Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

ALTAMIRO ARRUDA DE FIGUEIREDO NETO

DESENVOLVIMENTO DE UM JOGO 2D METROIDVANIA UTILIZANDO A ENGINE UNITY

Assis/SP
2021

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

ALTAMIRO ARRUDA DE FIGUEIREDO NETO

DESENVOLVIMENTO DE UM JOGO 2D METROIDVANIA UTILIZANDO A ENGINE UNITY

Trabalho de Conclusão de Curso apresentado ao Curso de Análise e Desenvolvimento de Sistemas do Instituto Municipal do Ensino Superior de Assis – IMESA e Fundação Educacional do Município de Assis – FEMA, como requisito para a obtenção do Certificado de Conclusão.

Orientando: Altamiro Arruda de Figueiredo Neto
Orientadora: Me.Diomara Martins Reigato Barros

Assis/SP
2021

FICHA CATALOGRÁFICA

F475d FIGUEIREDO NETO, Altamiro Arruda de
Desenvolvimento de um jogo 2D metroidvania utilizando a engine unity / Altamiro Arruda de Figueiredo Neto. – Assis, 2021.

56p.

Trabalho de conclusão do curso (Análise e Desenvolvimento de Sistemas). – Fundação Educacional do Município de Assis - FEMA

Orientadora: Me. Diomara Martins Reigato Barros

1.Jogos 2.C# 3.Unity

CDD005.133

DESENVOLVIMENTO DE UM JOGO 2D METROIDVANIA UTILIZANDO A ENGINE UNITY

ALTAMIRO ARRUDA DE FIGUEIREDO NETO

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, avaliado pela seguinte comissão examinadora:

Orientador: _____
Me.Diomara Martins Reigato Barros

Examinador: _____
Me.Douglas Sanches da Cunha

Assis/SP
2021

AGRADECIMENTOS

Em primeiro lugar agradeço meus familiares e amigos, por terem me apoiado e encorajado a seguir adiante com meus estudos. A minha orientadora, que me guiou e auxiliou na realização deste projeto. E também agradeço imensamente aos meus professores, por todo o conhecimento obtido durante a duração do curso. Por fim agradeço e dedico este trabalho de conclusão de curso ao meu pai, que me apoiou e me deu forças para seguir adiante e concluir esta etapa da minha jornada.

RESUMO

O trabalho em questão apresenta o desenvolvimento de um jogo eletrônico em 2D, Side-scrolling, MetroidVania utilizando a Engine Unity com Visual Studio Code e C#, através da elaboração de menus, sistema de salvamento de progresso, movimentação de personagem e interações com o ambiente e inimigos. O objetivo principal do desenvolvimento deste trabalho é o estudo e aprendizado das metodologias que auxiliam na elaboração e desenvolvimento de jogos eletrônicos bem como a utilização na prática.
Palavras-chave: Desenvolvimento de Jogos, C#, Unity, Jogos 2D.

ABSTRACT

The work in question presents the development of a 2D electronic game, side-scrolling, MetroidVania using the Unity Engine with Visual Studio Code and C#, through the elaboration of menus, progress saving system, character movement and interactions with the environment and enemies. The main objective in the development of this work is the study and learning of the methodologies that help in the elaboration and development of electronic games as well as the use in practice.

Keywords: Game Development, C#, Unity, 2D Games.

LISTA DE ILUSTRAÇÕES

Figura 1: Diagrama de Caso de Uso Menu Principal 16
Figura 2: Diagrama de Caso de Uso Menu de Pausa 18
Figura 3: Diagrama de Caso de Uso Menu de Inventário 20
Figura 4: Diagrama de Caso de Uso Interações do Inimigo 22
Figura 5: Diagrama de Caso de Uso Interações do Usuário 24
Figura 6: Diagrama de Caso de Uso Funções do Sistema 27
Figura 7: Cronograma 29
Figura 8: Código do menu principal 32
Figura 9: Código inventário 33
Figura 10: Código interface do inventario 34
Figura 11: Código dados de salvamento 35
Figura 12: Código sistema de salvamento 36
Figura 13: Código inimigo 37
Figura 14: Código inimigo continuação 38
Figura 15: Código inimigo Continuação 39
Figura 16: Código inimigo continuação 40
Figura 17: Código inimigo final 41
Figura 18: Código personagem 42
Figura 19: Código personagem continuação 43
Figura 20: Código personagem continuação 44
Figura 21: Código personagem continuação 45
Figura 22: Código personagem continuação 46
Figura 23: Código personagem continuação 47
Figura 24: Código personagem continuação 48

Figura 25: Código personagem fim.....49
Figura 26: Arte Personagem.....50
Figura 27: Arte inimigo 1.....50
Figura 28: Arte inimigo 2.....51
Figura 29: Arte inimigo 3.....51
Figura 30: Arte inimigo 4.....51
Figura 31: Arte cena de créditos.....52
Figura 32: Arte poções.....52
Figura 33: Arte tocha 52
Figura 34: Arte barra de vida 53
Figura 35: Arte menu de inventário.....53
Figura 36: Textura fase.....53
Figura 37: Textura de fundo primeira camada.....53
Figura 38: Textura de fundo segunda camada 54
Figura 39: Textura de fundo terceira camada.....54
Figura 40: Textura de fundo menu principal 54

SUMÁRIO

1.INTRODUÇÃO 11

1.1 OBJETIVO 12

1.2. JUSTIFICATIVAS 12

1.3. MOTIVAÇÃO 12

1.4. PERSPECTIVA DE CONTRIBUIÇÃO 12

1.5. PÚBLICO ALVO 13

2.TECNOLOGIAS UTILIZADAS PARA O DESENVOLVIMENTO DO JOGO 14

2.1. TECNOLOGIAS UTILIZADAS NA ANÁLISE 14

2.2 TECNOLOGIAS UTILIZADAS NO DESENVOLVIMENTO 14

2.2.1. Engine Unity 14

2.2.2. C# 15

2.2.3. Visual Studio Code 15

3. ANÁLISE E ESPECIFICAÇÃO DO JOGO 16

3.1. DIAGRAMAS DE CASO DE USO DO JOGO 16

3.1.1. UC01- MENU PRINCIPAL 16

3.1.2. UC02- MENU DE PAUSA 18

3.1.3. UC03- MENU DE INVENTÁRIO 20

3.1.4. UC04- INTERAÇÕES DO INIMIGO 22

3.1.5. UC05- INTERAÇÕES DO USUÁRIO 24

3.1.6. UC06- FUNÇÕES DO SISTEMA 27

3.2. CRONOGRAMA 29

4. DESENVOLVIMENTO DO JOGO 30

4.1. DESENVOLVIMENTO DOS ELEMENTOS 30

4.1.1 PERSONAGEM 30

4.1.2 INIMIGO 30

4.1.3 MENUS 30

4.1.4 SISTEMA DE SALVAMENTO DE PROGRESSO 31

4.1.5 INVENTARIO 31

4.1.6 FASE 31

4.2. CONTROLES 31

4.3. OBJETIVO DO JOGO 31
4.4. PRINCIPAIS CÓDIGOS FONTE DO JOGO 32
4.5. TEXTURAS E ARTES UTILIZADAS NO PROJETO 50
CONCLUSÃO 55
REFERÊNCIAS 56

1.INTRODUÇÃO

Existem vários gêneros, estilos, temáticas e mecânicas para o desenvolvimento do enredo de um game dando a liberdade de criar qualquer coisa que o desenvolvedor queira fazer como, por exemplo, uma aventura em um mundo medieval em que o personagem principal do enredo seja o jogador vivenciando a jornada de um cavaleiro atrás de respostas sobre determinado acontecimento e encontrando vários inimigos e tesouros em seu caminho, como também ter uma experiência imersiva sobre as grandes guerras mundiais.

Jogos digitais são atualmente um dos maiores mercados mundiais que, movimentam um grande capital e vem se expandido ano após ano com novas tecnologias e conceitos. Segundo uma reportagem do blog IMMAKERS (2020): o mercado brasileiro atualmente é o 13.º maior mercado do mundo em relação ao consumo de games. Suas principais plataformas para a utilização são: Personal Computer (PC), consoles (Xbox, Playstation) e os dispositivos mobile (Tablets, smartphones).

Conforme a reportagem do site WHOW (2020): o Brasil é um dos maiores consumidores de jogos eletrônicos do mundo que chega a movimentar anualmente 1,5 bilhões por ano e tem sua estimativa de crescimento de 5,3% até 2022. Em contrapartida, o desenvolvimento de games no país é muito menosprezado e não tem o devido estímulo para essa área de atuação, sendo assim se desenvolve a passos lentos. Consequentemente fazendo com que grandes desenvolvedores brasileiros dessa área tenham que procurar emprego e estímulos fora de nossa nação para poder trabalhar na indústria de jogos eletrônicos e se desenvolverem nesta área da tecnologia.

