Fractured-Heart: A 2D Zelda-Inspired Adventure Game

Project Title: Fractured-Heart
Student Name: Wildan, Mahtab, Marwan, Abizar, Yousif
Date: February 2025

Index

---

1. Introduction

Fractured-Heart is a 2D adventure game inspired by classic Zelda titles. The game features an expansive world rendered with efficient chunk loading and view culling techniques, dynamic enemy behavior with object pooling and area-based respawn, and an integrated NPC dialogue system for immersive storytelling. Players explore varied maps, engage in combat, cast magic, upgrade their character, and interact with NPCs—all while the game's performance remains optimized through our advanced design strategies.

Objectives

- **Engaging Gameplay:** Deliver dynamic combat, exploration, and immersive narrative.
- **Performance Optimization:** Utilize chunk loading, view culling, and enemy pooling to ensure smooth performance on large maps.

- **AI Integration:** Implement dialogue trees and branching conversations that adapt based on player choices.
- **Collaborative Development:** Divide tasks based on team members' strengths to streamline the development process.

---

2. Hardware/Software Requirements

Hardware Requirements

- **Processor:** Intel Core i3 or equivalent
- **RAM:** 4 GB minimum (8 GB recommended)
- **Graphics:** Integrated graphics or better
- **Storage:** At least 500 MB of free disk space
- **Display:** 1280 x 720 resolution or higher

Software Requirements

- **Operating System:** Windows 10 (or any OS supporting Python and Pygame)
- **Python:** Version 3.7 or higher
- **Pygame:** Version 2.0 or higher
- **Additional Libraries:** Standard Python libraries (os, sys, random, threading)
- **IDE/Editor:** PyCharm, VSCode, or any Python-compatible IDE

---

3. Design Strategies

Overall Architecture and Module Organization

The project is organized into modular components: - **main.py:** Entry point; manages game initialization, loading/transition screens, and the main game loop. - **level.py:** Manages map creation, collision detection, sprite groups, chunk loading, and enemy respawning. - **player.py & enemy.py:** Define the behaviors, animations, and interactions of the player and enemy entities. - **tile.py:** Manages individual map tiles and collision boundaries. - **support.py:** Provides utility functions (e.g., CSV parsing, asset loading). - **ui.py & upgrade.py:** Handle

user interface elements and the upgrade system. - **enemy_spawner.py:** Implements area-based, time-based enemy respawn using object pooling. - **ChunkedCameraGroup:** A custom sprite group that implements chunk loading and view culling.

Design Patterns and Optimization Techniques

- **Chunk Loading & View Culling:**
  Only sprites within or near the visible area are drawn, greatly reducing the rendering load on large maps.
- **Object Pooling for Enemy Respawns:**
  Instead of creating new enemy objects continuously, we reuse them via a time-based respawn mechanism that checks if the enemy's spawn point is off-screen.
- **Collision Grid Optimization:**
  Although not the primary focus in this version, the game also utilizes a collision grid to optimize collision checks.

Python Libraries Used

- **Pygame:** For game graphics, events, and sound.
- **Random:** For asset selection and enemy spawning logic.
- **Threading:** For asynchronous asset loading.
- **OS & CSV Modules:** (via support.py) for file management and CSV parsing.

Outline Sketches and Flow Diagrams

- **Game Flow Diagram:**
  1. **Loading Screen:** Animated "Loading..." while assets load asynchronously.
  2. **Start Menu:** Navigation options (Start Game, Settings, Scores, etc.).
  3. **Transition Screen:** Smooth fade-in transition into the gameplay.
  4. **Game Loop:** Level class manages map rendering, enemy updates, and UI overlays.
  5. **Game Over/Upgrade Screens:** Presented when the player dies or pauses.
- **Module Interaction Diagram:**
  Diagrams (attached separately) illustrate how main.py calls level.py, which in turn integrates tile.py, player.py, enemy.py, UI modules, and support functions.

4. Development Process

Throughout the project, we leveraged **GitHub** and **GitHub Projects** to manage our development workflow. We set up a repository to store all code, assets, and documentation. Using GitHub Projects, we created task boards where each task was broken down into subtasks and assigned to team members based on their strengths:

- **Asset/UI Designer - Marwan & Yousif:** Focused on creating and optimizing graphics (tilesets, NPCs, UI elements, and animations) using Krita and Tiled. The designer also prepared CSV files for map layouts.
- **AI Specialist - Mahtab:** Developed dialogue trees and branching conversations, integrating adaptive decision-making into the narrative. The dialogue system uses a file-based string dictionary and a binary tree search to navigate conversation nodes.
- **Core Programmers - Wildan & Abizar:** Implemented the game's main logic, including the Level, Player, Enemy, and enemy respawn systems. The programmer also integrated optimization techniques such as chunk loading and view culling.

Regular commits, pull requests, and code reviews on GitHub ensured that integration issues were minimized. GitHub Projects helped track progress, assign tasks, and plan milestones, resulting in a streamlined development process.

