ESTGOH
Escola Superior de Tecnologia e Gestão de Oliveira do Hospital
Instituto Politécnico de Coimbra

Licenciatura em Engenharia Informática

Multimédia e computação gráfica

Trabalho Prático Pygame

MAZECRAFT

Elaborado em: 02/12/2024

Mariana Magalhães a2022147454
Francisco Figueiras a2021155919

Índice

1. Introdução

O objetivo deste trabalho é criar um jogo em Pygame, optámos por um labirinto cujo tema é Minecraft. O objetivo do jogo, como o próprio nome indica, é encontrar a saída do labirinto, mas antes de chegar à saída é preciso encontrar a chave.

Todo o ambiente do jogo decorre numa dimensão específica do Minecraft, neste caso, a dimensão "The End", uma dimensão escura, semelhante ao espaço, composta por ilhas separadas no vazio, feitas de end stone e com pilares de obsidian. É habitada por endermen, shulkers e pelo ender dragon. Por ser um ambiente escuro, considerámos um bom tema para um labirinto, e por isso as paredes do labirinto são de obsidian, e a zona onde o jogador se move são blocos de end stone.

Desviando um pouco do tema original do Minecraft, onde, como jogadores, temos de matar o ender dragon para podermos sair, neste labirinto somos o próprio ender dragon, mais concretamente a sua cabeça, e a chave é o seu ovo (dragon egg). Portanto, o objetivo é o ender dragon apanhar o seu próprio ovo para poder finalmente sair do labirinto.

Para dificultar o jogo, optámos por aplicar um shader para dar um filtro de TV antiga e também implementámos uma câmara para seguir o jogador, mostrando apenas uma parte do que está à volta dele - ou seja, não será revelado o mapa completo. Adicionalmente, aplicámos um efeito de nevoeiro (fog) para escurecer o ambiente.

2. Desenvolvimento

O projeto não começou da melhor forma, uma vez que um dos membros não teve tempo no início devido a alguns problemas noutra unidade curricular. Após a resolução desses problemas e ao começar a implementar as funcionalidades em falta, deparámo-nos com uma situação preocupante: a estrutura do código.

O código não estava bem organizado e era difícil compreender como as funcionalidades foram desenvolvidas. Chegámos inclusivamente a classificá-lo como 'código martelado', devido à sua complexidade e falta de clareza. Esta situação dificultava, nomeadamente, a implementação de funcionalidades como o shader.

Por estas razões, decidimos refazer o código, desta vez com o envolvimento do elemento que tinha feito menos até ao momento. Esta decisão foi tomada porque ainda tínhamos tempo disponível e poderíamos reutilizar a base de código existente, melhorando a sua estrutura e legibilidade.

Sendo assim o projeto está organizado da seguinte maneira:

Figure 1 - organização do código.

2.1. Game.py

A classe Game representa o núcleo principal do jogo desenvolvido em Pygame. Na sua inicialização, o código realiza várias configurações essenciais:

1. **Inicialização de Componentes:**
   - Inicializa o Pygame e o mixer de som
   - Configura o ecrã em modo de ecrã inteiro e dimensionável
   - Cria uma superfície de jogo com resolução lógica de 1280x720
   - Esconde o cursor do rato
2. **Gestão de Recursos:**
   - **load_assets():**
     - Carrega ficheiros de recursos (sprites, fontes, música):
       - Define caminhos para diferentes tipos de ficheiros (e.g., sprites_dir para imagens, font_dir para fontes).
       - Carrega imagens, redimensiona fundos e define transparências.
     - Adiciona várias faixas de música para reprodução.
   - **load_state():**
     - Adiciona o estado inicial (StartScreen) à pilha de estados (state_stack).
   - **adjust_volume():**
     - Permite ajustar o volume da música (valores entre 0 e 1).
3. **Funcionalidades Principais:**
   - **Método run():** Loop principal do jogo
     - Captura de eventos do utilizador (get_events).
     - Atualização lógica (update).
     - Renderização gráfica (render).
   - **get_events():** Processa eventos de teclado
     - Garante que eventos do teclado e do rato são processados.
     - Mapeia teclas específicas para ações de jogador (e.g., W para "UP", Escape para "PAUSE").
   - **update():** Atualiza o estado atual do jogo
     - Atualiza o estado atual do jogo (o estado mais recente no state_stack).
   - **render():**
     - Renderiza o estado atual na superfície lógica (game_canvas).
     - Aplica o shader (efeito visual).
     - Escala e desenha a superfície no ecrã físico.
   - **get_dt()**:
     - Calcula o tempo decorrido desde o último frame, essencial para animações e lógica.
   - **change_music():** Alterna entre faixas musicais
     - Altera a faixa de música atual e reinicia a reprodução.
   - **draw_text():**
     - Desenha texto na superfície do jogo com fonte carregada nos ativos.
4. **Gestão de Estados**
   - A pilha de estados (state_stack) permite:
     - Alternar entre diferentes telas (e.g., início, pausa, jogo principal).
     - Manter o controlo sobre o estado atual.

```python
import pygame, os, time
from shaders.crt_shader import Shader
from states.start_screen import StartScreen
from pygame import mixer

class Game:
    def __init__(self):
        pygame.init()
        mixer.init()
        info = pygame.display.Info()
        self.GAME_LOGIC_SIZE, self.SCREEN_SIZE = (1280, 720), (info.current_w, info.current_h)
        self.NATIVE_SCREEN_SIZE = self.SCREEN_SIZE
        self.game_canvas = pygame.Surface(self.GAME_LOGIC_SIZE).convert((255, 65280, 16711680, 0))
        self.screen = pygame.display.set_mode(self.SCREEN_SIZE, pygame.FULLSCREEN | pygame.SCALED
| pygame.HWSURFACE | pygame.DOUBLEBUF | pygame.OPENGL)
        self.running, self.playing = True, True
        self.shader = Shader(self)
        self.player_actions = {'UP': False, 'DOWN': False, 'LEFT': False, 'RIGHT': False,
'SELECT': False, 'PAUSE': False}
        self.colors = {'WHITE': (255, 255, 255), 'BLACK': (0, 0, 0), 'GRAY': (200, 200, 200),
'BLUE': (0, 0, 255), 'GREEN': (0, 255, 0), 'RED': (255, 0, 0), 'YELLOW': (255, 255, 0)}
        self.dt, self.prev_time = 0, 0
        self.state_stack = []
        self.load_assets()
        self.load_state()
        pygame.mouse.set_visible(False)
        self.change_music()
    def run(self):
        while self.playing:
            self.get_dt()
            self.get_events()
            self.update()
            self.render()
    def get_events(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.handle_quit_event()
            if event.type == pygame.KEYDOWN:
                self.handle_key_event(event, True)
            if event.type == pygame.KEYUP:
                self.handle_key_event(event, False)

    def handle_quit_event(self):
        self.running, self.playing = False, False
```

```
    def handle_key_event(self, event, is_key_down):
        key_map = {
            pygame.K_w: 'UP',
            pygame.K_s: 'DOWN',
            pygame.K_a: 'LEFT',
            pygame.K_d: 'RIGHT',
            pygame.K_RETURN: 'SELECT',
            pygame.K_ESCAPE: 'PAUSE',
            pygame.K_UP: 'UP',
            pygame.K_DOWN: 'DOWN',
            pygame.K_LEFT: 'LEFT',
            pygame.K_RIGHT: 'RIGHT',
        }
        if event.key in key_map:
            self.player_actions[key_map[event.key]] = is_key_down
    def update(self):
        self.state_stack[-1].update(self.dt, self.player_actions)
    def render(self):
        self.state_stack[-1].render(self.dt, self.game_canvas)
        self.shader.render()
        self.screen.blit(pygame.transform.scale(self.game_canvas, self.SCREEN_SIZE), (0, 0))
        pygame.display.flip()
    def get_dt(self):
        curr_time = time.time()
        self.dt = curr_time - self.prev_time
        self.prev_time = curr_time
    def draw_text(self, surface, text, color, xy):
        text_surface = self.font.render(text, True, color)
        text_rect = text_surface.get_rect()
        text_rect.center = xy
        surface.blit(text_surface, text_rect)
    def load_state(self):
        self.state_stack.append(StartScreen(self))

    def change_music(self):
        self.current_track_index = (self.current_track_index + 1) % len(self.music_tracks)
        current_track = os.path.basename(self.music_tracks[self.current_track_index])
        mixer.music.load(self.music_tracks[self.current_track_index])
        mixer.music.play(-1)
        return current_track

    def adjust_volume(self, volume):
        # Volume should be between 0.0 and 1.0
        mixer.music.set_volume(max(0.0, min(1.0, volume)))

    def reset_player_actions(self):
        for action in self.player_actions:
            self.player_actions[action] = False
```

```
def load_assets(self):
        self.assets_dir = os.path.join('assets')
        self.sprites_dir = os.path.join(self.assets_dir, 'sprites')
        self.font_dir = os.path.join(self.assets_dir, 'fonts')
        self.effects_dir = os.path.join(self.assets_dir, 'effects')
        self.background = pygame.image.load('assets/backgrounds/end_portal.jpg')
        self.background = pygame.transform.scale(self.background, self.SCREEN_SIZE)
        self.wall = pygame.image.load(os.path.join(self.sprites_dir, 'obsidian.jpg'))
        self.path = pygame.image.load(os.path.join(self.sprites_dir, 'end_stone.png'))
        self.player = pygame.image.load(os.path.join(self.sprites_dir, 'Dragon_Head_29.jpg'))
        self.key= pygame.image.load(os.path.join(self.sprites_dir, 'Dragon_Egg.jpg'))
        self.font = pygame.font.Font(os.path.join(self.font_dir, 'Minecrafter.Reg.ttf'), 20)
        self.player = pygame.image.load(os.path.join(self.sprites_dir, 'Dragon_Head_29.jpg'))
        self.exit = pygame.image.load('assets/backgrounds/end_portal.jpg')
        self.exit.set_alpha(200)
        self.music_tracks = [
            os.path.join(self.effects_dir, 'ko0x - Galaxy Guppy [Chiptune].mp3'),# Add more
music track paths here
            os.path.join(self.effects_dir, 'Into the Maze.mp3'),
            os.path.join(self.effects_dir, 'Kubbi - Ember [Chiptune].mp3'),
            os.path.join(self.effects_dir, 'Daniel Fridell, Sven Lindvall - Trail to
Dolores.mp3'),
            os.path.join(self.effects_dir, 'NEON DRIVE by Ghostrifter.mp3'),
            os.path.join(self.effects_dir, 'Ava Low - Through the Prism.mp3'),
            os.path.join(self.effects_dir, 'd4vd - Remember Me.mp3'),
            os.path.join(self.effects_dir, 'Royal & the Serpent - Wasteland.mp3'),
            os.path.join(self.effects_dir, 'Stromae, Pomme - Ma Meilleure Ennemie English.mp3'),
            os.path.join(self.effects_dir, 'To Ashes and Blood.mp3'),
        ]
        self.current_track_index = 0
if __name__ == "__main__":
    game = Game()
    game.run()
```

