Universidade Federal de Juiz de Fora

Graduação em Ciência da Computação

Leonardo Vieira Silva - 202235038

Pablo Henrique Silva de Faria - 202235012

Trabalho de Orientação a Objetos

Juiz de Fora

2023

1. INTRODUÇÃO

1.1. Projeto

Esse projeto foi desenvolvido com a intenção de criar um jogo que utiliza Java Swing e uma engine personalizada como base. Dessa forma, o jogo criado se baseia no Fruit Ninja, tendo como objetivo não permitir que nenhuma entidade gerada caia no limite inferior da tela. Caso caia, o jogador perde uma vida e, ao chegar em 0 vidas, ele perde. Para cada elemento que o jogador intercepte antes de cair, um ponto será adicionado ao contador de pontos da rodada. A intenção é acumular o máximo de pontos numa única partida.

Tal jogo não possui história, pois foi feito para ser infinito e proporcionar um exemplo da engine e entretenimento sem limite de tempo, livrando o usuário do problema de ser uma jogatina demorada ou complexa, uma vez que ele pode ser parado a qualquer instante sem perda de detalhes históricos.

Link do repositorio: https://github.com/LeonardoVieira25/trabalho-poo

Link para clone: https://github.com/LeonardoVieira25/trabalho-poo.git

1.2. Compilação e execução

A compilação do projeto deve ser feita usando o script “compile.sh” localizado na pasta mãe do projeto, através do comando “bash compile.sh”, no qual “bash” pode ser substituído pelo shell de preferência do usuário. Dessa forma, um arquivo .jar será gerado na pasta “compiled”, o qual pode ser usado para executar o jogo usando o comando “bash runCompiled.sh”.

1.3. Descrição do funcionamento

O jogo é iniciado à partir da classe *Main*, que cria o *MenuInicial*, dando acesso a uma tela de login, registro e, caso o usuário seja o admin (fazendo login usando a senha “admin” e usuário “admin”), a tela de edição de níveis. Após fazer login, uma tela de início é exibida (Map0) e então a opção de jogar ou sair são oferecidas.

Os usuários só podem logar após o registro, o qual fica salvo em um arquivo local. A pontuação é exibida na tela durante a jogatina, bem como o número de vidas restantes e o nível atual.

2. Classes e arquivos

2.1. Main

É o ponto de partida do projeto. Ao executar o arquivo *.jar* gerado, a classe *Main* se encarrega de criar uma nova instância do menu inicial.

2.2. Janela

A classe Janela é usada para configurar a janela do jogo, além de possuir informações e métodos importantes e um listener para a entrada de eventos do teclado. Tais informações envolvem o tamanho da tela - altura e largura - e as funções responsáveis por desenhar componentes do jogo e fazê-los aparecer.

Quanto ao *listener* mencionado, o jogador pode sair do jogo pressionando *ESCAPE* (*Esc*). Entretanto, já que esse é um jogo baseado completamente no mouse, esse é o único comando que pode ser usado pelo teclado.

2.3. Scene

Essa classe possui muitas funcionalidades, ao longo do arquivo serão encontrados processos que fazem com que ela guarde o looping principal do jogo, adicione listeners para o movimento e cliques no mouse, selecione o mapa através do atributo *selectedMapId,* verifique por colisões, remova os objetos da lista *ObjectsToRemove* - a qual foi criada nesse mesmo arquivo, confira se há elementos no buffer do mapa selecionado e os adicione na lista de elementos para renderizar, armazene o valor literal de FPS, renderize o mapa atual. Além de tudo isso, ela também separa os processos em threads para o looping de renderização e de lógica do jogo.

2.4. Menu Inicial

Nessa tela o usuário deve digitar suas credenciais, caso já possua uma conta, ou optar por se registrar para que possa jogar. Quando o login é permitido, ou seja, quando as credenciais são inseridas corretamente, estão no banco de dados e o jogador clica em “Login”, ele é direcionado para a tela inicial do jogo, onde poderá escolher entre sair do aplicativo ou jogar.

2.5. Tela Registro

É executada quando o usuário clica em “Registro” durante o menu inicial. Essa tela cuida da entrada de dados para um novo jogador e faz a comunicação entre esses dados e o sistema de persistência de arquivos de dados do jogo, permitindo que haja vários jogadores diferentes na mesma máquina sem interferência de um no outro. Após o registro, o usuário pode voltar a tela de login e entrar com os dados cadastrados.

