Relatório do Projeto ZINF

1. Descrição do Trabalho Realizado

O presente trabalho detalha o desenvolvimento de "ZINF", um jogo 2D de aventura e exploração com visão de cima para baixo (*top-down*). Criado na linguagem C e utilizando a biblioteca Raylib para a renderização gráfica e gerenciamento de interações, o jogo desafia o jogador a navegar por múltiplos níveis, derrotar inimigos e coletar itens. A estrutura do projeto é modular, com fases (mapas) carregadas a partir de arquivos de texto externos, permitindo fácil expansão. O objetivo principal em cada fase é eliminar todos os monstros para poder avançar para a próxima.

2. Representação dos Elementos do Jogo

Os componentes do universo do jogo foram modelados utilizando estruturas de dados simples e eficientes:

- **Mapa:** O layout de cada fase é representado por uma matriz bidimensional de caracteres (char mapa[16][24]). Esses mapas são lidos de arquivos .txt, onde cada caractere possui um significado específico:
  - ‘P’: Parede, um obstáculo intransponível.
  - ‘.’: Chão, uma área livre para movimentação.
  - ‘J’: Posição inicial do jogador na fase.
  - ‘M’: Posição inicial de um monstro.
  - ‘E’: Espada, um item coletável que habilita o ataque.
  - ‘V’: Vida, um item que aumenta a vida do jogador.
- **Entidades (Jogador e Monstros):** As entidades dinâmicas do jogo são representadas por structs que agrupam seus respectivos dados e estados.
  - **struct personagem**: Armazena os dados do jogador, incluindo sua localização em coordenadas (vetor jogadorLoc), orientação (Norte, Sul, Leste, Oeste), pontuação, vidas restantes e um indicador booleano para a posse da espada.
  - **struct monstro**: Contém a localização, orientação e estado de vida de cada inimigo.
- **Highscores:** A tabela de pontuações mais altas é gerenciada pela struct Highscore, que armazena o nome do jogador e sua pontuação final. Estes dados são persistidos em um arquivo binário (ranking.bin), garantindo que o ranking seja mantido entre as sessões de jogo.

3. Implementação das Interações

A interatividade do jogo é controlada por um conjunto de funções que atualizam o estado dos elementos a cada ciclo do loop principal do jogo.

- **Movimento e Colisão com o Cenário:** A função AtualizaJogador monitora o input do teclado (teclas direcionais). Antes de efetivar um movimento, a função deveMover é

invocada para verificar se a célula de destino na matriz do mapa é uma parede ('P'). Se o caminho estiver livre, as coordenadas do jogador são atualizadas. O mesmo sistema é utilizado pelos monstros.

- **Inteligência Artificial dos Monstros:** A função atualizaMonstros dita o comportamento dos inimigos. Utilizando a "Distância de Manhattan", um monstro inicia a perseguição ao jogador se este estiver a uma distância de até 3 blocos. Caso contrário, o monstro se movimenta de forma aleatória. A IA também considera as paredes, tentando contorná-las para alcançar o jogador.
- **Coleta de Itens:** As funções verificaVidaColetada e verificaEspadaColetada comparam continuamente a posição do jogador com a dos itens no mapa. Se houver uma sobreposição, o atributo correspondente é atualizado na struct personagem e o caractere do item é removido da matriz do mapa, fazendo com que ele desapareça visualmente.
- **Sistema de Combate:** A lógica de combate é centralizada na função VerificarInteracoes:
    - **Dano ao Jogador:** Se o retângulo de colisão do jogador colidir com o de um monstro, o jogador perde uma vida e é reposicionado no ponto de partida da fase.
    - **Ataque do Jogador:** Se o jogador possui a espada (jogador.espada == 1) e pressiona a tecla 'J', o jogo projeta uma área de ataque retangular nos 3 blocos à sua frente. Se esta área colidir com um monstro, a vida do monstro é zerada, ele é removido do jogo e o jogador ganha pontos.

4. Estruturas e Funções Principais Utilizadas

- **Estruturas de Dados (structs):**
    - personagem: Agrega todos os dados e estados do jogador.
    - monstro: Agrega os dados e estados dos inimigos.
    - vetor: Estrutura auxiliar para armazenar coordenadas (x, y).
    - Highscore: Modela uma entrada na tabela de pontuação.
    - GameScreen: Um enum que gerencia o estado da aplicação, controlando qual tela é exibida (Menu, Gameplay, Pause, etc.).
- **Funções Principais:**
    - main(): Ponto de entrada do programa. Inicializa a janela, carrega os recursos gráficos (texturas) e executa o loop principal do jogo, que funciona como uma máquina de estados baseada na GameScreen atual.
    - carregarFase(): Orquestra o carregamento de um novo nível, chamando funções para ler o arquivo de mapa, posicionar o jogador e inicializar os monstros.
    - AtualizaJogador() e atualizaMonstros(): Contêm a lógica de input, movimento e comportamento do jogador e dos inimigos, respectivamente.
    - VerificarInteracoes(): Gerencia todas as colisões entre entidades e a lógica de combate.
    - IniciaMapa(): Itera sobre a matriz do mapa e desenha as texturas

correspondentes na tela.

5. Como Usar o Programa

1. **Compilação:** Para compilar o jogo, é necessário ter a biblioteca Raylib configurada no ambiente. Em um sistema Linux/Unix, utilize o script build.sh fornecido ou execute o comando diretamente no terminal:
   gcc -Wall -Werror -I/path/to/raylib/src -L/path/to/raylib/build/raylib -lraylib -o zinf_game main.c

   Substitua /path/to/raylib pelo caminho correto da sua instalação. O comando gera um arquivo executável chamado zinf_game.
2. **Execução:** Certifique-se de que o executável está no mesmo diretório que as pastas de recursos (sprites/ e mapas/). Execute o jogo pelo terminal:
   ./zinf_game

3. **Controles do Jogo:**
   - **Menu Principal:** Use as teclas 1 (Jogar), 2 (Ver Highscores) e 3 (Sair).
   - **Durante o Jogo:**
     - **Setas Direcionais:** Movem o personagem.
     - **Tecla 'J':** Ataca com a espada (após coletá-la).
     - **Tecla 'P':** Pausa ou despausa o jogo.
   - **Outras Telas:** Pressione ENTER para retornar ao menu ou avançar após as telas de pausa, highscores e fim de jogo.