2.2. game_world.py

A class GameWorld controla a interação entre o jogador, o labirinto, e a câmara, enquanto gere transições para outros estados, como pausa e avanço de nível.

1. **Inicialização:**
   - Herda da classe base State
   - Define tamanho da célula do labirinto (50 pixels)
   - Cria instâncias de:
     - Labirinto (Maze)
     - Jogador (Player)
     - Câmara (Camera)
2. **Método update():**
   - Atualiza o jogador, câmara e verifica eventos
   - Condições de mudança de estado:
     - Tecla de pausa: abre menu de pausa
     - Jogador coleta chave e chega à saída: avança para próximo nível
   - Incrementa dificuldade (aumentando nível do labirinto)
   - Regenera labirinto
   - Reinicia ações do jogador
3. **Método render:**
   - Preenche superfície com cor preta
   - Renderiza labirinto com:
     - Verificação de visibilidade pela câmara
     - Renderização limitada ao campo de visão
   - Aplica efeito de névoa (fog) através da câmara
4. **Integração dos Componentes:**
   - Labirinto (Maze):
     - Estrutura principal do nível. Pode ser gerado dinamicamente para níveis mais avançados.
   - Jogador (Player):
     - Interage com o labirinto e verifica colisões com a saída.
   - Câmara (Camera):
     - Controla a área visível do labirinto e aplica efeitos visuais como o nevoeiro.
5. **Notas Adicionais**
   1. Transições de Estados:
      - Utiliza um sistema de pilha (herdado da classe State) para transitar entre estados:
        - PauseScreen: Pausa o jogo.
        - Nextlevelscreen: Mostra o ecrã de transição antes do próximo nível.
   2. Escalabilidade:
      - O uso de níveis dinâmicos (gerados pelo método generate_maze) torna o jogo expansível.
   3. Efeitos Visuais:
      - A câmara melhora a imersão ao limitar o campo de visão do jogador.

```
from states.state import State
from states.pause_screen import PauseScreen
from states.nextlevel_screen import Nextlevelscreen
from components.maze import Maze
from components.player import Player
from components.camera import Camera


class GameWorld(State):
    def __init__(self, game):
        State.__init__(self, game)
        self.cell_size = 50
        self.level = 1
        self.maze = Maze(self.level)
        self.maze.cell_size = self.cell_size
        self.player = Player(self.maze, self.game)
        self.camera = Camera(self.maze)


    def update(self, dt, player_actions):
        self.player.update(player_actions, self.maze)
        self.camera.update(self.player)
        if player_actions['PAUSE']:
            new_state = PauseScreen(self.game)
            new_state.enter_state()
        if self.maze.key.collected and
self.player.collision.check_exit_collision(self.player.x, self.player.y):
            new_state = Nextlevelscreen(self.game)
            self.level += 2
            self.maze.generate_maze(self.level)
            self.player.update_maze(self.maze)
            self.camera.update_maze(self.maze)
            new_state.enter_state()
        self.game.reset_player_actions()

    def render(self, dt, surface):
        surface.fill(self.game.colors['BLACK'])
        self.maze.render(surface, self.cell_size, self.game,
                         visibility_check=self.camera.is_in_view_range,
                         player= self.player,
                         camera=self.camera)
        self.player.render(surface, self.cell_size, self.game, camera=self.camera)
        self.camera.apply_fog(surface, self.player)
```

