

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

Curso de Engenharia de Controle e Automação

Algoritmos e Programação (INF01202)

Profº Dr. Thiago L. T. da Silveira

Alberto Cinésio Veit, Arthur Dos Santos e João Francisco de Oliveira Dantas

TRABALHO FINAL

Jogo de Computador Utilizando a Linguagem C

INTRODUÇÃO	1
OBJETIVO	1
CAMINHOS	1
BIBLIOTECAS INCLUÍDAS	3
CÓDIGO DO JOGO “MAIN”	3
FUNCIONAMENTO PRÁTICO	14
CONCLUSÕES	14
REFERÊNCIAS	14

INTRODUÇÃO

Os jogos eletrônicos têm desempenhado um papel cada vez mais significativo no mundo do entretenimento, proporcionando experiências imersivas e emocionantes aos jogadores. Como alunos do primeiro semestre do curso de Engenharia de Controle e Automação, buscamos com este projeto consolidar os conceitos de programação que foram estudados ao longo do semestre, mas também aprimorarmos nossas habilidades de pesquisa direcionadas à área de programação. Dessa forma, produzimos um jogo inspirado no aclamado "The Binding of Isaac" misturando um pouco com outro muito conhecido "Minecraft".

Um dos aspectos cruciais que contribuem para tornar o jogo cativante é a aplicação da inteligência artificial (IA) para controlar os personagens não-jogadores. Com o intuito de criar fases desafiadoras e envolventes, investimos tempo extra na otimização do comportamento dos inimigos. Este relatório servirá como uma exploração detalhada dos passos e processos envolvidos na construção de nosso projeto final.

OBJETIVO

Os principais objetivos deste projeto de desenvolvimento de jogo incluíram a aplicação prática de conceitos de programação, aprimoramento das habilidades de trabalho em equipe, pesquisa independente para abordar elementos não cobertos em sala de aula, criação de uma experiência de jogo completa com menus interativos e feedback audiovisual, testes extensivos para garantir a jogabilidade e estabilidade do jogo, e finalmente, a entrega de um projeto de jogo funcional e completo que refletisse as habilidades adquiridas no semestre.

CAMINHOS

Após baixado o jogo você irá se deparar com uma pasta que armazena todas as informações necessárias para a execução correta do jogo.

Devido a necessidade de ter uma grande quantidade de arquivos decidimos implantar alguns caminhos extras para melhor organização.

1. Raiz

Na raiz da pasta do jogo encontramos o arquivo .c, nomeado jogo seguido da sua data e versão, é o arquivo fonte de execução. O arquivo .o, criado pelo compilador. O arquivo .exe sendo ele responsável pela execução do jogo.

Nome	Data de modificação	Tipo
resources	10/09/2023 21:42	Pasta de arquivos
jogo23_09_10_02	10/09/2023 23:45	Aplicativo
jogo23_09_10_02	10/09/2023 23:44	Arquivo Fonte C
jogo23_09_10_02.o	10/09/2023 23:45	Arquivo O

2. Recursos (Jogo\resources)

Essa pasta reúne outras pastas que contém informações para a execução correta do jogo,

saves	10/09/2023 23:51	Pasta de arquivos
maps	10/09/2023 23:12	Pasta de arquivos
sounds	10/09/2023 21:42	Pasta de arquivos
textures	10/09/2023 21:42	Pasta de arquivos
font	10/09/2023 21:19	Pasta de arquivos

3. Texturas (Jogo\resources\textures)

Nesta pasta temos a presença de várias imagens .png com nomes referentes a objetos no jogo

Nome	Data de modificação	Tipo
deepslate	08/09/2023 02:25	Arquivo PNG
stonebricks1	07/09/2023 20:06	Arquivo PNG
stonebricks3	07/09/2023 20:06	Arquivo PNG
stonebricks2	07/09/2023 20:06	Arquivo PNG
obsidian	07/09/2023 19:19	Arquivo PNG
projectile	07/09/2023 03:34	Arquivo PNG
portal	07/09/2023 02:15	Arquivo PNG
stone	07/09/2023 02:14	Arquivo PNG


4. Sons (Jogo\resources\sounds)

Aqui temos a presença de todas as músicas e efeitos especiais utilizados.

Nome	Data de modificação	Tipo
sound1	08/09/2023 02:13	Arquivo WAV
sound3	08/09/2023 02:12	Arquivo WAV
sound2	08/09/2023 02:12	Arquivo WAV
background	08/09/2023 01:37	Arquivo WAV



5. Fontes (Jogo\resources\font)

Na pasta de recursos temos o arquivo .ttf que é um pacote de fonte que é usado para todo texto escrito na tela durante a execução do jogo.

 font	02/02/2011 15:45	Arquivo de fonte ...	16 KB
--	------------------	----------------------	-------








6. Salvamentos (Jogo\resources\saves)

E caso já tenha realizado alguma run no jogo, estarão presentes os arquivos de save.bin, sendo um total de 3 espaços possíveis que tem papel de armazenar as informações importantes para o futuro carregamento de jogo.

 save2.bin	10/09/2023 23:51	Arquivo BIN	131 KB
 save1.bin	10/09/2023 23:51	Arquivo BIN	131 KB

7. Mapas (Jogo\resources\maps)

Todos os mapas listados com o nome map seguido da sua numeração .txt

 map000	10/09/2023 23:11	Documento de Te...	2 KB
 map002	10/09/2023 23:07	Documento de Te...	2 KB
 map010	10/09/2023 22:53	Documento de Te...	2 KB
 map008	10/09/2023 22:47	Documento de Te...	2 KB
 map003	10/09/2023 22:19	Documento de Te...	2 KB
 map005	10/09/2023 22:11	Documento de Te...	2 KB
 map004	10/09/2023 22:07	Documento de Te...	2 KB

BIBLIOTECAS INCLUÍDAS

Foram utilizados das seguintes bibliotecas para a realização do projeto:

- <math.h>
- <raylib.h>
- <string.h>
- <time.h>
- <stdlib.h>
- <stdio.h>

CÓDIGO DO JOGO

Para descrever melhor o nosso jogo iremos percorrer a main de uma forma descritiva e ilustrativa. Primeiramente inicializamos as funções de aleatoriedade, de criação de janela e ambiente de áudio, após isso foi feito configurado o FPS, a tecla de saída e a amplitude do áudio.