O mercado de jogos tem atualmente passado por várias mudanças além de novas tecnologias para o desenvolvimento de jogos principalmente os games “Indie” ou independentes, feitos por um grupo pequeno de desenvolvedores sem a influência de grandes empresas da área e possuindo um orçamento pequeno ou quase nulo, e utilizam na maioria das vezes ferramentas gratuitas para o desenvolvimento dos jogos. De acordo com Machado, Santuchi e Carletti (2018), o futuro econômico brasileiro depende dos profissionais desta área que precisam estar preparados para as adversidades que surgem no caminho, pois o Brasil tem muito a ganhar com investimentos na área de entretenimento eletrônico, em geral.

“Países como o Brasil possuem profissionais com grande talento porem pela falta de investimento interno, os mesmos saem à procura de seus objetivos em outros países, onde jogar vídeo game é tratado como um assunto sério” (Machado; Santuchi; Carletti, 2018, p.15).

1.1 OBJETIVO

O presente trabalho tem por objetivo desenvolver um jogo utilizando o motor Unity e a linguagem C# para a criação de um jogo 2D de forma Indie com o gênero ação e aventura no estilo MetroidVania. É também objetivo do presente trabalho apresentar a análise de todo o desenvolvimento do jogo.

1.2. JUSTIFICATIVAS

A justificativa para o desenvolvimento do trabalho proposto é o de conhecer os métodos, ferramentas e recursos para desenvolver jogos, assim adquirir conhecimento na área de desenvolvimento de jogos eletrônicos.

1.3. MOTIVAÇÃO

Atualmente o mercado de jogos eletrônicos é um dos maiores do mundo com os periféricos e hardwares para sua utilização, que vem se expandido cada dia mais com o avanço tecnológico e melhores ferramentas e metodologias para o desenvolvimento dos mesmos.

1.4. PERSPECTIVA DE CONTRIBUIÇÃO

Ao término do desenvolvimento deste presente trabalho, espera-se que ele possa estimular e contribuir com alunos e demais interessados, compartilhando conhecimentos e métodos para auxiliar em possíveis projetos que possam ser desenvolvidos posteriormente.

1.5. PÚBLICO ALVO

O público alvo deste Software que será desenvolvido de forma não comercial e para fins acadêmicos, são pessoas de todas as idades que queiram um meio de lazer e/ou gostem de consumir jogos eletrônicos.

2.TECNOLOGIAS UTILIZADAS PARA O DESENVOLVIMENTO DO JOGO

2.1. TECNOLOGIAS UTILIZADAS NA ANÁLISE

Para a análise do projeto se utiliza a linguagem UML (Unified Modeling Language), que por sua define uma sequência de artefatos que auxilia na documentação e modelagem dos sistemas e projetos desenvolvidos. O UML possui nove categorias de diagramas para modelar e facilitar na documentação do projeto.

O diagrama de caso de uso (UC) documenta as funcionalidades do sistema de uma forma que facilita a compreensão de tais dados do projeto, mas, não se aprofunda em detalhes técnicos ou avançados do sistema. UC é um derivado das especificações de requisitos, que ajuda a mostrar quem são os atores e o que eles fazem e como interagem com o sistema e a comunicação entre essas partes (Ribeiro, 2012).

2.2 TECNOLOGIAS UTILIZADAS NO DESENVOLVIMENTO

Para o desenvolvimento do jogo é utilizado como motor gráfico o Unity que se baseia em C# quem tem capacidade para desenvolver jogos 2D e 3D, e como linguagem utilizada temos o C# que é uma linguagem orientada a objeto simples e robusta. Por fim é utilizado como ferramenta de edição de código o VISUAL STUDIO CODE, que no que lhe concerne é um ambiente de desenvolvimento integrado desenvolvido pela Microsoft.

2.2.1. Engine Unity

Unity é um motor gráfico utilizado para o desenvolvimento de jogos eletrônicos em 2D ou 3D utilizando a linguagem C# para o desenvolvimento de jogos, além de oferecer essas capacidades aos usuários, também suporta as seguintes APIs: Direct3D no Windows e Xbox 360; OpenGL no MacOS, e Linux; OpenGL ES no Android e iOS; WebGL na Internet. Além de contar com diversas facilidades e ferramentas para auxiliar no desenvolvimento de jogos em sua plataforma, como importação de texturas e sprites, configurações de mipmaps, resolução, textura, sombras, efeitos de pós processamento,

entre outros recursos para o desenvolvimento rápido e prático da aplicação (UNITY, 2020).

2.2.2. C#

O C# além de ser uma linguagem orientada a objetos e bem tipada, que permite os desenvolvedores criarem as mais variadas categorias de aplicações robustas e seguras, de forma prática e rápida executadas no ecossistema do .NET. Além de ter uma sintaxe expressiva, simples e fácil de aprender, o C# pode criar aplicações em ambiente Windows, serviços Web XML, cliente-servidor, aplicativos de banco de dados entre outros tipos (MICROSOFT, 2020).

2.2.3. Visual Studio Code

O Visual Studio Code é uma ferramenta de edição de códigos, desenvolvido pela Microsoft para Windows, Linux e macOS. Ele inclui suporte para depuração, controle Git incorporado, realce de sintaxe, complementação inteligente de código, suporta várias extensões que auxiliam na codificação do usuário, além de ser um software livre e de código aberto e ser personalizável, possui um ambiente de desenvolvimento integrado, ou seja, é um programa que pode ser usado por muitos aspectos do desenvolvimento de software (VISUAL STUDIO CODE, 2020).

3. ANÁLISE E ESPECIFICAÇÃO DO JOGO

3.1. DIAGRAMAS DE CASO DE USO DO JOGO

3.1.1. UC01- MENU PRINCIPAL

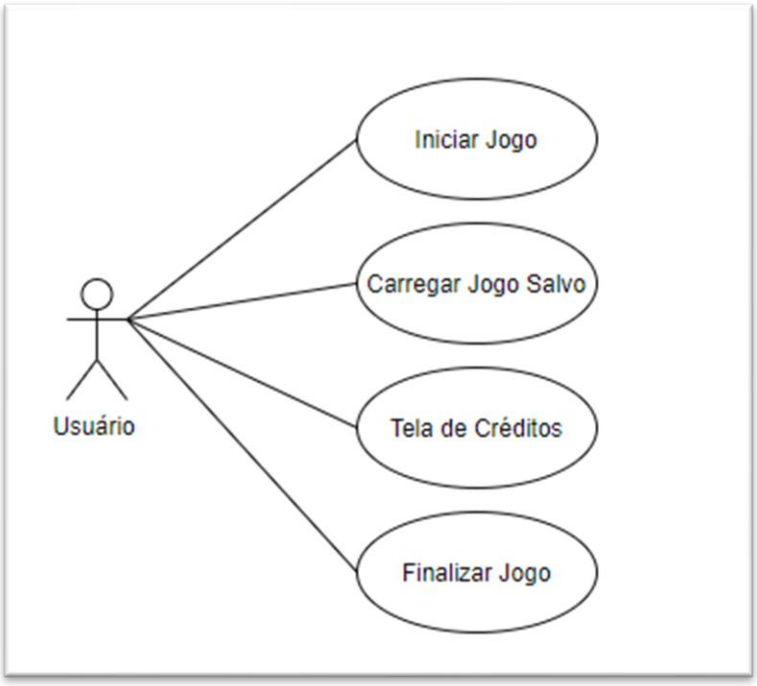

Figura 1: Diagrama de Caso de Uso Menu Principal
Fonte: Autoria Própria

IDENTIFICAÇÃO: UC01	
NOME: Menu principal	
ATORES: Usuário	
PRÉ-CONDIÇÕES: Jogo em execução	
PÓS-CONDIÇÕES: Acessar funcionalidades do jogo	
Fluxo Principal	
Ator	Sistema
1.O **Ator** inicia o caso de uso selecionando	2. O **Sistema** irá dar continuidade com a

a opção desejada [A1], [A2], [A3], [A4].	opção escolhida.
3. O **Ator** finaliza a utilização do menu principal.	4. caso de uso encerrado.

