvolvimento desse jogo seguiu um método empírico e após finalizar os protótipos de cada nível, balanceamentos e correções de bugs foram feitos até que o projeto estivesse completo para, no final, os menus serem incorporados ao restante do jogo.

5.1 Nível 01

Nesta seção estão o desenvolvimento, pontos importantes e técnicas utilizadas no desenvolvimento do nível 01 deste jogo.

5.1.1 Scripts e Componentes do Relp

Toda a programação dos três níveis foi feita em C# Script e nesta subseção o funcionamento dos scripts do protagonista será abordado, detalhando as funções e métodos utilizados.

Para condizer com a proposta de inspiração do primeiro nível, o jogo Super Mario Bros, nosso protagonista só poderia executar as simples ações de: correr, pular e interagir com itens. Ações essas que ocorreriam pelo contato dele com o cenário e os inimigos. Na Figura 42, podemos observar o objeto player e seus componentes no inspector e na Figura 43, os scripts que ele possui são mostrados com mais destaque, sendo possível notar que as variáveis públicas podem ser alteradas a qualquer momento no inspector. No nível 01, Relp possui um componente Animator para alterar os sprites durante o jogo, um Box Collider 2D, para detectar colisão com inimigos, um Circle Collider 2D para detectar sua colisão com o chão e um Rigidbody 2D para fazer com que ele simule os efeitos da gravidade, como peso e queda. Além dos componentes anteriormente citados, ele também possui 2 scripts, um para sua movimentação 5.1 e outro para suas vidas A.1.

Figura 42 – Relp e seus componentes, Fonte: O Autor

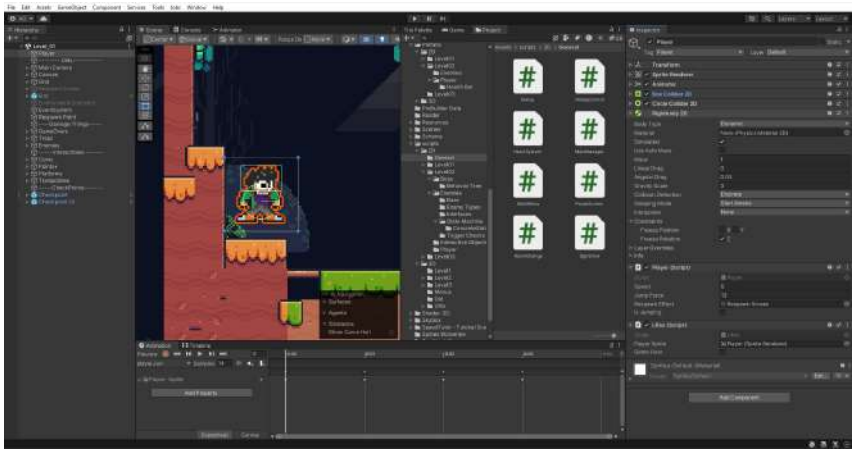
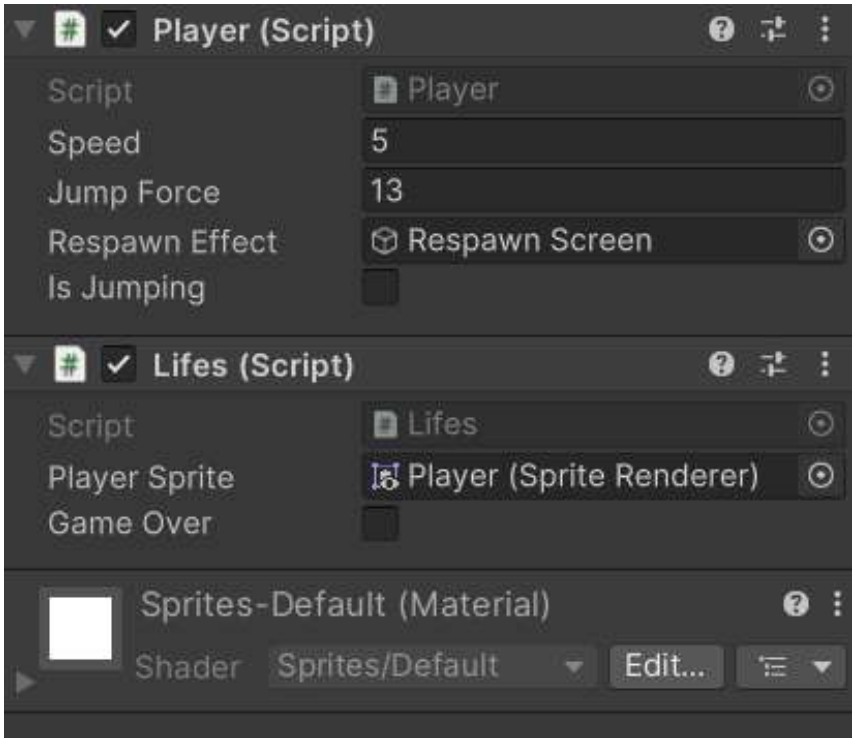


Figura 43 – Variáveis do Relp no nível 01, Fonte: O Autor



Programamos a velocidade e força de pulo do Relp e, além disso, havia um efeito de "Respawn"(reaparecimento), que seria ativado quando Relp morresse. Esse efeito se trata de um objeto chamado Respawn Screen, que consistia de um sprite 2D em branco com uma animação de aparecer e desaparecer, esse objeto era brevemente ativado quando o Relp morria, mostrando a animação na tela do jogador enquanto transportávamos Relp para o último ponto de renascimento definido. Já o script Lifes A.1 ficava encarregado de lidar com a contagem de vidas e o fim de jogo.

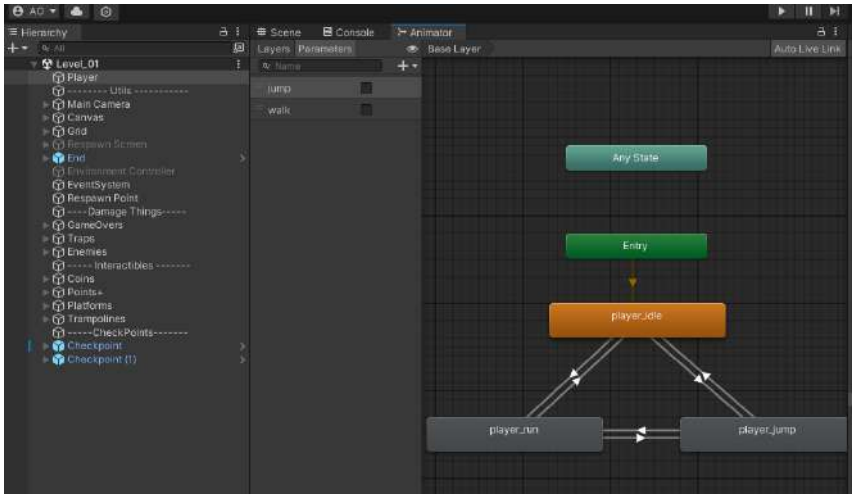
A movimentação funcionava por meio de 2 funções simples, Move e Jump, demonstradas no script (5.1) abaixo, chamado *Player*.

Listing 5.1 – Funções Move e Jump do script “Player”

```
1   void Move()
2   {
3       Vector3 movement = new Vector3(Input.GetAxis("Horizontal"), 0f,
            0f);
4       transform.position += movement * Time.deltaTime * Speed;
5
6       if(Input.GetAxis("Horizontal") > 0f)
7       {
8         anim.SetBool("walk", true);
9         transform.eulerAngles = new Vector3(0f,0f,0f);
10      }
11
12      if(Input.GetAxis("Horizontal") < 0f)
13      {
14          anim.SetBool("walk", true);
15          transform.eulerAngles = new Vector3(0f,180f,0f);
16      }
17
18      if(Input.GetAxis("Horizontal") == 0f)
19      {
20          anim.SetBool("walk", false);
21      }
22  }
23
24  void Jump()
25  {
26      if(Input.GetButtonDown("Jump")  && !isJumping)
27      {
28          rig.AddForce(new Vector2(0f, JumpForce), ForceMode2D.Impulse
                );
29          AudioController.current.PlayMusic(AudioController.current.
                jump);
30          anim.SetBool("jump", true);
31      }
32  }
```

A função Move funciona registrando se o jogador está pressionando algum direcional(teclas direcionais ou as teclas A e D) para esquerda ou direita. Quando isso ocorre, o componente Transform do Relp se movimenta na direção correspondente e a variável “walk”, Figura 44, do objeto player recebe o valor True, fazendo com que o jogador veja a animação do Relp andando. Caso o jogador não aperte nenhum botão, o animator ficará na animação do Relp parado (“player idle”), que é a padrão.

Figura 44 – Variáveis e Funcionamento do componente Animator do Relp no nível 01.



Fonte: O Autor

Além destas, há a função Jump que adiciona uma força de impulso no corpo do Relp fazendo com que ele pule. Para que Relp morra, ele precisa tocar num objeto com a tag “GameOver” ou “Enemy”, caso aconteça, ele vai renascer no início da fase ou no último checkpoint, como é possível observar no trecho (5.2) que é parte do script do Player(possível checar por completo no repositório no link no fim do trabalho):

Listing 5.2 – Método de colisão e função de renascimento no script “Player”

```
1
2     void OnCollisionEnter2D(Collision2D collision)
3     {
4      if(collision.gameObject.tag == "GameOver")
5      {
6          playerGo.loseLife();
7          AudioController.current.PlayMusic(AudioController.current.
               deathSFX);
8          Die();
9      }
10    }
11
12    IEnumerator Respawn()
13        {
14            rig.velocity = Vector3.zero;
15            playerSprite.enabled = false;
16            rig.simulated = false;
17            respawnEffect.SetActive(true);
18
19
20            yield return new WaitForSeconds(0.5f);
21
22
```

```
23              respawnEffect.SetActive(false);
24              transform.position = checkPointPos;
25              playerSprite.enabled = true;
26
27              rig.simulated = true;
28
29          }
```

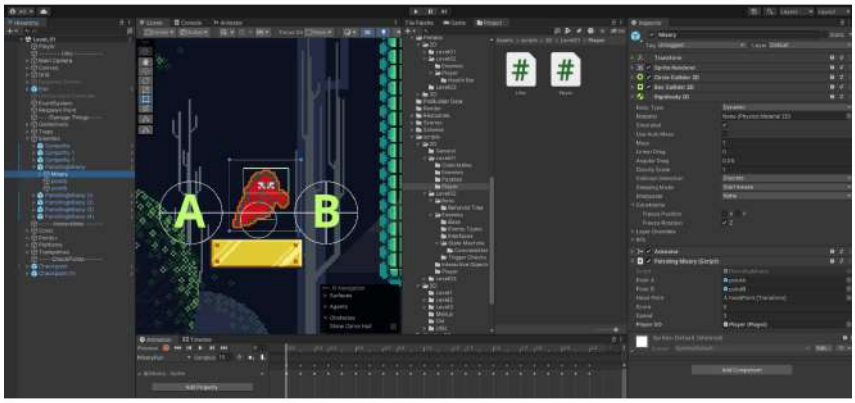
As Funções Move e Jump estão dentro do método Update, que os executa uma vez a cada frame. As demais funções e métodos são executados apenas quando há colisão do Relp com os elementos necessários, como inimigos e armadilhas. No apêndice A) há um exemplo de como isso ocorre, com sua função LoseLife, onde é verificado se o Relp ainda tem vidas extras, se não tiver, o GameOver é mostrado ao jogador.

5.1.2 Scripts e Componentes dos Inimigos

O primeiro inimigo criado para o jogo se chama Misery e seus componentes são praticamente os mesmos do Relp, com exceção do script de vidas, já que este morre com apenas 1 ataque e não retorna mais a menos que o jogador morra até chegar no estado de GameOver, assim resetando a fase. Misery tem somente 1 script, chamado PattrollingMisery (Apêndice B), ele faz com que Misery se mova de um ponto A até um ponto B, e retorne ao ponto anterior refazendo o caminho até ser morto. É possível observar os pontos de patrulha do Misery na Figura 45.

A lógica ocorre no método Update, checando se o ponto atual que o inimigo está é o ponto B ou ponto A, que são 2 objetos filhos do tipo Transform dentro do objeto pai (Misery). Se Misery iniciará se dirigindo até um destes pontos, recebendo valor de velocidade positiva(direita) para ir em direção ao ponto B e negativa(esquerda) quando chegar ao ponto B, para que o inimigo se mova em direção ao ponto A.

Figura 45 – Pontos de patrulha do Misery.



Fonte: O Autor

Presente nos scripts dos dois inimigos, também há um trecho encarregado de matá-los ou matar o jogador (apêndice A). Se o Relp pular na cabeça do inimigo causando colisão numa pequena caixa em cima destes, o inimigo é destruído, caso contrário o objeto player(Relp) é destruído, pois a colisão será fora da área segura, e então uma vida será retirada e ele retornará ao último checkpoint. Isto ocorre pela chamada da função Die que ativa o renascimento do Relp(corrotina Respawn) caso este ainda possua vidas extras.

Ao contrário de Misery, Sympathy é um personagem mais ativo e só para de se movimentar em uma direção quando encosta numa parede, ao tocar num componente com a layer “ground”, além disso ele também tem o ataque de pulo que é ativado a cada certa quantidade de segundos, seu pulo também funciona da mesma forma que o do Relp, exceto que Sympathy se torna invencível ao fazer isso. Seu script está disponível para consulta no Apêndice (B) e na Figura 46 é possível vê-lo em ação.

Figura 46 – Inimigo Sympathy.

Fonte: O Autor

5.1.3 Scripts das Mecânicas de Nível e Cenário

As únicas mecânicas presentes no nível 01 são os checkpoints, armadilhas, moedas coletáveis e o efeito parallax. O script dos checkpoints (Listing 5.3) funciona de forma simples, quando Relp encosta neles é definido um novo ponto de renascimento.

O processo funciona da seguinte forma: As linhas 7 até 15 ocorrem no script Checkpoint, a linha 11 utiliza uma instância do player(Relp) para chamar a função UpdateCheckpoint presente no script do Relp, que captura as coordenadas atuais dele no momento em que este entra em contato com um objeto de checkpoint. Após isso, a colisão com aquele checkpoint é desativada, a partir dali, Relp irá renascer naquele ponto.