Foram inicializadas todas as estruturas e variáveis utilizadas carregados as texturas e sons após a garantia de que as entidades estivessem com os status zerados e iniciado a chamada de funções de **carregamento de mapa e escaneamento de mapa**.

```

int main(void)
{

    srand(time(NULL)); //inicializa o gerador de numero aleatório
    InitWindow(MAP_PIXEL_WIDTH, MAP_PIXEL_HEIGHT+HUD_PIXEL_HEIGHT, "Joguinho do balacobaco");//Inicializa janela, com certo tamanho e titulo
    InitAudioDevice();
    SetTargetFPS(FPS);// Ajusta a execucao do jogo para 60 frames por segundo
    SetExitKey(KEY_NULL);
    SetMasterVolume(0.25);

    struct STRUCT_DATA data;
    struct STRUCT_TEXTURE texture;

    int i,j,k;//contadores declarados aqui pois sao enviados por referencia
    int past_x ,past_y;
    int option = 1;
    int enemies_alive;
    int intends_to_save_or_load = 0;
    char map_file_name[MAX_ARCHIVE_NAME] = "map000.txt";//nome da versao do mapa que estamos jogando
    bool menu_open = true;

    Texture2D textures[20];
    textures[0] = LoadTexture("resources/textures/steve.png");// Texture loading
    textures[1] = LoadTexture("resources/textures/skeleton.png");

```

Na função de **carregamento** de mapa “**load_map()**” utilizamos dos conhecimentos adquiridos em aula prática de arquivos de texto, onde lemos linha por linha e armazenamos em uma matriz de char dentro da estrutura de dados do mapa como ilustrado abaixo.

```

bool load_map(struct STRUCT_DATA *data,char archive_name[])//carrega o mapa de um arquivo txt
{
    //+1 devido o \n no fim de cada linha

    FILE *txt_arc = fopen(archive_name, "r");//o ponteiro txt_arc recebe o arquivo
    if (txt_arc == NULL)//testa se o arquivo existe na pasta
    {
        printf("Opening error\n");//se inexistente imprime na tela um erro
    }
    else//se aberto corretamente
    {
        for(int n_line = 0; n_line<30; n_line++)//vasculha linha por linha do mapa ate 1<30
        {
            if (fread(data->map[n_line], sizeof(char), (61), txt_arc) != (61))//realiza a leitura dessas linhas por caractere sendo um total de 61
            {
                printf("Reading error in line %d\n", n_line);//se a leitura der errado imprime na tela uma mensagem de erro e qual linha
                fclose(txt_arc);
                return 0;
            }
        }
    }
    fclose(txt_arc);//fecha o arquivo
    printf("Successful loading\n");//se tudo der certo imprime na tela sucesso no carregamento
    return 1;
}

```

Na função de **escaneamento** “**scan_map()**” nós enviamos essa matriz para que a função percorra por todas as linhas identificando cada caracter presente, exemplo ‘#’ como parede ou ‘I’ como o inimigo padrão.

```

bool scan_map (struct STRUCT_DATA *data)
{
    for (int y = 0; y < MAP_LINES; y++)//colunas
    {
        for (int x = 0; x < MAP_COLUMNS; x++)//linhas
        {
            switch(data->map[y][x])//interpretacao dos caracteres
            {
                case '#'://parede
                    data->wall[data->wall_counter].x = x;
                    data->wall[data->wall_counter].y = y;
                    data->wall[data->wall_counter].health_points = WALL_HP;
                    data->wall_counter=data->wall_counter+1;
                    break;
                case 'O'://parede indestrutivel
                    data->wall1[data->wall1_counter].x = x;
                    data->wall1[data->wall1_counter].y = y;
                    data->wall1_counter=data->wall1_counter+1;
                    break;
                case 'I'://inimigo ZUMBI
                    data->enemy[data->enemy_counter].x = x;
                    data->enemy[data->enemy_counter].y = y;
                    data->enemy[data->enemy_counter].health_points = ENEMY_HP;
                    data->enemy[data->enemy_counter].step.cooldown = CLOCKS_PER_SEC*ENEMY_STEP_COOLDOWN;
                    data->enemy[data->enemy_counter].action.cooldown = CLOCKS_PER_SEC*ENEMY_ACTION_COOLDOWN;
                    data->enemy_counter = data->enemy_counter+1;
                    break;
            }
        }
    }
}

```

Iniciamos o loop principal com “(!WindowShouldClose()){“ e iniciamos a música de fundo, após isso executamos a lógica do menu onde caso o menu esteja aberto o jogo permanece pausado.

```

while (!WindowShouldClose()){//Roda enquanto não for fechada a tela

    if (IsKeyPressed(KEY_F)) {
        ToggleFullscreen(); // Alternar entre tela cheia e janela ao pressionar a tecla F11
    }

    if(!IsSoundPlaying(background))
        PlaySound(background);

    if(IsKeyPressed(KEY_M)){
        menu_open = !(menu_open);
    }

    if (!menu_open) //Roda enquanto o menu não for aberto
    {
        enemies_alive = 0;

        for(i=0; i<MAX_ENEMIES; i++) if(data.enemy[i].health_points>0) enemies_alive++;
        for(i=0; i<MAX_ENEMIES1; i++) if(data.enemy1[i].health_points>0) enemies_alive++;
        for(i=0; i<MAX_ENEMIES2; i++) if(data.enemy2[i].health_points>0) enemies_alive++;
    }
}

```

A lógica do **jogador** que recebe os comandos do teclado W, A, S ou D para movimentos que são executados dentro da função que deve mover caso a condição seja atendida.

A função **deve mover** “**test_move()**” verifica se há algum obstáculo na direção da intenção de movimento e também não permite que os objetos passem na diagonal caso existam paredes perpendiculares ao jogador e adjacentes à diagonal.

```
bool test_move(struct STRUCT_DATA *data ,struct STRUCT_STATS *object)//semelhante a deve mover
{
    //comeca com flag 1, significando que pode mover, e vai testando condicoes que impede ele de mover
    bool flag = 1;

    //se a posicao desejada não for vazia, entao flag eh zero
    if(!(data->map[object->y+object->dy][object->x+object->dx]==' ')) flag = 0;

    //se a posicao desejada estiver fora dos limites do mapa, entao flag eh zero
    if(!(object->x+object->dx>=0 && object->x+object->dx<MAP_COLUMNS && object->y+object->dy>=0 && object->y+object->dy<MAP_LINES)) flag = 0;

    //se o objeto esta movendo em diagonal
    if(object->dx!=0 && object->dy!=0)
    {
        //se o objeto estiver se expremendo entre 2 objetos solidos, flag eh zero
        if(!(data->map[object->y+object->dy][object->x]==' ' || data->map[object->y][object->x+object->dx]==' ')) flag = 0;
    }

    return flag;
}
```

