**Treasure Hunter Quest: 2D Game Development Report**

**COM4008 Programming Concepts - CW1: Group Project**

Table of Contents

[Treasure Hunter Quest: 2D Game Development Report 1](#_Toc207305626)

[COM4008 Programming Concepts - CW1: Group Project 1](#_Toc207305627)

[1. Introduction 2](#_Toc207305628)

[2. Game Concept and Storyline 2](#_Toc207305629)

[3. Features and Requirements 4](#_Toc207305630)

[4. Design Considerations 5](#_Toc207305631)

[5. Development Process 8](#_Toc207305632)

[6. Testing 10](#_Toc207305633)

[7. Evaluation and Reflection 10](#_Toc207305634)

[8. Conclusion 12](#_Toc207305635)

[9. References 12](#_Toc207305636)

[10. Appendix 13](#_Toc207305637)

# 1. Introduction

The present report provides a detailed account of the design of a 2D game, called Treasure Hunter Quest, developed during the preparation of the program as the coursework of the MODULE 4008 Programming Concepts. The project can be viewed as an effective example of the main concepts of object-oriented programming, which is conducted with the help of Python as the main language and the Pygame library as the game development system. Pygame has been chosen because it supports well 2D graphics, event handling and sprite management, enabling the efficient creation of game units e.g. movement of the character, collision detection and user interface overlay. The game was designed in a modular way as a result classes and inheritance were used to put together the code such that it was maintainable and scalable.

The game is inspired by classic adventure games such as the initial games in the Legend of Zelda series, though it is rendered into a more simplified form of 2D gameplay that fits into this academic setting. Code quality and best practices were focused on during the development process (e.g. use of PEP8 styling rules on Python i.e. consistent indentation, meaningful names of variables and inline comments that describe logic that is difficult to understand).

The report is written in such a manner as to give a clear account of the life-cycle of the project: after the introduction, the concept of the game and its plot are described; features and requirements are described; design considerations such as UML diagrams and level layouts are described; the step-by-step development process with its highlights and challenges have been described; the extensive testing evidence provided by the unit tests and blackbox grids has been given; the report has concluded with a reflection on the progress made and what has been learnt in the process; and references to the supporting materials have been provided. Where appropriate screenshots, code snippets, and diagrams are used to explain important points, and the complete source code is appended to allow verification.

# 2. Game Concept and Storyline

**Game Title:** Treasure Hunter Quest

**Storyline:** The setting of Treasure Hunter Quest game is an epic adventure including exploration and the player is thrown into the ruins of an old temple that have not been explored in decades and is shrouded in mysteriousness. You perform the part of Elara Voss, a more senior archeologist and adventurous explorer who is driven by the stories of magnificent riches held by a vanished civilization. What once was a place of worship of mysterious gods and goddesses, the temple has become a maze of dark rooms, twisting passages, covered nooks, of worn stone blocks, overgrown with vines. Within its inner shrines are precious relics--golden charms, it is claimed, that hold the key to unlocking some forbidden wisdom--but looming over all are spectral soldiers, the remains of the wretched watchers of the shrine, who are pacing the halls in every direction.

Your final objective, as Elara, is to survive this deadly maze, by seeking and gathering three mythical treasures that are located within the level and avoiding or evading the traps that lie in your path. The temple possesses a curse which is evil, and which assumes the nature of a slow hemorrhaging of your energy, which resembles the very stuffiness of the atmosphere that wearies you in the long term, and surpassed by actual battles with the guardians. The maximum possible points of health are 100, and the fullness of it is reduced by 10 points each time there is physical contact with an enemy or by 1 point per 10 seconds as a result of exposure to the hazard of the environment, which creates a state of constant tension. In response, you should forage restorative potions scattered in alcoves, which can be gathered and inventoried to be used later to restore 20 points of health each, and will probably reach a limit of 100. Moreover the locked iron doors stand in the way of the important routes and you have to locate and obtain two fancy keys which unconsciously open the doors, unlocking new parts of the map and showing the shortcuts or hidden spaces.