Fluxo Alternativo

A1. O **Ator** seleciona “Iniciar jogo”

a. O **Sistema** redireciona o **Ator** para a primeira fase do jogo

Volta ao passo 1

A2. O **Ator** seleciona “Carregar jogo salvo”

a. O **Sistema** verifica nos diretórios a existência do arquivo no qual contém os dados de progresso do **Ator** [E1]

b. O **Sistema** recupera os dados armazenados do jogador o redireciona para a fase e local das últimas informações registradas

Volta ao passo 1

A3. O **Ator** seleciona “Tela de créditos”

a. O **Sistema** redireciona o **Ator** para a cena de créditos

Volta ao passo 1

A4. O **Ator** seleciona “Finalizar jogo”

a. O **Sistema** finaliza a execução do jogo

Volta ao passo 1

Fluxo de Exceção

E1. O **Ator** não tem um jogo salvo nos diretórios do jogo.

a. **O Sistema** Não permite a utilização da opção “Carregar Jogo”

O Caso de uso é finalizado

3.1.2. UC02- MENU DE PAUSA

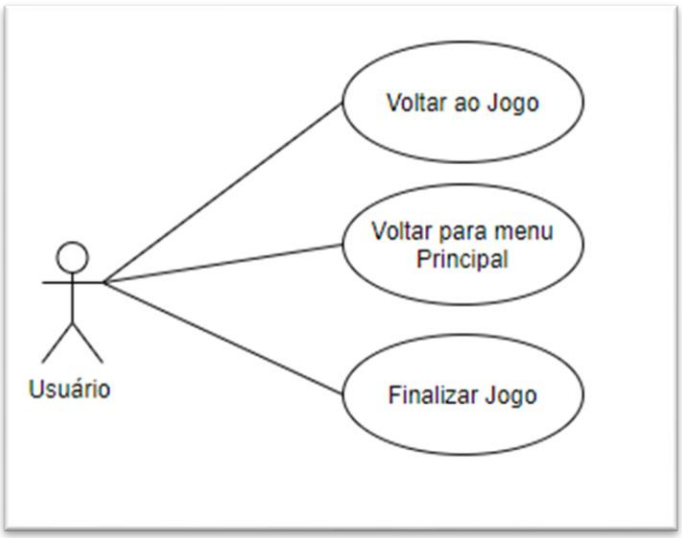

Figura 2: Diagrama de Caso de Uso Menu de Pausa
Fonte: Autoria Própria

IDENTIFICAÇÃO: UC02	
NOME: Menu de pausa	
ATORES: Usuário	
PRÉ-CONDIÇÕES: Estar em uma fase do jogo	
PÓS-CONDIÇÕES: Acessar funcionalidade menu de pausa	
Fluxo Principal	
Ator	Sistema
1.O **Ator** interage com o jogo pressionando a tecla vinculada ao menu de pausa	2. O **Sistema** irá pausar as atividades em andamento na fase e ativará o menu de pausa dentro da fase
3.O **Ator** seleciona a opção desejada [A1], [A2], [A3], [E1].	3. O **Sistema** irá dar continuidade com a opção escolhida.
5. O **Ator** finaliza a utilização do menu de Pausa.	6. caso de uso encerrado.
Fluxo Alternativo	

A1. O **Ator** seleciona “Voltar ao jogo” a. O **Sistema** irá desativa o menu de pausa e retomará as atividades em andamento na fase Volta ao passo 3
A2. O **Ator** seleciona “Voltar para o menu principal” a. O **Sistema** retorna o **Ator** para o menu principal do jogo Volta ao passo 3
A3. O **Ator** seleciona “Finalizar jogo” a. O **Sistema** finaliza a execução do jogo Volta ao passo 3
Fluxo de Exceção
E1. O **Ator** pressiona a tecla vinculada ao menu de pausa novamente a. O **Sistema** irá desativa o menu de pausa e retomará as atividades em andamento na fase O Caso de uso é finalizado

3.1.3. UC03- MENU DE INVENTÁRIO

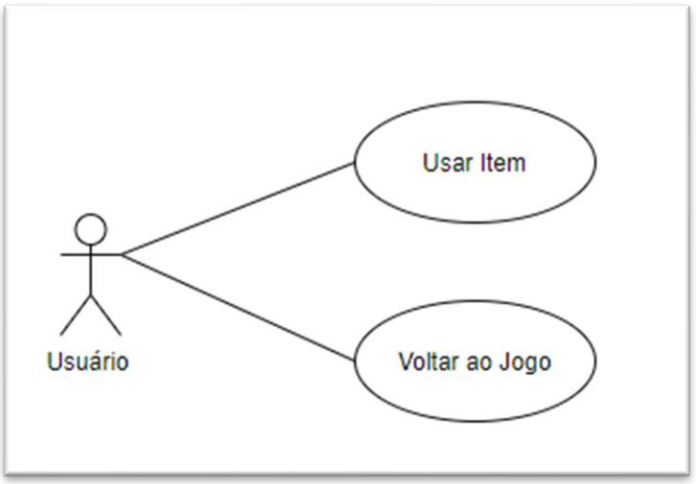

Figura 3: Diagrama de Caso de Uso Menu de Inventário
Fonte: Autoria Própria

IDENTIFICAÇÃO: UC03	
NOME: Menu de inventário	
ATORES: Usuário	
PRÉ-CONDIÇÕES: Estar em uma fase do jogo	
PÓS-CONDIÇÕES: Acessar funcionalidade menu de inventário	
Fluxo Principal	
Ator	Sistema
1.O **Ator** interage com o jogo pressionando a tecla vinculada ao inventário	2. O **Sistema** irá pausar as atividades em andamento na fase e ativará o menu de inventário dentro da fase
3.O **Ator** seleciona a opção desejada [A1], [A2], [E1].	3. O **Sistema** irá dar continuidade com a opção escolhida.
5. O **Ator** finaliza a utilização do menu de inventário.	6. caso de uso encerrado.
Fluxo Alternativo	

A1. O **Ator** com o cursor do mouse clica sobre o item que quer utilizar a. O **Sistema** irá receber as informações sobre o item Volta ao passo 3
A2. O **Ator** seleciona “Voltar para o menu principal” a. O **Sistema** retorna o **Ator** para o menu principal do jogo Volta ao passo 3
Fluxo de Exceção
E1. O **Ator** pressiona a tecla vinculada ao inventário novamente a. O **Sistema** irá desativa o menu de inventário e retomará as atividades em andamento na fase O Caso de uso é finalizado

3.1.4. UC04- INTERAÇÕES DO INIMIGO

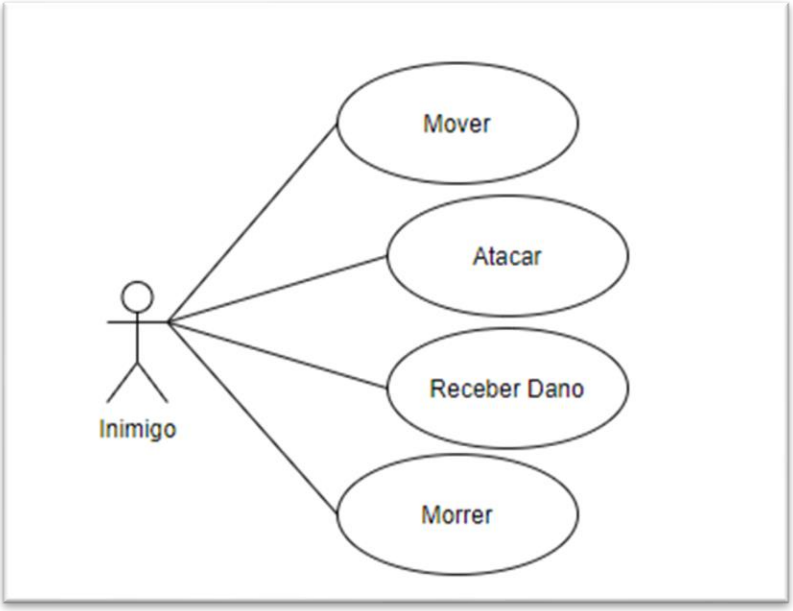

Figura 4: Diagrama de Caso de Uso Interações do Inimigo
Fonte: Autoria Própria

IDENTIFICAÇÃO: UC04	
NOME: Interações do inimigo	
ATORES: Inimigo	
PRÉ-CONDIÇÕES: Estar em uma fase do jogo	
PÓS-CONDIÇÕES: Interações com o jogador e ambiente	
Fluxo Principal	
Ator	Sistema
2.O **Ator** movimenta-se dentro das predefinições do sistema	1. O **Sistema** Define distancias padrões para a movimentação do **Ator**
4.O **Ator** fica patrulhando a área até encontrar o usuário [A1], [A2], [A3]	3. O **Sistema** Define uma área de detecção para encontrar o usuário
5. caso de uso encerrado.	
Fluxo Alternativo	