2.3. nextlevel_screen.py

1. **Inicialização:**
    - Herda da classe base State
    - Define dois opções de menu: "Next level" e "Exit game"
    - Inicializa cursor de menu na primeira opção
2. **Método update:**
    - Atualiza posição do cursor conforme ações do jogador
    - Ao selecionar (tecla Enter), executa ação correspondente
    - Reinicia ações do jogador
3. **Método handle_selected_option:**
    - "Next level": sai do estado atual (retorna ao jogo)
    - "Exit game": sai de dois estados (sai do jogo)
4. **Método render:**
    - Renderiza fundo do jogo
    - Mostra texto "Victory"
    - Desenha opções de menu:
        - Destaca opção selecionada em branco
        - Outras opções em cinza
    - Centraliza texto na tela
5. **Funcionamento Geral:**
    1. Interatividade:
        - O jogador usa as setas do teclado (ou controlos equivalentes) para mover o cursor entre as opções do menu.
        - Pressiona o botão de seleção (SELECT) para confirmar a escolha.
    2. Transições de Estado:
        - Escolher "Next level" sai do estado Nextlevelscreen e avança para o próximo nível.
        - Escolher "Exit game" termina o jogo, saindo de todos os estados.

```
from states.state import State
class Nextlevelscreen(State):
    def __init__(self, game):
        State.__init__(self, game)
        self.menu_options = {0: 'Next level', 1: 'Exit game'}
        self.menu_cursor = 0
    def update(self, dt, player_actions):
        self.update_cursor(player_actions)
        if player_actions['SELECT']:
            selected_option = self.menu_options[self.menu_cursor]
            self.handle_selected_option(selected_option)
        self.game.reset_player_actions()
    def handle_selected_option(self, option):
        if option == 'Next level':
            self.exit_state()
        if option == 'Exit game':
            self.exit_state()
            self.exit_state()
    def render(self, dt, surface):
        surface.blit(self.game.background, (0, 0))
        self.game.draw_text(surface, 'Victory', self.game.colors['WHITE'],
(self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 4))
        self.game.draw_text(surface, 'Next level', self.game.colors['GRAY'] if self.menu_cursor
!= 0 else self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2,
self.game.GAME_LOGIC_SIZE[1] / 3 + 40))
        self.game.draw_text(surface, 'Exit game', self.game.colors['GRAY'] if self.menu_cursor !=
1 else self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1]
/ 3 + 80))
```

2.4. pause_screen.py

A classe PauseScreen implementa o estado **PauseScreen**, que corresponde ao menu de pausa do jogo. Este estado permite ao jogador realizar ações como continuar o jogo, ajustar configurações ou sair. Abaixo segue uma explicação detalhada das funcionalidades implementadas.

1. **Inicialização:**
    - Herda da classe base State
    - Define 4 opções de menu:
        - Exit to menu
        - Change resolution
        - Configure Sound
        - Exit game
    - Adiciona funcionalidade de piscar texto
2. **Método update:**
    - Permite navegação no menu
    - Sai do menu de pausa ao pressionar tecla de pausa
    - Executa ação selecionada ao pressionar Enter
    - Reinicia ações do jogador
3. **Método handle_selected_option:**
    - "Exit to menu": sai de dois estados (retorna ao menu inicial)
    - "Change resolution": abre tela de resolução
    - "Configure Sound": abre configurações de som
    - "Exit game": encerra o jogo
4. **Método render:**
    - Renderiza fundo do jogo
    - Mostra texto "Game paused"
    - Texto "Press ESC to resume" pisca
    - Desenha opções de menu:
        - Destaca opção selecionada em branco
        - Outras opções em cinza
    - Centraliza texto na tela
5. **Funcionamento Geral:**

- Fluxo:
    - O jogador pode mover o cursor para navegar entre as opções e pressionar SELECT para confirmar.
    - Pressionar PAUSE fecha o menu e retorna ao jogo.
- Modularidade:
    - O estado utiliza outros estados modulares, como ResolutionScreen e ConfigureSound, promovendo uma arquitetura limpa e escalável.
- Interatividade:
    - O feedback visual no menu, como o efeito de texto intermitente e o destaque da opção selecionada, melhora a experiência do utilizador.

```
class PauseScreen(State):
    def __init__(self, game):
        State.__init__(self, game)
        self.menu_options = {0: 'Exit to menu', 1: 'Change resolution', 2: 'Configure Sound', 3:
'Exit game'}
        self.menu_cursor = 0
        self.is_text_hidden = False
        self.set_blink_timer()
    def update(self, dt, player_actions):
        self.update_cursor(player_actions)
        if player_actions['PAUSE']:
                self.exit_state()
        if player_actions['SELECT']:
            selected_option = self.menu_options[self.menu_cursor]
            self.handle_selected_option(selected_option)
        self.game.reset_player_actions()
    def handle_selected_option(self, option):
        if option == 'Exit to menu':
            self.exit_state()
            self.exit_state()
        if option == 'Change resolution':
            new_state = ResolutionScreen(self.game)
            new_state.enter_state()
        if option == 'Configure Sound':
            new_state = ConfigureSound(self.game)
            new_state.enter_state()
        if option == 'Exit game':
            self.game.running, self.game.playing = False, False
        self.game.reset_player_actions()
    def render(self, dt, surface):
        surface.blit(self.game.background, (0, 0))
        self.game.draw_text(surface, 'Game paused', self.game.colors['WHITE'],
(self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 4))
        self.blink_message(dt)
        if self.is_text_hidden:
            self.game.draw_text(surface, 'Press ESC to resume', self.game.colors['WHITE'],
(self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 3))
        self.game.draw_text(surface, 'Exit to menu', self.game.colors['GRAY'] if self.menu_cursor
!= 0 else self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2,
self.game.GAME_LOGIC_SIZE[1] / 3 + 40))
        self.game.draw_text(surface, 'Change resolution', self.game.colors['GRAY'] if
self.menu_cursor != 1 else self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2,
self.game.GAME_LOGIC_SIZE[1] / 3 + 60))
        self.game.draw_text(surface, 'Configure Sound', self.game.colors['GRAY'] if
self.menu_cursor != 2 else self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2,
self.game.GAME_LOGIC_SIZE[1] / 3 + 80))
        self.game.draw_text(surface, 'Exit game', self.game.colors['GRAY'] if self.menu_cursor != 3
else self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 3
+ 100))
```

2.5. resolution_screen.py

A classe ResolutionScreen implementa o estado **ResolutionScreen**, que permite ao jogador alterar a resolução do jogo ou alternar para o modo de tela cheia. Ele faz parte do sistema de gerenciamento de estados e proporciona uma interface interativa para configurações gráficas. Abaixo está a análise detalhada das funcionalidades implementadas.