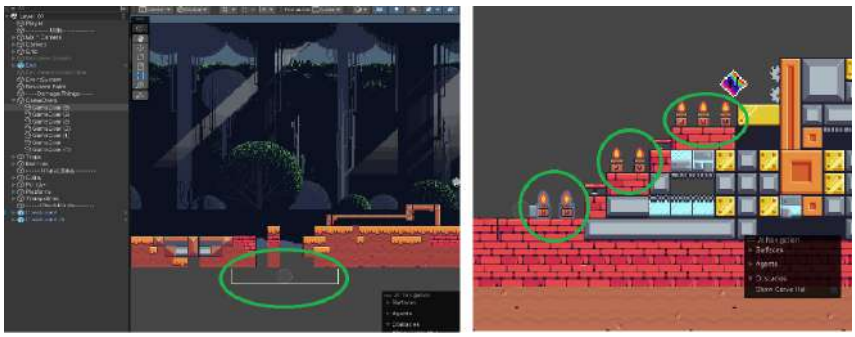
Listing 5.3 – Trechos do Script Checkpoint, Fonte: O Autor

```
1
2     public void UpdateCheckpoint(Vector2 pos)
3     {
4         checkPointPos = pos;
5     }
6
7      private void OnTriggerEnter2D(Collider2D collision)
8      {
9          if (collision.CompareTag("Player"))
10         {
11             playerChecked.UpdateCheckpoint(respawnPoint.position);
12             checkPointAnim.SetTrigger("CheckPoint");
```

```
13                AudioController.current.PlayMusic(AudioController.current.
                      ExtraPoints);
14
15                checkColl.enabled = false;
16            }
17        }
```

Já o tratamento das armadilhas, faz parte de um trecho de script do jogador e funciona da seguinte forma: Dentro do método OnCollisionEnter2D (método disponibilizado pela Unity que detecta a colisão entre objetos que possuem um colisor do tipo 2D), detectamos se Relp encostou em um objeto com a tag GameOver. Alguns exemplos podem ser vistos na Figura 47. Caso o faça, Relp perderá uma das vidas e a função Die será ativada chamando a função Respawn da instância do script player, função essa que retorna o Relp para a posição do último checkpoint ativado (caso o jogador ainda não tenha ativado nenhum, ele retorna para o início do nível).

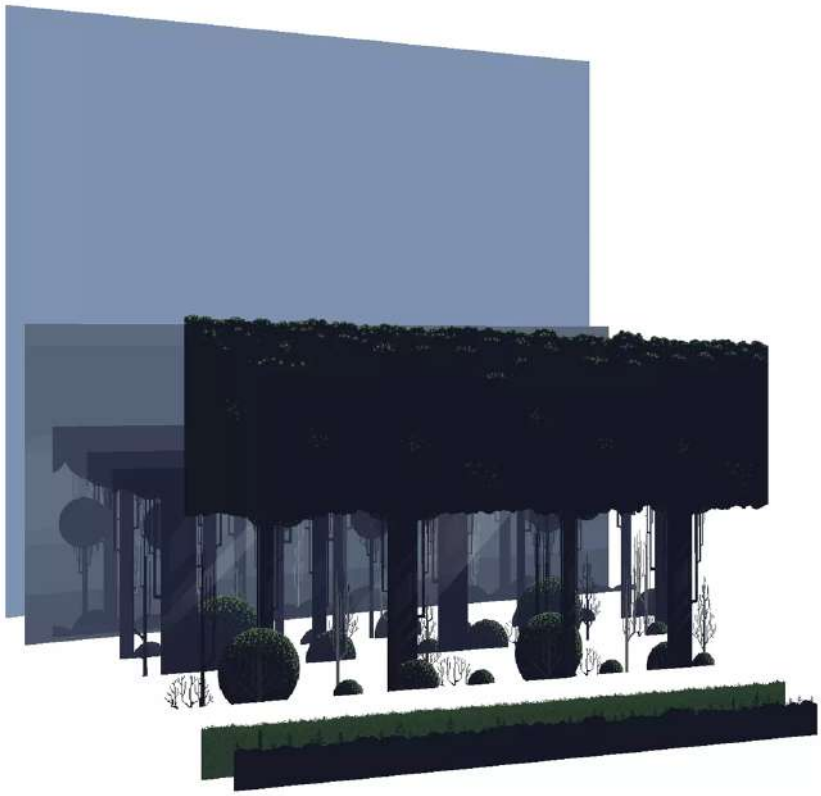
Figura 47 – À esquerda, caixa de colisão do vão que ocasiona a morte do Relp em caso de queda e à direita armadilhas.



Fonte: O Autor

Além destes recursos que interagem diretamente com o Relp, temos o efeito parallax no fundo do cenário deste nível, que interage indiretamente, fazendo o jogador ter uma sensação mais realista de que está avançando pelo cenário. O efeito parallax consiste em colocar algumas imagens sobrepostas e fazer com que se movam de acordo com a movimentação do personagem controlado pelo jogador, para causar o efeito de que o cenário está sendo deixado para trás conforme ele caminha. Obtivemos os sprites da Figura 48 na *Unity Asset Store* (MUNIZ, 2025b).

Figura 48 – Pacote de Efeito Parallax obtido na Unity Asset Store.



Fonte: (MUNIZ, 2025a)

O script do parallax trabalha em conjunto com o da câmera, portanto este deve ser abordado previamente. Chamado de CameraFollow2D (5.4), o script atua seguindo o componente transform do Relp dentro de limites pré-definidos nas variáveis da linha 04, para que a câmera não ultrapasse a visão desejada para o jogador e nem fique para trás, também seguindo o efeito parallax que ocorre dentro e fora dos limites da câmera.

Tudo ocorre dentro do método FixedUpdate que, diferente do Update convencional, é chamado apenas em intervalos fixos, o que o torna ideal para manipular a câmera de forma que pareça mais natural. Inicialmente, a nova posição da câmera é calculada, colocando-a na mesma posição que o transform do Relp, na linha 12, seu valor em Y também é colocado em 0.2, para que ela não fique muito alta, já que caso o Relp esteja pulando, ela também vai subir e se subir demais, pode revelar onde acabam os sprites do cenário. Após isso, a movimentação da câmera é suavizada com a função Lerp, interpolando linearmente o valor da posição atual da câmera com o valor da nova posição calculada, para que a câmera siga a movimentação controlada pelo jogador e não ande diretamente com ele(linhas 16 e 17). Por último, limitamos a movimentação da câmera nos eixos X e Y, para que ela não fique para trás e nem avance mais do que o necessário, é desejado que ela esteja sempre centrada no Relp.

Listing 5.4 – Script CameraFollow2D, Fonte: O Autor

```
public class CameraFollow2D : MonoBehaviour
{
    public Transform Player;
    public float minX, maxX, minY, maxY;

    //Camera movimentation
    public float timeLerp;

    private void FixedUpdate()
    {
        Vector3 newPosition = Player.position + new Vector3(0,0,-10);
        newPosition.y = -0.2f;

        transform.position = newPosition;

        newPosition = Vector3.Lerp(transform.position, newPosition,
            timeLerp);
        transform.position = newPosition;

        newPosition.y = Mathf.Clamp(newPosition.y, minY, maxY);

        transform.position = new Vector3(Mathf.Clamp(transform.position.
            x, minX, maxX),
         transform.position.y, transform.position.z);

    }
}
```

Enquanto o script CameraFollow2D funciona seguindo a movimentação do Relp, o script do parallax segue a câmera. Possuímos 2 scripts de parallax, fazemos um para o cenário, que tem de se mover mais rápido (árvores e luzes) e outro que deve se mover mais devagar (vegetação e ambiente de fundo). O script ”Parallax”(Apêndice (C)) é do primeiro tipo e funciona da seguinte forma: O sprite do GameObject em que este código é atribuído começa a movimentar sua posição inicial seguindo o jogador e faz isso calculando o valor da posição da câmera vezes a velocidade para aquele sprite, depois o valor da posição do componente transform do GameObject é ajustado atribuindo um novo Vector3 com os valores da posição inicial mais a distância percorrida. Por último, se a variável dist tiver chegado no tamanho máximo do sprite, os sprites daquele parallax serão transportados para frente, perpetuando o cenário de fundo. O truque está em reposicionar os sprites cada vez que o Relp se aproximar do final de sua extensão, pois os sprites do parallax são finitos. A diferença entre os dois scripts é que o primeiro script faz os sprites se movimentarem mais rápido que o segundo.

Por último iremos falar sobre os coletáveis do nível 01, as moedas. Seu script, chamado Coin (5.5), funciona de forma inversa ao do Relp, caso um objeto moeda colida

com um objeto que possui a tag player, este é destruído e contabilizado nos dados do objeto player, por meio do script EnvironmentController, responsável por diversas funções que iremos descrever a seguir.

Listing 5.5 – Trecho do script responsável pelos itens coletáveis, Fonte: O Autor

```
1
2     void OnTriggerEnter2D(Collider2D collider)
3     {
4         if(collider.gameObject.tag == "Player")
5         {
6             sr.enabled = false;
7             circle.enabled = false;
8             collected.SetActive(true);
9             AudioController.current.PlayMusic(AudioController.current.
                  Coin);
10
11            EnvironmentController.instance.playerScore +=  Score;
12            EnvironmentController.instance.UpdateScoreText();
13            //
14            Destroy(gameObject, 0.3f);
15
16            if(EnvironmentController.instance.playerScore >= 99)
17            {
18                EnvironmentController.instance.SetLives(+1);
19                //EnvironmentController.instance.playerLifes += 1;
20                EnvironmentController.instance.playerScore -= 100;
21
22                //
23            }
24
25        }
26    }
```

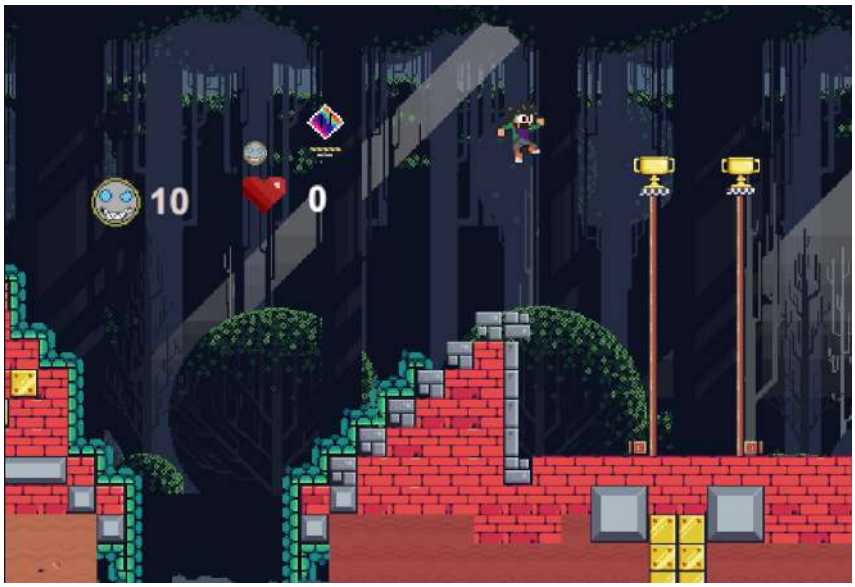
Como seu nome diz, o script EnvironmentController (Apêndice (C)) é utilizado para controlar o ambiente e isto é feito da seguinte forma:

- Método Start: Chamado apenas 1 vez a cada carregamento do nível, nessa parte do script a pontuação do jogador é inicializada e caso essa seja zero, o tutorial do nível 01 é acionado, além disso, adicionamos o evento de reiniciar o nível para o click no botão de Restart, que só aparece para o jogador caso ele perca todas as vidas;
- Função SetLives: Chamado quando Relp coleta 100 moedas, adicionando uma vida extra;
- Função UpdateScoreText: Chamado cada vez que o Relp coleta uma moeda, contabilizando as mesmas e sendo utilizado num componente do tipo text para mostrar o

valor da pontuação na tela do jogador;

- Função ShowGameOver: Chamado apenas quando o Relp esgota suas vidas, ativando a tela de fim de jogo que é um GameObject;
- Função RestartGame: Chamado quando o jogador clica no botão de Restart para iniciar o nível novamente;
- Função ShowWinningScreen: Chamado quando o Relp chega no final do nível ao pular nas bandeiras no final do nível (Figura 49);

Figura 49 – Bandeiras que representam o fim do primeiro nível.



Fonte: O Autor

5.2 Nível 02

Na programação do nível 02 surgiram alguns problemas no momento de programar a movimentação do protagonista, pois pelo estilo isométrico exigir uma movimentação diferente, Relp agora deveria ser capaz de se mover não apenas em duas, mas sim em oito direções, como é possível observar nos sprites da Figura 50.

Figura 50 – Sprites do Relp em 8 direções.

Fonte: O Autor

5.2.1 Scripts de Movimentação e Animação do Relp Isométrico

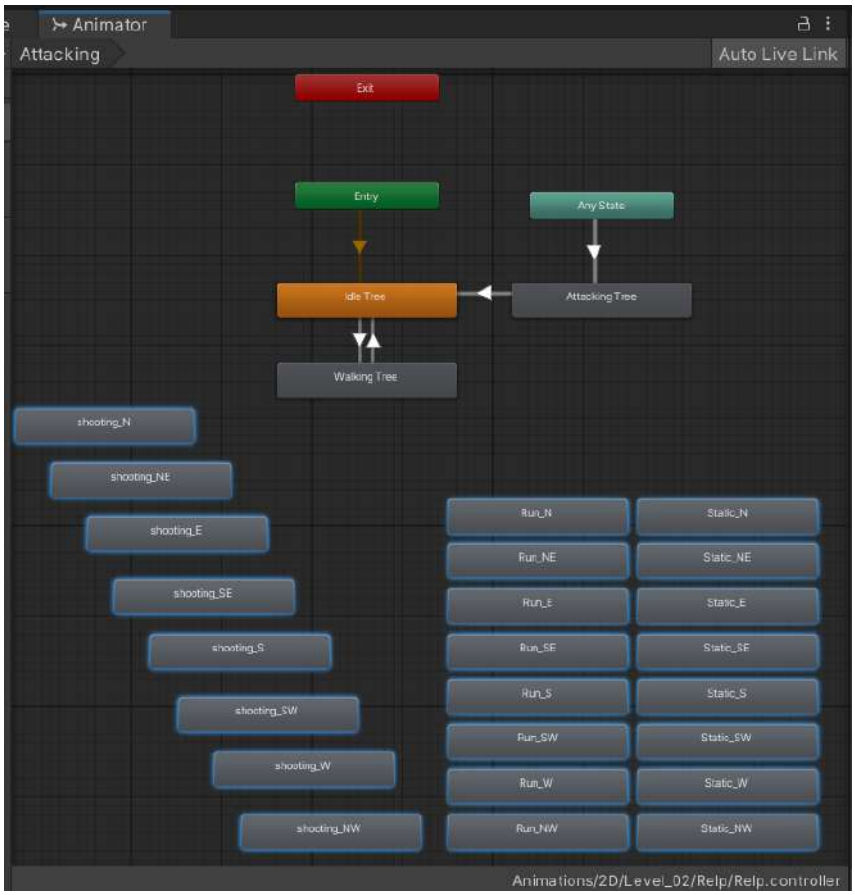
Para tratar da animação e movimentação do Relp no segundo nível, foi necessário recorrer a uma máquina de estados, recurso utilizado para organizar e tratar de um processo complicado, simplificando-o.

