SERVIÇO PÚBLICO FEDERAL - MINISTÉRIO DA EDUCAÇÃO
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
UNIDADE CONTAGEM: INFORMÁTICA III

JÚLIA BATISTA MOREIRA
LUAN SETA RAMOS
LUIZA ANDRESSA VAILANTE SILVA
MARIA EDUARDA OLIVEIRA PEREIRA
MARIA LUIZA FERREIRA MEIRELES
MEL RAPOSEIRAS RICALDONI
MIGUEL DOS SANTOS MIRANDA
NÚBIA TORRES DE OLIVEIRA
OTTO EMANUEL MARTINS ABREU
PEDRO GUIMARÃES DE DEUS CORRÊA
PEDRO LUCAS ALBANO FERNANDES
SARAH DOS SANTOS OLIVEIRA
SARAH RABELO ARAUJO
VICTOR HUGO GONÇALVES DA LUZ MIRANDA
VICTÓRIA FERNANDA SANTOS ROCHA
VINICIUS DOMINGOS CANDIDO

LINGUAGENS E TÉCNICAS DE PROGRAMAÇÃO II - “EQUIPE LIBGDX”

CONTAGEM
2024

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

O presente documento apresenta o relatório técnico para a disciplina de Laboratório de Linguagens e Técnicas de Programação II, ministrada pelo professor Alisson Rodrigo dos Santos. A prática consistiu em desenvolver a recriação do popular jogo *Baloons Tower Defense*, cujo objetivo é impedir que balões inimigos atravessem o mapa, posicionando estrategicamente torres para estourá-los antes que alcancem o final do trajeto.

Para a realização do projeto, a turma de Informática 3 foi dividida em duas equipes, com a proposta de desenvolver um *software* competitivo para ser "vendido" ao professor. As equipes funcionaram como empresas simuladas, organizadas em cinco subgrupos: Organização, Arte, Programação, Controle de Assets, Programação de Game Objects e Mapa. Cada grupo assumiu responsabilidades específicas, promovendo a colaboração e otimização do projeto. Como inspiração temática, foi escolhido o clássico *"Plants vs. Zombies"*.

Ademais, dentre os principais objetivos do trabalho encontrava-se a utilização da biblioteca LibGDX, que permite o desenvolvimento multiplataforma de jogos 2D e 3D, oferecendo suporte para controle de física, animações, áudio, gráficos, e integração com APIs. A linguagem de programação adotada foi o Java. Além de vivenciar condições de aprendizado de técnicas de programação, gerenciamento de projetos, e construção de aplicações multimédia.

Ao final, a recriação do *Tower Defense* foi concluída, proporcionando aos alunos a oportunidade de aprimorar suas habilidades em programação de jogos e trabalho em equipe, além de uma experiência prática em divisão de tarefas.

2. ARTE

A equipe de *Arte* foi responsável pela produção do audiovisual, fornecendo as imagens e sons apropriados para cada elemento do jogo, como o menu, as torres e os mapas.

2.1 Software

O software utilizado para o desenvolvimento das sprites foi o Aseprite, um programa dedicado à criação de sprites animados que oferece uma série de ferramentas para a construção de pixel art.

2.2 Dificuldades

Pelo fato do Aseprite ser um software com muitas ferramentas e funções, a dificuldade principal encontrada foi a adaptação ao próprio aplicativo. Portanto, pelo fato de haver diversos meios para a criação das sprites, o grupo precisou aprender como que funcionava cada função e logo após confeccionar os mapas, inimigos, torres, etc.

2.3 Resultados

Foram propostas as animações de 4 tipos de plantas, 4 tipos de projéteis, 3 tilesets completos de mapas e 6 inimigos para o jogo. Conforme esperado, todas as sprites foram concluídas, com suas respectivas animações e características dimensionais.

2.3.1 Plantas

Abaixo se encontra uma das sprites das plantas feitas pela equipe de Arte, a arte delas foi inspirada principalmente nas plantas encontradas no jogo “Plants vs Zombies”, é um jogo onde se há plantas para defender uma determinada casa atacada por Zombies. Como se trata de uma releitura do Tower Defense, o grupo decidiu que as plantas iriam representar as torres do jogo.

Figura 1 - SMG Tower

Fonte: Autoria Própria

2.3.2 Projéteis

Por haver as plantas, se tornou necessário a confecção de projéteis destinados a cada uma delas, sendo assim, foram desenvolvidos 4 projéteis para suas respectivas plantas. Na figura 3 se tem um dos projéteis criados pelo grupo.