1. **Inicialização:**
   - Herda da classe State
   - Define opções de resolução:
     - Fullscreen
     - 1920x1080
     - 1280x720
     - 1024x768
     - Voltar
2. **Método update:**
   - Navega pelas opções do menu
   - Executa ação selecionada
   - Reinicia ações do jogador
3. **Método handle_selected_option:**
   - Fullscreen: ativa modo tela cheia
   - Resoluções específicas: ajusta tamanho da janela
   - Opção "Voltar": sai da tela de resolução
4. **Método render:**
   - Renderiza fundo do jogo
   - Mostra título "Change resolution"
   - Desenha opções de menu:
     - Destaca opção selecionada em branco
     - Outras opções em cinza
5. **Métodos de manipulação de resolução:**
   - handle_fullscreen_option: Configura tela cheia
   - handle_resolution_option: Ajusta resolução da janela

```
import pygame
from states.state import State
class ResolutionScreen(State):
    def __init__(self, game):
        State.__init__(self, game)
        self.menu_options = {0: 'Fullscreen', 1: '1920x1080', 2: '1280x720', 3: '1024x768', 4: 'Back'}
        self.menu_cursor = 0
    def update(self, dt, player_actions):
        self.update_cursor(player_actions)
        if player_actions['SELECT']:
            selected_option = self.menu_options[self.menu_cursor]
            self.handle_selected_option(selected_option)
        self.game.reset_player_actions()
    def handle_selected_option(self, option):
        if option == 'Fullscreen':
            self.handle_fullscreen_option()
        if option == '1920x1080':
            self.handle_resolution_option(1920, 1080)
        if option == '1280x720':
            self.handle_resolution_option(1280, 720)
        if option == '1024x768':
            self.handle_resolution_option(1024, 768)
        if option == 'Back':
            self.exit_state()
        self.game.reset_player_actions()
    def render(self, dt, surface):
        surface.blit(self.game.background, (0, 0))
        self.game.draw_text(surface, 'Change resolution', self.game.colors['WHITE'],
(self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 4))
        self.game.draw_text(surface, 'Fullscreen', self.game.colors['GRAY'] if self.menu_cursor != 0 else
self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 3))
        self.game.draw_text(surface, '1920x1080', self.game.colors['GRAY'] if self.menu_cursor != 1 else
self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 3 + 20))
        self.game.draw_text(surface, '1280x720', self.game.colors['GRAY'] if self.menu_cursor != 2 else
self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 3 + 40))
        self.game.draw_text(surface, '1024x768', self.game.colors['GRAY'] if self.menu_cursor != 3 else
self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 3 + 60))
        self.game.draw_text(surface, 'Back', self.game.colors['GRAY'] if self.menu_cursor != 4 else
self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 3 + 80))
    def handle_fullscreen_option(self):
        self.game.SCREEN_SIZE = self.game.NATIVE_SCREEN_SIZE
        self.game.screen = pygame.display.set_mode(self.game.SCREEN_SIZE, pygame.FULLSCREEN | pygame.SCALED
| pygame.HWSURFACE | pygame.DOUBLEBUF | pygame.OPENGL)
    def handle_resolution_option(self, width, height):
        self.game.SCREEN_SIZE = (width, height)
        self.game.screen = pygame.display.set_mode(self.game.SCREEN_SIZE, pygame.HWSURFACE |
pygame.DOUBLEBUF | pygame.OPENGL)
```

2.6. sound_screen.py

A classe ConfigureSound implementa o estado **ConfigureSound**, que permite ao jogador ajustar o volume do jogo, alternar entre faixas musicais e voltar ao menu anterior. Este estado é essencial para personalizar a experiência auditiva do jogador. Abaixo está a análise detalhada.

1. **Inicialização:**
   - Herda da classe State
   - Opções de menu: Volume, Change Music, Back
   - Configura volume e música atual
2. **Método update:**
   - Navega pelas opções
   - Ajusta volume com teclas esquerda/direita
   - Troca música
   - Pausa/despausa música
   - Reinicia ações do jogador
3. **Métodos de música:**
   - change_music_track: Próxima/anterior música
   - get_previous_track: Reproduz música anterior
   - toggle_music_pause: Pausa/despausa música
4. **Método render:**
   - Mostra tela de configuração de som
   - Exibe volume atual
   - Mostra música corrente e estado (tocando/pausado)
   - Destaca opção selecionada

```
import pygame
import os
from states.state import State
class ConfigureSound(State):
    def __init__(self, game):
        State.__init__(self, game)
        self.menu_options = {0: 'Volume', 1: 'Change Music', 2: 'Back'}
        self.menu_cursor = 0
        self.volume = pygame.mixer.music.get_volume()  # Default volume at 50%
        self.is_music_paused = False
        self.current_music = os.path.basename(self.game.music_tracks[self.game.current_track_index])

    def update(self, dt, player_actions):
        self.update_cursor(player_actions)
        if self.menu_cursor == 0 and (player_actions['LEFT'] or player_actions['RIGHT']):
            self.volume = max(0.0, min(1.0, round((self.volume + (0.1 if player_actions['RIGHT'] else -
0.1)) * 10) / 10))
            self.game.adjust_volume(self.volume)
        elif self.menu_cursor == 1 and (player_actions['LEFT'] or player_actions['RIGHT']):
            direction = 1 if player_actions['RIGHT'] else -1
            self.change_music_track(direction)
        if player_actions['SELECT']:
            if self.menu_cursor == 1:
                self.toggle_music_pause()
            else:
                selected_option = self.menu_options[self.menu_cursor]
                self.handle_selected_option(selected_option)
        self.game.reset_player_actions()
```

```
    def change_music_track(self, direction):
        self.current_music = self.game.change_music() if direction > 0 else
self.get_previous_track()
    def get_previous_track(self):
        prev_index = (self.game.current_track_index - 1 + len(self.game.music_tracks)) %
len(self.game.music_tracks)
        self.game.current_track_index = prev_index
        current_track = os.path.basename(self.game.music_tracks[prev_index])
        pygame.mixer.music.load(self.game.music_tracks[prev_index])
        pygame.mixer.music.play(-1)
        return current_track
    def toggle_music_pause(self):
        if self.is_music_paused:
            pygame.mixer.music.unpause()
            self.is_music_paused = False
        else:
            pygame.mixer.music.pause()
            self.is_music_paused = True
    def handle_selected_option(self, option):
        if option == 'Change Music':
            self.current_music = self.game.change_music()
        if option == 'Back':
            self.exit_state()
        self.game.reset_player_actions()
    def render(self, dt, surface):
        surface.blit(self.game.background, (0, 0))
        self.game.draw_text(surface, 'Configure Sound', self.game.colors['WHITE'],
(self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 4))
        volume_text = f'Volume: {int(self.volume * 100)}%'
        self.game.draw_text(surface, volume_text, self.game.colors['WHITE'] if self.menu_cursor
== 0 else self.game.colors['GRAY'], (self.game.GAME_LOGIC_SIZE[0] / 2,
self.game.GAME_LOGIC_SIZE[1] / 3))
        music_status = 'Paused' if self.is_music_paused else 'Playing'
        current_music_text = f'Current Music: {self.current_music} ({music_status})'
        self.game.draw_text(surface, current_music_text, self.game.colors['WHITE'] if
self.menu_cursor == 1 else self.game.colors['GRAY'], (self.game.GAME_LOGIC_SIZE[0] / 2,
self.game.GAME_LOGIC_SIZE[1] / 3 + 20))
        self.game.draw_text(surface, 'Back', self.game.colors['WHITE'] if self.menu_cursor == 2
else self.game.colors['GRAY'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] /
3 + 40))
```

2.7. start_screen.py

Aclasse StartScreen serve como a tela inicial do jogo, exibindo o título e instruções para começar. Abaixo está a análise detalhada.

1. **Inicialização:**
   - Herda da classe State
   - Representa a tela inicial do jogo MAZECRAFT
   - Inicializa o estado do jogo
   - Define is_text_hidden como False
   - Configura um timer para piscar mensagens
2. **Método update:**
   - Verifica se o jogador pressionou a tecla 'SELECT'
   - Se acionado, muda para a tela de título (TitleScreen)
   - Reseta as ações do jogador
3. **Método render:**
   - Desenha o plano de fundo do jogo
   - Escreve "MAZECRAFT" no centro superior da tela
   - Chama blink_message para efeito visual
   - Mostra texto "Press Enter to start" quando is_text_hidden é verdadeiro

```
from states.state import State
from states.title_screen import TitleScreen
class StartScreen(State):
    def __init__(self, game):
        State.__init__(self, game)
        self.is_text_hidden = False
        self.set_blink_timer()
    def update(self, dt, player_actions):
        if player_actions['SELECT']:
            new_state = TitleScreen(self.game)
            new_state.enter_state()
        self.game.reset_player_actions()
    def render(self, dt, surface):
        surface.blit(self.game.background, (0, 0))
        self.game.draw_text(surface,'MAZECRAFT', self.game.colors['WHITE'],
(self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 3))
        self.blink_message(dt)
        if self.is_text_hidden:
            self.game.draw_text(surface, 'Press Enter to start',
self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2,
self.game.GAME_LOGIC_SIZE[1] / 2))
```

2.8. state.py

A classe State, fornece funcionalidades comuns para todos os estados no sistema de gerenciamento de estados do jogo. Essa classe serve como base para criar e gerenciar estados como menus, telas de jogo, ou qualquer outro estado necessário.