Uma máquina de estados consiste na criação de um modelo abstrato de um sistema onde, o sistema deve se comportar de forma adequada para cada entrada. A máquina de estados criada para o Relp possui dois estados principais: andar e atacar, inicialmente parece ser algo relativamente simples, contudo a problemática entra na relação direção-animação, pois o personagem agora tem três tipos de animação para cada uma das oito direções(parado, andando e atirando). Para que isso funcionasse, foi necessário fazer os scripts trabalharem em conjunto com o animator do Relp.

Organizados os estados nos dois, as escolhas de sprites ativos seriam definidas de acordo com os valores obtidos nas coordenadas X e Y, ainda obtidas ao apertar as teclas WASD ou direcionais. No animador o processo é mais visual, na Figura 51 é possível observar à esquerda os estados de animação para as direções dos tiros e à direita os estados para o personagem parado e andando, sendo as iniciais N, NE, E, SE, S, SW, W e NW

correspondentes às direções: norte, nordeste, leste, sudeste, sul, sudoeste, oeste e noroeste.

Figura 51 – Componente Animator do personagem Relp e seus estados no nível 02.

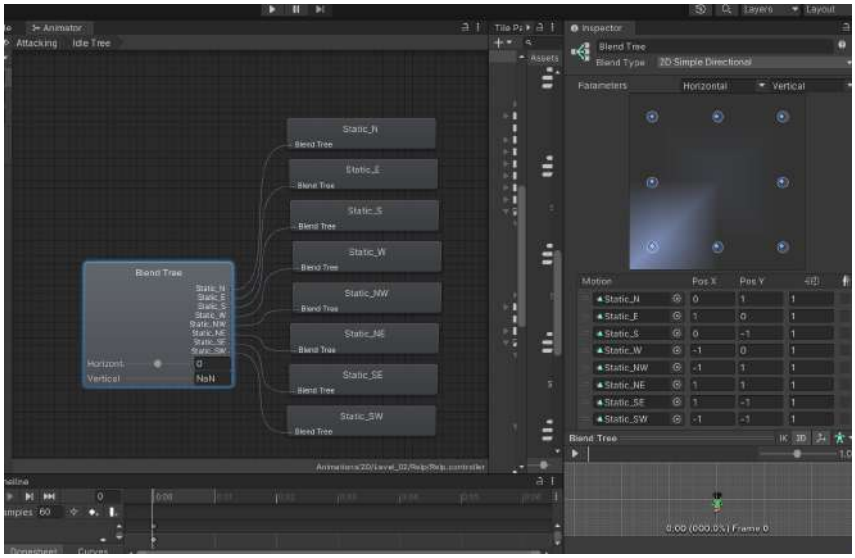


Fonte: O Autor

As animações se mantêm ativas pelos estados que estão na imagem acima, porém elas alternam entre si por meio de Blend Trees (Figura 52). Uma Blend Tree (Árvore de Combinação) é uma ferramenta utilizada quando há a necessidade de alternar entre várias animações de forma fluida, de forma que pareçam naturais. Na figura abaixo está a Blend Tree das animações do protagonista parado, neste caso são apenas frames estáticos, porém a troca de animações ocorre de acordo com as variáveis “Horizontal” e “Vertical” obtidas a partir do código que discutiremos mais tarde.

Existe uma Blend Tree para cada ação do protagonista no nível 02, a da Figura 52 é a árvore para o Relp enquanto parado, nomeada de Idle Tree, e na Figura 51 é possível observar que também existem as árvores Attacking Tree e Walking Tree.

Figura 52 – Blend Tree do Relp estático.



Fonte: O Autor

O código PlayerMovement (Apêndice (A) possui as funções que regem o comportamento das animações e movimentação do Relp, e funciona da seguinte forma:

- Enum PlayerState: variável do tipo “enum”(enumerate) que separa os estados possíveis para o Relp;
- Método Update: Neste método detectamos se o jogador apertou o botão de ataque e iniciamos a corrotina de ataque para que o Relp fique parado por um momento e atire;
- Método Fixed Update: Os dados da movimentação são capturados aqui pela variável change que é do tipo Vector3(coordenadas X,Y e Z). Utilizamos a função Input.GetAxisRaw para obter as coordenadas com valores normalizados indo de -1 até 1, isso era utilizado na nossa Blend Tree, mostrado na Figura 52, as combinações de direções envolvem números nessa faixa, como por exemplo a direção Static N(Norte), onde devemos possuir os valores: 0 para posição X e 1 para posição Y(personagem olhando para o norte pois o jogador movimentou o Relp para cima);
- Corrotina AttackCo: alterna o valor da variável booleana attacking no animator do Relp para verdadeiro e o deixa estacionado por um breve momento para realizar a animação de tiro, após isso retorna a variável ao seu estado de falso para que o Relp volte ao estado walk;
- Função UpdateAnimationAndMove: Detecta se há mudanças na variável change(jogador apertando os direcionais para se movimentar) e avisa ao animator que o Relp está andando, setando a variável booleana “moving” como verdadeira e enviando os valores das coordenadas X e Y para as Blend Trees do animator;

- Função MoveCharacter: Este de fato movimenta o Relp pelo cenário. Obtendo os valores dos inputs horizontais e verticais nas linhas 82 e 83 e criando um novo vector2 com esses valores, após isso essas coordenadas são utilizadas para impulsionar o componente rigidybody do Relp utilizando o valor da velocidade vezes o valor do tempo;
- Função MechanismActivation:: Esta função serve para guardar dentro do Relp a informação de quantos mecanismos ele ativou para que a porta da sala do chefe seja liberada.

5.2.2 Scripts dos Inimigos

Dois scripts foram usados para os inimigos do nível 02, um faz o inimigo correr em direção ao Relp caso ele esteja na sua "área de vigilância" e o outro é semelhante ao script de patrulha do inimigo Misery do primeiro nível.

Iremos falar a respeito do script de perseguição primeiro. Existem dois tipos de Misery no nível 02, o tipo 01 persegue o Relp assim que ele entra no raio de um círculo que fica em volta dele, caso Relp consiga se distanciar o suficiente, o inimigo para de persegui-lo e fica "estacionado" no local esperando-o entrar em seu círculo de caça novamente. Os métodos e funções presentes no script desse inimigo funcionam da seguinte forma:

- Função SearchPlayer: Verifica se há colisão de um objeto com a tag Player no raio do círculo do colisor dentro do objeto inimigo, caso aconteça, permanece capturando as coordenadas do componente transform do Relp e armazenando estas na variável target que é utilizada no método Update como requisito para ativar a função de perseguição;
- Função Move: Utilizado para perseguir o Relp, move o componente rigidy body do inimigo em direção às coordenadas do Relp-(variável target) enquanto este está no raio do círculo de caça, caso contrário utiliza a função StopMoving para estacionar o inimigo no local.

O inimigo do segundo script no entanto, tem um comportamento mais pacífico, ele não ataca o Relp mesmo sendo atacado, porém ficará rondando uma área enquanto estiver vivo e, caso o jogador seja descuidado, Relp vai acabar recebendo dano dele, já que ele se move numa velocidade maior. O inimigo tipo 02 se move entre 04 pontos, A, B, C e D. Quando chega no último ponto ele retorna ao primeiro e repete o ciclo até que seja interrompido. Os métodos presentes no trecho do script B.4 funcionam da seguinte forma:

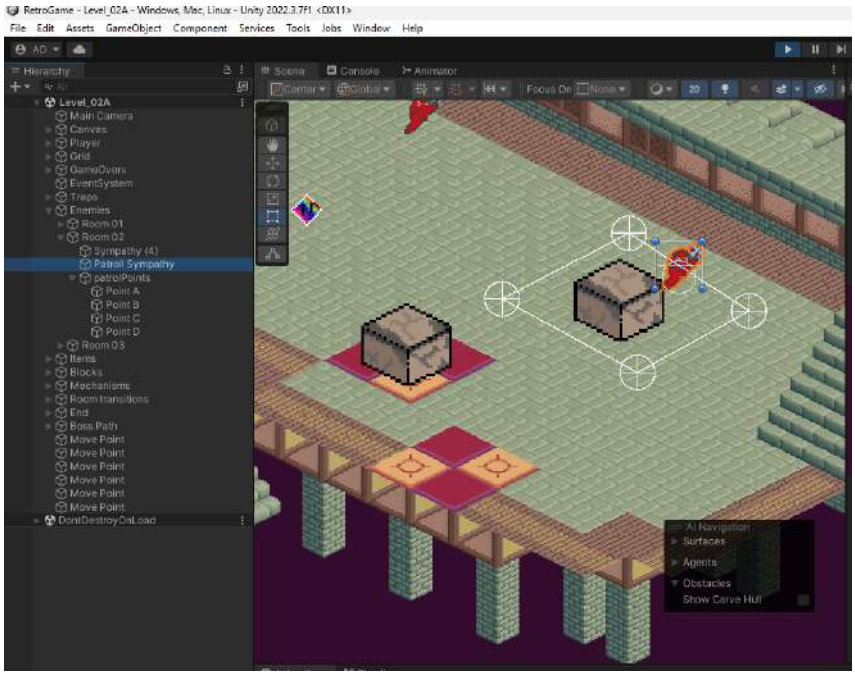
- Função PatrolingBehavior: Na linha 09 o objeto confere se a sua posição atual é igual ao próximo ponto de patrulha, caso não seja, na linha 13 o código começa

a mover o GameObject inimigo em direção ao ponto de referência utilizando a posição seu componente transform com o método MoveTowards provido pela Unity. O método MoveTowards já vem “de fábrica” no tipo Vector3 e seus parâmetros recebem: posição do objeto que vamos locomover, ponto até onde queremos que ele se mova e velocidade com a qual este irá se mover;

- Função increaseTargetInt: Alterna o ponto de patrulha alvo cada vez que o inimigo chega num dos pontos, por exemplo: se o inimigo está no ponto A, seu ponto alvo será o ponto B, quando ele chegar no B as linhas 09 até 11 serão executadas, fazendo o ponto C ser o próximo ponto alvo e assim por diante até chegar no último ponto e ultrapassar a contagem de pontos de patrulha, quando for o caso as linhas 18 até 20 irão resetar a contagem fazendo com que o próximo ponto alvo seja o ponto A;

- Método OnDrawGizmosSelected: Método advindo da Unity para auxiliar no posicionamento dos nossos alvos, pode ser utilizado para outros propósitos mas em nosso trabalho foi usado para estabelecer linhas que conectam cada um dos GameObject’s alvo. Nas linhas 26 até 29 nós desenhamos círculos ao redor do GameObject’s para sabermos onde estes se localizam(como são apenas objetos vazios, eles são “invisíveis” até clicarmos neles no editor e a interface da Unity nos mostrar onde eles estão como é possível observar na Figura 53. Na linha 31 nós utilizamos o método DrawLine para traçar uma linha da posição do transform do GameObject ponto A até o transform do GameObject Ponto B, o mesmo processo é feito com os demais pontos até que todos se interliguem;

Figura 53 – Funcionamento dos pontos de patrulha do PatrollingMisery no editor da Unity, Fonte: O Autor



Estes dois inimigos morrem quando seus corpos colidem com objetos do tipo “PlayerBullets” duas vezes, nesse caso é necessário receberem dois tiros para serem destruidos. É possível conferir os scripts dos dois tipos de inimigo do nível 02 no apêndice (B).

5.2.3 Scripts e Componentes do Chefe e Finalização do Nível 02

Para encerrar o conteúdo do nível 02, o funcionamento chefe deste nível será abordado, como dito previamente no final do capítulo 5.2.1, o Relp terá a passagem para a sala do mesmo liberada apenas após ativar todos os mecanismos.

Ao adentrar na sala a batalha se inicia e o chefe começa a atacar o Relp, para que o jogador tivesse uma experiência diferente dos inimigos anteriores, foi determinado que o comportamento do chefe seria de seguir o Relp e ativar o seu ataque após o mesmo entrar em contato com sua área de ataque, ao contrário dos inimigos anteriores que atacavam apenas encostando no Relp, o chefe do nível 02 invoca alguns inimigos do tipo 01 para que eles ataquem na frente. O plano é fazer com que o chefe se proteja atrás de seus lacaios e vá aumentando a quantidade de inimigos até se tornar um problema para o jogador.

O chefe possui três scripts principais, sendo estes: script de parâmetros do inimigo, “script de alerta” e um script para iniciar a invocação dos fantasmas ao disparar o gatilho de proximidade com o Relp.

Primeiramente é necessário abordar o script de parâmetros chamado “EnemySecond”. Criado utilizando implementação de interfaces, herança e máquina de estados, tudo isto com auxílio de dois vídeos dos canais (CODEMONKEY, 2024) e (SASQUATCHBSTUDIOS, 2024), de onde foram extraídos alguns trechos de código para formar a “entidade” deste chefe.

As interfaces do script são as seguintes:

- IDamageable: Para objetos que podem sofrer dano, determina que o chefe pode sofrer os ataques do Relp, ao contrário do cenário, onde ao entrar em contato com as balas faz com que essas desapareçam;
- IEnemyMoveable: Para inimigos que podem se mover;
- ITriggerCheckable: Para objetos que verificam colisões com o uso de gatilhos, essa interface foi utilizada para as áreas de detecção do chefe. Se o Relp adentrar nelas, o alerta será emitido para os scripts do chefe.

Interfaces são uma forma de implementar um conjunto de funcionalidades a uma classe. Um script com as funcionalidades necessárias é criado e depois outro script

diretamente interligado com nosso objetivo de implementação herda estas por meio da técnica de Herança. Geralmente, são utilizadas para evitar que os scripts fiquem muito centralizados, protegendo a integridade do projeto.

Os estados presentes na máquina de estados são os seguintes:

- IdleState: Chefe parado, utilizado no início do nível antes do Relp entrar no alcance de alerta;
- ChaseState: Estado de perseguição;
- ATKState: Estado onde o chefe invoca os fantasmas para atacar o Relp.

Dos citados acima, iremos falar sobre o resultado do ChaseState permanecer ativo enquanto o Relp está no alcance do chefe. Se ele entrar no círculo de alerta, o estado SetStrikingDistanceBool é ativado, fazendo com que os fantasmas sejam invocados; para que isso aconteça, o chefe precisa estar previamente em estado de alerta (estado controlado por uma variável bool chamada "IsAggroed").

Todos esses estados são manipulados de acordo com os gatilhos de colisão das áreas em volta do chefe, como os scripts para a atividade de cada área são semelhantes, abordaremos somente o script 5.6, que é utilizado para ativação do ataque.