2.6. Tela de edição de níveis

É acessada fazendo o login com o nome de usuário “admin” e a senha “admin”. Nessa tela é possível selecionar quais inimigos aparecerão em cada nível.

2.7. Exceções

2.7.1. UserNotLogged

É uma exceção disparada quando o usuário tenta fazer login utilizando dados que não têm correspondência nos dados salvos.

2.8. Maps

O arquivo *Maps* contém o construtor de um “mapa” - que são as telas executadas após o login. Tal construtor possui duas listas de *GameObjects*, uma lista dos que estão sendo executados imediatamente e outra que é o buffer, que contém os elementos que serão inseridos na primeira lista, além de um manuseador de eventos.

2.8.1. Map0

Map0 é a janela que dá a opção de jogar ou sair do aplicativo e possui *id* igual a 0. Os botões dela foram criados usando a classe Button que será detalhada mais à frente. Ao clicar em jogar, o id selecionado será alterado, fazendo com que o jogador passe para a tela do jogo propriamente dito, e ao clicar em sair o jogo é fechado.

2.8.2. Map1

Map1 corresponde à tela do jogo durante a jogatina. É nela que o jogador de fato joga. Durante essa tela, as vidas e a pontuação do usuário são exibidas conforme o jogo passa. Nesse mapa, novamente são utilizadas duas listas de *GameObject*, sendo uma delas o buffer, além de serem criados 4 *spawners* (que também são do tipo *GameObject*) que geram os inimigos escondidos no topo da tela, onde o usuário não pode enxergar.

2.8.3. Map2

Corresponde à tela de fim de jogo (Game Over). Essa tela aparece quando o jogador chega a 0 vidas durante o *Map1*. Quando *Map2* é executado, é sorteada uma frase de fim de jogo dentro da lista de *frases* para ser exibida no topo da tela. Além disso, nessa tela também é mostrado um ranking de jogadores registrados na mesma máquina de acordo com suas pontuações.

2.9. Utils

2.9.1. EventManager

Utiliza o padrão de desenvolvimento Observer para permitir a adição de ouvintes de eventos a determinados componentes do jogo. Essa classe possui as funções *addListener(java.util.function.Consumer<Object> listener)*, *removeListener(java.util.function.Consumer<Object> listener)* e *trigger(Object event)*, as quais foram utilizadas em outras partes do projeto.

2.9.2. XmlLoader

É responsável pela manipulação dos XMLs que guardam os dados do jogo, tais como registros e propriedades dos usuários. Esse arquivo possui as funções *loadLevels(), saveNewPlayer(Player player), getPlayer(String username, String password), loadNiveis(), saveNiveis(List<String> niveis), updatePlayer(Player player), getEnemyTypes(int currentLevel) e getPlayersRankings()*, todas responsáveis por manusear os XMLs, recuperando dificuldades, dados e registros, os quais são usados para gerar o jogo para o jogador, para comparar os dados na hora do login ou para salvá-los no registro.

2.10. Assets

Essa pasta guarda os arquivos XML e imagens usadas no jogo.

2.10.1. Níveis

É o arquivo XML responsável por armazenar os inimigos de cada nível.

2.10.2. Players

Guarda os dados dos usuários, como nome de usuário, senha, vida inicial, pontuação máxima, taxa de spawn das partículas que saem do mouse e taxa de espalhamento dessas partículas.

2.11. Behaviors

2.11.1. BounceHorizontal

Recebe um *PhysicsObject* e, no método *update()*, confere se a posição desse objeto é válida no eixo X, ou seja, se está saindo da tela para algum dos lados.

2.11.2. BounceVertical

É responsável por tratar quando um *PhysicsObject* entra na tela e bate nas bordas verticais dela (eixo Y). Possui dois construtores: um que recebe apenas um *PhysicsObject* e um que recebe um *PhysicsObject* e uma função para ser executada quando o objeto bater na parte inferior da janela. Essa classe confere se o objeto recebido encostou no limite inferior ou superior da janela e altera a velocidade dele para alterar sua direção usando a função *update()*.