Já a **move** executa a movimentação dos objetos de acordo com a intenção de movimento, incrementando dx ao x e dy ao y e atualiza a visualização no mapa

```
void move(struct STRUCT_DATA *data ,struct STRUCT_STATS *object)//funcao que movimenta os objetos na matriz
{
    char temp;//variavel temporaria

    temp = data->map[object->y][object->x];//igual a o tipo de caractere (P, o, I) a posicao inicial da estrutura
    data->map[object->y][object->x] = ' ';//entao zera a posicao inicial da matriz

    object->x = object->x + object->dx;//encrementa dx ao x, ou seja cria o movimento e gera a nova posicao
    object->y = object->y + object->dy;//encrementa dy ao y, ou seja cria o movimento e gera a nova posicao

    data->map[object->y][object->x] = temp;//a nova posicao recebe entao o valor temporario (P, o, I) para fazer o objeto aparecer de novo
}
```

Ainda na lógica do jogador é feito uma verificação de existência de bombas, ou seja, caso o jogador passe em cima de uma bomba é percorrido o vetor de bombas para identificar qual foi, removendo ela do cenário e armazenando uma cópia no inventário do jogador para ser utilizada depois.

A mesma coisa ocorre quando o jogador passa por portais, mas ao invés de armazenar um portal, ele incrementa o nível da fase e carrega um novo mapa.

```

if (IsKeyDown(KEY_D) || IsKeyDown(KEY_A) || IsKeyDown(KEY_W) || IsKeyDown(KEY_S))//se alguma das teclas estaõ sendo pressionadas
{
    data.player.step.timer_current = clock();
    if (data.player.step.timer_current - data.player.step.timer_start >= data.player.step.cooldown)//se o cooldown for zero, pega a intenção de movimento
    {
        //determina a intencao de movimento de acordo com a combinacao de teclas
        if (IsKeyDown(KEY_D)) data.player.dx++;
        if (IsKeyDown(KEY_A)) data.player.dx--;
        if (IsKeyDown(KEY_W)) data.player.dy--;
        if (IsKeyDown(KEY_S)) data.player.dy++;
        data.player.step.timer_start = clock();
    }
}

if(test_if_object_is_there(&data,&data.player,data.player.dx,data.player.dy,'B')){
    for(i=0; i<MAX_BOMBS; i++){
        if(data.bomb[i].x==(data.player.x+data.player.dx)&&(data.bomb[i].y==(data.player.y+data.player.dy))){
            data.map[data.bomb[i].y][data.bomb[i].x] = ' ';
            clear_data(&data.bomb[i]);
            data.bomb_inventory = data.bomb_inventory+1;
        }
    }
}

```

Também temos uma função para verificar se o jogador morreu “**test_if_dead**”, que recebe a estrutura do jogador e verifica se suas vidas acabaram “ $HP \leq 0$ ” e se for o caso faz o jogador desaparecer.

```

bool test_if_dead(struct STRUCT_DATA *data ,struct STRUCT_STATS *object)//testa se o objeto morreu
{
    bool flag = false;

    if(object->health_points<=0)
    {
        data->map[object->y][object->x] = ' ';
        object->dx = 0;
        object->dy = 0;
        object->x = -1;
        object->y = -1;
        object->health_points = 0;
        flag = true;
    }

    return flag;
}

```

Já na lógica para as ações de **tiro e bomba** é verificado se foi clicado as teclas de setas e determina a direção do tiro de acordo com a posição do jogador executando a função “**shoot()**”.

```

if (IsKeyDown(KEY_RIGHT) || IsKeyDown(KEY_LEFT) || IsKeyDown(KEY_UP) || IsKeyDown(KEY_DOWN))//ve se alguma das teclas estao sendo pressionadas
{
    data.player.action.timer_current = clock();
    if (data.player.action.timer_current - data.player.action.timer_start >= data.player.action.cooldown)//se o cooldown for zero, pega a intenção de movimento
    {
        //determina a intencao de tiro de acordo com a combinacao de teclas
        if (IsKeyDown(KEY_RIGHT)) data.player.head_direction_x++;
        if (IsKeyDown(KEY_LEFT)) data.player.head_direction_x--;
        if (IsKeyDown(KEY_UP)) data.player.head_direction_y--;
        if (IsKeyDown(KEY_DOWN)) data.player.head_direction_y++;
        data.player.action.timer_start = clock();
    }
}

if (IsKeyDown(KEY_E)) data.intends_to_put_bomb = true;

if(data.player.head_direction_x!=0||data.player.head_direction_y!=0)//se alguma das direções não for nula
{
    if(data.intends_to_put_bomb==true&&data.bomb_inventory>0){
        if(place_bomb(&data,&data.player, data.player.head_direction_x, data.player.head_direction_y)) data.intends_to_put_bomb = false;
    }else{
        if(!shoot(&data,&data.player, &data.projectile[data.shot_counter], data.player.head_direction_x, data.player.head_direction_y)){
            damage_position(&data,data.player.x+data.player.head_direction_x,data.player.y+data.player.head_direction_y,1);
        }//atira
        data.intends_to_put_bomb = false;
    }
}
}

```

A função shoot é responsável por criar o projétil e fazer ele aparecer na tela, também controla os aspectos de movimentação do mesmo.

```

bool shoot(struct STRUCT_DATA *data, struct STRUCT_STATS *shooter, struct STRUCT_STATS *shot, int initial_x,int initial_y)
{
    //funcao do tiro
    if(test_if_object_is_there(data,shooter,initial_x,initial_y, ' '))//testa se pode ser "criado" o tiro
    {
        shot->x = shooter->x+initial_x;//a posicao do tiro eh a posicao do player mais a posicao inicial do tiro
        shot->y = shooter->y+initial_y;
        shot->dx = initial_x;//o deslocamento do tiro eh igual a posicao inicio do tiro
        shot->dy = initial_y;//basicamente pega a direcao inicial do tiro e diz para qual direção sera seu deslocamento
        shot->health_points = 1;

        data->map[shot->y][shot->x] = 'o';//a matriz recebe o char do tiro e aparece na tela
        data->stream[data->stream_counter].x = shot->x;
        data->stream[data->stream_counter].y = shot->y;
        data->stream[data->stream_counter].lifetime.timer_start = clock();
        data->stream[data->stream_counter].lifetime.cooldown = STREAM_LIFETIME*CLOCKS_PER_SEC;

        if(initial_x==0&&initial_y==1)data->stream[data->stream_counter].type = 0;
        if(initial_x==1&&initial_y==1)data->stream[data->stream_counter].type = 2;
        if(initial_x==1&&initial_y==0)data->stream[data->stream_counter].type = 4;
        if(initial_x==1&&initial_y==1)data->stream[data->stream_counter].type = 6;
        if(initial_x==0&&initial_y==1)data->stream[data->stream_counter].type = 8;
        if(initial_x==1&&initial_y==1)data->stream[data->stream_counter].type = 10;
        if(initial_x==1&&initial_y==0)data->stream[data->stream_counter].type = 12;
        if(initial_x==1&&initial_y==1)data->stream[data->stream_counter].type = 14;
        data->smoke[data->smoke_counter].x = shot->x;
        data->smoke[data->smoke_counter].y = shot->y;
        data->smoke[data->smoke_counter].lifetime.timer_start = clock();
        data->smoke[data->smoke_counter].lifetime.cooldown = SMOKE1_LIFETIME*CLOCKS_PER_SEC;
        data->smoke[data->smoke_counter].type = 1;
    }
}