Figura 2 - Projétil da SMG Tower

Fonte: Autoria Própria

2.3.3 Mapas

Inicialmente, foi definido que seria preciso 3 mapas para o jogo, uma vez que no jogo Plants vs Zombies se tem 3 mapas: um mapa para representar o dia, um mapa para representar a noite e um que se baseia num telhado de uma casa. Desta forma, a equipe de arte desenvolveu os mapas planejados. Para exemplificar, abaixo está o tileset diurno.

Figura 3 - Mapa Diurno

Fonte: Autoria Própria

2.3.4 Sons

Os desenvolvedores selecionaram uma música para cada mapa e um som para o menu do jogo. Os sons foram escolhidos levando em consideração a ambientação de cada mapa.

2.3.5 Inimigos

Todas as sprites, tanto dos inimigos quanto dos mapas, foram criadas utilizando o software de pixel art Aseprite, uma ferramenta amplamente utilizada para garantir a qualidade visual e coesão estética do jogo. Vários testes foram realizados para ajustar o equilíbrio entre as torres e os inimigos, garantindo uma jogabilidade fluida e desafiadora.

Figura 4 - Inimigos

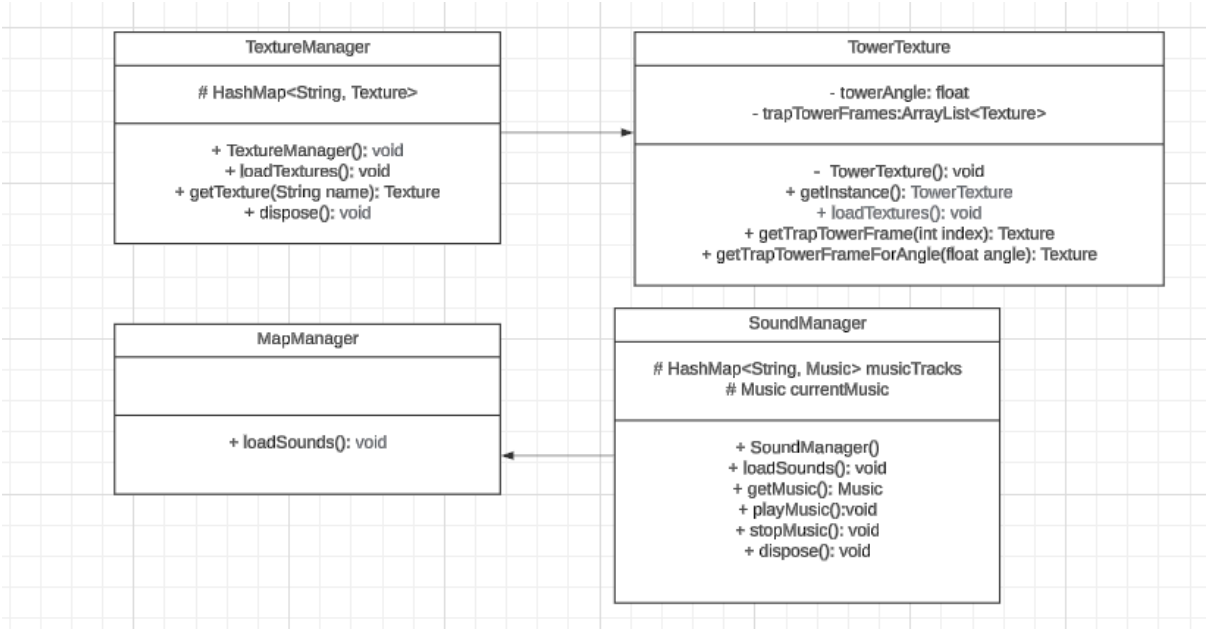
Fonte: Autoria Própria

3. PROGRAMAÇÃO CONTROLE DE ASSETS

O foco principal do controle de assets é carregar, armazenar e fornecer as texturas utilizadas no jogo.

3.1 Organização do código

Figura 5 - Diagrama de classes da equipe Controle de Assets



Fonte: Acervo próprio

3.1.1 Classe *TextureManager*:

É a classe que gerencia as imagens das torres, com apenas um método implementado.