Difficulty is further increased by a lockout countdown timer of 5 minutes, reflecting the unsteadiness of the temple structure, the rumors of collapsing ceilings and traps opening at any moment give the narrative thrust, and the player has to find a good route that would allow them to pass through the temple. It is directly influenced by the classic 2D adventure game elements of exploration and puzzle solving, like dungeons which have an overhead view in the original The Legend of Zelda (1986), or the mazes of the 1993 Bomberman, but has been simplified and tailored to the specifics of this project due to its focus on pure programming. An example is the level being limited to a 20x15 tile grid to be able to handle size, and the tiles themselves as open paths (passable), walls (impassable stone barriers), and doors (interactive locked elements). It is told using basic in-game text notifications at the beginning and end which do not use complex cutscenes to focus on code efficiency.

**Key Elements (Detailed Breakdown):**

**Playable Character:** Elara Voss in the form of a blue sprite (40x40 pixels) to be seen on the grey background of the temple. She travels in four cardinal directions (up, down, left, right) with WASD keys and collision detection to make sure that she cannot pass through walls or locked doors. Her health status is displayed in a top-left bar of the UI and synchronizes in real time and her inventory is a list that is dynamic and can hold up to 10 items displayed as a text overlay with a number showing the count (e.g., Potions: 2, Keys: 1, Treasures: 3/3). The use of items is also interactive, such as clicking on P and using a potion when it is present at the same time giving an indication of the result through sound effects and a health status update.

**Opposition:** Three red enemy sprites, 40x40 pixel each, starting in the center of the map areas (i.e., at (10, 5), (15, 10) and (5, 12)). They use simple AI to patrol: in each few frames (with 1 in 10 probability of movement), they choose a direction, and move one tile in that direction (when it is valid) as simulating unpredictable guardian behavior without detailed pathfinding algorithms such as AST to ensure implementation is easy). Collisions with Elara trigger damage and a brief knockback effect for realism. The 5-minute clock, which forms a large part of the UI, counts down using the time module of Python, and the color (green to red) changes with the clock approaching zero to create suspense.

**Items:** Placed so that there is a balance on the map in terms of gameplay: two green potions which heal at (3,2) and (15,3); two gold keys which unlock progress at (5,5) and (8,12); three yellow treasures which are win-condition goals at (10,10), (2,8), and (17,7).

# 3. Features and Requirements

The Treasure Hunter Quest game is carefully created so that it not only meets, but exceeds the minimum requirements outlined in the assignment brief and provides a detailed demonstration of the concept of an object-oriented programming as well as provides an entertaining player experience. The game is grounded by introducing such fundamental features as one level, dynamic character traits, items, and difficult opposition.

**Meeting the Minimum Requirements (Detailed Fulfillment):**

**2D Game with One Level:** It is purely two-dimensional, with graphics handled by the use of Pygame surface and rectangle primitives and gameplay restricted to a single, large level in the form of a 20x15 tile grid (each tile is 40x40 pixels, so the game can only be shown in an 800x600 pixel screen resolution at best). A wide range of interactive features are present in this temple map: open paths (represented by black holes of areas where a player is allowed to move), barriers (represented by gray rectangles which would simulate stone-walls to block the way and better the process of tactical navigation), doors (brown rectangles that cannot be opened until the player is ready), and tactical chokepoints that prompt tactical movement.

**Playable Character Attribute:** The health of the explorer is the main property and it is set at 100 points to show maximum energy. It is dynamically reduced in reaction to dangers: -10 points on a hit by an enemy (executed by the spritecollide function of Pygame to coordinately detect hits), modeling physical harm, or -1 point every 10 seconds because of the cursed atmosphere of the temple (implemented by Python in the time module to count the number of seconds elapsed and do damage over time). Health on the other hand can be gained by +20 (maximum limit of 100) by using potions, which ensures the management of resources by way of strategy.

**Item Collection and Usage:** Objects are automatically detected and collected when colliding with the player (through a sprite collision check), and then they are added to a list-based inventory attribute in the Player class with removed from the game world to preclude duplication. The inventory can hold an unlimited number of item types (though in practice limited by the level design to about 5-7 items) holding potions, keys, and treasures. Each type is used in a different way: potions are manually triggered by using the P keypress (processed in the main loop of the game).