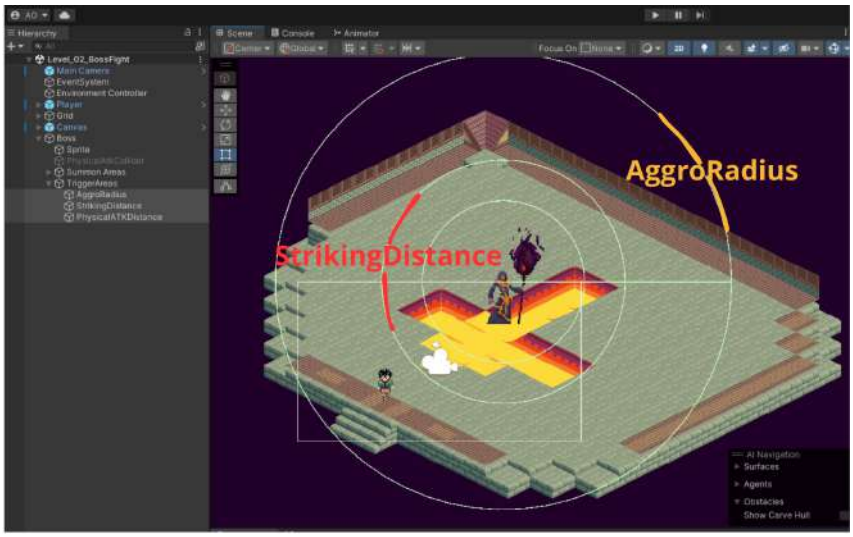
Listing 5.6 – Script EnemyStrikingDistanceCheck

```
1
2  public class EnemyStrikingDistanceCheck : MonoBehaviour
3  {
4      public GameObject PlayerTarget { get; set; }
5      public EnemySecond _enemy;
6
7      private void Awake()
8      {
9          PlayerTarget = GameObject.FindGameObjectWithTag("Player");
10
11         _enemy = GetComponentInParent<EnemySecond>();
12     }
13
14     private void OnTriggerEnter2D(Collider2D collision)
15     {
16         if (collision.gameObject == PlayerTarget)
17         {
18             _enemy.SetStrikingDistanceBool(true);
19         }
20     }
21
22     private void OnTriggerExit2D(Collider2D collision)
```

```
23      {
24          if (collision.gameObject == PlayerTarget)
25          {
26              _enemy.SetStrikingDistanceBool(false);
27          }
28      }
29  }
```

Ao iniciar a batalha contra o chefe, o jogador vai se encontrar na área AggroRadius, o campo de visão do chefe, fazendo com que o mesmo avance em direção ao Relp para atacá-lo e, caso o chefe consiga fazer com que o mesmo adentre a área StrikingDistance, o ataque se inicia. É possível observar as áreas na Figura 54.

Figura 54 – Áreas de ataque do chefe do nível 02, Fonte: O Autor



Após derrotar o chefe, a função Die é ativada no script EnemySecond, fazendo com que o jogador conclua o nível 02 e veja a tela de vitória para seguir para o nível 03.

5.3 Nível 03

Nessa seção abordaremos o desenvolvimento e as técnicas utilizadas no desenvolvimento do último nível do projeto.

5.3.1 Mecânicas e programação do Nível 03

No terceiro nível, foi necessário realizar ajustes nos scripts relacionados à vida e à movimentação do protagonista. Nesse nível, Relp volta a se movimentar em duas dimensões, conforme ocorre no nível 01 do jogo. Entretanto, devido à nova mecânica que permite que ele resista a múltiplos golpes antes de ser derrotado, tornou-se indispensável

implementar um sistema que impeça a movimentação do personagem enquanto este estiver sob ataque. Adicionalmente, foram desenvolvidos scripts específicos para o comportamento do chefe e para a contabilização de rounds, sendo que este último envolve a contabilização por: vida e tempo esgotado. As próximas seções abordarão essas atualizações e os novos scripts.

5.3.1.1 Scripts da Mecânica de Combates

Iniciando no script responsável pelo sistema de Rounds, chamado de RoundCount, este script controla diversas variáveis, entre estas: o componente Transform, do Relp e do Nostalghia, a contagem de tempo do Round e os dados de vida.

O script possui variáveis do tipo "Life" para acessar os dados do Nostalghia e do Relp sempre que um round se inicia, assim o RoundCount fica a par de quem está com vantagem de rounds sempre que é executado, isso é atualizado com a função UpdateHUD.

Porém a parte importante está na função GameSet. Função esta que, por sua vez, chama a função GettingRoundResults se um dos personagens tiver sido nocauteado, fazendo com que na função RoundCheck(chamada na GettingRoundResults) adicionemos uma vitória para Nostalghia ou para Relp, respectivamente, utilizando os dados das variáveis playerData e bossData. Caso um destes tenha enviado o dado "fainted", contabilizaremos a vitória para o lado oposto. É possível conferir um trecho com as funções responsáveis por isso no Listing (5.7).

Listing 5.7 – Funções utilizadas no sistema de Rounds

```
1    void GameSet()
2    {
3        if (playerData.fainted || bossData.fainted)
4        {
5            GettingRoundResults();
6            isUpdateEnabled = false; //para de rodar no update
7        }
8
9    }
10     private void RoundCheck()
11     {
12         if (playerData.fainted == true)
13         {
14
15             RoundSingleton.Instance.RecordRoundWinner(winner:
                   RoundSingleton.Side.Right);
16             playerData.fainted = false;
17
18         }
19
```

```
20          else if (bossData.fainted == true) //lembrando que preciso
                fazer isso pros 2 rounds(ganhos)
21          {
22
23              RoundSingleton.Instance.RecordRoundWinner(winner:
                    RoundSingleton.Side.Left);
24              bossData.fainted = false;
25
26          }
27      }
```

Para encerrar essa subseção, vamos falar sobre a função UpdateHUD (5.8) que ficou encarregada de informar ao jogador quem está vencendo a luta.

Nela obtemos as informações sobre a contagem de vitórias para cada lado por meio de um Singleton criado com o intuito de manter a contagem intacta entre cenas. Um Singleton é uma forma de garantir que uma classe tenha apenas uma instância durante a execução de um programa. Fizemos isso para que os dados da luta pudessem ficar guardados num objeto que não seria perdido no recarregamento da cena, já que a cada nocaute ou fim de Round por limite de tempo, recarregávamos a mesma para redefinir as posições dos personagens e suas barras de vida.

Listing 5.8 – Função UpdateHUD, Fonte: O Autor

```
1  void UpdateHUD()
2  {
3      if(lifeIcon != null && lifeIcon.Length > 0)
4      {
5          int rightWins = RoundSingleton.Instance.CurrentWinsOf(
               RoundSingleton.Side.Right);
6          int leftWins = RoundSingleton.Instance.CurrentWinsOf(
               RoundSingleton.Side.Left);
7
8          Debug.Log($"Victory on boss side: " + rightWins);
9          Debug.Log($"Victory on Player side: " + leftWins);
10
11         lifeIcon[0].sprite = rightWins > 0 ? hollow : full;
12         lifeIcon[1].sprite = rightWins > 1 ? hollow : full;
13         lifeIcon[3].sprite = leftWins > 0 ? hollow : full;
14         lifeIcon[2].sprite = leftWins > 1 ? hollow : full;
15
16         Debug.Log("HUD updated");
17     }
18     else Debug.LogError("lifeIcon array retornou null");
19 }
```

Após obter os valores de vitórias, nas linhas 11 até 14 do script 5.8, desativamos

o sprite do respectivo coração de quem perdeu o Round passado, como é possível ver na Figura 55. Os sprites ficavam localizados num vetor do tipo Image chamado lifeIcon, onde os vetores 0 e 1 correspondiam às chances do jogador e os outros correspondiam às chances do Nostalghia.

Figura 55 – Mudança no HUD contabilizando a vitória do jogador, Fonte: O Autor

5.3.2 Scripts e Funcionamento do Relp

Como dito no início do capítulo 5.3.1, algumas mudanças na movimentação foram realizadas e estas, em conjunto com o novo formato de vida para este nível, serão discutidas a seguir.

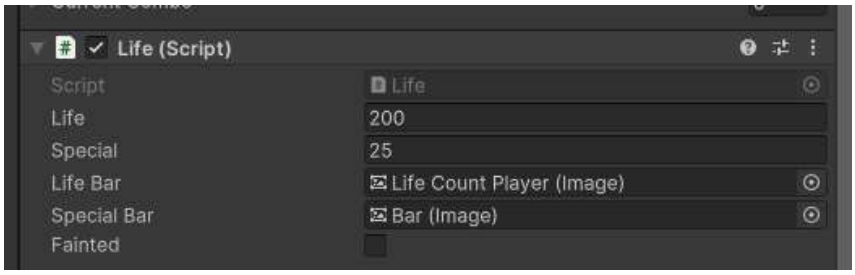
5.3.2.1 Scripts de Movimentação, Barras de Vida e Ataque Especial

O script de movimentação do Relp no nível 03 se chama PlayerWalk, sendo uma adaptação do script de movimentação do Relp no nível 01. Fizemos poucas alterações nele e entre as mais importantes estão:

- Função resetAnimation: Utilizado quando Relp recebe golpes, afinal precisamos interromper qualquer animação que ele esteja fazendo para a animação de quando está recebendo dano, então setamos as booleanas de movimentação como falsas e retiramos a possibilidade de movimentação do personagem por um breve momento;
- Função Facing: Como em um jogo de luta o personagem deve estar sempre olhando na direção do seu oponente, precisamos que seus colisores estejam de acordo e para isso, verificamos se a posição do inimigo no eixo X é maior que a posição do Relp, o que realmente verificamos nessa condicional é quem está mais à direita, para que caso seja o Relp, ele passe a olhar e direcionar seus ataques para a esquerda.

Tratada a movimentação, seguiremos para o funcionamento das barras de vida e ataque especial. Como dito anteriormente, Relp agora pode receber mais golpes antes de ser derrotado, então adicionamos uma variável de vida para isso, conforme a Figura 56. Além disso, a vida trabalha em conjunto com a barra de ataque especial, onde esta funciona como uma mecânica de vingança. Nosso objetivo é fazer com que, caso o personagem esteja perdendo a luta, possa virar o jogo contra seu oponente utilizando um ataque especial, já que essa barra aumenta conforme ele recebe dano.

Figura 56 – Variáveis do script de vida no nível 03, Fonte: O Autor



Esse script foi utilizado tanto no Relp quanto no Nostalghia e a única função implementada nele foi o TakeDamage, que é chamada toda vez que um golpe de um dos personagens colide com o corpo de seu oponente, reduzindo a vida pelo dano recebido e contabilizando o nocaute se a vida chegar a zero. É possível conferir isto no script Life (5.9).

Listing 5.9 – Script de Vida no nível 03

```
1  public class Life : MonoBehaviour
2  {
3      public float life;
4      public float special = 25;
5      public Image lifeBar;
6      public Image specialBar;
7      public bool fainted;
8
9      private void Start()
10     {
11         fainted = false;
12     }
13     private void Update()
14     {
15         lifeBar.fillAmount = life / 200f;
16         specialBar.fillAmount = special/ 100f;
17     }
18
19     public void TakeDamage(float damage)
20     {
21         life -= damage;
```

```
22
23          special += damage * 2f;
24
25          if (special >= 100)
26          {
27              special = 100;
28          }
29
30          if(life <= 0)
31          {
32              Debug.Log("Game over pra esse aqui");
33              fainted = true;
34          }
35
36      }
37  }
```

5.3.2.2 Scripts do Sistema de Golpes e Combos para o Relp

O nível 03 foi inspirado no jogo Street Fighter 2, onde os personagens possuem golpes de diferentes forças como: fraco, médio e forte e também possuem combos envolvendo a combinação destes golpes. Portanto, tentamos fazer o terceiro nível possuir uma mecânica parecida; apesar da quantidade de tempo ser pequena, conseguimos implementar o possível.

O script responsável pelo sistema de combate do Relp se chama PlayerCombo e possui apenas três funções. Abaixo descreveremos seu funcionamento:

- Função CheckInputs: Executada no método Update, esta função fica encarregada de checar as teclas de ataque que o jogador aperta para contabilizar em que parte do combo o jogador está, por exemplo, o combo básico segue os inputs: ataque fraco, ataque médio e ataque forte. Caso o jogador os execute nessa ordem o combo vai "sair", porém se os ataques apertados não seguirem uma das ordens determinadas possíveis por nós, a condicional de "falha do combo" é executada e o Relp volta ao estado iddle após um tempo(por exemplo, tentar o combo básico mas pular o golpe médio e ir direto pro golpe forte, o combo não será executado);

- Função PlayHit: Este método aciona o ataque correspondente do Relp(animação e colisão), o adiciona na contagem de combos e também reseta o Timer de combos, para que a contagem de acertos continue ativa na tela, já que ela desaparece com o tempo, caso o jogador não esteja acertando golpes;

- Função ResetCombo: Como o próprio nome diz, reseta o contador de combos e além disso rebobina a animação do Relp para o estado idle. Este método é chamado em

duas situações: quando o jogador erra a sequência de golpes e quando ele decide parar de bater;

É possível conferir o código na íntegra no Apêndice (A).

5.3.2.3 Script do Sistema de Defesa

Além da habilidade de atacar, Relp também pode se defender dos ataques do Nostalghia. Ao apertar a tecla L, Relp entrará em um estado estacionado, de modo que ele não conseguirá se mover enquanto a defesa estiver ativa; contudo, "segurar a defesa" irá reduzir todo o dano recebido pela metade.

Contudo, é importante informar que não há corte algum de dano nos scripts do Relp, o corte acontece no código de combate do Nostalghia, pois ao setarmos o valor da variável isDefending como verdadeira, quando há a colisão dos golpes com o Relp, o próprio script dele estará encarregado de enviar somente metade do dano.

5.3.2.4 Barra de Ataque Especial

Por último, falaremos sobre o script da barra de ataque especial. Esta mecânica consiste em acumular poder com base no dano recebido, até que a barra fique completamente cheia. A partir do momento em que isso acontece, o jogador pode apertar a tecla R para ativar o ataque especial, onde serão executadas as seguintes ações:

- Ativação da câmera virtual: Por um breve momento ativamos uma câmera secundária para dar destaque ao personagem que está usando seu ataque especial;
- Ativação do som do ataque especial: Após o jogador apertar o botão e executar o especial, o áudio dele vai tocar e o valor da barra será zerado.

É possível conferir o trecho de código da barra de especial no Apêndice (A). E assim, encerramos a seção que trata da programação do jogador no nível 03; as próximas seções abordarão o chefe do jogo, Nostalghia.

5.3.3 Scripts e Funcionamento do Nostalghia

Encerrando os assuntos do terceiro nível, os scripts responsáveis pelo comportamento de Nostalghia serão abordados, o principal inimigo deste jogo e aquele que prendeu o protagonista em um mundo de jogos. Nostalghia tem uma programação mais robusta que os inimigos previamente trabalhados; apesar de mais rápidos, os inimigos anteriores não tinham a mesma quantidade de vida ou mesmo a capacidade de desferir diferentes ataques

contra o protagonista. Contudo, é necessário ressaltar que, pelo curto tempo disponível para a produção deste jogo que compõe três estilos diferentes de jogatina, a IA entregue é relativamente simples para os padrões atuais.