3.1.2 Classe *TowerTexture*:

É responsável pela gestão das texturas específicas das torres no jogo Bomber Tower

Estrutura:

loadTextures(): Carrega as texturas estáticas para cada tipo de torre (Trap, Bomber, SMG, Sniper) e os frames de animação para a TrapTower.

getTrapTowerFrameForAngle(): Método voltado para a TrapTower e torres com animações rotacionais. Ele calcula o frame de animação correto com base no ângulo recebido, dividindo o ângulo total (360 graus) pelo número de frames, retornando a textura correspondente ao índice calculado.

3.2 Desenvolvimento

A metodologia utilizada envolveu a implementação de um sistema de gerenciamento de texturas a fim de resolver problemas relacionados à animação, associação de assets, e a otimização do carregamento de recursos. Buscando maneiras mais eficientes de codificar e organizar os assets, com foco em otimizar o desempenho do jogo.

3.3 Dificuldades enfrentadas e erros

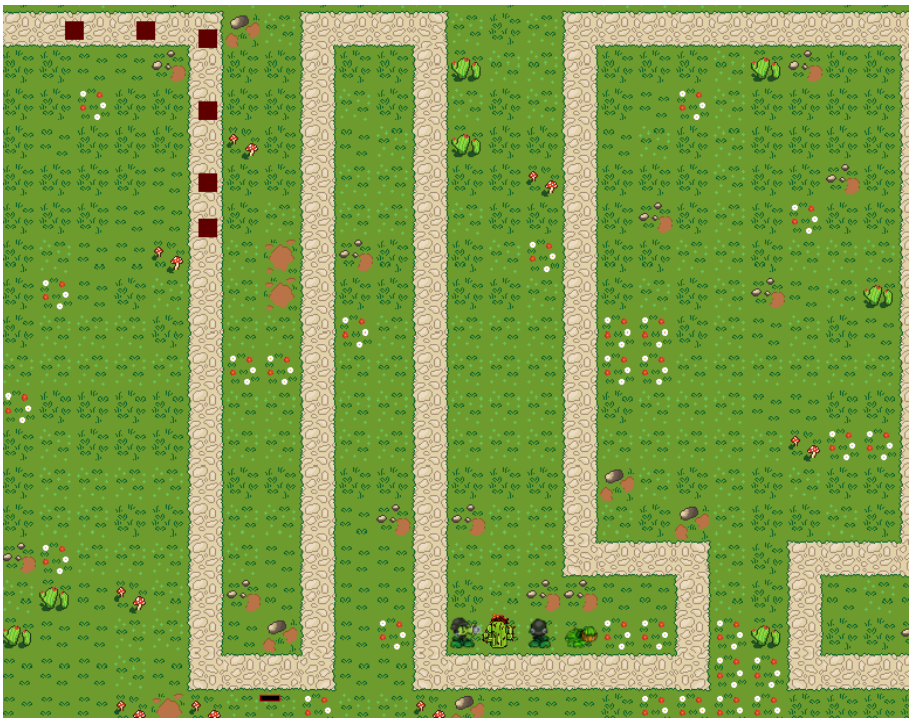
3.3.1 Lógica de Animação dos Personagens

Foi um desafio implementar a rotação das torres com base na movimentação dos inimigos. A lógica consistiu em calcular o ângulo entre a torre e o inimigo e associar esse ângulo a uma imagem correspondente, de forma a simular a rotação da torre. Para isso, utilizou-se a herança de uma classe "super", padronizando o código e evitando repetições desnecessárias. Entretanto, a rotação dos assets foi implementada com sucesso.

3.3.2 Direção da Bala

A definição da direção da bala em relação à torre enfrentou problemas durante a implementação inicial. Pois, a bala estava atirando sempre para uma posição fixa com “x” e “y” iguais a 0:

Figura 6 - Erro na direção da bala



Fonte: Acervo próprio

Tal fato é perceptível, pois os inimigos estão chegando perto das torres, mas as torres estão atirando para o canto inferior à esquerda

3.3.3 Erro na alocação de Assets

Houve dificuldade em associar corretamente os assets a cada torre, o que causou problemas visuais no jogo. Na figura anterior é possível perceber que (em ordem da esquerda para a direita) a primeira e a terceira torre são os mesmos assets. Porém, com ajuda em equipe, foi percebido que a referência de quando desenha a torre quando calcula o ângulo estava duplicada.