```

Caso o jogador possua bombas no inventário e clique na tecla ‘E’ ele irá segurar a bomba, sendo a próxima tecla de seta responsável por posicionar a bomba na direção da seta de acordo com a função posiciona bomba “**place_bomb()**”.


```

bool place_bomb(struct STRUCT_DATA *data, struct STRUCT_STATS *placer, int initial_x, int initial_y)
{
    //funcao de botar bomba

    if(test_if_object_is_there(data, placer, initial_x, initial_y, ' '))//testa se pode ser "criado" o tiro
    {
        data->bomb[data->bomb_counter].x = placer->x+initial_x;//a posicao do tiro eh a posicao do player mais a posicao inicial do tiro
        data->bomb[data->bomb_counter].y = placer->y+initial_y;
        data->bomb[data->bomb_counter].health_points = 1;

        data->map[data->bomb[data->bomb_counter].y][data->bomb[data->bomb_counter].x] = 'B';//a matriz recebe o char da bomba e aparece na tela

        data->bomb_inventory = data->bomb_inventory - 1;
        data->bomb_counter = data->bomb_counter+1;//conta quantos projeteis foram acionados
        if(data->bomb_counter==MAX_PROJECTILES)//se a contagem for igual ao maximo de projeteis na tela
        {
            data->bomb_counter = 0;//zera a contagem
        }
        return true;
    }else return false;
}

```

Para a movimentação dos disparos, é percorrido o vetor de projéteis e feito as verificações de “Deve mover” e caso seja possível movimentar o tiro é chamado a função move, ainda dentro da verificação também são executados todos os efeitos de disparo, que são baseados na direção criando linhas para

```

for(i=0; i<MAX_PROJECTILES; i++)//percorre o vetor de projéteis
{
    if(test_move(&data,&data.projectile[i]))//se o projétil puder mover
    {
        move(&data,&data.projectile[i]);//move o projétil
        data.stream[data.stream_counter].x = data.projectile[i].x;
        data.stream[data.stream_counter].y = data.projectile[i].y;
        data.stream[data.stream_counter].lifetime.timer_start = clock();
        data.stream[data.stream_counter].lifetime.cooldown = STREAM_LIFETIME*CLOCKS_PER_SEC;

        if(data.projectile[i].dx==0&&data.projectile[i].dy==1)data.stream[data.stream_counter].type = 0;
        if(data.projectile[i].dx==1&&data.projectile[i].dy==1)data.stream[data.stream_counter].type = 2;
        if(data.projectile[i].dx==1&&data.projectile[i].dy==0)data.stream[data.stream_counter].type = 4;
        if(data.projectile[i].dx==1&&data.projectile[i].dy==1)data.stream[data.stream_counter].type = 6;
        if(data.projectile[i].dx==0&&data.projectile[i].dy==1)data.stream[data.stream_counter].type = 8;
        if(data.projectile[i].dx==1&&data.projectile[i].dy==1)data.stream[data.stream_counter].type = 10;
        if(data.projectile[i].dx==1&&data.projectile[i].dy==0)data.stream[data.stream_counter].type = 12;
        if(data.projectile[i].dx==1&&data.projectile[i].dy==1)data.stream[data.stream_counter].type = 14;

        data.smoke[data.smoke_counter].x = data.projectile[i].x;
        data.smoke[data.smoke_counter].y = data.projectile[i].y;
        data.smoke[data.smoke_counter].lifetime.timer_start = clock();
        data.smoke[data.smoke_counter].lifetime.cooldown = SMOKE1_LIFETIME*CLOCKS_PER_SEC;
        data.smoke[data.smoke_counter].type = 1;

        data.smoke_counter = data.smoke_counter+1;
        data.stream_counter = data.stream_counter+1;

        if(data.stream_counter>=MAX_SMOKES)data.stream_counter = 0;
        if(data.smoke_counter>=MAX_SMOKES)data.smoke_counter = 0;
    }
}

```

Caso ele não possa mover significa que o disparo se chocou com algo, então chama a função de dano “damage_position()” e decrementa a vida do projétil.

```

void damage_position(struct STRUCT_DATA *data, int x_position, int y_position, int damage)//funcao que movimenta os objetos na matriz
{
    int i;

    if(data->player.x == x_position && data->player.y == y_position && data->player.health_points>=1)//se a posição onde quer infligir dano é a mesma do player
    {
        data->player.health_points = data->player.health_points - damage;//decrementa da vida
    }

    for(i=0; i<MAX_ENEMIES1; i++)//percorre o vetor de inimigos ESQUELETO
    {
        if(data->enemy1[i].x == x_position && data->enemy1[i].y == y_position && data->enemy1[i].health_points>=1)//se a posição onde quer infligir dano é a mesma do inimigo
        {
            data->enemy1[i].health_points = data->enemy1[i].health_points - damage;//decrementa da vida
        }
    }
}

```

Essa função checa a posição de todos os objetos do jogo e caso a posição do objeto coincidir com a posição que a função deseja dar dano ele decrementa a vida desse elemento em relação ao dano que deve dar.

Ainda para o tiro é chamada a função “**test_if_dead()**” novamente, porém, dessa vez enviando os projéteis para verificar se eles “tomam dano” fazendo eles sumirem da tela.