2.11.3. ClickableArea

Caracteriza uma área clicável, recebendo um *GameObject* e um *Runnable* no construtor. A função *isInside(int X, int Y)* confere se os parâmetros X e Y recebidos na chamada da função correspondem a valores dentro do retângulo feito pelos valores de X, Y, largura e altura do *GameObject* recebido e retorna verdadeiro caso esteja dentro ou falso caso esteja fora.

2.11.4. Collision

Possui três construtores: um que recebe apenas um *PhysicsObject*, um que, além disso, recebe uma função de o que fazer ao colidir e outro que ainda recebe um parâmetro booleano de *trigger*. A função *update()* faz as verificações (ex: se o outro objeto que está colidindo de fato é válido) e alterações necessárias, como mudanças nas velocidades verticais e horizontais.

2.11.5. DestroyAfterTime

Classe encarregada de destruir um determinado objeto após determinado tempo. Seu construtor recebe um *GameObject* e um número *double* que indica o tempo que o parâmetro anterior deve durar até ser destruído. Esse processo é feito através da variável *accumulatedTime* e do atributo *removeNextIteration* - ambos utilizados na função *update()* dessa classe - que, ao ser passado para *true*, permite que o objeto seja removido da lista de objetos a serem renderizados.

2.11.6. SpawnGameObject

Essa classe possui 3 construtores:

- *SpawnGameObject(GameObject gameObject, List<GameObject> objectsList, float timeToSpawn, GameObject objectToSpawn)*
- *SpawnGameObject(GameObject gameObject, List<GameObject> objectsList, float timeToSpawn, List<GameObject> objectsToSpaw)*
- *SpawnGameObject(GameObject gameObject, List<GameObject> objectsList, float timeToSpawn, List<GameObject> objectsToSpaw, Function<GameObject, Void> beforeSpawn)*

A primeira e a segunda se diferem pois a primeira recebe apenas 1 *GameObject* no último parâmetro, enquanto a segunda recebe uma lista desses objetos. A terceira, por sua vez, além de receber uma lista de *GameObject*, também recebe uma função para executar antes de gerar o objeto. Essa classe também conta com uma função *update()* que usa uma variável *double* chamada *accumulatedTime* para verificar se o tempo *timeToSpawn já* foi alcançado e, só então, prosseguir. Após o tempo ter sido atingido, verifica se a lista de objetos é nula

e, se não for, seleciona um objeto aleatório de dentro dela para ser gerado. A geração desse objeto é feita através da função *clone(), que é implementada no objeto a ser clonado e retorna uma cópia equivalente do mesmo*. Depois, a função também verifica se o parâmetro *beforeSpawn é* nulo e, se não for, aplica o valor passado no objeto recém gerado. Então, o objeto é adicionado à lista de objetos *objectsList* e *accumulatedTime* passa a ser 0.

2.12. Components

2.12.1. GameObject

É a superclasse de várias classes da mesma pasta. Implementa a interface *Cloneable* e determina um *GameObject*. Seu construtor consiste em valores do tipo *double* para os parâmetros X, Y, largura e altura do elemento a ser criado. Possui uma função *clone()* que clona todos os behaviors de um dado elemento para uma lista e retorna o (do tipo *Object)* clone. Tem também uma função resetDeltaTimeRender() que muda o atributo *lastTimeRender* para o momento atual do computador. Isso é usado em cálculos em outras linhas do código. Além disso, também contém a função *update()* e *draw(Graphics2D g2d)*. A primeira mantém a variável *currentTimeNano* com o valor do sistema, faz um update em cada elemento dentro da lista *behaviors,* uma conta para determinar o valor de *deltaTimeRender* e, em seguida, atualiza o valor de *lastTimeRender* para o valor salvo em *currentTimeNano*. Todos esses valores são usados em cálculos durante o jogo, especialmente envolvendo FPS

2.12.2. Button

Essa classe é responsável pela criação de botões após a interface de login. Possui dois construtores que recebem uma *String* com o texto que será escrito no botão (String *name),* as coordenadas do centro do botão (X e Y). O segundo construtor recebe tudo isso e uma lista de *GameObject*.