1. **Inicialização:**
   - game: Representa o objeto principal do jogo, que fornece acesso a recursos e configurações globais.
   - prev_state: Guarda uma referência ao estado anterior, facilitando transições entre estados e permitindo o retorno ao estado anterior quando necessário.
2. **Método blink_message:**
   - Controla o efeito de "piscar" em mensagens de texto.
   - Reduz o temporizador (em segundos) pelo valor de dt (delta time).
   - Alterna o estado de visibilidade (is_text_hidden) quando o temporizador chega a zero.
3. **Método set_blink_timer:**
   - Define o temporizador para o intervalo padrão de 0.75 segundos.
4. **Método update_cursor:**
   - vê se as teclas direcionais para cima (**UP**) ou para baixo (**DOWN**) foram pressionadas.
5. **Métodos handle_arrow_up_key e handle_arrow_down_key:**
   - Ajustam a posição do cursor no menu.
   - Utiliza o operador módulo (%) para garantir que o cursor "dê a volta" ao atingir o início ou fim das opções do menu.
6. **Métodos enter_state e exit_state:**
   - Pilha de Estados (state_stack):
     - O jogo utiliza uma pilha para gerenciar estados, onde o estado ativo é sempre o topo da pilha.
   - Entrada em um Novo Estado (enter_state):
     - Guarda o estado anterior (se houver) como referência em prev_state.
     - Empilha o novo estado.
   - Saída de um Estado (exit_state):
     - Remove o estado atual da pilha, retornando automaticamente ao estado anterior.
7. **Funcionamento Geral**
   - A classe State é abstrata e fornece funcionalidades reutilizáveis para os estados derivados:
   - Efeito de Piscar:
     - Ideal para destacar mensagens interativas.
   - Navegação no Menu:
     - Permite controlar a posição do cursor com as setas do teclado.
   - Transições de Estados:
     - Simplifica a entrada e saída de estados com um sistema de pilha.

```python
class State():
    def __init__(self, game):
        self.game = game
        self.prev_state = None
    def blink_message(self, dt):
        self.blink_timer_seconds -= dt
        if self.blink_timer_seconds < 0:
            self.is_text_hidden = not self.is_text_hidden
            self.set_blink_timer()
    def set_blink_timer(self):
        self.blink_timer_seconds = 0.75
    def update_cursor(self, keys):
        if keys['UP']:
            self.handle_arrow_up_key()
        if keys['DOWN']:
            self.handle_arrow_down_key()
    def handle_arrow_up_key(self):
        self.menu_cursor = (self.menu_cursor - 1) % len(self.menu_options)
    def handle_arrow_down_key(self):
        self.menu_cursor = (self.menu_cursor + 1) % len(self.menu_options)
    def enter_state(self):
        if len(self.game.state_stack) > 1:
            self.prev_state = self.game.state_stack[-1]
        self.game.state_stack.append(self)
    def exit_state(self):
        self.game.state_stack.pop()
```

2.9. title_screen.py

A classe TitleScreen, que representa a tela inicial do jogo com um menu principal. Ela permite ao jogador selecionar entre diferentes opções, como iniciar o jogo, ajustar configurações, ou sair. A classe herda de State, aproveitando funcionalidades genéricas como navegação de cursor e gerenciamento de estados.

1. **Inicialização:**
   - menu_options: Um dicionário com as opções disponíveis no menu principal.
   - menu_cursor: Indica a posição atual do cursor no menu, que é controlada pelo jogador.
2. **Método update:**
   - Entrada do Jogador:
     - Utiliza update_cursor (herdado de State) para alterar a posição do cursor com base nas ações do jogador (UP ou DOWN).
     - Executa a opção selecionada ao pressionar SELECT.
   - Reset de Ações:
     - Após processar os inputs, limpa os estados das ações com reset_player_actions.
3. **Método handle_selected_option:**
   - Start: Inicia o jogo, criando uma instância de GameWorld e empilhando-a no sistema de estados.
   - Change resolution: Abre a tela de configuração de resolução (ResolutionScreen).
   - Configure Sound: Navega para a tela de som (ConfigureSound).
   - Exit game: Finaliza o jogo ajustando os flags de execução.
4. **Método render:**
   - Fundo:
     - Exibe a imagem de fundo (background) carregada no objeto principal do jogo (game).
   - Título:
     - Renderiza o texto "Menu" no topo da tela.
   - Opções do Menu:
     - Cada opção é desenhada com cores diferentes:
       - Branco: Opção atualmente selecionada (menu_cursor).
       - Cinza: Opções não selecionadas.
5. **Funcionamento Geral**
   - A classe TitleScreen organiza o menu principal e suas funcionalidades de forma clara e eficiente:
     - Navegação: Utiliza as setas do teclado para percorrer as opções do menu.
     - Seleção: Executa diferentes ações com base na opção escolhida.
     - Modularidade:
       - Faz uso de outras classes de estado, como GameWorld, ResolutionScreen, e ConfigureSound, delegando responsabilidades específicas para essas classes.

```
from states.state import State
from states.game_world import GameWorld
from states.resolution_screen import ResolutionScreen
from states.score_screen import Score
from states.sound_screen import ConfigureSound
class TitleScreen(State):
    def __init__(self, game):
        State.__init__(self, game)
        self.menu_options = {0: 'Start', 1: 'Change resolution', 2: 'Configure Sound', 3: 'Score', 4:
'Exit game'}
        self.menu_cursor = 0
    def update(self, dt, player_actions):
        self.update_cursor(player_actions)
        if player_actions['SELECT']:
            selected_option = self.menu_options[self.menu_cursor]
            self.handle_selected_option(selected_option)
        self.game.reset_player_actions()
    def handle_selected_option(self, option):
        if option == 'Start':
            new_state = GameWorld(self.game)
            new_state.enter_state()
        if option == 'Change resolution':
            new_state = ResolutionScreen(self.game)
            new_state.enter_state()
        if option == 'Configure Sound':
            new_state = ConfigureSound(self.game)
            new_state.enter_state()
        if option == 'Score':
            new_state = Score(self.game)
            new_state.enter_state()
        if option == 'Exit game':
            self.game.running, self.game.playing = False, False
    def render(self, dt, surface):
        surface.blit(self.game.background, (0, 0))
        self.game.draw_text(surface, 'Menu', self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0]
/ 2, self.game.GAME_LOGIC_SIZE[1] / 4))
        self.game.draw_text(surface, 'Start', self.game.colors['GRAY'] if self.menu_cursor != 0 else
self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 3 + 20))
        self.game.draw_text(surface, 'Change resolution', self.game.colors['GRAY'] if
self.menu_cursor != 1 else self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2,
self.game.GAME_LOGIC_SIZE[1] / 3 + 40))
        self.game.draw_text(surface, 'Configure Sound', self.game.colors['GRAY'] if self.menu_cursor
!= 2 else self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1]
/ 3 + 60))
        self.game.draw_text(surface, 'Score', self.game.colors['GRAY'] if self.menu_cursor != 3 else
self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 3 + 80))
        self.game.draw_text(surface, 'Exit game', self.game.colors['GRAY'] if self.menu_cursor != 4
else self.game.colors['WHITE'], (self.game.GAME_LOGIC_SIZE[0] / 2, self.game.GAME_LOGIC_SIZE[1] / 3 +
100))
```

2.10. camera.py

A classe Camera é responsável por gerenciar a visualização do jogador em um labirinto, adicionando uma camada de "neblina" para limitar a visibilidade e criar um efeito imersivo de visão restrita. A câmera acompanha o movimento do jogador e aplica a restrição visual com base na posição do jogador e no tamanho das células do labirinto.