Para os inimigos fizemos ações diferentes de acordo com sua classe, caso o inimigo seja o **ZUMBI** ele terá sua intenção de movimento de acordo com a posição do jogador, ou seja ele irá seguir o jogador.

Ele verifica se um inimigo está morto “**test_if_dead()**“, controla um tempo de cooldown, verifica a posição do jogador e determina a intenção de movimento comparando a posição do jogador com a posição do inimigo. Se o cooldown permitir, o inimigo tenta se mover “**test_move()**“ na direção desejada através da “**test_move()**“, mas se encontrar obstáculos, poderá causar dano a objetos próximos. Se ele não puder mover, o inimigo escolhe uma nova direção aleatória, como consideramos tiros um objeto em que o inimigo não pode cruzar por cima é possível que inimigos desviem de tiros. eventualmente.

```

for(i=0; i<MAX_ENEMIES; i++)//percorre o vetor de inimigos
{
    test_if_dead(&data,&data.enemy[i]);
    data.enemy[i].step.timer_current = clock();
    if (data.enemy[i].step.timer_current - data.enemy[i].step.timer_start >= data.enemy[i].step.cooldown)//se o cooldown for zero, pega a intenção de movimento
    {
        if (data.player.x > data.enemy[i].x) data.enemy[i].dx = 1;
        if (data.player.x < data.enemy[i].x) data.enemy[i].dx = -1;
        if (data.player.y > data.enemy[i].y) data.enemy[i].dy = 1;
        if (data.player.y < data.enemy[i].y) data.enemy[i].dy = -1;
        data.enemy[i].step.timer_start = clock();

        if(test_move(&data,&data.enemy[i]))//testa se ele pode mover
        {
            move(&data,&data.enemy[i]);//move
        }
        else if((data.enemy[i].x>=0 && data.enemy[i].x<MAP_COLUMNS && data.enemy[i].y>=0 && data.enemy[i].y<MAP_LINES))
        {
            data.enemy[i].action.timer_current = clock();
            if((test_if_object_is_there (&data,&data.enemy[i],data.enemy[i].dx,data.enemy[i].dy,'#')||test_if_object_is_there (&data,&data.enemy[i],data.enemy[i].dx,data.enemy[i].dy,'#'))
            {
                damage_position(&data,data.enemy[i].x+data.enemy[i].dx,data.enemy[i].y+data.enemy[i].dy,1);
                data.enemy[i].action.timer_start = clock();
            }
            data.enemy[i].dx = (-1 + (rand()%3));
            data.enemy[i].dy = (-1 + (rand()%3));
            if(test_move(&data,&data.enemy[i]))//testa se ele pode mover
            {
                move(&data,&data.enemy[i]);//move
            }
        }
    }
}

```

Se o inimigo for um **ESQUELETO** verifica se o inimigo está morto, em seguida, verifica se o tempo de cooldown para ações do inimigo, como disparar projéteis, atacar ou mover-se, foi atingido. Se o cooldown permitir, o código avalia a posição do jogador em relação à posição do inimigo e, com base nisso, determina a intenção de ação do inimigo. Se o jogador estiver alinhado horizontalmente (mesma linha), o inimigo tentará atirar usando a “**shoot()**” em direção ao jogador na mesma linha. Se

estiver alinhado verticalmente (mesma coluna), o inimigo tentará atirar na direção do jogador na mesma coluna. Além disso, o código também lida com o movimento do inimigo. Ele calcula se o jogador está mais próximo na horizontal ou na vertical e, em seguida, move o inimigo na direção apropriada.

```
for(i=0; i<MAX_ENEMIES1; i++)//percorre o vetor de inimigos
{
    test_if_dead(&data,&data.enemy1[i]);
    data.enemy1[i].step.timer_current = clock();
    data.enemy1[i].action.timer_current = clock();
    if (data.enemy1[i].action.timer_current - data.enemy1[i].action.timer_start >= data.enemy1[i].action.cooldown)//se o cooldown for zero, pega a intenção de movimento
    {
        if(data.player.x == data.enemy1[i].x){//se estiver alinhado em x
            if(data.player.y > data.enemy1[i].y){//e o player estiver abaixo
                shoot(&data,&data.enemy1[i],&data.projectile[data.shot_counter],0,1);
                data.enemy1[i].action.timer_start = clock();
            }
            if(data.player.y < data.enemy1[i].y){//e o player estiver abaixo
                shoot(&data,&data.enemy1[i],&data.projectile[data.shot_counter],0,-1);
                data.enemy1[i].action.timer_start = clock();
            }
        }
        if(data.player.y == data.enemy1[i].y){//se estiver alinhado em y
            if(data.player.x > data.enemy1[i].x){//e o player estiver a direita
                shoot(&data,&data.enemy1[i],&data.projectile[data.shot_counter],1,0);
                data.enemy1[i].action.timer_start = clock();
            }
            if(data.player.x < data.enemy1[i].x){//e o player estiver a esquerda
                shoot(&data,&data.enemy1[i],&data.projectile[data.shot_counter],-1,0);
                data.enemy1[i].action.timer_start = clock();
            }
        }
    }
}
```

Caso o inimigo seja um **CREEPER** ele se movimenta como o zumbi tentando se aproximar do jogador e quebrando paredes para chegar perto, caso ele esteja na mesma posição que o jogador ele “morre”, se o inimigo estiver morto, uma explosão é gerada “**explode()**”em sua posição anterior.

```
for(i=0; i<MAX_ENEMIES2; i++)//percorre o vetor de inimigos
{
    past_x = data.enemy2[i].x;
    past_y = data.enemy2[i].y;

    data.enemy2[i].step.timer_current = clock();
    if (data.enemy2[i].step.timer_current - data.enemy2[i].step.timer_start >= data.enemy2[i].step.cooldown)//se o cooldown for zero, pega a intenção de movimento
    {
        if (data.player.x > data.enemy2[i].x)
            data.enemy2[i].dx = 1;
        if (data.player.x < data.enemy2[i].x)
            data.enemy2[i].dx = -1;
        if (data.player.y > data.enemy2[i].y)
            data.enemy2[i].dy = 1;
        if (data.player.y < data.enemy2[i].y)
            data.enemy2[i].dy = -1;
        data.enemy2[i].step.timer_start = clock();

        if(test_move(&data,&data.enemy2[i]))//testa se ele pode mover
        {
            move(&data,&data.enemy2[i]);//move
        }
        else if((data.enemy2[i].x>0 && data.enemy2[i].x<MAP_COLUMNS && data.enemy2[i].y>0 && data.enemy2[i].y<MAP_LINES))
        {
            data.enemy2[i].action.timer_current = clock();
            if(test_if_object_is_there (&data,&data.enemy2[i],data.enemy2[i].dx,data.enemy2[i].dy,'J')){
                data.enemy2[i].health_points = 0;
            }else if(test_if_object_is_there (&data,&data.enemy2[i],data.enemy2[i].dx,data.enemy2[i].dy,'#')&&data.enemy2[i].action.timer_current - data.enemy2[i].action.cooldown >= 0){
                explode(&data,&data.enemy2[i].x,&data.enemy2[i].y);
            }
        }
    }
}
```