A1. O **Ator** ao encontrar o usuário move-se em direção do usuário até que o mesmo esteja dentro de sua área de ataque efetua um golpe contra o usuário a. O **Sistema** irá receber as informações do dano do golpe e fará a subtração do valor na vida do usuário Volta ao passo 4
A2. O **Ator** ao receber o dano do usuário a. O **Sistema** subtrai o dano causado pelo usuário na vida do **Ator** Volta ao passo 4
A3. Caso a vida do **Ator** chegar a zero a. O **Sistema** após alguns segundos retira o **Ator** da fase Caso de uso é finalizado

3.1.5. UC05- INTERAÇÕES DO USUÁRIO

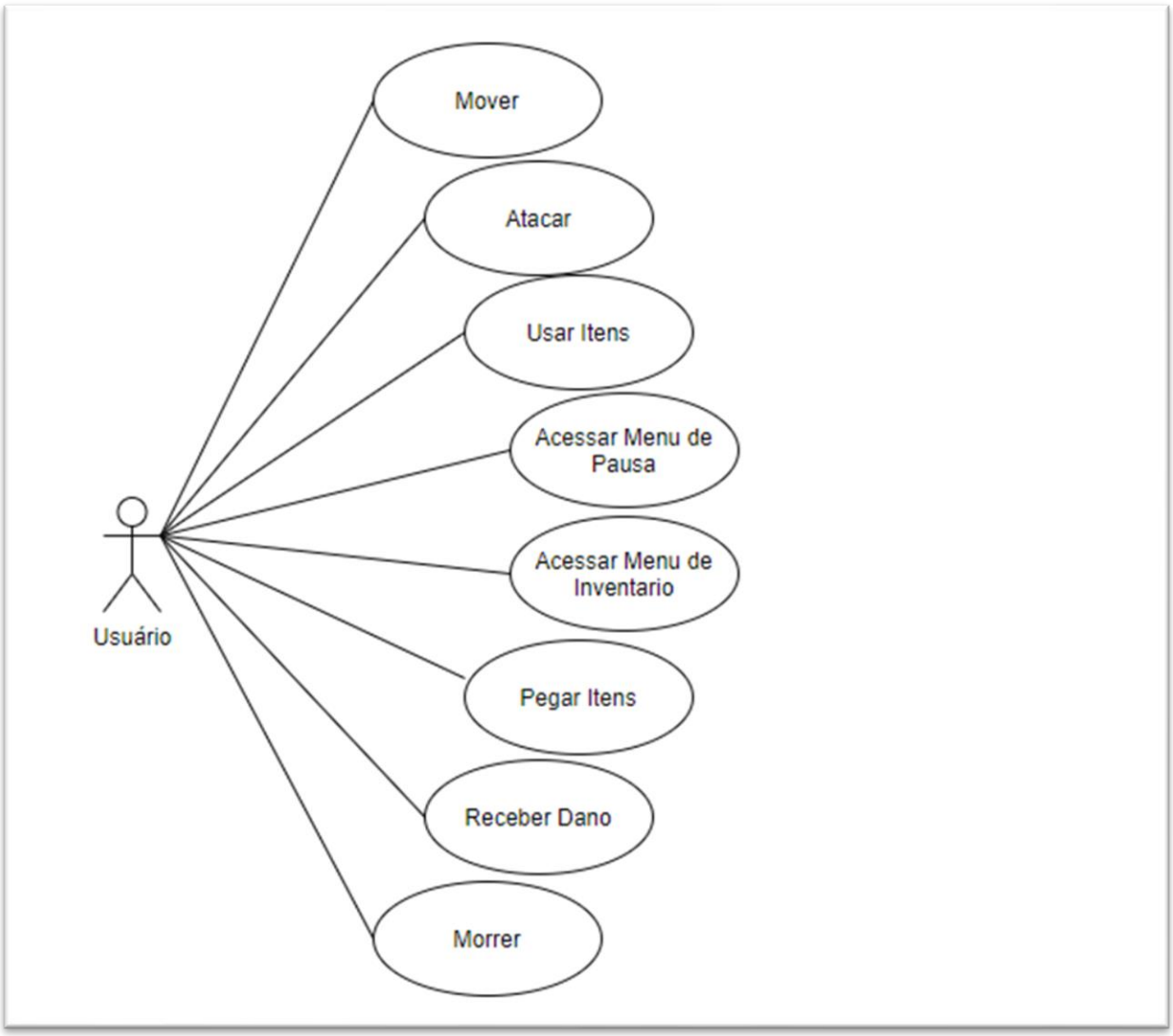

Figura 5: Diagrama de Caso de Uso Interações do Usuário
Fonte: Autoria Própria

IDENTIFICAÇÃO: UC05
NOME: Interações do usuário
ATORES: Usuário
PRÉ-CONDIÇÕES: Estar em uma fase do jogo
PÓS-CONDIÇÕES: Interações com o ambiente e inimigos
Fluxo Principal

Ator	Sistema
1.O **Ator** inicia o caso de uso interagindo com o ambiente[A1], [A2], [A3], [A4], [A5], [A6], [A7], [A8].	2. O **Sistema** irá dar continuidade de acordo com a interação escolhida.
3. O **Ator** chega ao final da fase	4. O **Sistema** passa para a próxima fase do jogo
5. Caso de uso encerrado	
Fluxo Alternativo	
A1. O **Ator** pressiona qual tecla vinculada ao movimento para o personagem andar a. O personagem se move para a direção desejada pelo usuário Volta ao passo 1	
A2. O **Ator** pressiona a tecla vinculada ao ataque a. A animação do ataque é feita[E1] Volta ao passo 1	
A3. O **Ator** utiliza um item a. O **Sistema** obtém os atributos do item e atualiza as informações do personagem b. O **Sistema** remove o Item do inventario Volta ao passo 1	
A4. O **Ator** pressiona a tecla vinculada ao Menu de Pausa a. O **Sistema** ativa o menu de pausa b. O **Ator** seleciona uma opção c. O **Sistema** retorna à opção desejada Volta ao passo 1	
A5. O **Ator** pressiona a tecla vinculada ao Menu de inventário a. O **Sistema** ativa o menu de inventário b. O **Ator** seleciona uma opção	

c. O **Sistema** retorna à opção desejada Volta ao passo 1
A6. O **Ator** se move por cima de algum item a. O **Sistema** remove o item do mapa e adiciona o item ao inventario do personagem
A7. O **Ator** ao receber o dano do Inimigo a. O **Sistema** subtrai o dano causado pelo inimigo na vida do **Ator** Volta ao passo 1
A8. Caso a vida do **Ator** chegar a zero a. O **Sistema** após alguns segundos é feito o reinicio da fase[E2] Volta ao passo 1
Fluxo de Exceção
E1. Caso haja um inimigo dentro da área de ataque do personagem a. O **Sistema** recupera os dados de dano que o personagem causa e faz a subtração do dano na vida do inimigo Volta ao passo 1
E2. Caso o **Ator** tenha chegado em um ponto de salvamento ele retorna para o último ponto de salvamento Volta ao passo 1

3.1.6. UC06- FUNÇÕES DO SISTEMA

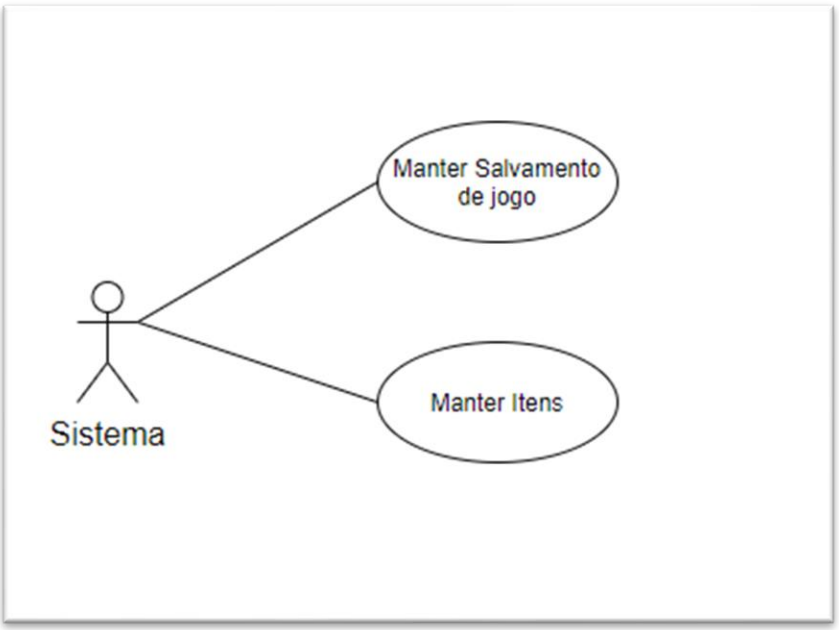

Figura 6: Diagrama de Caso de Uso Funções do Sistema
Fonte: Autoria Própria

IDENTIFICAÇÃO: UC06	
NOME: Funções do Sistema	
PRÉ-CONDIÇÕES: Estar em uma fase do jogo	
PÓS-CONDIÇÕES: Interações com o ambiente	
Fluxo Principal	
Sistema	
1.O **Sistema** durante a inicialização da fase obtém as informações dos itens que serão posicionados na fase	2.Após a obtenção dos dados o **Sistema** efetua a substituição dos objetos preposicionados na fase pelos itens
3.O **Sistema** obtém as informações do usuário[A1] para posicioná-lo na fase	4. caso de uso encerrado
Fluxo Alternativo	
O **Sistema** fará uma verificação nos diretórios para encontrar algum arquivo que estejam armazenando dados de salvamento, caso haja dados salvo o **Sistema**	