Através da função *update(),* essa classe verifica se o botão possui uma lista de *GameObject* e se o último estado de hover (mouse sobre o botão) é diferente do atual, caso ambas sejam verdadeiras, *lastHoverState* passará a ser o estado de hover atual (*onHover,* verdadeiro ou falso*)* e, caso *lastHoverState* seja verdadeiro, partículas serão criadas e espalhadas a partir da posição do mouse no botão.

Por último, essa classe possui uma função *void draw(Graphics2D g2d)* responsável por desenhar o botão, alterar a cor enquanto o mouse estiver sobre ele

- e alterar de novo quando sair de cima dele e centralizar a string passada no construtor.

2.12.3. Contador

Contador é a classe dedicada inteiramente ao contador visível durante a jogatina (*Map1)*. Ele possui um construtor *Contador(String baseText, String id, int x, int y)*, em que *baseText* e o texto inicial dele, *id* e a identificação (usada para alterar valores), *X* e *Y* são as coordenadas do centro dele.

O construtor possui um *listener* ligado ao evento de mudança de pontos e, quando o jogador destrói um inimigo, ele aumenta seu valor em 1. A função *void draw(Graphics2D g2d)* se encarrega de posicionar o contador no local desejado e centralizar a escrita nele.

2.12.4. GameController

Essa classe é responsável por criar a lista de componentes da tela de upgrade (que é mostrada sempre que o usuário muda de nível), observar os eventos disparados e efetuar ações com base nos mesmos. Como exibir o botão de game over, mostrar a tela de upgrade, e removê-la assim que o usuário escolher um power up.

2.12.5. PhysicsObject

Representa um objeto com física. Seu construtor *PhysicsObject(double positionX, double positionY, double width, double height)* apenas chama o construtor da superclasse (*GameObject)* com os valores passados.

Possui uma função *Object clone()* que clona e retorna um clone com os mesmos valores dos atributos e *behaviors* da instância referenciada.

A função *void draw(Graphics2D g2d)* apenas chama a função *draw()* da superclasse.

Ela é responsável por simular efeitos de força e velocidade do objeto, permitindo que o mesmo se mova.

2.12.6. Player

Possui atributos int *id, String username, String password, int vidaInicial, int maxPontuacao, float spawnRate, float spread*, com getters e setters para todos eles, além do atributo *int vida* que possui apenas getter, *List<GameObject> objectsList que possui apenas setter e* os atributos *int lastPositionX, int lastPositionY que não possuem getters nem setters*.

Seu construtor *Player(List<GameObject> objectsList)* executa construtor da superclasse, recebe apenas uma lista de *GameObject* e a coloca na variável da instância.

Há também uma função *void update()*, a qual é responsável por detectar e calcular a variação na posição do mouse e criar instâncias de *SlashParticle* de acordo com as variáveis *accumulatedTime, spawnRate* e *velocity.*

2.12.7. SlashParticle

Essa classe estende *PhysicsObject* e é responsável por criar, aleatorizar e remover as partículas que saem do mouse após determinado tempo. Essas partículas possuem colisão e são o que permite o jogador marcar pontos. Seus construtores *SlashParticle(double positionX, double positionY, double width, double height)* e *SlashParticle(double positionX, double positionY, double width, double height, float initialVelocity, float timeToLive)* diferem em parâmetros e na aleatoriedade usada para criar as particulas

2.12.8. Spawner

Caracteriza os *GameObjects* usados como criadores de inimigos no topo da tela. Seu construtor é *Spawner(int positionX, int positionY, int width, int height, List<GameObject> objectsListBuffer)* e, basicamente, atribui ao spawner uma localização e tamanho e uma lista buffer, além de fazer com que ele gerem inimigos no alto da janela.

A função *void update()* executa a função super no game object, dessa forma calculando o deltaTime (essa variavel que garante consistência de movimento e tempo independente da velocidade da máquina testada) e garante que a posição Y do spawner seja no topo da janela.

2.12.9. Inimigos

2.12.9.1. Inimigo

Caracteriza um inimigo. Estende *PhysicsObject* e possui um construtor: *Inimigo(double positionX, double positionY, int width, int height)*. Esse construtor chama o construtor da superclasse com os mesmos parâmetros para inicializar um *PhysicsObject*. Também contém um parâmetro *int dano* que começa com valor -1 e possui um getter. Todos os *inimigos* se diferenciam numericamente (dimensões) e visualmente.