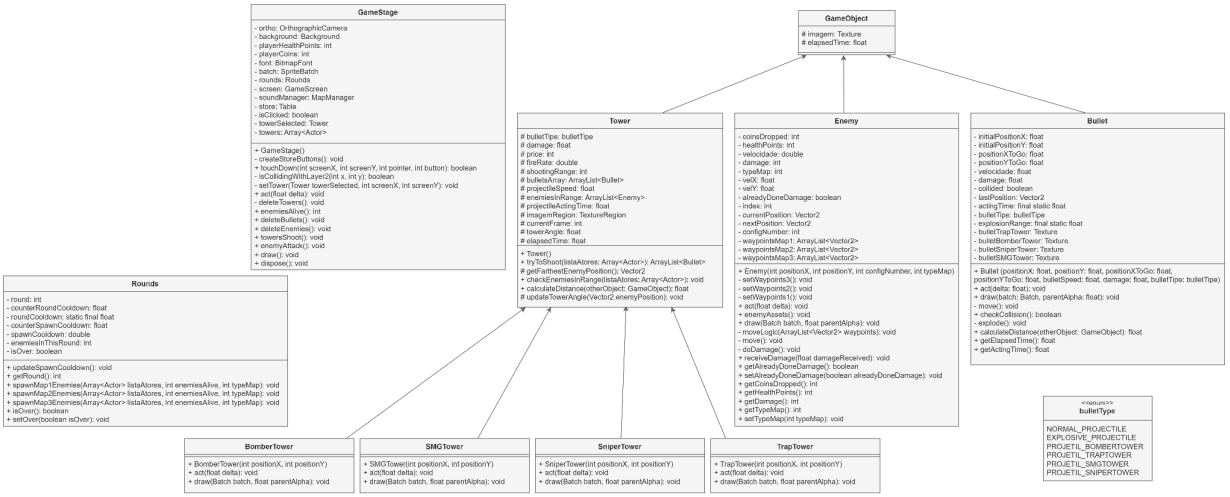
4. PROGRAMAÇÃO GAME OBJECT

A equipe de programação *Game Objects* foi responsável por implementar as estruturas base do jogo, desenvolvendo as principais mecânicas e classes para o controle dos objetos, como torres, inimigos, sistema de disparos, *rounds* e moedas.

4.1 Organização do código

O grupo implementou os personagens e principais funções nos pacotes *GameActors*, *GameStage* e *GameRounds*, utilizando herança e polimorfismo no relacionamento entre as classes. A figura 01 ilustra o esquema de classes aplicado.

Figura 7 - Diagrama de classes da equipe *Game Objects*



Fonte: Criação própria

4.2 Torres

O jogo possui quatro torres com características distintas: Torre *Sniper*, Torre Submetralhadora, Torre Bombardeiro e Torre de Armadilha. Inicialmente, foram definidas suas propriedades, como comportamento, raio de ataque, dano e velocidade dos projéteis. Para os primeiros testes, foram criados objetos representando cada torre, que eram ativadas na tela pelas teclas "B", "S", "T" e "P". Além disso, foi implementado um sistema de raio de ataque e *tracking*, onde as torres miram no inimigo com a maior posição X.

4.2.1 Incremento da explosão da Torre Bombardeiro

A torre bombardeiro é diferente das outras em relação ao tiro. Como o nome sugere, ela lança bombas que causam uma explosão. Para essa execução, foi criado um *enum bulletTipe*, para diferenciar os projéteis entre si.

4.2.2 Mudança na Torre Armadilha

Inicialmente, a Torre Armadilha foi planejada para lançar armadilhas no caminho dos inimigos. No entanto, na etapa final do projeto, a equipe geral, em consenso, decidiu que ela iria apenas atirar nos inimigos, como as outras torres. A lógica implementada foi similar às demais, com ajustes apenas nos valores.

4.3 Inimigos

O jogo conta com 5 tipos de inimigos, cada um com atributos próprios, como pontos de vida, velocidade, dano e moedas ao ser abatido, definidos ao longo das semanas para tornar o jogo mais dinâmico. Inicialmente, os inimigos foram criados para testes e inseridos manualmente com a tecla "espaço". Após a finalização, o *spawn* dos inimigos passou a ser automático em cada round, com quantidade e variação predefinidas. Diversos testes foram realizados para garantir o balanceamento entre torres e inimigos.