verificará se o usuário está em um novo jogo ou em um jogo já salvo

a. Se for um novo jogo o **Sistema** irá remover esse arquivo dos diretórios do jogo, caso contrário o **Sistema** passará esses dados salvos para o personagem e terminará a inicialização do jogo

Volta ao passo 3

3.2. CRONOGRAMA

O cronograma é utilizado para estipular prazos e metas para o desenvolvimento de um projeto, tornando-o eficiente e organizado, assim podendo evitar maiores transtornos e o descumprimento de datas vitais para o andamento e conclusão do projeto.

Atividade	Ano/Mês										
	2020			2021							
	OUT	NOV	DEZ	JAN	FEV	MAR	ABRI	MAI	JUN	JUL	AGO
Pré-Projeto											
Escolha do Tema											
Análise do Projeto											
Desenvolvimento do Projeto											
Entrega do Projeto											
Qualificação											
Estudo Sobre as Tecnologias											
Cursos Para o Desenvolvimento											
Análise e Especificação do Jogo											
Documentação											
Banca de Qualificação											
Desenvolvimento											
Desenvolvimento											
Documentação do Desenvolvimento											
Testes											
Banca de Defesa											
Entrega											

Figura 7: Cronograma
Fonte: Autoria Própria

4. DESENVOLVIMENTO DO JOGO

4.1. DESENVOLVIMENTO DOS ELEMENTOS

4.1.1 PERSONAGEM

A principal parte de um jogo e onde ficam concentrados a maioria das funcionalidades e interações com os outros elementos do jogo. Foi atribuído ao personagem uma movimentação padrão com sistema de ataque (causar e receber dano) a outras criaturas como também uma tela de alerta onde pode se ver a vida atual em forma de barra além de uma interface de inventário atribuída ao personagem.

4.1.2 INIMIGO

O inimigo foi desenvolvido com uma mecânica onde é definido dois pontos um, à esquerda do local onde ele irá se posicionar e outro a direita, para ser feito uma movimentação de patrulha na área. Essa movimentação de patrulha só é interrompida caso o personagem do usuário chegue no campo de percepção do inimigo, fazendo com que o inimigo persiga o personagem até chegar no alcance para desferir um ataque, caso o mesmo não alcance o seu objetivo ele retorna para a área de patrulha.

4.1.3 MENUS

Os menus são um dos principais elementos para auxiliar o usuário na utilização do jogo, muitas vezes estando conectados e com as mesmas funcionalidades. O menu de pausa tem uma característica já descrita em seu nome que em que interrompe o andamento e progresso da fase quando está sendo utilizado, para que o usuário possa fazer a utilização do mesmo sem se preocupar em qual ambiente ele solicitou a função do menu.

Já o menu principal é o primeiro contato do usuário com o jogo tendo que ser dinâmico e simples de se utilizar e com alguns aspectos animados como, por exemplo, o fundo do menu, onde se movimenta constantemente.

4.1.4 SISTEMA DE SALVAMENTO DE PROGRESSO

O sistema de salvamento é feito para que o usuário guarde suas informações de progresso no jogo e o estado atual, no caso deste jogo o salvamento é responsável pelo armazenamento da fase atual aonde se encontra o personagem bem como posição e vida atuais.

4.1.5 INVENTARIO

Realizado em duas partes o sistema de inventário é constituído de uma parte onde se armazena as informações dos itens e a quantidade de cada, e outra parte onde são feitas as interações com os dados de forma limpa e clara para melhor entendimento do usuário.

4.1.6 FASE

A parte mais artística do desenvolvimento de um jogo, a elaboração da fase é o momento do desenvolvimento em que é mesclado tudo que foi desenvolvido além de construir toda a sua estrutura, como locais para os itens e inimigos, posição dos pontos de salvamento bem como o início e fim da fase.

4.2. CONTROLES

Os comandos para interagir com o jogo são:

- Teclas “A” e “D” para movimentar o personagem para esquerda ou direita;
- Tecla “Barra de Espaço” para fazer o personagem pular;
- Botão esquerdo do mouse para interagir com os menus e atacar;

4.3. OBJETIVO DO JOGO

O jogo em questão tem por objetivo fazer com que o personagem explore o ambiente para interagir com itens, objetos e inimigos, podendo prosseguir com a sua aventura e chegar ao final da fase para alcançar o seu objetivo. Após concluir sua jornada o jogador pode visualizar a quantidade de inimigos eliminados, itens obtidos e utilizados.

4.4. PRINCIPAIS CÓDIGOS FONTE DO JOGO

Na Figura 8, encontram-se as funções utilizadas nas configurações dos botões de novo jogo, carregar jogo e finalizar jogo, bem como a verificação da existência de um salvamento de jogo para poder utilizar a função carregar jogo dentro do menu principal.

```
14      private  int scene;
15
16      public void NewGame()
17      {
18          SceneManager.LoadScene(1);
19           string path = Application.persistentDataPath + "/playerdata.data";
20           if(File.Exists(path))
21          {
22           File.Delete(path);
23          }
24      }
25
26      public void LoadGame()
27      {
28        string path = Application.persistentDataPath + "/playerdata.data";
29        if(File.Exists(path))
30        {
31
32        PlayerData data = SaveSystem.LoadGame();
33        scene =  data.scene;
34
35        if(scene !=0)
36        {
37          SceneManager.LoadScene(scene);
38        }
39
40        }
41
42      }
43
44      public void Credits()
45      {
46
47        SceneManager.LoadScene(2);
48
49      }
50
51
52      public void QuitGame()
53         {
54         // Usando o editor da unity
55        // UnityEditor.EditorApplication.isPlaying = false;
56         //Jogo Compilado
57         Application.Quit();
58         }
59
60  }
```

Figura 8: Código do menu principal
Fonte: Autoria Própria

Pode-se observar na Figura 9 as funções do menu de inventário, no qual é realizado os processos de adição, utilização e remoção do item no inventário, e também a criação de uma lista inventário onde são armazenados os itens obtidos pelo jogador.

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

[System.Serializable]
public class Inventory
{
  public event EventHandler OnItemListChanged;


  private List<Item> itemList;
  private Action<Item> UseItemAction;

  public Inventory(Action<Item> UseItemAction)
  {
      this.UseItemAction = UseItemAction;
      itemList = new List<Item>();




  }

   public void AddItem(Item item)
   {
       itemList.Add(item);
       OnItemListChanged?.Invoke(this, EventArgs.Empty);


   }
   public void UseItem(Item item)
   {
    UseItemAction(item);
   }

   public void RemoveItem(Item item)
   {
       itemList.Remove(item);
       OnItemListChanged?.Invoke(this, EventArgs.Empty);
   }

   public List<Item> GetItemList()
   {
       return itemList;
   }

```

Figura 9: Código inventário
Fonte: Autoria Própria

A Figura 10 mostra o código da interface do menu, onde se obtém a lista contendo os itens do jogador sendo feito a organização dos itens no menu de inventário, atribui-se também a função de utilizar o item clicando sobre a imagem do item.