2.12.9.2. Inimigo1

Representa o inimigo do tipo 1, cujo construtor é *Inimigo1(double positionX, double positionY),* com cada parâmetro correspondendo às posições X e Y do inimigo. Esse inimigo é representado como um meteoro em chamas.

Contém uma função *void draw(Graphics2D g2d)* responsável por desenhar cada um dos inimigos.

Também há uma função *Object clone()* que clona os atributos e behaviors da instância dessa classe e retorna um clone.

2.12.9.3. Inimigo2

Representa o inimigo do tipo 2, cujo construtor é igual ao *Inimigo1.* Possui os mesmos métodos de Inimigo1. Inimigo 2 é representado na tela do jogador como um asteroide grande.

2.12.9.4. Inimigo3

Representa o inimigo do tipo 3, contendo o mesmo construtor de *Inimigo2 e os mesmos métodos*. *Inimigo3* é representado por um asteroide parecido com uma rocha.

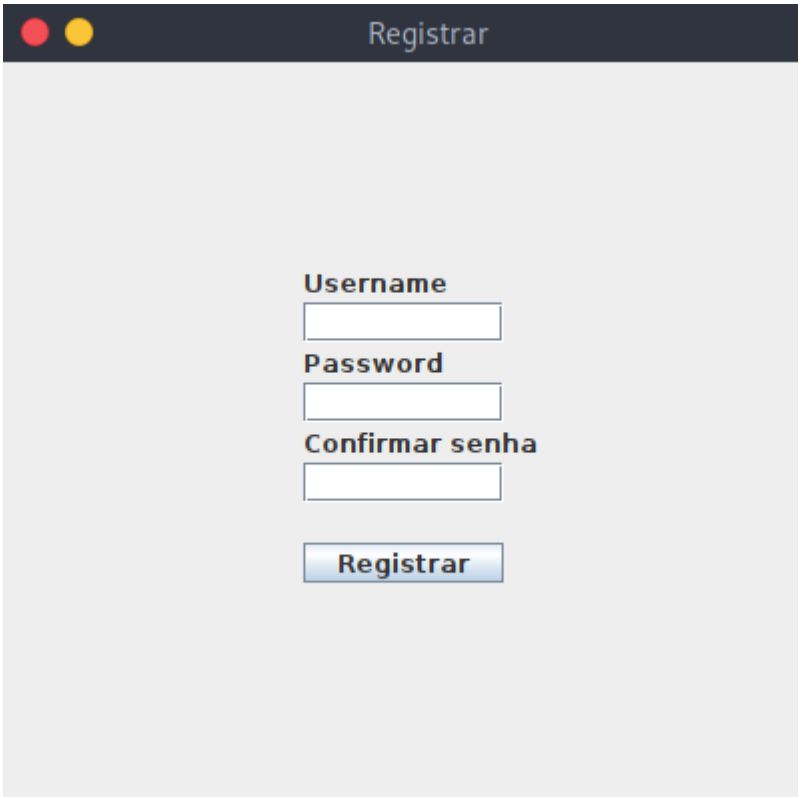
2.12.9.5. Inimigo4

Representa o inimigo do tipo 4 e possui o mesmo construtor que os outros inimigos e os mesmos métodos também. *Inimigo4* é representado por um asteroide em chamas cartoonizado.