Nostalghia possui três scripts organizados da seguinte forma:

- BossFacingDirection: Controla a direção em que ele estará direcionado, de forma que esteja sempre olhando para o Relp;
- BossMovimentation: Controla atributos como velocidade, área de ataque e o comportamento do chefe. Este script é como o cérebro do personagem, é nele que é decidida aleatoriamente qual ação o chefe vai tomar entre um conjunto de 6 ações diferentes;
- BossCombat: Responsável por executar as ações determinadas no script BossMovimentation, BossCombat tem um dez métodos, sendo seis para atacar o protagonista e os outros quatro para controle do ataque especial e de um movimento de finta onde ele recua um pouco antes de continuar avançando novamente contra o jogador.

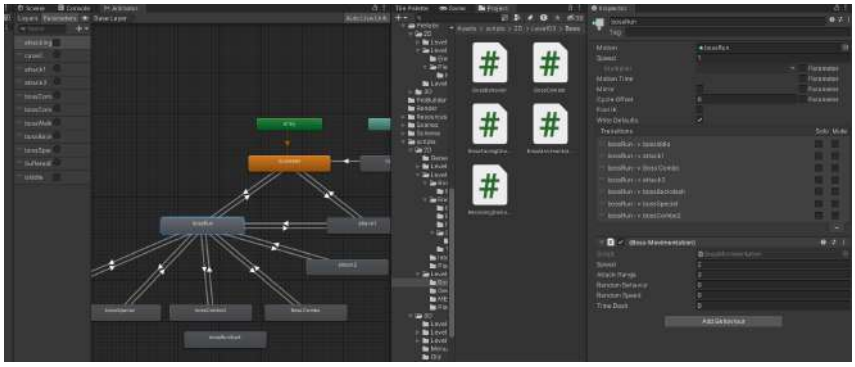
Apenas os scripts BossMovimentation e BossCombat serão abordados nesta seção, pois o BossFacingDirection nada mais é do que uma simples inversão de direções.

5.3.3.1 Script BossMovimentation

O objetivo deste script era fazer a IA tomar uma decisão para cada ação do jogador, porém a ideia se revelou mais difícil de implementar do que o imaginado, então foi decidido que, ao se aproximar o suficiente do jogador, um SwitchCase seria executado com um número aleatório de 0 a 5 e o valor determinaria qual das 6 ações disponíveis seria executada.

O script BossMovimentation (5.10) foi inserido não no GameObject do chefe, mas sim no seu estado BossRun no Animator, como é possível observar na Figura 57. Este script é do tipo máquina de estados, e foi feito dessa forma para que a ligação fosse feita entre cada ação e sua animação a partir da decisão tomada quando Nostalghia está próximo o suficiente do Relp.

Figura 57 – Script BossMovimentation inserido no animator para controlar o comportamento do chefe, Fonte: O Autor



Cada ação é ativada por seu respectivo gatilho e este é resetado após a ação de seu número sorteado ter acabado, isso acontece a partir da linha 122, onde todos os gatilhos de ação são resetados. Além disso, no método OnStateEnter, que é disponibilizado pela Unity, a velocidade e o estado de perseguição do chefe são decididos a cada mudança de estado, para que ele não vá enfrentar o jogador sempre na mesma velocidade.

Listing 5.10 – Trecho do script BossMovimentation, Fonte: O Autor

```
1  public class BossMovimentation : StateMachineBehaviour
2  {
3   override public void OnStateEnter(Animator animator, AnimatorStateInfo
        stateInfo, int layerIndex)
4   {
5       player = GameObject.FindWithTag("Player").transform;
6       rb = animator.GetComponentInParent<Rigidbody2D>();
7       bossSpecialBar = GameObject.Find("Enemy Test").GetComponent<Life>()
            ;
8
9       randomSpeed = Random.Range(1, 3);
10      if(randomSpeed == 1) { speed = 2f; } else { speed = 3f; }
11  }
12
13  //OnStateUpdate is called on each Update frame between OnStateEnter and
        OnStateExit callbacks
14  override public void OnStateUpdate(Animator animator, AnimatorStateInfo
        stateInfo, int layerIndex)
15  {
16      if(isGettingHit == false) {
17
18          animator.SetBool("isIddle", false); //andando
19
20          tempoPassado += Time.fixedDeltaTime;
21          //regulagem de velocidade
22          if (tempoPassado > 2.5f) { speed = 2f; }
23
```

```
24          //Back walking behavior
25          if (lastState == 5)
26          {
27              Vector2 target = new Vector2(player.position.x, rb.position
                    .y);
28              Vector2 newPos = Vector2.MoveTowards(rb.position, target, -
                    speed * Time.fixedDeltaTime);
29              rb.MovePosition(newPos);
30
31              tempoPassado += Time.fixedDeltaTime;
32              if (tempoPassado >= timeToAction)
33              {
34                  newPos = Vector2.MoveTowards(rb.position, target, speed
                        * Time.fixedDeltaTime);
35                  rb.MovePosition(newPos);
36              }
37          }
38          else
39          {
40              Vector2 target = new Vector2(player.position.x, rb.position
                    .y);
41              Vector2 newPos = Vector2.MoveTowards(rb.position, target,
                    speed * Time.fixedDeltaTime);
42              rb.MovePosition(newPos);
43          }
44
45
46
47          if (Vector2.Distance(player.position, rb.position) <=
                attackRange)
48          {
49              if (tempoPassado > timeToAction)
50              {
51                  //Comportamento
52                  randomBehavior = Random.Range(1, 6);
53
54                  lastState = randomBehavior; //usado para mudan a de
                        velocidade
55
56                  switch (randomBehavior)
57                  {
58                      case 0:
59                          //dash para tr s
60                          rbOriginalGravity = rb.gravityScale;
61
62                          currentTimeDash = timeDash;
63                          animator.SetTrigger("bossBackdash");
64                          timeDash -= Time.deltaTime;
```

```
65
66                          if (timeDash <= 0)
67                          {
68                              Debug.Log("finalizou");
69                              rb.gravityScale = rbOriginalGravity;
70                          }
71
72                          break;
73                      case 01:
74                          animator.SetTrigger("attack1");
75                          new WaitForSeconds(1);
76                          break;
77                      case 02:
78                          animator.SetTrigger("attack2");
79                          new WaitForSeconds(1);
80                          break;
81                      case 03:
82                          if (bossSpecialBar.special >= 100)
83                          {
84                              animator.SetTrigger("bossCombo2");
85                              bossSpecialBar.special = 0;
86                              break;
87                          }
88                          else
89                          {
90                              animator.SetTrigger("bossCombo");
91                              new WaitForSeconds(1);
92                          }
93                          break;
94                      case 04:
95                          if (bossSpecialBar.special >= 100)
96                          {
97                              animator.SetTrigger("bossSpecialAttack");
98                              bossSpecialBar.special = 0;
99                          }
100                         break;
101                     case 05:
102                         animator.SetTrigger("bossWalkBack");
103                         tempoPassado = 0;
104                         break;
105                 }
106                 tempoPassado = 0;
107
108             }
109         }
110     } else if(isGettingHit == true)
111     {
112         Debug.Log("entramos no true");
```

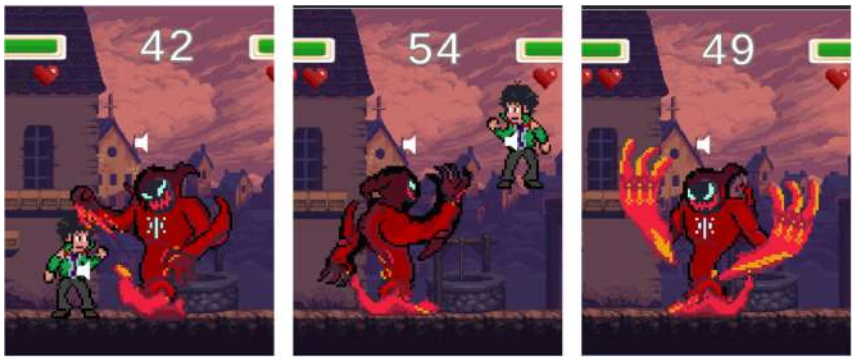
```
113             animator.SetBool("isIddle", true);
114
115             if (isGettingHit == false) return;
116
117         }
118
119     }
120
121     //OnStateExit is called when a transition ends and the state machine
        finishes evaluating this state
122     override public void OnStateExit(Animator animator, AnimatorStateInfo
        stateInfo, int layerIndex)
123     {
124         animator.ResetTrigger("attack1");
125         animator.ResetTrigger("attack2");
126         animator.ResetTrigger("bossCombo");
127         animator.ResetTrigger("bossWalkBack");
128         animator.ResetTrigger("bossBackdash");
129         animator.ResetTrigger("bossSpecialAttack");
130     }
131 }
```

5.3.3.2 Script BossCombat

Este script está inteiramente encarregado de executar as ações do chefe contra o protagonista, se o BossMovimentation era o cérebro, o BossCombat é como os membros do Nostalghia. De forma resumida, Nostalghia agirá de acordo para cada função quando forem chamadas, assim como na Figura 58 onde o mesmo está executando as funções:

- bossAttack1: Ataca com um soco rápido, é o golpe mais fraco dele;
- bossAttack2: Ataca com um gancho com tanta força que lança o Relp para longe, esse golpe possui um dano maior para que a ideia de impacto seja transmitida ao jogador;
- BossComboThirdHit: Como o nome diz, esse é o terceiro acerto do combo, essa função é chamada em sequência após as duas primeiras funções de combo e corresponde ao caso 03 do SwitchCase no script *BossMovimentation* (5.10).

Figura 58 – Diferentes golpes do chefe Nostalghia.



Fonte: O Autor

Após nocautear o chefe duas vezes, Relp enfim vence o desafio e fica livre. Para transmitir ao jogador que ele finalizou o jogo, uma tela de vitória foi criada, Figura 59, para dar parabéns por sua conquista e finalizar a história.

Figura 59 – Tela de Finalização do jogo, Fonte: O Autor

6 DESAFIOS E ANÁLISE DE RESULTADOS

Este capítulo tratará dos principais desafios enfrentados durante o projeto e também dos resultados finais de desenvolvimento.

6.1 Principais Desafios

A diferença de gênero de cada nível tornou o trabalho de implementação do jogo mais complexo do que o esperado, apesar das ferramentas disponibilizadas pela Unity facilitarem grandemente o desenvolvimento. Além disso, o desenvolvimento de forma independente demonstrou como desenvolver jogos é uma tarefa demorada e complicada, já que, apesar de antigos, os jogos tomados como modelo são provenientes de produções muito maiores do que a desenvolvida neste trabalho.

A cada nível foi necessário aprender um novo tipo de lógica de programação para que fosse possível reproduzir os gêneros escolhidos, ocasionando um gasto maior tempo. Além disso, era necessário animar e redesenhar o protagonista para cada nível, o que também não foi uma tarefa fácil.

6.2 Resultados

Produziu-se um jogo com resultados satisfatórios para o objetivo do projeto. O nível 01 possui uma jogatina com desafios que vão aumentando conforme o jogador passa pela floresta; no nível 02, a animação do protagonista e o ambiente construído permitiram uma grande imersão do jogador com a proposta do jogo; e por último no nível 03, houve notável melhora na implementação de mecânicas e animações, pois os sprites podiam ser maiores e mais detalhados, tornando possível dar mais vida aos personagens, principalmente ao chefe final, onde a melhoria nas habilidades de animação em Pixel-Art foi evidente, além da criação de uma IA com melhor qualidade que as apresentadas nos níveis anteriores.

Vale citar que no decorrer deste projeto foi possível obter conhecimentos a respeito de implementação jogos como: interfaces visuais, menus que interligam as fases ao menu principal, animação e arte em Pixel-Art, máquinas de estado, implementação dos conceitos de herança e polimorfismo estudados durante o curso de Engenharia da Computação.

6.3 Trabalhos Futuros

Algumas melhorias que podem ser adicionadas a este trabalho são: construir IA's mais robustas para os inimigos, implementar interfaces visuais com animações, tornando-as mais atrativas e animações dos personagens mais trabalhadas.

7 CONCLUSÃO

Este projeto apresentou um breve estudo sobre a influência da pixel-art nos jogos indies atuais e também a descrição do desenvolvimento de um jogo 2D na Unity Engine criado com base na inspiração nos jogos icônicos Super Mario Bros, Boktai The Sun Is In Your Hand e Street Fighter 2. Com base nisso, tivemos a oportunidade de aprender novas técnicas de programação orientadas à criação de jogos, além da experiência de iniciar na pixel-art.

A escalada foi complicada, mas as ferramentas utilizadas no caminho se provaram extremamente úteis; a Unity nos proveu com diversas ferramentas que encurtaram o processo de trabalho em anos, pois várias funções presentes em jogos já vêm prontas nela, provendo um maior desempenho ao desenvolvedor; o Aseprite também nos auxiliou grandemente com as timelines, a função de camadas e as paletas de cores das décadas de 80 e 90, proporcionando um ambiente de trabalho adaptável e confortável.

O objetivo deste projeto se manteve. No decorrer do trabalho, foi possível observar jogos indies e trabalhos acadêmicos que sofreram influência de jogos e gêneros famosos do auge da pixel-art do período de 1980 e 1990. Além de demonstrar um pouco da experiência de desenvolvimento de um jogo com as ferramentas atuais, este trabalho pode vir a servir de referência para os próximos desenvolvedores indie brasileiros.

Os resultados obtidos também demonstram que a Pixel art e as mecânicas dos jogos clássicos continuam exercendo influência na produção de jogos independentes, seja como uma forma de resgatar a identidade visual e a jogabilidade desses títulos ou como escolha pessoal. O estudo também evidenciou desafios no desenvolvimento, como a adaptação de mecânicas retrô a um ambiente moderno e a complexidade da implementação da perspectiva isométrica na Unity.

Apesar de cumprir os objetivos principais, melhorias ainda podem ser implementadas para enriquecer o trabalho e melhorar o jogo como um todo. Entre elas, se destacam a correção de bugs, aprimoramento da IA e da pixel-art.

Por último, espera-se que com este trabalho seja possível ter uma noção do processo de desenvolvimento de um jogo 2D na Unity engine.

O código-fonte do projeto será disponibilizado de forma aberta através de um repositório público disponibilizado na plataforma *GitHub*(clique ou acesse: https://github.com/Shubunaga/Re
Game-2D.git).

REFERÊNCIAS

AMORIM, A. Jogos eletrônicos interativos: A origem dos jogos eletrônicos. **Universidade de São Paulo, São Paulo, Brasil**, 2006.