```
15    public void SetInventory(Inventory inventory)
16    {
17        this.inventory = inventory;
18        inventory.OnItemListChanged += Inventory_OnItemListChanged;
19        RefreshInventoryItems();
20    }
21
22    private void Inventory_OnItemListChanged(object sender, System.EventArgs e)
23    {
24        RefreshInventoryItems();
25
26    }
27
28     private void RefreshInventoryItems()
29     {
30
31         foreach(Transform child in itemSlotContainer)
32         {
33           if(child == itemSlotTemplate) continue;
34           Destroy(child.gameObject);
35
36         }
37
38         int x = 0;
39         int y = 0;
40         float itemSlotCellSize = 100f;
41         foreach( Item item in inventory.GetItemList())
42         {
43             RectTransform itemSlotRectTransform = Instantiate(itemSlotTemplate, itemSlotContainer).GetComponent<RectTransform>();
44             itemSlotRectTransform.gameObject.SetActive(true);
45             itemSlotRectTransform.anchoredPosition = new Vector2(x * itemSlotCellSize, y * itemSlotCellSize);
46             Image image = itemSlotRectTransform.Find("Image").GetComponent<Image>();
47             image.sprite = item.GetSprite();
48
49             itemSlotRectTransform.GetComponent<Button_UI>().ClickFunc = () => {
50               inventory.UseItem(item);
51               inventory.RemoveItem(item);
52             };
53
54
55             x++;
56             if(x > 6)
57             {
58                 x = 0;
59                 y--;
60             }
61         }
```

Figura 10: Código interface do inventario
Fonte: Autoria Própria

Percebe-se na Figura 11 a classe que obtém os dados a serem salvos pelo sistema.

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


[System.Serializable]

public class PlayerData
{
    public int health;

    public float positionX, positionY;

     public  int scene;


    public PlayerData (Player player)
    {

        scene = SceneManager.GetActiveScene().buildIndex;

        health = player.currentHealth;


        positionX = player.transform.position.x;
        positionY = player.transform.position.y;
    }

}

```

Figura 11: Código dados de salvamento
Fonte: Autoria Própria

O código do sistema de salvamento que é observado na Figura 12, consiste em duas funções que fazem o salvamento dos dados e o carregamento dos mesmos. Após os dados serem obtidos pela classe playerdata o sistema gera um arquivo “.data” onde é feito o armazenamento dos dados e então o arquivo é serializado e salvo nos diretórios do dispositivo. Para reaver esses dados o sistema procura por este arquivo e faz a desserialização dos dados e retorna ao jogador o seu progresso.

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using UnityEngine;

public static class SaveSystem
{

    public static void SaveGame(Player player)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        string path = Application.persistentDataPath + "/playerdata.data";
        FileStream stream = new FileStream(path, FileMode.Create);

        PlayerData data = new PlayerData(player);
        formatter.Serialize(stream, data);
        stream.Close();


    }

     public  static PlayerData LoadGame()
     {
         string path = Application.persistentDataPath + "/playerdata.data";
         if(File.Exists(path))
         {
             BinaryFormatter formatter = new BinaryFormatter();
             FileStream stream = new FileStream (path, FileMode.Open);
             PlayerData data = formatter.Deserialize(stream) as PlayerData;
             stream.Close();
             return  data;
         }
         else
         {
           return null;
         }
     }


}

```

Figura 12: Código sistema de salvamento
Fonte: Autoria Própria

Neste trecho do código da Figura 13 se inicia a rotina do inimigo onde existem as verificações, chamados de funções e o início do estado de patrulha que é dependente dos gatilhos definidos no objeto inimigo na fase e as animações do mesmo.

```
34
35      void Awake()
36      {
37          currentHealth = maxHealth;
38          SelectTarget();
39
40          intTimer = timer;
41          anime = GetComponent<Animator>();
42
43      }
44
45      // Update is called once per frame
46      void Update()
47      {
48          if(!attackMode)
49          {
50              Move();
51          }
52
53          if(!InsideofLimits() && !inRange && !anime.GetCurrentAnimatorStateInfo(0).IsName("ATK"))
54          {
55              SelectTarget();
56          }
57
58          if(inRange)
59          {
60              EnemyLogic();
61          }
62      }
63
64
```

Figura 13: Código inimigo
Fonte: Autoria Própria

Dentro da Figura 14 pode-se observar as funções *EnemyLogic* e Move, onde é atribuído o movimento do inimigo e o estado de patrulha da região pré-determinada na da fase, e suas respectivas animações.

```
64
65
66      void EnemyLogic()
67      {
68
69          distance = Vector2.Distance (transform.position, target.position);
70          if(distance > attackDistance)
71          {
72              StopAttack();
73          }
74          else if(attackDistance >= distance && cooling == false)
75          {
76              Attack();
77          }
78
79          if(cooling)
80          {
81              Cooldown();
82              anime.SetBool("ATK", false);
83          }
84
85
86      }
87
88      void Move()
89      {
90
91
92          anime.SetBool("move", true);
93
94
95
96          if(!anime.GetCurrentAnimatorStateInfo(0).IsName("ATK"))
97          {
98              Vector2 targetPosition = new Vector2(target.position.x, transform.position.y);
99              transform.position = Vector2.MoveTowards(transform.position, targetPosition, moveSpeed * Time.deltaTime);
100
101         }
102
103     }
104
```

Figura 14: Código inimigo continuação
Fonte: Autoria Própria

Na parte exibida da Figura 15 temos as funções de *Attack*, *TriggerATK*, *OnDrawGizmosSelect* e pôr fim a função *Cooldown*. Tais funções são responsáveis pelo sistema de ataque do inimigo através da definição do ataque, da área de alcance, o tempo de espera para atacar novamente e causar danos ao jogador.

```
105     void Attack()
106     {
107         timer = intTimer;
108         attackMode = true;
109
110
111         anime.SetBool("move", false);
112
113         if(cooling == false)
114         {
115         anime.SetBool("ATK", true);
116         }
117
118      }
119
120       public void TriggerATK()
121     {
122          Collider2D[] hitPlayer = Physics2D.OverlapCircleAll(ATKpoint.position, attackrange, playerlayer);
123         foreach(Collider2D player in hitPlayer)
124         {
125
126         Dplayer.GetComponent<Player>().TakeDamage(pdamage);
127
128         }
129
130     }
131
132       void OnDrawGizmosSelected()
133    {
134        if( ATKpoint == null)
135        return;
136        Gizmos.DrawWireSphere(ATKpoint.position, attackrange);
137    }
138
139      void Cooldown()
140      {
141          timer -= Time.deltaTime;
142
143          if(timer <= 0 && cooling && attackMode)
144          {
145              cooling = false;
146              timer = intTimer;
147          }
148      }
```

Figura 15: Código inimigo Continuação
Fonte: Autoria Própria

O código da Figura 16 observa-se as funções que terminam o sistema de dano ao jogador e a rotação da movimentação do jogador.

Nota-se as funções *StopAttack* que é responsável por determinar a finalização das animações e funções primarias do ataque, em seguida temos a função *TriggerColing* que auxilia no tempo de espera do ataque do inimigo, por fim temos as funções InsideofLimits e *SelectTarget* que são responsáveis por parte do sistema de patrulha do inimigo onde ele realiza a verificação dos objetos em jogo para realizar suas ações primarias, além da função Flip, onde é feito a rotação do inimigo para a direita ou esquerda.

```
149     void StopAttack()
150     {
151         cooling = false;
152         attackMode = false;
153         anime.SetBool("ATK", false);
154 
155     }
156 
157     public void TriggerCooling()
158     {
159         cooling = true;
160     }
161 
162     private bool InsideofLimits()
163     {
164         return transform.position.x > leftLimit.position.x && transform.position.x < rightLimit.position.x;
165     }
166 
167     public void SelectTarget()
168     {
169         float distanceToLeft = Vector2.Distance(transform.position, leftLimit.position);
170         float distanceToRight = Vector2.Distance(transform.position, rightLimit.position);
171 
172         if(distanceToLeft > distanceToRight)
173         {
174             target = leftLimit;
175         }
176         else
177         {
178           target = rightLimit;
179         }
180 
181         Flip();
182     }
183 
184     public void Flip()
185     {
186         Vector3 rotation = transform.eulerAngles;
187         if(transform.position.x > target.position.x)
188         {
189             rotation.y = 180f;
190         }
191         else
192         {
193             rotation.y = 0f;
194         }
```

Figura 16: Código inimigo continuação
Fonte: Autoria Própria

Por fim temos a Figura 17 onde é possível observar as funções *TakeDamage* e *Die* e a corrotina *DestroyCoroutine*, onde a função *TakeDamage* é responsável pelo recebimento de dano e a verificação da vida do inimigo, já a função die é realizada após a vida do inimigo chegue a zero e então chama-se a corrotina *DestroyCoroutine* que remove o objeto inimigo da fase.