4.3.1 Dano no jogador

Foi implementado o sistema de vidas do usuário; caso ele falhe em eliminar um inimigo e ele saia do mapa pelo lado direito, o jogador sofre um dano com valor a depender do tipo de inimigo, sendo que cada um possui um dano diferente.

4.4 Classe Bullet

Dentre as principais dinâmicas do jogo está o controle dos projéteis lançados pelas torres em direção aos inimigos. Para controlar as características e movimentações das balas foi criada a classe ***Bullet***, em que seus atributos e métodos estão presentes na figura 06 acima. Suas principais funções são tratar dos danos dos projéteis – incluindo a colisão –, definir sua direção – no caso do jogo, em direção ao inimigo mais próximo – e desenhar a bala na tela.

4.4.1 Sistema de dano das torres

Cada torre foi configurada com um nível de dano aos inimigos. Inicialmente, a função "*checkCollision()*" foi criada na classe ***Bullet*** para verificar se os projéteis atingiam os inimigos e aplicar o dano, removendo o projétil após a colisão. No entanto, devido a um erro de índice, a função foi ajustada para retornar um valor booleano indicando a colisão. Para resolver o problema, as funções "*deleteBullets()*" e "*deleteEnemies()*" foram adicionadas à classe **GameStage**, removendo balas e inimigos após a interação.

4.5 Mecânicas-chave implementadas

Na versão original de *Bloons Tower Defense*, o sistema de evolução e o de moedas para compra de torres são fundamentais. Por sua importância, essas mecânicas foram implementadas no jogo como descritas abaixo, e apresentadas no apêndice A.

4.5.1 Criação do sistema de *rounds*

A estrutura de evolução tratava-se de um item obrigatório. Com isso, o sistema de rounds foi criado para aumentar gradualmente a quantidade e a força dos inimigos em cada rodada. A função "*spawnEnemies()*", na classe ***GameStage***, gerencia o surgimento dos inimigos com base na rodada atual.

4.5.2 Criação do sistema de moedas

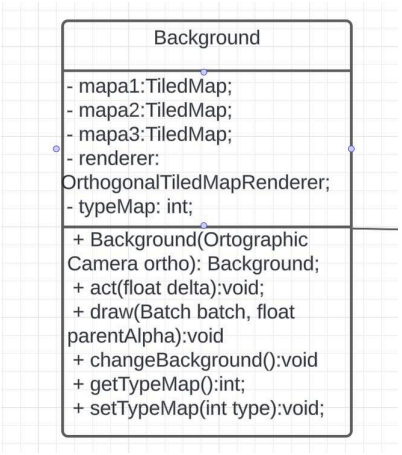
A compra de novas torres é essencial para derrotar os inimigos no jogo. Assim, foi implementado o sistema de moedas, onde os inimigos, ao serem eliminados (com seus pontos de vida reduzidos a zero), aumentam a quantidade de moedas do usuário. Cada inimigo na classe ***Enemy*** recebe um atributo que define quantas moedas serão concedidas ao jogador.

5. PROGRAMAÇÃO MAPA

A equipe de programação do Mapa foi responsável por desenhar os mapas e seus respectivos gerenciadores, desenvolvendo várias funcionalidades essenciais para a experiência do usuário. Isso incluiu implementar a loja, onde os jogadores podem adquirir torres; estruturar o menu e a lógica por trás dele, garantindo uma navegação intuitiva e fluida entre as diferentes opções; definir as trocas de mapa, permitindo transições suaves entre diferentes áreas do jogo; e implementar o TiledMap, que facilitou a criação de mapas complexos e detalhados. Essas tarefas foram fundamentais para criar um ambiente imersivo e dinâmico dentro do jogo.

5.1 Organização do código

Figura 8 - Diagrama de classes da equipe Mapa



Fonte: Autoria Própria

5.2 Mapas

A estruturação dos mapas na classe Background é projetada para permitir a renderização de diferentes cenários em um jogo, utilizando o formato de mapas "tiled". Três mapas distintos são carregados: mapa1, mapa2 e mapa3, correspondendo a cenários diurno, noturno e telhado, respectivamente. Cada mapa é representado como um objeto da classe TiledMap, que facilita o gerenciamento e a renderização de elementos em uma grade.

O renderizador ortogonal OrthogonalTiledMapRenderer é utilizado para desenhar o mapa atual na tela, configurando a visualização por meio de uma câmera ortográfica. O método changeBackground() permite alternar entre os mapas durante a execução do jogo, utilizando uma lógica simples para determinar qual mapa deve ser exibido.