CENGIZ, S. **What is an isometric game How to make it**. 2021. Disponível em: <https://mobidictum.com/what-is-isometric-game-how-to-make-it/>.

CODEMONKEY. **How to Boss Fight in Unity with Stages and Increasing Difficulty! (Unity Tutorial)**. 2024. Disponível em: <https://youtu.be/qZC1VYWnHZ8?si=ws3_M9xgFJDd4hW->.

DENISYUK, Y. **What is an isometric game How to make it?** 2022. Disponível em: <https://pinglestudio.com/blog/full-cycle-development/game-development-stages#:~:text=The%207%20stages%20of%20game%20development%20(Planning%2C%20Pre%2Dproduction,structured%20throughout%20the%20development%20process.>

DESCONHECIDO. **Indie Games And The Love Of The Retro Aesthetic**. 2019. Disponível em: <https://www.indiegamewebsite.com/2019/01/30/indie-games-and-the-love-of-the-retro-aesthetic/>.

DOMíNGUEZ, A. **The Design of Indie Games, a Different Paradigm**. 2019. Disponível em: <https://epub.uni-bayreuth.de/id/eprint/4330/1/THE%20DESIGN%20OF%20INDIE%20GAMES%20A%20DIFFERENT%20PARADIGM.pdf>.

GOGONI, R. **Street Fighter: a trajetória de um dos jogos de luta mais famosos**. 2020. Disponível em: <https://meiobit.com/419291/street-fighter-a-trajetoria-de-um-dos-jogos-de-luta-mais-famosos/>.

GUINNESSWORLDRECORDS. **Best-selling fighting videogame series (excluding crossovers)**. 2024. Disponível em: <https://guinnessworldrecords.com.br/world-records/95565-best-selling-fighting-videogame-series-excluding-crossovers>.

MARCUSSEN SOFIE SCHMIDT, E. Z. S. **Inferno Master Thesis Report**. 2023. Disponível em: <https://kglakademi.dk/sites/default/files/imported-file/project-downloads/masters_thesis_report_-_sofie_kjaer_schmidt_-_052523_-_copy.pdf>.

MATTOS, A. S. de. **Developers: Shigeru Miyamoto**. 2011. Disponível em: <https://www.nintendoblast.com.br/2011/02/developers-shigeru-miyamoto.html>.

MCAULEY, C. **Top Ten Pixel Art Games Coming in 2018**. 2018. Disponível em: <https://gametyrant.com/news/top-ten-pixel-art-games-coming-in-2018>.

METHIOU. **Ryu Fan Spritesheet**. 2016. Disponível em: <https://www.deviantart.com/methiou/art/Ryuspritesheet-595008837>.

MOTTA, R. L. Short game design document (sgdd) documento de game design aplicado a jogos de pequeno porte e advergames um estudo de caso do advergame rockergirl bikeway. **SBC**, 2013.

MUNIZ, E. **Página do Eder na Unity Asset Store**. 2025. Disponível em: <https://assetstore.unity.com/publishers/26424>.

MUNIZ, E. **Página do efeito parallax na Unity Asset Store**. 2025. Disponível em: <https://assetstore.unity.com/packages/2d/textures-materials/nature/free-pixel-art-forest-133112>.

PIXELHUNTED. **Boktai: The Sun Is In Your Hand (Game Boy Advance, 2003)**. 2021. Disponível em: <https://pixelhunted.wordpress.com/2021/05/26/boktai-the-sun-is-in-your-hand-game-boy-advance-2003/>.

RYAN, T. **Learning the Ways of the Game Development**. 2009. Disponível em: <https://www.gamedeveloper.com/design/learning-the-ways-of-the-game-development-wiki>.

SASQUATCHBSTUDIOS. **Enhance Your Game: EASY Enemy Logic Using Scriptable Objects (State Machine PART 2) | Unity Tutorial**. 2024. Disponível em: <https://youtu.be/iOYo7flBUW4?si=CIsVi-4hTK0lr57n>.

SCHELL, J. The art of game design. morgan kaufman publishers. **Amsterdam ; Boston : Elsevier/Morgan Kaufmann**, 2008.

SOURAV, B. **2D Platform Game Developed Using Unity Game Engine**. Dissertação (Mestrado) — Centria University of Applied Sciences, 2017.

SPRITERSRESOURCE. **Django Sprite Sheet**. 2024. Disponível em: <https://www.spriters-resource.com/fullview/5588/>.

SPRITERSRESOURCE. **Mega Man Sprite Sheet**. 2024. Disponível em: <https://www.spriters-resource.com/nes/mm/sheet/144772/>.

SUPERMARIOWIKI. **Super Mario Bros**. 2025. Disponível em: <https://www.mariowiki.it/Mondo_1-1_%28Super_Mario_Bros.%29>.

TASAKA, E. **Chrono Trigger (SNES) ainda é a aventura perfeita e no tempo certo**. 2016. Disponível em: <https://jogoveio.com.br/retro-review-chrono-trigger/>.

UNITY. **Unity Documentation**. 2025. Disponível em: <https://docs.unity3d.com/Manual/UsingTheEditor.html>.

WHITEHEAD, T. **Mega Man Legends 3 Demake**. 2014. Disponível em: <https://www.nintendolife.com/news/2014/09/mega_man_legends_3_8_bit_fan_demake_is_available_now>.

APÊNDICE A – SCRIPTS RELACIONADOS AO RELP

Listing A.1 – Script Lifes (Nível 01)

```
public class Lifes : MonoBehaviour
{
    [SerializeField] SpriteRenderer playerSprite;
    public bool gameOver = false;

    EnvironmentController envControll;

    void Start()
    {
        playerSprite = GetComponent<SpriteRenderer>();
        EnvironmentController.instance.UpdateScoreText();

        envControll = GameObject.Find("Canvas").GetComponent<
            EnvironmentController>();
    }

    public void loseLife()
    {
        if(envControll.playerLifes <= 0)
        {
            playerSprite.enabled = false;
            envControll.playerLifes = 0;
            EnvironmentController.instance.ShowGameOver();
        }
        else
        {
            EnvironmentController.instance.SetLives(-1);
        }
    }
}
```

Fonte: O Autor

Listing A.2 – Trecho de script responsável pela morte dos inimigos ou do Relp no Nível 01

```

    bool playerDestroyed;
     void OnCollisionEnter2D(Collision2D collision) {
         if (collision.gameObject.tag == "Player")
         {
             float height = collision.contacts[0].point.y - headPoint.
                 position.y;

             if (height > 0 && !playerDestroyed)
             {
```

```
10                  collision.gameObject.GetComponent<Rigidbody2D>().
                        AddForce(Vector2.up * 7, ForceMode2D.Impulse);
11                  anim.SetTrigger("death");
12                  AudioController.current.PlayMusic(AudioController.
                        current.jumpInEnemySFX);
13                  speed = 0;
14                  boxCol.enabled = false;
15                  cirCol.enabled = false;
16                  Destroy(transform.parent.gameObject, 1.35f);
17                  EnvironmentController.instance.playerScore +=  Score;
18                  EnvironmentController.instance.UpdateScoreText();
19              }
20              else
21              {
22                  collision.gameObject.GetComponent<Lifes>().loseLife();
23
24                  playerDestroyed = true;
25                  AudioController.current.PlayMusic(AudioController.
                        current.deathSFX);
26                  playerGO.Die();
27                  //EnvironmentController.instance.ShowGameOver();
28
29              }
30          }
```

Fonte: O Autor

Listing A.3 – Script da Movimentação do Relp no Nível 02

```
1  public enum PlayerState
2  {
3      walk,
4      attack,
5      interact
6  }
7
8  public class PlayerMovement : MonoBehaviour
9  {
10
11         [SerializeField] private int mechanismCount;
12         [SerializeField] private GameObject bossPath;//abertura da
               passagem pra sala do chefe
13
14         public PlayerState currentState;
15
16         public float speed;
17         public Rigidbody2D rbody;
18         private Vector3 change;
19         private Animator animator;
```

```
20
21          void Start()
22          {
23              Application.targetFrameRate = 60;
24              currentState = PlayerState.walk;
25              animator = GetComponent<Animator>();
26              rbody = GetComponent<Rigidbody2D>();
27          }
28
29      void Update()
30      {
31
32          if (Input.GetButtonDown("attack") && currentState != PlayerState
                .attack)
33          {
34              StartCoroutine(AttackCo());
35          }
36
37      }
38
39      void FixedUpdate()
40      {
41          change = Vector3.zero;
42          change.x = Input.GetAxisRaw("Horizontal");
43          change.y = Input.GetAxisRaw("Vertical");
44
45          if (currentState == PlayerState.walk)
46          {
47              UpddateAnimationAndMove();
48          }
49      }
50
51      private IEnumerator AttackCo()
52      {
53          animator.SetBool("attacking", true);
54          currentState = PlayerState.attack;
55          yield return null;
56          animator.SetBool("attacking", false);
57          rbody.velocity = Vector2.zero;
58          yield return new WaitForSeconds(.33f);
59          currentState = PlayerState.walk;
60          AudioController.current.PlayMusic(AudioController.current.
                shooting);
61      }
62
63      void UpddateAnimationAndMove()
64      {
65          if (change != Vector3.zero)
```

```
66          {
67              MoveCharacter();
68              animator.SetFloat("Horizontal", change.x);
69              animator.SetFloat("Vertical", change.y);
70              animator.SetBool("moving", true);
71          }
72          else
73          {
74              animator.SetBool("moving", false);
75          }
76      }
77
78      void MoveCharacter()
79      {
80
81          Vector2 currentPos = rbody.position;
82          float horizontalInput = Input.GetAxis("Horizontal");
83          float verticalInput = Input.GetAxis("Vertical");
84          Vector2 inputVector = new Vector2(horizontalInput, verticalInput
                );
85          inputVector = Vector2.ClampMagnitude(inputVector, 1);
86          Vector2 movement = inputVector * speed;
87          Vector2 newPos = currentPos + movement * Time.fixedDeltaTime;
88          //isoRenderer.SetDirection(movement);
89          rbody.MovePosition(newPos);
90      }
91
92      public void MechanismActivation()
93      {
94          mechanismCount += 1;
95          Debug.Log("Mechanism count: " + mechanismCount);
96
97          if(mechanismCount >= 6)
98          {
99              bossPath.SetActive(false);
100             //dar play na cutscene aqui de alguma forma
101         }
102     }
103
104 }
```

Fonte: O Autor

Listing A.4 – Trechos Alterados no script de movimentação para o nível 03

```
1  public class PlayerWalk : MonoBehaviour
2  {
3
4      void Update()
```

```
5       {
6           if(canMove == true)
7           {
8               Move();
9               Jump();
10              Facing();
11          }
12
13          if(canMove == false)
14          {
15              resetAnimation();
16          }
17
18      }
19
20      void Move()
21      {
22          {
23              movement = new Vector3(Input.GetAxis("Horizontal"), 0f, 0f);
24
25              switch (isFacingRight)
26              {
27                  case true: //olhando para a direita
28                      if (movement.x > 0)
29                      {
30                          anim.SetBool("isRunning", true);
31                      }
32                      else if (movement.x < 0)
33                      {
34                          anim.SetBool("runningBack", true);
35                      }
36                      else
37                      {
38                          anim.SetBool("isRunning", false);
39                          anim.SetBool("runningBack", false);
40                      }
41                      break;
42                  case false: //olhando para a esquerda
43                      if (movement.x > 0)
44                      {
45                          anim.SetBool("runningBack", true);
46                      }
47                      else if (movement.x < 0)
48                      {
49                          anim.SetBool("isRunning", true);
50                      }
51                      else
52                      {
```

```
53                          anim.SetBool("isRunning", false);
54                          anim.SetBool("runningBack", false);
55                      }
56                      break;
57              }
58
59              transform.position += movement * Time.deltaTime *
                    playerSpeed;
60          }
61
62      }
63
64      #region Player's FacingDir
65      public GameObject Enemy;
66      private float fightersDistance;
67      void Facing()
68      {
69          Vector3 scale = transform.localScale;
70
71          if (Enemy.transform.position.x > transform.position.x)
72          {
73              scale.x = Mathf.Abs(scale.x);
74              isFacingRight = true;
75          }
76          else
77          {
78              scale.x = Mathf.Abs(scale.x) * -1;
79              isFacingRight = false;
80          }
81          transform.localScale = scale;
82      }
83      #endregion
84
85      void resetAnimation()
86      {
87          if(canMove == false)
88          {
89              anim.SetBool("isRunning", false);
90              anim.SetBool("runningBack", false);
91          }
92      }
93  }
```

Fonte: O Autor

Listing A.5 – Script PlayerCombo

```
1  public class PlayerCombo : MonoBehaviour
2  {
```

```
3       public Combo[] combos;
4
5       public AttackData attackData;
6
7       public List<string> currentCombo;
8
9       private Animator anim;
10      private bool startCombo;
11      private Hit currentHit, nextHit;
12      private float comboTimer;
13
14      private bool canHit = true;
15      private bool resetCombo;
16
17      private void Awake()
18      {
19          anim = GetComponentInChildren<Animator>();
20      }
21      void Start()
22      {
23
24      }
25
26      void Update()
27      {
28          CheckInputs();
29      }
30
31      void CheckInputs()
32      {
33          for (int i = 0; i < combos.Length; i++)
34          {
35              if(combos[i].hits.Length > currentCombo.Count)
36              {
37                  if (Input.GetButtonDown(combos[i].hits[currentCombo.
                        Count].inputButton))
38                  {
39                      if (currentCombo.Count == 0)
40                      {
41                          Debug.Log("Primeiro hit foi adicionado");
42                          PlayHit(combos[i].hits[currentCombo.Count]);
43                          break;
44                      }
45                      else
46                      {
47                          bool comboMatch = false;
48                          for (int y = 0; y < currentCombo.Count; y++)
49                          {
```

```
50                              if (currentCombo[y] != combos[i].hits[y].
                                    inputButton)
51                              {
52                                  Debug.Log("Input nao pertence ao combo
                                        atual");
53                                  comboMatch = false;
54                                  break;
55                              }
56                              else
57                              {
58                                  comboMatch = true;
59                              }
60                          }
61
62                          if (comboMatch && canHit)
63                          {
64                              Debug.Log("Hit adicionado ao combo");
65                              canHit = false;
66                              nextHit = combos[i].hits[currentCombo.Count
                                    ];
67                              break;
68                          }
69                      }
70
71                  }
72              }
73          }
74          if (startCombo)
75          {
76              comboTimer += Time.deltaTime;
77              if(comboTimer >= currentHit.animationTime && !canHit)
78              {
79                  PlayHit(nextHit);
80              }
81
82              if(comboTimer >= currentHit.resetTime)
83              {
84                  ResetCombo();
85              }
86          }
87      }
88
89      void PlayHit(Hit hit)
90      {
91          comboTimer = 0f;
92          attackData.SetAttack(hit);
93          startCombo = true;
94          currentCombo.Add(hit.inputButton);
```

```
95          currentHit = hit;
96          anim.Play(hit.animation);
97          canHit = true;
98      }
99
100     void ResetCombo()
101     {
102         startCombo = false;
103         comboTimer = 0f;
104         currentCombo.Clear();
105         anim.Rebind();
106         canHit = true;
107     }
108 }
```