1. **Inicialização:**
   - maze: Representa o labirinto no qual a câmera opera.
   - view_range: Define o alcance de visão do jogador, em múltiplos do tamanho das células do labirinto.
   - fog_surface: Superfície para aplicar a neblina.
   - offset_x e offset_y: Define o deslocamento da câmera em relação à posição do jogador.
   - create_fog_surface(): Método chamado no início para inicializar a superfície da neblina com um raio máximo definido.
2. **Método update:**
   - Cálculo do Centro: A câmera é centralizada em torno do jogador. O cálculo de center_x e center_y ajusta a posição considerando o tamanho da tela e das células.
   - Offsets: Atualiza os deslocamentos horizontais e verticais para alinhar a visualização ao jogador.
3. **Método create_fog_surface:**
   - Criação da Superfície de Neblina
   - Raio Máximo: Define a área da neblina com base em um múltiplo do tamanho da célula.
   - Superfície Transparente: Cria uma superfície que suporta transparência (SRCALPHA) para desenhar a neblina.
4. **Método apply_fog:**
   - Aplicação da Neblina
   - Superfície de Neblina Completa:
   - self.fog_surface.fill((0, 0, 0, 250)): Cria uma camada preta semi-transparente para cobrir toda a tela.
   - Área de Visão:
   - Um círculo é desenhado na superfície fog_clear, com transparência decrescente em direção às bordas para criar um efeito de desfoque.
   - Posicionamento do Círculo:
   - O círculo transparente é posicionado na área correspondente à posição do jogador na tela, calculada com base nos offsets.
   - Mesclagem:
   - Usa BLEND_RGBA_SUB para "subtrair" transparência da neblina na área circular.
   - Aplicação Final:
   - A superfície de neblina é sobreposta à tela principal.
5. **Método apply:**
   - Converte as coordenadas de mundo para coordenadas de tela aplicando o deslocamento da câmera e o tamanho das células do labirinto.
6. **Método update_maze:**
   - Atualiza o labirinto referenciado pela câmera e recria a superfície de neblina.
7. **Método is_in_view_range:**
   - Determina se um ponto está dentro do alcance de visão do jogador, calculando a distância euclidiana entre os pontos e comparando com o alcance definido.

```
import pygame
import math
class Camera:
    def __init__(self, maze, view_range=10):
        self.maze = maze
        self.view_range = view_range
        self.fog_surface = None
        self.offset_x = 0
        self.offset_y = 0
        self.create_fog_surface()
    def update(self, player):
        center_x = pygame.display.get_surface().get_width() // (2.9 *
self.maze.cell_size)
        center_y = pygame.display.get_surface().get_height() // (2.9 *
self.maze.cell_size)
        self.offset_x = player.x - center_x
        self.offset_y = player.y - center_y
    def create_fog_surface(self):
        max_radius = int( 5 * self.maze.cell_size)
        self.fog_surface = pygame.Surface((max_radius * 2, max_radius * 2),
pygame.SRCALPHA)
    def apply_fog(self, surface, player):
        screen_width, screen_height = surface.get_size()
        self.fog_surface = pygame.Surface((screen_width, screen_height),
pygame.SRCALPHA)
        self.fog_surface.fill((0, 0, 0, 250))
        max_radius = int(3 * self.maze.cell_size)
        fog_clear = pygame.Surface((max_radius * 2, max_radius * 2), pygame.SRCALPHA)
        for r in range(max_radius, 0, -1):
            alpha = int(255 * (r / max_radius) ** 2)
            pygame.draw.circle(fog_clear, (0, 0, 0, 255 - alpha), (max_radius,
max_radius), r)
        player_screen_x = (player.x - self.offset_x) * self.maze.cell_size
        player_screen_y = (player.y - self.offset_y) * self.maze.cell_size
        clear_rect = fog_clear.get_rect(center=(player_screen_x, player_screen_y))
        self.fog_surface.blit(fog_clear, clear_rect,
special_flags=pygame.BLEND_RGBA_SUB)
        surface.blit(self.fog_surface, (0, 0))
    def apply(self, x, y):
        return (x - self.offset_x) * self.maze.cell_size, (y - self.offset_y) *
self.maze.cell_size
    def update_maze(self, maze):
        self.maze = maze
        self.create_fog_surface()
    def is_in_view_range(self, x, y, player):
        distance = math.sqrt((x - player.x)**2 + (y - player.y)**2)
        return distance <= self.view_range * self.maze.cell_size
```

2.11. collision.py

A classe collision é projetada para gerenciar e verificar colisões no jogo, incluindo colisões com paredes, a chave e a saída do labirinto.

1. **Inicialização:**
   - Inicializa a instância com os atributos key e exit
   - Armazena a posição da chave e da saída do labirinto
2. **Método check_wall_collision:**
   - Verifica se uma posição (x, y) representa uma colisão com parede
   - Primeiro, checa se a posição está dentro dos limites válidos do labirinto
   - Se a posição estiver fora dos limites, retorna True (considerando como parede)
   - Depois, verifica se a posição específica é uma parede usando maze.is_wall()
   - Retorna True se for uma parede, False caso contrário
3. **Método check_key_collision:**
   - Verifica se o jogador está na mesma posição da chave
   - Se estiver, chama o método collect() da chave
   - Provavelmente para "pegar" ou "coletar" a chave no jogo
4. **Método check_exit_collision:**
   - Verifica se o jogador chegou à posição de saída
   - Retorna True se as coordenadas corresponderem à saída
   - Retorna False caso contrário

```python
class collision:
    def __init__(self, key, exit):
        self.key = key
        self.exit = exit
    def check_wall_collision(self, x, y, maze):
        # First check if the position is within valid maze boundaries
        if not maze.is_valid_position(x, y):
            return True  # Treat out-of-bounds positions as walls
        # Then check if the position is a wall
        return maze.is_wall(x, y)
    def check_key_collision(self, x, y):
        if x == self.key.position[0] and y == self.key.position[1]:
            self.key.collect()
    def check_exit_collision(self, x, y):
        if x == self.exit[0] and y == self.exit[1]:
            return True
        return False
```

2.12. entities.py

Dentro deste ficheiro temos 2 classes Classe Entity que serve como base para entidades no jogo, recebe uma posição inicial no construtor, serve como classe pai para outros objetos do jogo

A classe Key (herda de Entity) que representa a chave no jogo de labirinto.