```
195
196             transform.eulerAngles = rotation;
197         }
198
199         public void TakeDamage(int damage)
200         {
201
202             if(canDamge && currentHealth > 0)
203             {
204                 currentHealth -= damage;
205                 anime.SetTrigger("hurt");
206             }
207
208             if(currentHealth <= 0)
209             {
210                 canDamge = false;
211                 anime.SetBool("dead", true);
212
213                 Die();
214
215             }
216             else  if(inRange)
217             {
218                 EnemyLogic();
219             }
220
221         }
222
223         void Die()
224         {
225             StartCoroutine(DestroyCoroutine());
226             this.enabled = false;
227
228         }
229
230         IEnumerator DestroyCoroutine()
231         {
232             if( canDamge == false)
233             {
234                 yield return new WaitForSeconds(4f);
235                 Destroy(gameObject);
236
237             }
238         }
239     }
```

Figura 17: Código inimigo final
Fonte: Autoria Própria

Na Figura 18 observa-se os códigos iniciais do jogador onde são atribuídos diversos dados e componentes para o início da fase, além das funções para salvar os dados do jogo e carregar os mesmos para o início do jogo, através das funções *SavePlayer* e *LoadPlayer*.

```
// Start is called before the first frame update
void Start()
{
    Time.timeScale = 1f;
    Cursor.lockState = CursorLockMode.Locked;
    Cursor.visible = false;

    sprite = GetComponent<SpriteRenderer> ();
    rb2d = GetComponent<Rigidbody2D> ();
    anime = GetComponent<Animator> ();

    currentSpeed = speed;
    currentHealth = maxHealth;
    SetMaxHealth(maxHealth);
    inventory = new Inventory(UseItem);
    uiInventory.SetInventory(inventory);
    string path = Application.persistentDataPath + "/playerdata.data";
    if(File.Exists(path))
    {
        LoadPlayer();
    }
}

public void SavePlayer ()
{

    SaveSystem.SaveGame(this);
    Debug.Log("Game Saved");
}

public void LoadPlayer()
{
    PlayerData data = SaveSystem.LoadGame();

    currentHealth = data.health;
    SetHealth(currentHealth);

    Vector2 position;
    position.x = data.positionX;
    position.y = data.positionY;
    this.transform.position = position;

}
```

Figura 18: Código personagem
Fonte: Autoria Própria

Observa-se na Figura 19 o início do sistema de obtenção e utilização dos itens durante a fase, através das funções *OnTriggerEnter2D* e *UseItem.* O jogador quando se aproxima do colisor do item ativa a função *OnTriggerEnter2D* e o item em questão é adicionado no inventário onde pode-se fazer sua utilização através da função *UseItem*, no qual verifica o tipo do item e retorna os seus atributos.

```
private void OnTriggerEnter2D(Collider2D collider)
{
    ItemWorld itemWorld = collider.GetComponent<ItemWorld>();
    if(itemWorld != null)
    {
        inventory.AddItem(itemWorld.GetItem());

        itemWorld.DestroySelf();

    }
}

private void UseItem(Item item)
{

  switch (item.itemType)
  {
      case Item.ItemType.HealthPotionM :

      TakeHeal(100);
      inventory.RemoveItem(new Item{ itemType = Item.ItemType.HealthPotionM, amount =1});
      break;
      case Item.ItemType.HealthPotionP :

      TakeHeal(50);
      inventory.RemoveItem(new Item{ itemType = Item.ItemType.HealthPotionP, amount =1});
      break;
  }
}
```

Figura 19: Código personagem continuação
Fonte: Autoria Própria

A Figura 20 mostra a parte do código onde é feita a associação de algumas teclas e botões com os elementos presentes no jogo.

```
// Update is called once per frame
void Update()
{
    if(!isPaused)
    {
    grounded = Physics2D.Linecast(transform.position, groundcheck.position, 1 << LayerMask.NameToLayer("Ground"));
    if( Input.GetButtonDown("Jump") && grounded)
    {

     jumping = true;

    }

     SetAnimations();

     if (Input.GetButton ("Fire1") && grounded && Time.time > nextattack)
     {

        Attack();

     }

    }
      if(Input.GetKeyDown(KeyCode.I))
      {
         InventoryScreen();
      }

      if(Input.GetKeyDown(KeyCode.Escape))
   {
       PauseScreen();
   }
```

Figura 20: Código personagem continuação
Fonte: Autoria Própria

Observa-se na Figura 21 as funções que habilitam e desabilitam os menus de pausa e de inventário bem como elementos adicionais que são responsáveis pelo congelamento do jogo e a exibição do ponteiro do mouse nos menus.

```
 public void InventoryScreen()
{
    if(isPaused)
    {
        isPaused = false;
        Time.timeScale = 1f;
        InventarioMenu.SetActive(false);
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }
    else
    {
        isPaused = true;
        Time.timeScale = 0f;
        InventarioMenu.SetActive(true);
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;


    }
}



public void PauseScreen()
{
    if(isPaused)
    {
        isPaused = false;
        Time.timeScale = 1f;
        pauseMenu.SetActive(false);
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }
    else
    {
        isPaused = true;
        Time.timeScale = 0f;
        pauseMenu.SetActive(true);
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;

    }
}
```

Figura 21: Código personagem continuação
Fonte: Autoria Própria

Na Figura 22 é possivel observar as funções *TakeHeal* e *Flip*, os quais são semelhantes as funções de mesmo nome localizadas no código do inimigo.

A função *FixedUpdate* é utilizada para constante verificação das ações do jogador durante a fase no qual possui algumas funções ou variáveis que necessitam de constante verificação e ou atribuição.

```
    public void TakeHeal(int heal)
  {
    if(currentHealth > 0 && currentHealth < maxHealth)
    {

     currentHealth += heal;

     SetHealth(currentHealth);

    }
  }



void FixedUpdate()
{

  move = Input.GetAxis("Horizontal");
  rb2d.velocity = new Vector2(move * currentSpeed, rb2d.velocity.y);
  anime.SetBool ("Walk", rb2d.velocity.x != 0f && grounded);
  if (move < 0f && facingrigth || move > 0f && !facingrigth)
  {

   flip();

  }

  if(jumping)
  {
      rb2d.AddForce(new Vector2(0f, jumpforce));
      jumping = false;

  }

}

void flip()
{

facingrigth = !facingrigth;
transform.localScale = new Vector3(-transform.localScale.x , transform.localScale.y, transform.localScale.z);

}
```

Figura 22: Código personagem continuação
Fonte: Autoria Própria

Nesta parte do código exibida na Figura 23 observa-se a função *SetAnimations* que realiza a instanciação das animações que são ativadas por condições da fase, há também a função *Attack* que é responsável por causar dano aos inimigos no raio de ataque do jogador.

```
void SetAnimations()
{

   anime.SetFloat ("VelY", rb2d.velocity.y);
   anime.SetBool ("JumpFall", !grounded);


}

void Attack()
{

    anime.SetTrigger ("ATK");
    StartCoroutine(StopCoroutine());
    nextattack = Time.time + attackrate;
    Collider2D[] hitEnemies = Physics2D.OverlapCircleAll(ATKpoint.position, attackrange, enemylayer);
    foreach(Collider2D enemy in hitEnemies)
    {

      enemy.GetComponent<ENEMY1>().TakeDamage(ATKdamage);



    }

}
```

Figura 23: Código personagem continuação
Fonte: Autoria Própria

Pode-se observar no código da Figura 24 as funções *OnDrawGizmosSelec*t, *TakeDamage* e *Die,* que são responsaveis por parte do sistema de ataque do jogador, sendo assim semelhante ao sistema de ataque do inimigo.

A função *ReloadLevel* quando solicitada é responsável pelo recarregamento da fase atual em determinadas circunstancias.

```
void OnDrawGizmosSelected()
{
    if( ATKpoint == null)
    return;
    Gizmos.DrawWireSphere(ATKpoint.position, attackrange);
}

  public void TakeDamage(int damage)
  {
    if(canDamge && currentHealth > 0)
    {
     canDamge = false;
     currentHealth -= damage;
     anime.SetTrigger("Hurt");
     SetHealth(currentHealth);
     StartCoroutine(DamageCoroutine());
    }
     else if(currentHealth <= 0)
     {
        canDamge = false;
        anime.SetBool("Die", true);

        Die();
     }




 }

  void Die()
  {
   dieSc.SetActive(true);
   Invoke("ReloadLevel",2.5f);
   this.enabled = false;


  }

    void ReloadLevel()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene ().buildIndex, LoadSceneMode.Single);
    }
```

Figura 24: Código personagem continuação
Fonte: Autoria Própria

Na Figura 25 temos as corrotinas *StopCorourine* e *DamageCoroutine*, que são semelhantes as corrotinas do inimigo. Por fim temos as funções *SetMaxHealth* e *SetHealth* que são responsáveis por manipular a barra de vida do jogador e mantê-la atualizada.