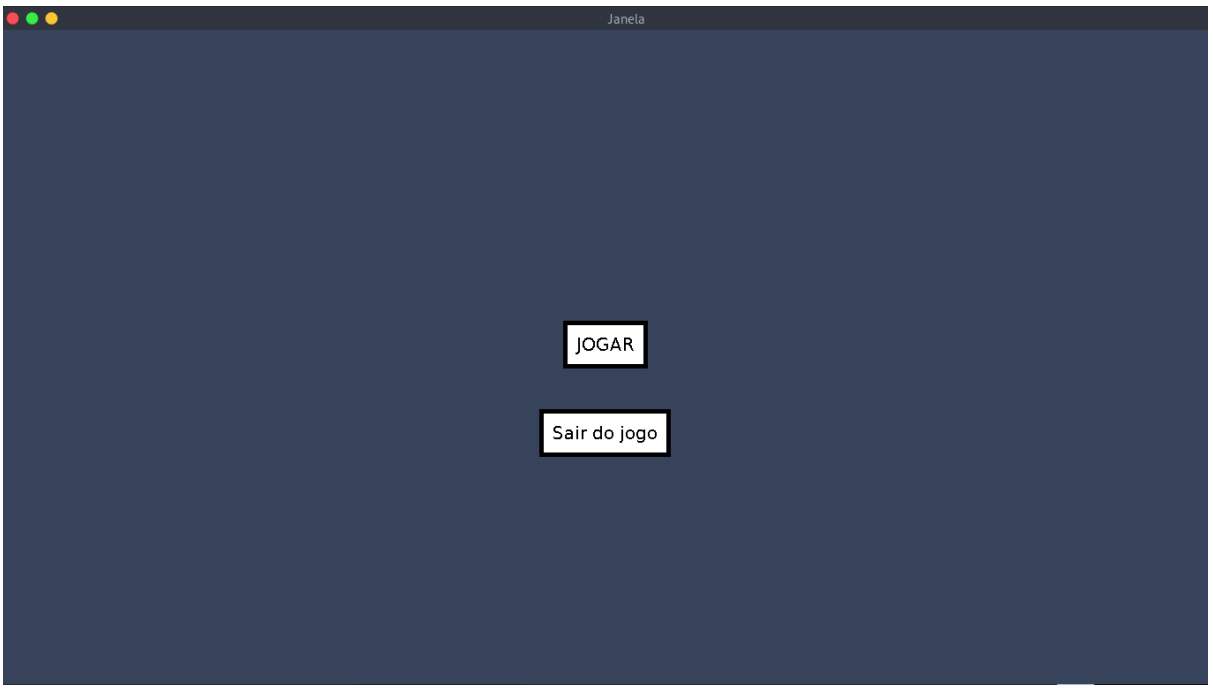
3. Interface gráfica

A interface gráfica usada nesse projeto durante a tela de login/registrar foi criada utilizando Swing, feita para ser direta e simples de entender o que se pede. Entretanto, após o login, a GUI se altera para uma personalizada apenas baseada em Swing, criada pelos alunos com intenção de dar mais interação ao usuário de maneira que a janela do jogo ficasse mais divertida.

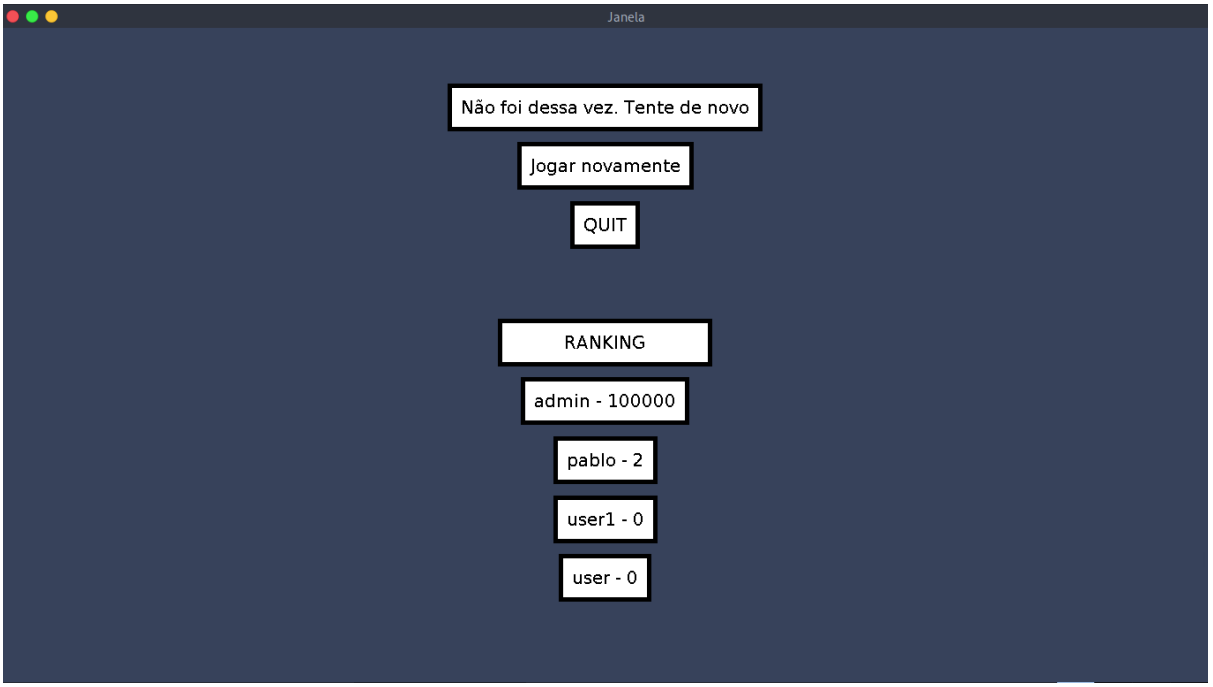
Tela de login.