Os mapas são trocados com base no número de rounds do jogo. O mapa2 é exibido quando o jogador chega ao round 10 e derrota os inimigos presentes. Da mesma forma, o

mapa3 é ativado ao atingir o round 20, após a derrota dos inimigos. Essa mecânica não apenas adiciona uma camada de progressão ao jogo, mas também mantém o interesse do jogador, introduzindo novos cenários conforme os desafios se intensificam.

5.2.1 Waypoints

A estruturação dos waypoints dos inimigos na classe `Enemy` é projetada para permitir que os inimigos sigam um caminho definido em cada mapa, proporcionando uma movimentação previsível e estratégica. Três conjuntos de waypoints são criados, um para o mapa 1, outro para o mapa 2, e outro para o mapa 3, armazenados em *waypointsMap*1, *waypointsMap2* e *waypointsMap3,* respectivamente. Cada waypoint é representado como um objeto `Vector2`, que contém as coordenadas x e y onde o inimigo deve se mover. Durante a execução do jogo, o método `move()` é responsável por determinar qual conjunto de waypoints usar, com base no `typeMap`. A lógica de movimentação é gerenciada no método `moveLogic()`, que ajusta a velocidade do inimigo com base na posição atual e na posição do próximo waypoint. Quando o inimigo chega a um waypoint, o índice é incrementado, permitindo que ele se desloque para o próximo ponto na sequência. Essa abordagem não só proporciona uma movimentação fluida e organizada, mas também permite personalizar facilmente os caminhos dos inimigos em diferentes mapas, contribuindo para a variedade e a dinâmica do jogo.

5.3 Menu

O menu na classe `MenuStage` oferece uma interface interativa ao jogador, gerenciando a visualização com uma `OrthographicCamera` e permitindo a entrada do usuário. Um fundo de imagem preenche a tela, e no método `createButtons()`, são criados três botões principais: "Jogar", "Configurações" e "Sair", estilizados com fontes grandes e coloridas. Os eventos de clique são tratados com `addListener`, onde cada botão realiza uma ação específica ao ser clicado, como iniciar o jogo, acessar as configurações ou encerrar o aplicativo. Os botões são organizados em uma tabela (`Table`), garantindo um layout consistente e responsivo, promovendo uma navegação fluida e uma experiência de usuário agradável.

5.4 Loja

A loja é projetada para permitir que o jogador compre torres durante a partida, utilizando moedas acumuladas. Uma tabela (`Table`) é utilizada para organizar os botões de compra das torres.

No método `createStoreButtons()`, são criados quatro botões, cada um representando uma torre diferente: `TrapTower`, `BomberTower`, `SMGTower` e `SniperTower`. Cada botão é estilizado com uma imagem correspondente e possui um listener que, ao ser clicado, verifica se o jogador possui moedas suficientes para a compra. Se o jogador clicar em um botão e tiver moedas, a torre correspondente é selecionada e o valor é descontado das moedas do jogador.

O método `touchDown()` é responsável por posicionar a torre selecionada na tela quando o jogador clica, utilizando as coordenadas do mouse. A loja não só proporciona uma mecânica de compra estratégica, mas também incentiva o gerenciamento de recursos, tornando o jogo mais dinâmico e interativo.

6. RESULTADOS

Durante o processo de desenvolvimento de “Zombie Infection”, o grupo enfrentou grandes desafios referentes à programação e organização, cujas soluções foram essenciais para a conclusão do trabalho. Em síntese, foi concluído o objetivo final de construir um jogo no estilo tower defense, ampliando o conhecimento de técnicas de programação em Java, como waypoints, TiledMap, polimorfismo e programação orientada a objetos. Além disso, foram desenvolvidas habilidades de organização de equipes e tarefas, que foram subdivididas e gerenciadas conforme as necessidades do trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

LIBGDX. Wiki. *LibGDX*. Disponível em: https://libgdx.com/wiki/. Acesso em: 10 de out. 2024

OGZKRT. *A Tower Defense game written in Java by using LibGDX*. GitHub. Disponível em: https://github.com/ogzkrt/OTD/blob/master/desktop/src/com/javakaian/game/entity/Bullet.java. Acesso em: 28 de set. 2024.

APÊNDICE A - TELA DO JOGO EM EXECUÇÃO