```
IEnumerator DamageCoroutine()
{
    if(canDamge == false)
    {
    if(currentHealth <= 0)
    {
       canDamge = false;
       anime.SetBool("Die", true);

        Die();
    }
    currentSpeed = 0;
    yield return new WaitForSeconds(0.5f);
    canDamge = true;
    currentSpeed = speed;

    }

}

IEnumerator StopCoroutine()
{

    currentSpeed = 0;
    yield return new WaitForSeconds(0.5f);
    currentSpeed = speed;
}

  public void SetMaxHealth (int hp)
{
    slider.maxValue = hp;

}

  public void SetHealth (int ch)
{
    slider.value = ch;
}
```

Figura 25: Código personagem fim
Fonte: Autoria Própria

4.5. TEXTURAS E ARTES UTILIZADAS NO PROJETO

Na seção abaixo estão presentes as imagens e texturas utilizadas nos elementos visuais desenvolvidos no jogo.

Figura 26: Arte Personagem
Fonte: https://itch.io/game-assets/free/tag-pixel-art

Figura 27: Arte inimigo 1
Fonte: https://itch.io/game-assets/free/tag-pixel-art

Figura 28: Arte inimigo 2
Fonte: https://itch.io/game-assets/free/tag-pixel-art

Figura 29: Arte inimigo 3
Fonte: https://itch.io/game-assets/free/tag-pixel-art

Figura 30: Arte inimigo 4
Fonte: https://itch.io/game-assets/free/tag-pixel-art

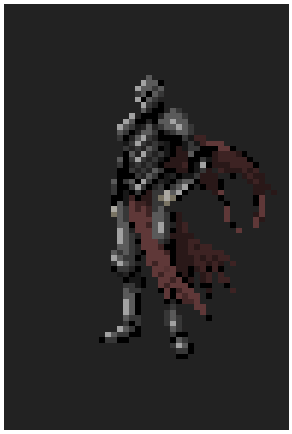

Figura 31: Arte cena de créditos
Fonte: https://itch.io/game-assets/free/tag-pixel-art

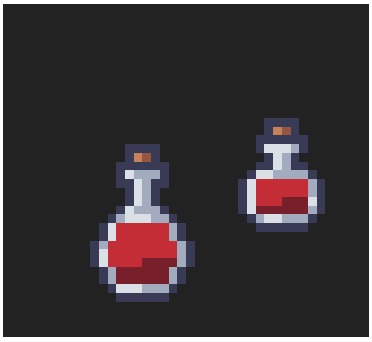

Figura 32: Arte poções
Fonte: https://itch.io/game-assets/free/tag-pixel-art

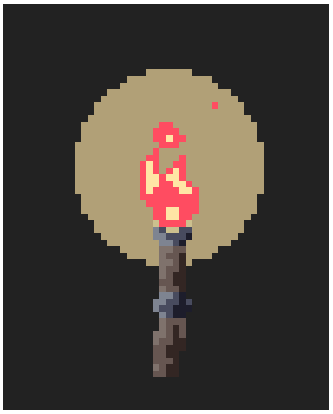

Figura 33: Arte tocha
Fonte: https://itch.io/game-assets/free/tag-pixel-art

Figura 34: Arte barra de vida
Fonte: https://itch.io/game-assets/free/tag-pixel-art

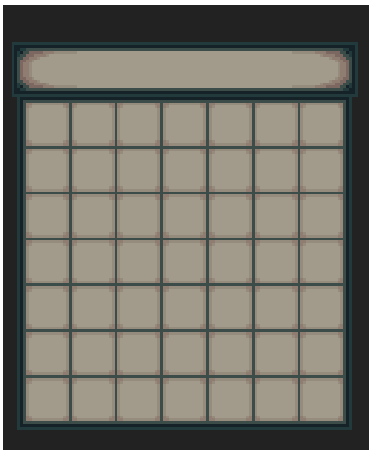

Figura 35: Arte menu de inventário
Fonte: https://itch.io/game-assets/free/tag-pixel-art

Figura 36: Textura fase
Fonte: https://itch.io/game-assets/free/tag-pixel-art

Figura 37: Textura de fundo primeira camada
Fonte: https://itch.io/game-assets/free/tag-pixel-art

Figura 38: Textura de fundo segunda camada
Fonte: https://itch.io/game-assets/free/tag-pixel-art

Figura 39: Textura de fundo terceira camada
Fonte: https://itch.io/game-assets/free/tag-pixel-art

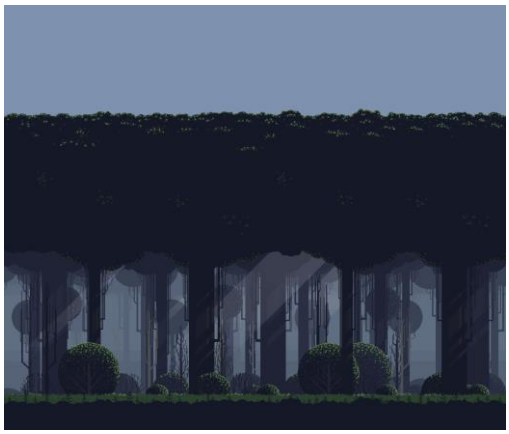

Figura 40: Textura de fundo menu principal
Fonte: https://itch.io/game-assets/free/tag-pixel-art

CONCLUSÃO

No andamento do projeto foi possível compreender parcialmente as etapas de planejamento e desenvolvimento de jogos e seus conceitos técnicos e conceituais, bem como a utilização de tecnologias e ferramentas para auxiliar no processo de elaboração do projeto.

O gênero MetroidVania em que foi baseado este trabalho é bem quisto no mercado atual de jogos, havendo vários lançamentos desse mesmo gênero. Onde sua simplicidade e diversificação de mecânica agradam à maioria dos consumidores de jogos da atualidade.

Esse trabalho de conclusão de curso trouxe inúmeros conhecimentos sobre a área de desenvolvimento de jogos, suas facilidades e dificuldades, bem como um melhor entendimento de lógica de programação geralmente e conceitos simples de design em jogos. Durante o seu desenvolvimento houve algumas modificações e adaptações para alcançar o produto final do projeto, pois diferente de um sistema comercial o desenvolvimento de jogos é volátil e extenso podendo durar meses até anos e dentro desse período mudar inúmeras vezes seus aspectos.

REFERÊNCIAS

DAFONT: Fontes Bitmap. Disponível em: < https://www.dafont.com/pt/bitmap.php >. Acesso em: 23 jun. 2021.

IMMAKERS: O Mercado de Jogos Digitais no Brasil e no mundo. São Paulo. Disponível em < http://immakers.com/mercado-de-jogos-digitais-no-brasil-e-no-mundo/#!/contact >. Acesso em: 16 out. 2020.

ITCH.IO: Game Assets Free. Disponível em: < https://itch.io/game-assets/free/tag-pixel-art >. Acesso em: 15 abr. 2021.

MACHADO; SANTUCHI; CARLETTI. Carlos Eduardo Martins; Rafael Pinheiro e Ednéa Zandonadi Brambila. O mercado de jogos eletrônicos e seus impactos na sociedade. 2018. 17p. Monografia- Curso de Sistemas da Informação - Faculdade Multivix Cachoeiro de Itapemirim, Espirito Santo, Cachoeiro de Itapemirim, 2018.

MICROSOFT: Introdução à Linguagem C# e .NET. Disponível em: < https://docs.microsoft.com/pt-br/dotnet/csharp/getting-started/ >. Acesso em: 07 nov. 2020.

OLIVIERA GOULARTE; COCCA FUKADA, Guilherme e João Francisco. O mercado de jogos eletrônicos independentes. Disponível em < http://www.each.usp.br/petsi/jornal/?p=2553 >. Acesso em: 16 out. 2020.

RIBEIRO, Leandro. O que é UML e Diagrama de Caso de Uso: Introdução Prática à UML. Disponível em: <https://www.devmedia.com.br/o-que-e-uml-e-diagramas-de-caso-de-uso-introducao-pratica-a-uml/23408> acesso em: 10 mar. 2021.

UNITY: Unity Para Jogos. Disponível em: < https://unity.com/pt/solutions/game >. Acesso em: 07 nov. 2020.

UNITY: Unity Manual. Disponível em: < http://docs.unity3d.com/Manual/index.html >. Acesso em: 05 jul. 2021.

VISUAL STUDIO CODE: Code.editing.Redefined. Disponível em: < https://code.visualstudio.com>. Acesso em: 07 nov. 2020.

WHOW: A disputa pelo mercado de games no Brasil. São Paulo. Disponível em < https://www.whow.com.br/negocios/disputa-mercado-games-brasil/ >. Acesso em: 15 out. 2020.