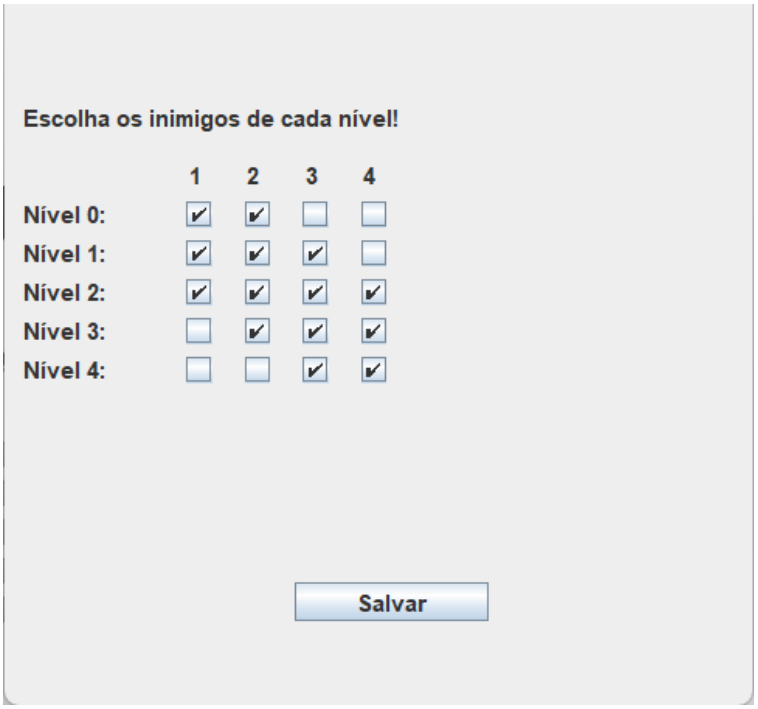
Tela de registro.



Tela de início após login (*Map0).*



Tela de Game Over (*Map2)*.



Tela de edição de níveis, disponível apenas para Admin.

4. Utilização dos conceitos de Orientação a Objetos

Todos os conceitos aprendidos na disciplina foram aplicados nesse projeto, tais como herança, polimorfismo, interfaces, bibliotecas, GUI com Swing, tratamento de exceções, manipulação de arquivos, múltiplos construtores, entre outros. Essas ideias e conceitos foram aplicados seguindo a necessidade do projeto conforme ele crescia e se mostraram muito úteis ao longo do desenvolvimento, especialmente para gerar entidades parecidas e implementar comportamentos parecidos.

5. Expectativas

Esperava-se construir um jogo simples, divertido, infinito e sem história para a demonstração da engine desenvolvida. Além disso, pudemos implementar a utilização de múltiplas threads do processador a fim de distribuir os cálculos executados por cada parte dele e observar o comportamento, as dificuldades e facilidades de adicionar essa característica em Java.

Sobretudo, o projeto se mostrou muito interessante para os desenvolvedores, os quais já estavam interessados em fazer um jogo e tiveram a oportunidade de fazer isso.

6. Conclusão

Ao fim do projeto, ele é capaz de

- Autenticar usuários

- Registrar usuários
- Editar características dos níveis (opção para admin)
- Exibir animações
- Contar tempo para animações que devem desaparecer
- Detectar colisões
- Clonar inimigos para geração múltipla de entidades
- Simular física de forma simplificada

Sendo assim, esse trabalho é capaz de demonstrar o quão poderoso Java Swing com uma engine personalizada podem ser. Demonstrando complexidade na criação de um jogo infinito com registro e autenticação de usuários e proporcionando entretenimento ao usuário.