Fonte: O Autor

Listing A.6 – Script de Ataque Especial

```
1  public class SpecialBarAttack : MonoBehaviour
2  {
3      private Life spBar;
4      public Transform spInstance;
5      public GameObject bltStrikePrefab;
6      public float spSpeed;
7      public bool spIsActive;
8      private PlayerWalk facingDir;
9      private Animator anim;
10
11     [SerializeField] private float timeLeft;
12     [SerializeField] private GameObject[] _virtualCameras;
13
14     public AudioManager audioManager;
15     private void Awake()
16     {
17         audioManager = GameObject.FindWithTag("Audio").GetComponent<
               AudioManager>();
18     }
19     void Start()
20     {
21         spBar = GameObject.Find("Player").GetComponent<Life>();
22         facingDir = GameObject.Find("Player").GetComponent<PlayerWalk>()
               ;
23         anim = GameObject.Find("Player").GetComponentInChildren<Animator
               >();
24     }
25
26      private void FixedUpdate()
27      {
```

```
28          if (spBar.special >= 100 && Input.GetKey(KeyCode.R))
29          {
30              SpecialATK();
31              StartCoroutine(CameraReturn());
32          }
33
34      }
35
36      void SpecialATK()
37      {
38          CameraChange();
39          anim.SetTrigger("Special");
40          audioManager.PlaySFX(audioManager.specialATK);
41
42          spBar.special = 0;
43          Debug.Log("Ativamos o special!");
44          spIsActive = true;
45
46          GameObject bullet = Instantiate(bltStrikePrefab, spInstance.
                position, transform.rotation);
47
48          if (facingDir.isFacingRight == true)
49          {
50              bullet.GetComponent<Rigidbody2D>().velocity = new Vector2(
                    spSpeed, 0);
51          }
52          else
53          {
54              bullet.GetComponent<Rigidbody2D>().velocity = new Vector2(-
                    spSpeed, 0);
55          }
56
57      }
```

Fonte: O Autor

APÊNDICE B – SCRIPTS DOS INIMIGOS

Listing B.1 – Trecho do Script do Misery(Nível 01)

```
public class Misery : MonoBehaviour
void Update()
    {

        if(currentPoint == pointB.transform)
        {
            rig.velocity = new Vector2(speed, 0);
        }
        else
        {
            rig.velocity = new Vector2(-speed, 0);
        }

        if(Vector2.Distance(transform.position, currentPoint.position)
           < 0.5f && currentPoint == pointB.transform)
        {
            currentPoint = pointA.transform;
            transform.localScale = new Vector2(transform.localScale.x *
                -1f, transform.localScale.y);
        }
        if(Vector2.Distance(transform.position, currentPoint.position)
           < 0.5f && currentPoint == pointA.transform)
        {
            currentPoint = pointB.transform;
            transform.localScale = new Vector2(transform.localScale.x *
                -1f, transform.localScale.y);
        }
    }

    }

    private void OnDrawGizmosSelected()
    {
        Gizmos.DrawWireSphere(pointA.transform.position, 0.5f);
        Gizmos.DrawWireSphere(pointB.transform.position, 0.5f);
        Gizmos.DrawLine(pointA.transform.position, pointB.transform.
           position);
    }
```

Fonte: O Autor

Listing B.2 – Trecho do Script do Sympathy

```
    void Update()
    {
```

```
3           rig.velocity = new Vector2(speed, rig.velocity.y);
4
5           colliding = Physics2D.Linecast(rightCol.position, leftCol.
                position, layer);
6
7           if (colliding)
8           {
9               transform.localScale = new Vector2(transform.localScale.x *
                    -1f, transform.localScale.y);
10              speed *= -1f;
11          }
12
13          if (timerOn)
14          {
15              if (timeLeft > 0)
16              {
17                  timeLeft -= Time.deltaTime;
18                  updateTimer(timeLeft);
19              }
20              else
21              {   //pulo
22                  rig.AddForce(new Vector2(0f, JumpForce), ForceMode2D.
                        Impulse);
23                  anim.SetBool("Jump", true);
24                  timeLeft = 3.5f;
25              }
26          }
27
28      }
```

Fonte: O Autor

Listing B.3 – Métodos e Funções do script de caça do Misery no nível 02

```
1       private Transform target;
2
3       [SerializeField]
4       private float visionRadius;
5
6       [SerializeField]
7       private LayerMask LayerVisionArea;
8
9       protected override void Update() {
10          SearchPlayer();
11          if(target != null){
12              Move();
13          } else {
14              StopMoving();
15          }
```

```
16      }
17
18      private void OnDrawGizmosSelected() {
19          Gizmos.DrawWireSphere(transform.position, visionRadius);
20      }
21
22      private void SearchPlayer(){
23          Collider2D colisor = Physics2D.OverlapCircle(transform.position
                , visionRadius, LayerVisionArea);
24          if (colisor != null && colisor.transform.CompareTag("Player")){
25              target = colisor.transform;
26          } else {
27              target = null;
28          }
29      }
30
31      private void Move()
32      {
33          Vector2 targetPosition = target.position;
34          Vector2 actualPosition = transform.position;
35
36          float distance = Vector2.Distance(actualPosition, targetPosition
                );
37          if(distance >= minimumDistance){
38              // Movimento do inimigo
39              Vector2 direction = targetPosition - actualPosition;
40              direction = direction.normalized;
41
42              rig.velocity = velocity * direction;
43
44              if(rig.velocity.x > 0) { //direita
45                  spriteRen.flipX = false;
46              } else if (rig.velocity.y < 0){ //esquerda
47                  spriteRen.flipX = true;
48              }
49
50          } else {
51              StopMoving();
52          }
53      }
```

Fonte: O Autor

Listing B.4 – Métodos e Funções do Script PattrolingMisery, Fonte: O Autor

```
1       public GameObject pointA, pointB, pointC, pointD;
2       public Transform[] patrolPoints;
3       public int targetPoint;
4       private Transform currentPoint;
```

```
5       private Rigidbody2D rig;
6
7       void PatrollingBehavior()
8       {
9           if(transform.position == patrolPoints[targetPoint].position)
10          {
11              increaseTargetInt();
12          }
13          transform.position = Vector3.MoveTowards(transform.position,
                patrolPoints[targetPoint].position, moveSpeed * Time.deltaTime
                );
14      }
15
16      void increaseTargetInt(){
17          targetPoint++;
18          if(targetPoint >= patrolPoints.Length){
19              targetPoint = 0;
20          }
21      }
22
23
24      private void OnDrawGizmosSelected()
25      {
26          Gizmos.DrawWireSphere(pointA.transform.position, 0.2f);
27          Gizmos.DrawWireSphere(pointB.transform.position, 0.2f);
28          Gizmos.DrawWireSphere(pointC.transform.position, 0.2f);
29          Gizmos.DrawWireSphere(pointD.transform.position, 0.2f);
30
31          Gizmos.DrawLine(pointA.transform.position, pointB.transform.
                position);
32          Gizmos.DrawLine(pointB.transform.position, pointC.transform.
                position);
33          Gizmos.DrawLine(pointC.transform.position, pointD.transform.
                position);
34          Gizmos.DrawLine(pointD.transform.position, pointA.transform.
                position);
35      }
```

Fonte: O Autor

Listing B.5 – Scripts de Parâmetros de inimigo utilizado no Nível 02

```
1
2   public class EnemySecond : MonoBehaviour, IDamageable, IEnemyMoveable,
        ITriggerCheckable
3   {
4       [field: SerializeField] public float MaxHealth { get; set; } = 15f;
5       public float CurrentHealth { get; set; }
6       public Rigidbody2D rbody { get; set; }
```

```
7       public bool IsFacingRight { get; set; } = true;
8       public bool IsAggroed { get; set; }
9       public bool IsWithinStrikingDistance { get; set; }
10
11      public bool IsWithinPhysicalATKdistance { get; set; }
12
13      #region State Machine Variables
14      public EnemyStateMachine StateMachine { get; set; }
15      public EnemyIdleState IdleState { get; set; }
16      public EnemyChaseState ChaseState { get; set; }
17      public EnemyAttackState AttackState { get; set; }
18
19      #endregion
20
21      #region Summoning Attack Variables
22
23      public GameObject phantomPrefab;
24      public Animator anim;
25
26      public GameObject summonPointA;
27      public GameObject summonPointB;
28      public GameObject summonPointC;
29      #endregion
30
31      #region Idle Variables
32      public float RandomMovementRange = 5f;
33      public float RandomMovementSpeed = 2f;
34      #endregion
35
36      private void Awake()
37      {
38          StateMachine = new EnemyStateMachine();
39          IdleState = new EnemyIdleState(this, StateMachine);
40          ChaseState = new EnemyChaseState(this, StateMachine);
41          AttackState = new EnemyAttackState(this, StateMachine);
42      }
43
44      private void Start()
45      {
46          CurrentHealth = MaxHealth;
47
48          rbody = GetComponent<Rigidbody2D> ();
49          anim = GetComponentInChildren<Animator> ();
50          StateMachine.Initialize(IdleState);
51      }
52
53      private void Update()
54      {
```

```
55          StateMachine.CurrentEnemyState.FrameUpdate();
56      }
57
58      private void FixedUpdate()
59      {
60          StateMachine.CurrentEnemyState.PhysicsUpdate();
61      }
62
63      #region Health/Die Functions
64      public void Damage(float damageAmout)
65      {
66          CurrentHealth -= damageAmout;
67          anim.SetTrigger("Damage");
68          AudioController.current.PlayMusic(AudioController.current.
                bossHit);
69
70          if (CurrentHealth <= 0) {
71              anim.SetBool("isDead", true);
72              AudioController.current.PlayMusic(AudioController.current.
                    bossDeath);
73              Die();
74          }
75      }
76
77      public void Die()
78      {
79          Destroy(this.gameObject, 2f);
80          Time.timeScale = 0f;
81          SceneManager.LoadScene("VictoryLevel02", LoadSceneMode.Single);
82          //tela de vitoria aqui e som do boss morrendo
83      }
84
85      #endregion
86
87      #region Movement
88
89      public void MoveEnemy(Vector2 velocity)
90      {
91          rbody.velocity = velocity;
92          CheckForLeftOrRightFacing(velocity);
93      }
94      #endregion
95
96      #region Animation Triggers
97
98      private void AnimationTriggerEvent(AnimationTriggerType triggerType)
99      {
100         StateMachine.CurrentEnemyState.AnimationTriggerEvent(triggerType
```

```
            );
    }

    public enum AnimationTriggerType
    {
        EnemyDamaged,
        PlayFootsteps,
    }
    #endregion

    #region Distance checks
    public void SetAggroStatus(bool isAggroed)
    {
        IsAggroed = isAggroed;
    }

    public void SetStrikingDistanceBool(bool isWithinStrikingDistance)
    {
        IsWithinStrikingDistance = isWithinStrikingDistance;
    }
    #endregion

    #region Collision detection

    private void OnTriggerEnter2D(Collider2D collision)
    {

        if (collision.gameObject.CompareTag("PlayerBullets"))
        {
            Damage(1f);
            Debug.Log("Vida do boss: " + CurrentHealth);
        }
    }

    #endregion
}
```

Fonte: O Autor

APÊNDICE C – SCRIPTS DE MECÂNICAS ESPECÍFICAS DOS NÍVEIS

Listing C.1 – Script Parallax

```
public class Parallax : MonoBehaviour
{
    public GameObject MainCamera;
    private float length, startPos;
    public float speedParallax;

    void Start()
    {
        startPos = transform.position.x;
        length = GetComponent<SpriteRenderer>().bounds.size.x;
    }


    void FixedUpdate()
    {
        float temp = (MainCamera.transform.position.x * (1 -
            speedParallax));
        float dist = (MainCamera.transform.position.x * speedParallax);

        transform.position = new Vector3(startPos + dist, transform.
            position.y, transform.position.z);

        if(temp > startPos + length / 2)
        {
            startPos += length;
        }
        else if(temp < startPos - length / 2)
        {
            startPos -= length;
        }
    }
}
```

Fonte: O Autor

Listing C.2 – Script Parallax2

```
public class Parallax2 : MonoBehaviour
{
    public GameObject MainCamera;
    private float length, startPos;
    public float speedParallax;

    void Start()
    {
```

```
9           startPos = transform.position.x;
10          length = GetComponent<SpriteRenderer>().bounds.size.x;
11      }
12
13      void FixedUpdate()
14      {
15          float temp = (MainCamera.transform.position.x * (1 -
                speedParallax));
16          float dist = (MainCamera.transform.position.x * speedParallax);
17
18          transform.position = new Vector3 (startPos + dist, transform.
                position.y, transform.position.z);
19
20          if(temp > startPos + length)
21          {
22              startPos += length;
23          }else if(temp < startPos - length){
24              startPos -= length;
25          }
26      }
27  }
```

Fonte: O Autor

Listing C.3 – Trechos do script EnvironmentController

```
1       void Start()
2       {
3
4           UpdateScoreText();
5           if (playerScore == 0)
6           {
7               dialogPrefab.SetActive(true);
8           }
9           else dialogPrefab.SetActive(false);
10
11          if (restartButton != null)
12          {
13              restartButton.onClick.AddListener(() => RestartGame(
                    SceneManager.GetActiveScene().name));
14          }
15      }
16
17       public void SetLives(int life)
18       {
19           playerLifes += life;
20           if(playerLifes >= 0 ) { UpdateScoreText(); }
21
22       }
```

```
23
24      public void UpdateScoreText()
25      {
26          scoreText.text = playerScore.ToString();
27          lifesText.text = playerLifes.ToString();
28      }
29
30      public void ShowGameOver()
31      {
32          Debug.Log("Ativamos o game over");
33          gameOver.SetActive(true);
34      }
35
36      public void RestartGame(string lvlName)
37      {
38          playerLifes = 1;
39          gameOver.SetActive(false);
40          SceneManager.LoadScene(lvlName);
41      }
42
43      public void ShowWinningScreen()
44      {
45          winningScreen.SetActive(true);
46      }
```