5. Team Contributions

Asset and UI Design

- **Asset Designer:**
  - Created a seamless grass tileset that transitions naturally between tiles using Krita.
  - Colored and animated assets (using PNG sequences) for NPCs, enemies, and environmental elements.
  - Designed multiple maps using Tiled, which included object layers with border blocks for collisions.
- **UI Designer:**
  - Developed a clean and responsive UI with a start menu, upgrade menu, and in-game HUD.
  - Prepared CSV files for map layers (boundaries, grass, objects, entities) and optimized UI elements for performance.

AI Integration

- **AI Specialist:**
  - Implemented dialogue trees with branching conversations that adapt based on player choices.
  - Developed a file-based string dictionary and binary tree search algorithm to manage dialogue nodes.
  - Integrated adaptive enemy behavior and narrative elements that converge to a unified story outcome.
  - *(Note: Dialogue tree code will be attached separately.)*

---

6. Code Overview

File Structure and Key Modules

- **main.py:** Initializes the game, handles transitions, and runs the main loop.
- **level.py:** Manages map creation, collision, sprite grouping, chunk loading, and enemy respawning.
- **player.py & enemy.py:** Define core entity behaviors and interactions.
- **tile.py:** Manages individual tiles and collision logic.
- **support.py:** Utility functions for CSV and asset imports.
- **ui.py & upgrade.py:** Manage the user interface and upgrade system.
- **enemy_spawner.py:** Implements area- and time-based enemy respawning using object pooling.
- **ChunkedCameraGroup:** A custom sprite group for efficient rendering via chunk loading.

Explanation of Main Components

- **Level Class:**
  Reads CSV files to build the map and records enemy spawn points. Integrates chunk loading and creates an enemy spawner to manage enemy respawns based on area and time.
- **Player and Enemy Classes:**
  Manage movement, combat, animations, and interactions. The Enemy class incorporates sound caching and a death process that plays a death animation before removal.
- **ChunkedCameraGroup:**
  Efficiently draws only those sprites in nearby chunks, greatly improving performance.
- **Enemy Spawner:**

Checks each enemy spawn point periodically. If the spawn point is off-screen and no enemy is present, it creates a new enemy, ensuring respawned enemies are visible and correctly initialized.

- **Dialogue & AI:**
  Dialogue trees and branching narratives (developed by our AI Specialist) integrate adaptive storytelling into gameplay.

Selected Code Snippets

Enemy Spawner – enemy_spawner.py:

```
import pygame
from enemy import Enemy

class EnemySpawner:
    def __init__(self, level, area_size=100, respawn_interval=5000, update_interval=1000):
        self.level = level
        self.area_size = area_size
        self.respawn_interval = respawn_interval
        self.update_interval = update_interval
        self.last_update = pygame.time.get_ticks()
        self.spawn_times = {spawn: pygame.time.get_ticks() for spawn in self.level.enemy_spa

    def enemy_in_area(self, pos):
        player = self.level.player
        cam_width = self.level.display_surface.get_width()
        cam_height = self.level.display_surface.get_height()
        camera_rect = pygame.Rect(0, 0, cam_width, cam_height)
        camera_rect.center = player.rect.center
        if camera_rect.collidepoint(pos):
            return True
        area_rect = pygame.Rect(0, 0, self.area_size, self.area_size)
        area_rect.center = pos
        for enemy in self.level.attackable_sprites:
            if enemy.rect.colliderect(area_rect):
                return True
        return False

    def update(self):
        current_time = pygame.time.get_ticks()
        if current_time - self.last_update < self.update_interval:
            return
        self.last_update = current_time
        for spawn in self.level.enemy_spawn_points:
            last_spawn_time = self.spawn_times.get(spawn, 0)
            if current_time - last_spawn_time >= self.respawn_interval:
```

```
if not self.enemy_in_area(spawn[1]):
    monster_name, pos = spawn
    new_enemy = Enemy(
        monster_name,
        pos,
        [self.level.visible_sprites, self.level.attackable_sprites],
        self.level.obstacle_sprites,
        self.level.damage_player,
        self.level.trigger_death_particles,
        self.level.add_exp
    )
    new_enemy.frame_index = 0
    new_enemy.status = 'idle'
    new_enemy.vulnerable = True
    self.spawn_times[spawn] = current_time
```

7. Conclusion

Fractured-Heart is a 2D adventure game that marries engaging gameplay with robust performance optimizations. Through techniques like chunk loading, view culling, and object pooling for enemy respawns, the game maintains smooth performance on expansive maps. Our development process leveraged GitHub and GitHub Projects to organize tasks and assign them based on team members' strengths, ensuring efficient collaboration. The asset and UI design, combined with advanced AI dialogue integration, result in an immersive game experience.

This documentation, along with the attached code files, provides a comprehensive overview of the project's design, development, and implementation.