1. Inicialização:

- Usa super().__init__() para herdar a posição da classe pai
- Adiciona um atributo collected inicialmente como False

2. Método collect():

- Muda o estado da chave para collected = True
- Indica que a chave foi coletada pelo jogador

3. Método render():

- Responsável por desenhar a chave na tela
- Parâmetros:
  - surface: superfície de desenho
  - cell_size: tamanho de cada célula do labirinto
  - game: objeto do jogo
  - camera (opcional): para lidar com rolagem/movimento da câmera
  - visibility_check (opcional): para verificar se a chave está visível
  - player (opcional): jogador para verificação de visibilidade

4. Método render:

- Só renderiza se a chave não foi coletada
- Calcula a posição da chave na tela
- Verifica se a chave está no campo de visão (se houver verificação)
- Desenha a chave usando a imagem do jogo, redimensionada para o tamanho da célula

```
import pygame
class Entity:
    def __init__(self, position):
        self.position = position
class Key(Entity):
    def __init__(self, position):
        super().__init__(position)
        self.collected = False
    def collect(self):
        self.collected = True
    def render(self, surface, cell_size, game, camera=None, visibility_check=None,
player=None):
        if not self.collected:
            if camera:
                key_screen_x, key_screen_y = camera.apply(self.position[0],
self.position[1])
            else:
                key_screen_x = self.position[0] * cell_size
                key_screen_y = self.position[1] * cell_size
            # Render key and exit if in view range
            if visibility_check is None or visibility_check(self.position[0],
self.position[1], player):
                surface.blit(pygame.transform.scale(game.key, (cell_size, cell_size)),
(key_screen_x, key_screen_y))
```

2.13. maze.py

A classe maze gera os labirintos para um jogo de acordo com o nivel.

1. **Inicialização:**
   - Inicializa dicionário de tipos de posição (OPEN e WALL)
   - Chama generate_maze() para criar o labirinto
2. **Método generate_maze():**
   - Calcula tamanho do labirinto baseado no nível (aumenta com níveis)
   - Inicializa grade como paredes
   - Gera posição inicial aleatória
   - Cria caminhos usando DFS
   - Posiciona chave e saída
3. **Métodos de Verificação:**
   - is_valid_position(): Checa limites do labirinto
   - is_wall(): Verifica se uma posição é parede
4. **Método dfs() (Busca em Profundidade):**
   - Transforma posições em caminhos
   - Explora direções aleatoriamente
   - Cria corredores no labirinto
5. **Método place_key_and_exit():**
   - Seleciona posições aleatórias para chave e saída
   - Garante que não sejam na posição inicial
6. **Métodos de Renderização:**
   - render(): Desenha labirinto completo
   - draw_maze(): Renderiza células (paredes/caminhos)
   - draw_borders(): Adiciona bordas ao labirinto
   - draw_start_and_exit(): Desenha início e saída

```python
import pygame, random
from components.entities import Key
class Maze:
    def __init__(self, level):
        self.position_type = {'OPEN': 0, 'WALL': 1}
        self.start, self.exit, self.key = None, None, None
        self.generate_maze(level)
    def generate_maze(self, level):
        self.width = 10 + (level - 1) * 5
        self.height = 10 + (level - 1) * 5
        self.grid = [[1 for _ in range(self.width)] for _ in range(self.height)]
        # Generate a random start position
        self.start = self.random_position('WALL')
        # Generate maze paths using DFS
        self.dfs(self.start[0], self.start[1])
        # Place key and exit after maze generation
        self.place_key_and_exit()
    def is_valid_position(self, x, y):
        # Check if a position is within maze bounds
        return 0 <= x < self.width and 0 <= y < self.height
    def is_wall(self, x, y):
        return self.grid[y][x] == self.position_type['WALL']
    def dfs(self, x, y):
        self.grid[y][x] = self.position_type['OPEN']
        directions = [(0, -2), (0, 2), (-2, 0), (2, 0)]
        random.shuffle(directions)
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if self.is_valid_position(nx, ny) and self.is_wall(nx, ny):
                self.grid[y + dy // 2][x + dx // 2] = self.position_type['OPEN']
                self.dfs(nx, ny)
    def place_key_and_exit(self):
        # Get all open positions
        open_positions = [
            (x, y) for y in range(self.height)
            for x in range(self.width)
            if self.grid[y][x] == self.position_type['OPEN']
            and (x, y) != self.start
        ]
        # Randomly select positions for key and exit
        key_position = random.choice(open_positions)
        self.key = Key(key_position)
        open_positions.remove(key_position)
        self.exit = random.choice(open_positions)
    def random_position(self, position_type):
        target_value = self.position_type[position_type]
        while True:
            x, y = random.randint(0, self.width - 1), random.randint(0, self.height - 1)
            if self.grid[y][x] == target_value:
                return (x, y)
```

```
    def render(self, surface, cell_size, game, visibility_check=None, player = None,
camera=None):
        self.draw_maze(surface, cell_size, visibility_check, player, game, camera)
        self.draw_borders(surface, cell_size, game)
        self.key.render(surface, cell_size, game, camera, visibility_check, player)
        self.draw_start_and_exit(surface, cell_size, game, camera, visibility_check, player)
    def draw_maze(self, surface, cell_size, visibility_check,  player, game, camera):
        for y in range(self.height):
            for x in range(self.width):
                screen_x = x * cell_size
                screen_y = y * cell_size
                if camera:
                    screen_x, screen_y = camera.apply(x, y)
                if visibility_check and not visibility_check(x, y, player):
                    continue
                if self.grid[y][x] == 1:
                    surface.blit(pygame.transform.scale(game.wall, (cell_size, cell_size)),
(screen_x, screen_y))
                else:
                    surface.blit(pygame.transform.scale(game.path, (cell_size, cell_size)),
(screen_x, screen_y))  # Light color for paths
    def draw_borders(self, surface, cell_size, game):
        border_check_sides = {
            'left': any(self.grid[y][0] == 0 for y in range(self.height)),
            'right': any(self.grid[y][self.width-1] == 0 for y in range(self.height)),
            'top': any(self.grid[0][x] == 0 for x in range(self.width)),
            'bottom': any(self.grid[self.height-1][x] == 0 for x in range(self.width))
        }
        self.draw_continuous_borders(surface, cell_size, game, border_check_sides)
    def draw_continuous_borders(self, surface, cell_size, game, border_check_sides):
        if border_check_sides['left']:
            for y in range(self.height):
                border_rect = pygame.Rect(-cell_size, y * cell_size, cell_size, cell_size)
                pygame.draw.rect(surface, game.colors['BLACK'], border_rect)
        if border_check_sides['right']:
            for y in range(self.height):
                border_rect = pygame.Rect(self.width * cell_size, y * cell_size, cell_size,
cell_size)
                pygame.draw.rect(surface, game.colors['BLACK'], border_rect)
        if border_check_sides['top']:
            for x in range(self.width):
                border_rect = pygame.Rect(x * cell_size, -cell_size, cell_size, cell_size)
                pygame.draw.rect(surface, game.colors['BLACK'], border_rect)
        if border_check_sides['bottom']:
            for x in range(self.width):
                border_rect = pygame.Rect(x * cell_size, self.height * cell_size, cell_size,
cell_size)
                pygame.draw.rect(surface, game.colors['BLACK'], border_rect)
        self.fill_gaps_between_borders(surface, cell_size, game, border_check_sides)
```

```
    def fill_gaps_between_borders(self, surface, cell_size, game, border_check_sides):
        # Fill corner gaps where two borders meet
        if border_check_sides['left'] and border_check_sides['top']:
            corner_rect = pygame.Rect(-cell_size, -cell_size, cell_size, cell_size)
            pygame.draw.rect(surface, game.colors['BLACK'], corner_rect)
        if border_check_sides['left'] and border_check_sides['bottom']:
            corner_rect = pygame.Rect(-cell_size, self.height * cell_size, cell_size, cell_size)
            pygame.draw.rect(surface, game.colors['BLACK'], corner_rect)
        if border_check_sides['right'] and border_check_sides['top']:
            corner_rect = pygame.Rect(self.width * cell_size, -cell_size, cell_size, cell_size)
            pygame.draw.rect(surface, game.colors['BLACK'], corner_rect)
        if border_check_sides['right'] and border_check_sides['bottom']:
            corner_rect = pygame.Rect(self.width * cell_size, self.height * cell_size, cell_size,
cell_size)
            pygame.draw.rect(surface, game.colors['BLACK'], corner_rect)

    def draw_start_and_exit(self, surface, cell_size, game, camera, visibility_check, player):
        # Render key and exit with camera transformation
        if camera:
            Start_screen_x, Start_screen_y = camera.apply(self.start[0], self.start[1])
            exit_screen_x, exit_screen_y = camera.apply(self.exit[0], self.exit[1])
        else:
            Start_screen_x = self.start[0] * cell_size
            Start_screen_y = self.start[1] * cell_size
            exit_screen_x = self.exit[0] * cell_size
            exit_screen_y = self.exit[1] * cell_size
        if visibility_check is None or visibility_check(self.exit[0], self.exit[1], player):
            surface.blit(pygame.transform.scale(game.exit, (cell_size, cell_size)),
(exit_screen_x, exit_screen_y))
        if visibility_check is None or visibility_check(self.start[0], self.start[1], player):
            pygame.draw.rect(surface, game.colors['GREEN'], pygame.Rect(Start_screen_x,
Start_screen_y, cell_size, cell_size))  # Green color for the start
```