Fonte: O Autor

Listing C.4 – Trecho inicial do script RoundCount

```
1  public class RoundCount : MonoBehaviour
2  {
3      //dados dos personagens para quando acabar o round
4      [SerializeField] Life playerData;
5      [SerializeField] Life bossData;
6      private TimeCounter timeCount;
7      float resultsTime = 4f;
8
9      [SerializeField] Transform playerTransform;
10     [SerializeField] Transform bossTransform;
11     //
12     private PlayerWalk playerWalk;
13     private PlayerCombo playerCombo;
14     //
15
16     public Image[] lifeIcon;
17     public Sprite full;
18     public Sprite hollow;
19
20     private bool isUpdateEnabled;
```

```
21
22      void Start()
23      {
24          playerData = GameObject.Find("Player").GetComponent<Life>();
25          bossData = GameObject.Find("Enemy Test").GetComponent<Life>();
26          timeCount = GameObject.Find("Main Camera").GetComponent<
                TimeCounter>();
27
28          playerWalk = GameObject.Find("Player").GetComponent<PlayerWalk
                >();
29          playerCombo = GameObject.Find("Player").GetComponent<
                PlayerCombo>();
30
31          Debug.Log($"Scene Reloaded - Player and boss Initialized");
32
33          if (RoundSingleton.Instance.FinalRound() && (RoundSingleton.
                Instance.finalWinner == RoundSingleton.Side.Left))
34          {
35              isUpdateEnabled = false;
36              victoryScreen.SetActive(true);//ATIVAR A TELA FINAL AQUI
37          }
38          else if (RoundSingleton.Instance.FinalRound() && (RoundSingleton
                .Instance.finalWinner == RoundSingleton.Side.Right))
39          {
40              isUpdateEnabled = false;
41              gameOverScreen.SetActive(true);//ATIVAR TELA DE DERROTA AQUI
42              RoundSingleton.Instance.Reset();
43          }
44          else isUpdateEnabled = true;
45
46          UpdateHUD();
47      }
```

Fonte: O Autor

APÊNDICE D – SCRIPT BOSSCOMBAT

Listing D.1 – Código responsável pelos ataques do chefe no nível 03

```
public class BossCombat : MonoBehaviour
{
    public LayerMask playerLayer;
    private PlayerWalk playerDirection;

    [SerializeField] AudioManager audioManager;

    #region Dados de ataque
    [SerializeField] Transform punchAttack;
    [SerializeField] Transform specialAttack;
    public float punchRange = 0.5f;
    private float punchDamage = 16;
    public float repulsionForce = 100f;

    public float repulsionY = 0.9f;
    public float repulsionX = -4f;
    #endregion

    private void Awake()
    {
        bossRb = GameObject.Find("Enemy Test").GetComponentInParent<
            Rigidbody2D>();
        bossAnim = GetComponentInChildren<Animator>();
        playerDirection = GameObject.Find("Player").GetComponent<
            PlayerWalk>();
        audioManager= GameObject.FindGameObjectWithTag("Audio").
            GetComponent<AudioManager>();
    }

    private void Start()
    {
        originalGravity = bossRb.gravityScale;
    }

    //Dados funciondo no componente "Sprite" do boss
    public void bossBackdash()
    {
        if (playerDirection.isFacingRight == true)
        {
            bossRb.velocity = new Vector2(speedDash, bossRb.velocity.y);
            //bossAnim.SetBool("Backdash", true);
        }
        else
        {
```

```
42              bossRb.velocity = new Vector2(-speedDash, bossRb.velocity.y)
                    ;
43              //bossAnim.SetBool("Backdash", true);
44          }
45      }
46
47      public void bossAttack1()
48      {
49          Collider2D[] hitPlayer = Physics2D.OverlapCircleAll(punchAttack.
                position, punchRange, playerLayer);
50
51          foreach (Collider2D playerCol in hitPlayer)
52          {
53
54              Life player = playerCol.GetComponent<Life>();
55              //aplicar for a de repuls o
56              Rigidbody2D playerRb = playerCol.GetComponent<Rigidbody2D>()
                    ;
57
58              if (PlayerDefense.isDefending == true && playerRb != null)
59              {
60                  audioManager.PlaySFX(audioManager.bossPunches);
61                  player.TakeDamage(punchDamage / 2); //diminuimos o dano
                        pegando a vari vel isDefending
62                  StartCoroutine(dealingDamage(0.3f));
63              }
64              else if (PlayerDefense.isDefending == false && playerRb !=
                    null)
65              {
66                  float directionX;
67                  if(playerDirection.isFacingRight == true) { directionX =
                        repulsionX; } else {  directionX = -repulsionX; }
68
69
70                  Vector2 repulsionDirection = (Vector2)playerRb.position
                        - (Vector2)punchAttack.position;
71                  repulsionDirection.Normalize();
72
73
74                  repulsionDirection.y += repulsionY;
75                  repulsionDirection.x += directionX;//repulsionX;
76
77                  player.TakeDamage(punchDamage);
78                  playerDirection.canMove = false;
79
80                  playerRb.AddForce(repulsionDirection * repulsionForce,
                        ForceMode2D.Force);
81                  StartCoroutine(dealingDamage(1f));
```

```
82                  audioManager.PlaySFX(audioManager.bossPunches);
83              }
84          }
85
86      }
87
88      void BossAttack2()
89      {
90          //Debug.Log("acertamos o ataque2");
91
92          Collider2D[] hitPlayer = Physics2D.OverlapCircleAll(punchAttack.
                position, punchRange * 2, playerLayer);
93
94          foreach (Collider2D playerCol in hitPlayer)
95          {
96              Life player = playerCol.GetComponent<Life>();
97              //aplicar for a de repuls o
98              Rigidbody2D enemyRb = playerCol.GetComponent<Rigidbody2D>();
99
100             if (PlayerDefense.isDefending == true && enemyRb != null)
101             {
102                 float directionX;
103                 if (playerDirection.isFacingRight == true) { directionX
                        = repulsionX; } else { directionX = -repulsionX; }
104
105                 Vector2 repulsionDirection = (Vector2)enemyRb.position -
                        (Vector2)punchAttack.position;
106                 repulsionDirection.Normalize();
107
108
109                 repulsionDirection.y += repulsionY;
110                 repulsionDirection.x += 0.2f;//repulsionX;
111
112                 player.TakeDamage(punchDamage / 3);
113                 enemyRb.AddForce(repulsionDirection * repulsionForce,
                        ForceMode2D.Force);
114                 StartCoroutine(dealingDamage(0.3f));
115                 audioManager.PlaySFX(audioManager.bossPunches);
116                 return;
117             }
118             else if (PlayerDefense.isDefending == false && enemyRb !=
                    null)
119             {
120                 //primeiro hit do combo
121                 float directionX;
122                 if (playerDirection.isFacingRight == true) { directionX
                        = repulsionX; } else { directionX = -repulsionX; }
123
```

```
124                 Vector2 repulsionDirection = (Vector2)enemyRb.position -
                        (Vector2)punchAttack.position;
125                 repulsionDirection.Normalize();
126
127                 repulsionDirection.y += 15;
128                 repulsionDirection.x += directionX;
129
130                 player.TakeDamage(punchDamage * 2);
131                 playerDirection.canMove = false;
132
133                 enemyRb.AddForce(repulsionDirection * (repulsionForce),
                        ForceMode2D.Force);
134                 audioManager.PlaySFX(audioManager.bossPunches);
135                 StartCoroutine(dealingDamage(1f));
136             }
137         }
138     }
139
140     public void bossComboFirstHit()
141     {
142
143         Collider2D[] hitPlayer = Physics2D.OverlapCircleAll(punchAttack.
                position, punchRange, playerLayer);
144
145         foreach (Collider2D playerCol in hitPlayer)
146         {
147             Life player = playerCol.GetComponent<Life>();
148             //aplicar for a de repuls o
149             Rigidbody2D enemyRb = playerCol.GetComponent<Rigidbody2D>();
150
151             if (PlayerDefense.isDefending == true && enemyRb != null)
152             {
153                 //fazer ele parar o ataque aqui
154                 audioManager.PlaySFX(audioManager.bossPunches);
155                 return;
156             }
157             else if (PlayerDefense.isDefending == false && enemyRb !=
                    null)
158             {
159                 //primeiro hit do combo
160                 float directionX;
161                 if (playerDirection.isFacingRight == true) { directionX
                        = repulsionX; } else { directionX = -repulsionX; }
162
163                 Vector2 repulsionDirection = (Vector2)enemyRb.position -
                        (Vector2)punchAttack.position;
164                 repulsionDirection.Normalize();
165
```

```
166                 repulsionDirection.y += repulsionY;
167                 repulsionDirection.x += directionX;
168 
169                 player.TakeDamage(punchDamage / 2);
170                 playerDirection.canMove = false;
171 
172                 enemyRb.AddForce(repulsionDirection * (repulsionForce *
                        0.6f), ForceMode2D.Force);
173                 audioManager.PlaySFX(audioManager.bossPunches);
174                 StartCoroutine(dealingDamage(1f));
175             }
176         }
177     }
178 
179     public void BossComboSecondHit()
180     {
181 
182 
183         Collider2D[] hitPlayer = Physics2D.OverlapCircleAll(punchAttack.
                position, punchRange, playerLayer);
184 
185         foreach (Collider2D playerCol in hitPlayer)
186         {
187             Life player = playerCol.GetComponent<Life>();
188             //aplicar for a de repuls o
189             Rigidbody2D enemyRb = playerCol.GetComponent<Rigidbody2D>();
190 
191             if (PlayerDefense.isDefending == false && enemyRb != null)
192             {
193                 //segundo hit do combo
194                 float directionX;
195                 Vector2 repulsionDirection = (Vector2)enemyRb.position -
                        (Vector2)punchAttack.position;
196                 repulsionDirection.Normalize();
197                 if (playerDirection.isFacingRight == true) { directionX
                        = repulsionX; } else { directionX = -repulsionX; }
198 
199                 repulsionDirection.y += 8f;
200                 repulsionDirection.x = 0f;
201 
202                 player.TakeDamage(punchDamage / 2);
203                 playerDirection.canMove = false;
204 
205                 enemyRb.AddForce(repulsionDirection * (repulsionForce),
                        ForceMode2D.Force);
206                 audioManager.PlaySFX(audioManager.bossPunches);
207                 StartCoroutine(dealingDamage(1f));
208             }
```

```
209         }
210     }
211 
212     public void BossComboThirdHit()
213     {
214 
215         Collider2D[] hitPlayer = Physics2D.OverlapCircleAll(punchAttack.
               position, punchRange, playerLayer);
216 
217         foreach (Collider2D playerCol in hitPlayer)
218         {
219             Life player = playerCol.GetComponent<Life>();
220             //aplicar for a de repuls o
221             Rigidbody2D enemyRb = playerCol.GetComponent<Rigidbody2D>();
222 
223             if (PlayerDefense.isDefending == false && enemyRb != null)
224             {
225                 //hit mais forte do combo
226                 float directionX;
227                 Vector2 repulsionDirection = (Vector2)enemyRb.position -
                       (Vector2)punchAttack.position;
228                 repulsionDirection.Normalize();
229                 if (playerDirection.isFacingRight == true) { directionX
                       = repulsionX; } else { directionX = -repulsionX; }
230 
231                 repulsionDirection.y += repulsionY;
232                 repulsionDirection.x += directionX;
233 
234                 player.TakeDamage(punchDamage);
235                 playerDirection.canMove = false;
236 
237                 enemyRb.AddForce(repulsionDirection * (repulsionForce *
                       5f), ForceMode2D.Force);
238                 audioManager.PlaySFX(audioManager.bossPunches);
239                 StartCoroutine(dealingDamage(1f));
240 
241             }
242         }
243     }
244     #region Ataque Especial do chefe
245     [SerializeField] private GameObject[] _virtualCameras;
246     [SerializeField] private float timeLeft = 3.0f;
247     public void BossSpecialAttack()
248     {
249         playerDirection.canMove = false;
250 
251         Collider2D[] hitPlayer = Physics2D.OverlapCircleAll(
               specialAttack.position, (float)specialAttack.GetComponent<
```

```
            CircleCollider2D>().radius, playerLayer);
252
253         CameraChange();
254
255         foreach (Collider2D playerCol in hitPlayer)
256         {
257             Life player = playerCol.GetComponent<Life>();
258             //aplicar for a de repuls o
259             Rigidbody2D enemyRb = playerCol.GetComponent<Rigidbody2D>();
260
261             if (PlayerDefense.isDefending == true && enemyRb != null) //
                    vai ter repuls o mesmo na defesa por ser um special,
                    quero passar a sensa  o de for a
262             {
263                 float directionX;
264                 if (playerDirection.isFacingRight == true) { directionX
                        = repulsionX; } else { directionX = -repulsionX; }
265
266                 Vector2 repulsionDirection = (Vector2)enemyRb.position -
                        (Vector2)punchAttack.position;
267                 repulsionDirection.Normalize();
268
269                 repulsionDirection.y += repulsionY * 10;
270                 repulsionDirection.x += directionX * 0.1f;
271
272                 //aplicar metade do dano do special aqui
273                 enemyRb.AddForce(repulsionDirection * (repulsionForce *
                        0.2f), ForceMode2D.Force);
274                 player.TakeDamage(punchDamage * 3);
275                 audioManager.PlaySFX(audioManager.bossSpecialATK);
276                 return;
277             }
278             else if (PlayerDefense.isDefending == false && enemyRb !=
                    null)
279             {
280                 float directionX;
281                 if (playerDirection.isFacingRight == true) { directionX
                        = -repulsionX; } else { directionX = repulsionX; }
282
283                 Vector2 repulsionDirection = (Vector2)enemyRb.position -
                        (Vector2)punchAttack.position;
284                 repulsionDirection.Normalize();
285
286                 repulsionDirection.y += repulsionY * 30;
287                 repulsionDirection.x += directionX * repulsionX;
288
289                 player.TakeDamage(punchDamage * 6);
290                 playerDirection.canMove = false;
```

```
291
292                 enemyRb.AddForce(repulsionDirection * repulsionForce,
                        ForceMode2D.Force);
293                 audioManager.PlaySFX(audioManager.bossSpecialATK);
294                 StartCoroutine(dealingDamage(1f));
295
296             }
297         }
298         StartCoroutine(CameraReturn());
299     }
300
301
302     void CameraChange()
303     {
304         _virtualCameras[1].SetActive(true);
305     }
306
307     IEnumerator CameraReturn()
308     {
309         Time.timeScale = 0.001f;
310         yield return new WaitForSecondsRealtime(timeLeft);
311         _virtualCameras[1].SetActive(false);
312         Time.timeScale = 1f;
313     }
314     #endregion
315
316     //Aqui eu desativo os controles do jogador quando ele estiver
            ativamente apanhando
317     IEnumerator dealingDamage(float timeToMove)
318     {
319         yield return new WaitForSecondsRealtime(timeToMove);
320         playerDirection.canMove = true;
321     }
322
323     private void OnDrawGizmosSelected()
324     {
325         if (punchAttack == null)
326             return;
327
328         Gizmos.DrawWireSphere(punchAttack.position, punchRange);
329     }
330 }
```

Fonte: O Autor