A função “**explode()**” acima citada é responsável por gerar a explosão, ela recebe as posições de atuação “x e y” e um raio, a função faz com que a área ao redor da explosão cause um dano conforme a proximidade da origem. Em seguida ela entra em um loop que cria feixes de explosão e calcula a partir de funções trigonométricas, (sendo que a hipotenusa é o raio da explosão) a posição que será afetada pela explosão arredondando os valores de cateto adjacente e cateto oposto para posições inteiras onde causará dano através da “**damaged_position()**”, vale ressaltar que ele não da dano 2 vezes em uma mesma posição, mas quando mais longe da origem da explosão menor será o dano infligido. Se a explosão atingir paredes ‘#’, o loop do raio é interrompido prematuramente para

evitar que o dano atravessasse. Para cada posição danificada, ela cria o efeito de fumaça no jogo, definindo as coordenadas, tempo de vida e tipo da fumaça.

```
bool explode(struct STRUCT_DATA *data, int x_origin, int y_origin, float radius, int beam_n, int damage)
{
    int i, k, x, y;
    float j;
    float radians, beam_interval = beam_n/(2*PI);
    int flag, damaged_position[MAP_LINES][MAP_COLUMNS] = {0};

    for(i=0; i<beam_n; i++)
    {
        flag = 1;
        radians = beam_interval*i;
        for(j=0; j<=radius&&flag==1; j=j+1){
            x = (int)round(cos(radians)*j);
            y = (int)round(sin(radians)*j);
            if((y_origin+y)>=0&&(y_origin+y)<MAP_LINES&&(x_origin+x)>=0&&(x_origin+x)<MAP_COLUMNS){
                if(damaged_position[y_origin+y][x_origin+x]==0){
                    damage_position(data, x_origin+x, y_origin+y, (int)round(damage -(j/radius)*damage));
                    damaged_position[y_origin+y][x_origin+x]=1;
                }
                if(data->map[y_origin+y][x_origin+x]=='#'){
                    flag = 0;
                }
            }else{
                flag = 0;
            }
        }
    }
}
```

O seguinte trecho de código lida com o comportamento dos inimigos do tipo **SLIME**. Primeiro, ele verifica se o inimigo está morto usando a função “**test_if_dead()**”. O seu movimento é do tipo aleatório, por isso ele começa com uma intenção de movimento aleatória e ao se chocar com algum outro objeto ele muda essa intenção novamente. Se ele puder se mover “**test_move()**”, a função “**move()**” é chamada para efetuar o movimento, caso o objeto que ele se chocou for um jogador ele causa dano ao mesmo.

```
for(i=0; i<MAX_ENEMIES3; i++)//percorre o vetor de inimigos
{
    test_if_dead(&data, &data.enemy3[i]);
    data.enemy3[i].step.timer_current = clock();
    if (data.enemy3[i].step.timer_current - data.enemy3[i].step.timer_start >= data.enemy3[i].step.cooldown)//se o cooldown for zero, pega a intenção de movimento
    {
        data.enemy3[i].step.timer_start = clock();

        if(test_move(&data, &data.enemy3[i]))//testa se ele pode mover
        {
            move(&data, &data.enemy3[i]); //move
        }
        else
        {
            if(test_if_object_is_there (&data, &data.enemy3[i], data.enemy3[i].dx, data.enemy3[i].dy, 'J')){
                damage_position(&data, data.enemy3[i].x+data.enemy3[i].dx, data.enemy3[i].y+data.enemy3[i].dy, 1);
            }
            do{
                data.enemy3[i].dx = (-1 + (rand()%3));
                data.enemy3[i].dy = (-1 + (rand()%3));
            }while(!test_move(&data, &data.enemy3[i])&&data.enemy3[i].dx==0||data.enemy3[i].dy==0);
        }
    }
}
```

Já para as **ARMADILHAS** verifica se um jogador 'J' está presente na posição da armadilha usando a função “**test_if_object_is_there()**”. Se um jogador estiver presente, a armadilha inicia um temporizador, então verifica se o tempo de cooldown para a ação da armadilha “**data.trap[i].action.cooldown**” foi atingido. Se o cooldown permitir, a armadilha causa dano à posição em que está localizada usando a função “**damage_position()**”. Este dano provavelmente afeta o jogador que estava na posição da armadilha.

```
for(i=0; i<MAX_TRAPS; i++)//percorre o vetor de inimigos
{
    if(test_if_object_is_there(&data,&data.trap[i],0,0,'J')){
        data.trap[i].action.timer_current = clock();
        if (data.trap[i].action.timer_current - data.trap[i].action.timer_start >= data.trap[i].action.cooldown)//se o cooldown for zero, pega a intenção de movimento
        {
            damage_position(&data,data.trap[i].x ,data.trap[i].y , 1);
            data.trap[i].action.timer_start = clock();
        }
    }
}
```

Também foi implementado paredes destrutíveis e para isso atribuímos vidas para sua estruturas e para que ela quebrasse foi feita a chamada da função “**test_if_dead()**” que verifica essa condição e também remove a parede caso ela tenha sido quebrada.

```
for(i=0; i<MAX_WALLS; i++)//percorre o vetor de inimigos
{
    test_if_dead(&data,&data.wall[i]);
}
```

Para executar a ação de explosão das bombas tivemos que criar variáveis que armazenam a posição dela. Verifica-se se a bomba está dentro dos limites do mapa (MAP_COLUMNS e MAP_LINES), feito para garantir que a bomba esteja em uma posição válida no jogo. Certifica a condição de vida pela “**test_if_dead()**” e caso ela esteja “morta”, ela executa a função “**explode()**”, mesma que o inimigo CREEPER usa.

```
for(i=0; i<MAX_BOMBS; i++)
{
    past_x = data.bomb[i].x;
    past_y = data.bomb[i].y;
    if(data.bomb[i].x>=0&&data.bomb[i].x<MAP_COLUMNS&&data.bomb[i].y>=0&&data.bomb[i].y<MAP_LINES){
        if(test_if_dead(&data,&data.bomb[i])){
            explode(&data,past_x,past_y,BOMB_RADIUS,180,BOMB_DAMAGE);
        }
    }
}
```