2.14. player.py

A classe player gere a movimentação do player, deteção de colisões para não não sair do labirinto e renderização com suporte a câmara.

1. Iniciaização
    - Inicializa o jogador na posição inicial do labirinto
    - Cria um objeto de colisão usando a chave e saída do labirinto
    - Armazena referência do jogo
2. Método update():
    - Verifica colisão com a chave
    - Chama move_player() para movimentação
3. Método move_player():
    - Define movimentos possíveis (cima, baixo, esquerda, direita)
    - Verifica se movimento é válido (sem colisão com paredes)
    - Atualiza coordenadas do jogador se movimento for permitido
4. Método render():
    - Chama start_player() para desenhar o jogador
    - Suporta sistema de câmera opcional
5. Método start_player():
    - Calcula posição do jogador na tela
    - Renderiza sprite do jogador no tamanho da célula
    - Aplica transformação da câmera se disponível
6. Método update_maze():
    - Atualiza objeto de colisão
    - Reposiciona jogador no início do novo labirinto

```python
import pygame
from components.collision import collision
class Player:
    def __init__(self, maze, game):
        # Inicializa o player na posição inicial do labirinto
        self.x = maze.start[0]  # Posição X inicial
        self.y = maze.start[1]  # Posição Y inicial
        self.game = game
        self.collision = collision(maze.key, maze.exit)

    def update(self, player_actions, maze):
        self.collision.check_key_collision(self.x, self.y)
        self.move_player(player_actions, maze)

    def move_player(self, player_actions, maze):
        movements = {
            'UP': (0, -1),
            'DOWN': (0, 1),
            'LEFT': (-1, 0),
            'RIGHT': (1, 0)
        }
        for action, (dx, dy) in movements.items():
            if player_actions[action] and not
self.collision.check_wall_collision(self.x + dx, self.y + dy, maze):
                self.x += dx
                self.y += dy

    def render(self, surface, cell_size, game, camera=None):
        # Use camera's apply method if provided
        self.start_player(surface, cell_size, game, camera)

    def start_player(self, surface, cell_size, game, camera=None):
        # Desenha o player
        screen_x = self.x * cell_size
        screen_y = self.y * cell_size
        if camera:
            screen_x, screen_y = camera.apply(self.x, self.y)
        # Render player
        surface.blit(pygame.transform.scale(game.player, (cell_size,
cell_size)),pygame.Rect(screen_x, screen_y, cell_size, cell_size))
    def update_maze(self, maze):
        self.collision = collision(maze.key, maze.exit)
        self.x = maze.start[0]
        self.y = maze.start[1]
```

2.15. Shader

Esta secção do trabalho baseia-se no que foi desenvolvido no ano passado, quando um elemento do grupo o ano passado implementou o mesmo. Com a devida permissão do colega que originalmente trabalhou na implementação, reutilizei e modifiquei ligeiramente o código para adaptá-lo às necessidades atuais.

O módulo **crt_shader** aplica um efeito visual ao ecrã, simulando características estéticas semelhantes às de monitores CRT (tubos de raios catódicos). Este efeito é alcançado através do uso de shaders, pequenos programas que processam gráficos diretamente na GPU. O processo consiste em renderizar o conteúdo do jogo para uma textura e depois aplicar transformações através de shaders customizados.

A classe `Shader` inicializa o contexto gráfico utilizando a biblioteca `moderngl` e define os vértices e coordenadas de textura para mapear o efeito sobre o ecrã. O código faz uso de shaders de vértice e fragmento, carregados a partir de ficheiros externos (`vertex_shader.glsl` e `fragment_shader.glsl`), para aplicar o efeito gráfico final. Durante a execução, o método `render` atualiza a textura do ecrã e renderiza a imagem transformada com as propriedades configuradas no shader.

Este efeito melhora a apresentação gráfica do jogo, conferindo um aspeto retro e nostálgico, adequado para jogos que se inspiram em épocas anteriores.

3. Conclusão

Com este trabalho, concluímos que conseguimos criar um jogo funcional com níveis infinitos. Se tivéssemos mais tempo, poderíamos ter implementado funcionalidades adicionais como um sistema de pontuação ou um modo multiplayer, cuja ausência deveu-se, em parte, aos trabalhos de outras unidades curriculares.

Tivemos o cuidado de otimizar o código, evitando a confusão da primeira versão, onde era difícil compreender a estrutura e a lógica. Embora esta preocupação nos tenha consumido tempo, consideramos crucial, pois não basta ter um código funcional se não soubermos identificar funções e variáveis.

A implementação da câmara foi deixada para o final, reconhecendo à partida que seria uma tarefa desafiante. Talvez tivesse sido mais simples se a tivéssemos implementado desde o início, mas acreditamos que o importante é aprender com os erros e com a prática, numa busca contínua de melhoria.

A otimização de código é um processo iterativo, e estamos conscientes de que podem sempre existir aspetos a aperfeiçoar. No entanto, consideramos que desenvolvemos uma base sólida para desenvolvimentos futuros.

Referências

Code, Clear. “Cameras in Pygame.” *YouTube*, 12 Feb. 2022, www.youtube.com/watch?v=u7LPRqrzry8. Accessed 28 Nov. 2024.

In-game, an. “Pygame Game States Tutorial: Creating an In-Game Menu Using States.” *YouTube*, 23 June 2021, youtu.be/b_DkQrJxpck?si=2jwv85JJOnmn92zz. Accessed 1 Nov. 2024.

ShawCode. “Pygame Camera - Pygame RPG Tutorial #8.” *YouTube*, 9 May 2021, www.youtube.com/watch?v=KnAkhpF3mIU. Accessed 28 Nov. 2024.

Como. “Pygame - Como Resolver Efeito Fantasma Da Imagem.” *Stack Overflow Em Português*, 20 May 2017, pt.stackoverflow.com/questions/206211/pygame-como-resolver-efeito-fantasma-da-imagem. Accessed 27 Nov. 2024.

“GameMaker - Sprite Invisível.” *GameMaker Community*, 15 Oct. 2023, forum.gamemaker.io/index.php?threads/sprite-invisivel.106434/. Accessed 27 Nov. 2024.

Schwabe, Ben. “How to Use Pygame Set_alpha() on a Picture.” *Stack Overflow*, 4 Sept. 2012, stackoverflow.com/questions/12255558/how-to-use-pygame-set-alpha-on-a-picture. Accessed 27 Nov. 2024.

Repositório Github:

https://github.com/Fiugas/GamePygame