Por último, ainda no loop principal, a execução do ambiente de desenho é feita através do “**BeginDrawing()**”, função especial da raylib. Existe uma verificação de condição if “**menu_open**” que controla o que será desenhado. Se o menu estiver aberto, ele desenha o menu principal do jogo, e o jogador pode navegar pelo menu usando as teclas direcionais e a tecla "Enter" para selecionar opções.

```

BeginDrawing();//inicializa ambiente para desenho
if(menu_open){
    draw_menu(option,textures, font);
    if (option>=1&&option<=5){
        if (IsKeyPressed(KEY_DOWN)&&option<5&&option>=1) option++;
        if (IsKeyPressed(KEY_UP)&&option>1&&option<=5) option--;

        if ((IsKeyPressed(KEY_ENTER)&&option==1)||IsKeyPressed(KEY_N)){
            clear_all_data(&data);
            data.player.health_points = PLAYER_HP;
            update_arc_name(map_file_name,0);
            load_map(&data,map_file_name);//FUNCAO DE CARREGAMENTO DO MAPA
            scan_map(&data);//FUNCAO DE SCANEAMENTO DE MAPA
            menu_open = false;
        }
        if ((IsKeyPressed(KEY_ENTER)&&option==2)||IsKeyPressed(KEY_L)){
            option = option+6;
            intends_to_save_or_load = -1;
        }
        if ((IsKeyPressed(KEY_ENTER)&&option==3)||IsKeyPressed(KEY_S)){
            option = option+6;
            intends_to_save_or_load = 1;
        }
        if ((IsKeyPressed(KEY_ENTER)&&option==4)||IsKeyPressed(KEY_Q)) CloseWindow();
        if ((IsKeyPressed(KEY_ENTER)&&option==5)) menu_open = false;
    }else{
        if (IsKeyPressed(KEY_DOWN)&&option<11&&option>=7) option++;
        if (IsKeyPressed(KEY_UP)&&option>7&&option<=11) option--;
    }
}

```

Dentro do menu, dependendo da opção selecionada “(option)”, várias ações podem ocorrer se o jogador pressionar "Enter" na primeira opção do menu "New Run", limpa todos os dados do jogo, restaura a saúde do jogador, carrega um novo mapa e sai do menu.

Se o jogador pressionar "Enter" em outras opções, ele pode estar selecionando opções de **salvar** “salva o exato estado atual do jogo em 5 possíveis espaços de memória ex: save1.bin” ou **carregar o jogo** “carrega os saves gerados”.

Se o jogador pressionar "Enter" na última opção do menu, o jogo é fechado usando “CloseWindow()”.

Se o jogador não estiver no menu, ele desenha o mapa do jogo, os efeitos visuais e verifica se o jogador perdeu todas as vidas. Se o jogador perder todas as vidas, uma mensagem de "Você Morreu" é exibida na tela.

O jogo é atualizado no loop, e a função “EndDrawing()” é chamada para finalizar o desenho que continua até que o jogador escolha sair do jogo ou até que outra condição de saída seja atendida. E por fim, o jogo fecha o áudio e a janela usando “CloseAudioDevice()” e “CloseWindow()”, respectivamente, antes de retornar 0 para encerrar o programa.

```

        if ((IsKeyPressed(KEY_ENTER)&&option==10)||IsKeyPressed(52)){
            if(intends_to_save_or_load==1) save_data(data,4);
            if(intends_to_save_or_load==1) load_data(&data,4);
            option = option-6;
            menu_open = false;
            intends_to_save_or_load = 0;
        }
        if ((IsKeyPressed(KEY_ENTER)&&option==11)||IsKeyPressed(53)){
            if(intends_to_save_or_load==1) save_data(data,5);
            if(intends_to_save_or_load==1) load_data(&data,5);
            option = option-6;
            menu_open = false;
            intends_to_save_or_load = 0;
        }
    }
}
}else{
    draw_map(&data,textures, font);
    draw_effects(&data);
    if(data.player.health_points==0){
        DrawRectangle(0,0,MAP_PIXEL_WIDTH,MAP_PIXEL_HEIGHT,(Color){0,0,0,190});//limpa a tela e define cor de fundo
        DrawTextEx(font,"You Died", (Vector2){(MAP_PIXEL_WIDTH/2)-200, (MAP_PIXEL_HEIGHT/2)-50}, 100,0, WHITE);
        DrawTextEx(font,"Press M to open Menu", (Vector2){(MAP_PIXEL_WIDTH/2)-270, (MAP_PIXEL_HEIGHT/2)+80}, 50,0, WHITE);
    }
}
draw_hud(&data,textures, font);
EndDrawing();
}
CloseAudioDevice();
CloseWindow();// Fecha a janela e o contexto OpenGL
return 0;

```

FUNCIONAMENTO PRÁTICO

Estamos satisfeitos em relatar que todas as nossas expectativas foram atendidas e até mesmo superadas durante essa fase de desenvolvimento. As mecânicas de jogo, os elementos visuais e sonoros, bem como a experiência geral do jogador, funcionaram conforme o planejado.

A parte gráfica do jogo foi uma experiência gratificante, inspirada pelo estilo visual único de Minecraft. Escolhemos essa abordagem para criar uma sensação de nostalgia e familiaridade, ao mesmo tempo em que introduzimos novas mecânicas de jogo.



Ele inicia pausado na tela de menu onde é possível iniciar um jogo novo, carregar um arquivo salvo, salvar o jogo atual, sair do jogo e fechar este menu.



É possível selecionar os saves que desejamos carregar ou que desejamos salvar, com um total de 5 espaços de armazenamento.



Como explicado nos comentários de código o inimigo padrão é o zumbi, ele possui essa textura característica e sua mecânica é perseguir o jogador, já o jogador pode utilizar das teclas de comando para desviar e montar a sua estratégia para matar o inimigo.



Os mapas ficaram bastante diversos com uma variedade grande de arquitetura de fases e uma dificuldade crescente

Projetamos que cada classe de inimigo apresentaria um nível de dificuldade distinto. No início do jogo, nossa intenção era que a experiência não fosse excessivamente desafiadora, portanto,

optamos por introduzir os inimigos do tipo "slime" como os primeiros adversários. Embora esses inimigos possam ser considerados chatos de enfrentar, são os mais simples de derrotar, o que os torna uma escolha adequada para iniciar o jogo.

Além disso, buscamos aprimorar a experiência do jogador ao longo do jogo, introduzindo elementos que facilitariam sua jornada. Com isso em mente, equilibramos cuidadosamente os desafios e os benefícios, procurando proporcionar a melhor experiência possível.

FASE	MAÇÃ	TNT	DOURADA	SLIME	ZUMBI	ESQUELETO	CREEPER
1	2	2			2		
2	1	1		2	2		
3	1	3		10	1		
4		3		15			
5		3	1	2		2	1
6		2		2		3	2
7			1	2		4	2
8		2		4		4	2
9			1	2	1	4	6
10	3	3	2	2	3	4	6

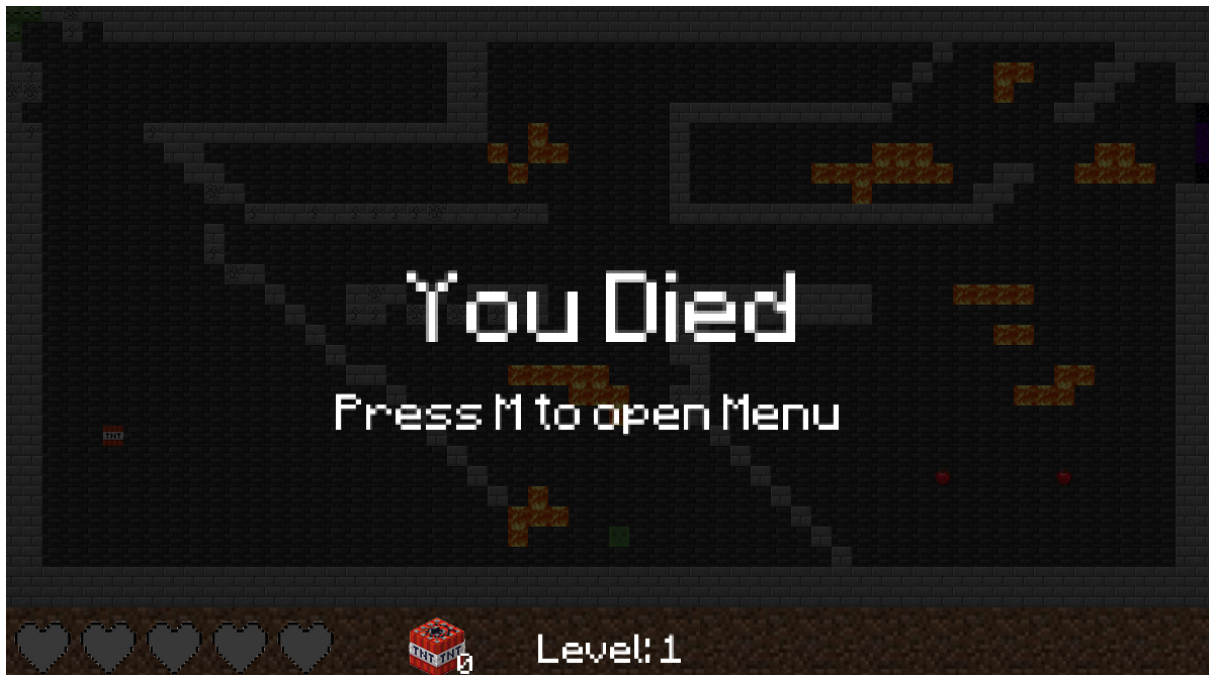
Quando o jogador sofre dano ou coleta itens, implementamos um sistema de HUD (Heads-Up Display) que fornece informações claras sobre o estado atual do personagem e seu desempenho. Esse HUD é projetado de forma a ser intuitivo e fácil de entender, permitindo que o jogador acompanhe seu progresso de forma eficaz.

Por meio do HUD, o jogador pode visualizar instantaneamente informações cruciais, como a quantidade de vida restante, e as bombas coletadas. Além disso, incorporamos um timer que exibe o tempo decorrido, permitindo ao jogador avaliar seu desempenho atual e definir metas para melhorar seu tempo.

Dessa forma, nosso objetivo é garantir que o jogador tenha acesso imediato a todas as informações relevantes, proporcionando uma experiência de jogo mais envolvente e acessível.



Quando o jogador perde no jogo, uma mensagem "Você Morreu" é exibida na tela, e ele pode retornar ao menu principal pressionando a tecla 'M' para começar uma nova partida. Isso proporciona uma transição suave e orientada após a derrota.



CONCLUSÕES

A legibilidade do código foi uma prioridade desde o início, com nomes descritivos para variáveis e funções e comentários claros. Isso tornou o trabalho e a manutenção e o demais acessíveis para a equipe. A lógica dos inimigos foi projetada com inteligência e desafio em mente. Eles respondem ao jogador de forma eficaz, tornando o combate no jogo envolvente.

Em resumo, o jogo demonstra uma implementação complexa de mecânicas de jogo, desde a gestão de personagens e inimigos até o controle de explosões e interações com o jogador. Com uma variedade de elementos como menus, mapas e efeitos visuais, ele oferece uma experiência de jogo completa. Além disso, a capacidade de salvar e carregar progresso adiciona uma camada adicional de profundidade. Em última análise, este jogo destaca a aplicação prática de conceitos de programação em um ambiente de entretenimento, ilustrando a versatilidade e criatividade que a programação pode proporcionar na criação de experiências interativas para os jogadores.

REFERÊNCIAS

RAYLIB. Cheatsheet da raylib. Disponível em:
<https://www.raylib.com/cheatsheet/cheatsheet.html>. Acesso em: 10 de setembro de 2023.

MINECRAFT WIKI. Minecraft Wiki. Disponível em:
https://minecraft.fandom.com/pt/wiki/Minecraft_Wiki. Acesso em: 10 de setembro de 2023.

THE BINDING OF ISAAC WIKI. **The Binding of Isaac Wiki**. Disponível em:
https://bindingofisaac.fandom.com/wiki/The_Binding_of_Isaac_Wiki. Acesso em: 10 de setembro de 2023.

TUTORIALSPPOINT. **Página sobre a biblioteca time.h**. Disponível em:
https://www.tutorialspoint.com/c_standard_library/time_h.htm. Acesso em: 10 de setembro de 2023.

SICKYAY. **Repositório do GitHub: ufrgs_game**. Disponível em:
https://github.com/sickyay/ufrgs_game. Acesso em: 10 de setembro de 2023.