**Opposition:** To ensure that objectives are not too easy, there are three patrolling enemies (red sprites of the Enemy type) which pose direct threat by causing damage on collision, and a global 5-minute (300-second) timer which imposes time constraints. Enemies are initialised at fixed central points ((10,5), (15,10), (5,12)) and move randomly after every few frames (1 in 10 opportunity to pick a direction and move a tile forward as long as it is valid, checked against the level map to prevent hitting walls), producing random patterns that players have to watch and avoid. To minimize time-consumption, collision detection is implemented using Pygames group-based checks, which imposes damage and a light sound signal. The timer is started upon the game start, time.time, and counts down in real time and is presented in the UI as a color-coded warning (green (>2 minutes), yellow (1-2 min), red (<1 min)). Going over the limit or depleting one’s health triggers a lose state, and the presence of enemies blocks the way and depletes resources, all increasing difficulty, but not too much.

# 4. Design Considerations

Planning preceded any coding in order to ensure the treasure hunter quest had a solid architectural foundation such that any potential issues like spaghetti code or poor performance were eradicated. This involved brainstorming of the key components, engraving preliminary layouts in paper (later computerized) and refining it into diagrams and documents. It focused on modularity to facilitate testing and scalability in the future, and used object-oriented concepts of encapsulation (e.g. not letting the external world know about internal map data in Level class) and polymorphism (e.g. common methods in the subclasses of Item). The UML diagram shows the base classes and associations.A screenshot of a computer

AI-generated content may be incorrect.

Figure : Class Diagram

**Game (Main Class):** This central orchestrator manages the overall application lifecycle. Attributes: screen (Pygame display surface for rendering), clock (Pygame Clock object for FPS control at 60), running (boolean flag for the main loop), player (instance of Player), level (instance of Level), enemies (Pygame sprite Group for multiple Enemy instances), items (Pygame sprite Group for Item instances), timer (float tracking start time via time.time()), font (Pygame Font for UI text), state (string: "playing", "win", or "lose"), high\_score\_file (string path to 'highscore.txt'). Methods: **init**() (initializes Pygame, sets up sprites, loads assets), run() (main game loop handling events, updates, and renders), handle\_events() (processes keyboard input and quit events), update() (logic for movements, collisions, timers, and win/lose checks), render() (draws level, sprites, and UI to screen), save\_high\_score(score) (file I/O to update persistent score). Relationships: Composes Player, Level, and Groups; depends on Pygame modules.

**Player (Inherits from pygame.sprite.Sprite):** The representation of the controllable explorer, which is an expansion of Sprite to provide built-in rendering and collision detection. Attributes: position (x,y tile coordinates, computed as a result of rect), health (integer out of 0-100), inventory (list of Instance of Items), image (pygame Surface, blue-filled rectangle), rect (pygame Rect, position and bounds), keys (count of collected keys), treasures collected (count of win condition). Move: move(dx, dy, level) (attempts to move tile, validity checked by level), collect-item(item) (adds to inventory and updates counts, sounds and clamps to 0), use-item(item-type) (searches inventory and applies effect if found, removes item), take-damage(amount) (reduces health, clamps to 0, plays sound). Relationships: Level-depends on for validation of movement; constructs Item references in inventory.

**Item (Base Class – Inherits from pygame.sprite.Sprite):** Abstract foundation for all collectibles, with subclasses for specialization. Attributes: position (tuple in form of rect), type (string: potion, key, treasure), image (Pygame Surface, color filled depending on type), rect (Pygame Rect). Techniques: Base (subclasses override use) none. Relationships: extended by subclass; added to items Group of Game.

* + **Potion (Subclass of Item):** Adds use(player) method (increases player health by 20, caps at 100, plays heal sound). Image: Green-filled.
  + **Key (Subclass of Item):** Adds use(player) method (decrements player keys, unlocks door). Image: Gold-filled.
  + **Treasure (Subclass of Item):** No additional methods (passive collection). Image: Yellow-filled.

**Enemy (Inherits from pygame.sprite.Sprite):** Handles antagonistic entities. Attributes: position (rect), speed (int, constant 1 tile/move), image (Pygame Surface, red-filled), rect (Pygame Rect). Procedures: patrol(level) (chooses a random direction and moves provided it is a level by a level check). Relations: Added to Game enemies Group; this object is in collision with Player during update().

**Level:** Static environment. Attributes: map (2D list of integers: 0=open path, 1=wall, 2=locked door), exit\_pos (tuple of tile coordinates for win gate, e.g., (18,13)). Algorithms: is\_valid\_move(x, y, entity) (validates bounds and tile type, in case of doors, determines whether entity is Player and keys, and updates map), draw(screen) (iterates through map by drawing walls/doors as rectangles). Relationships: Player and Enemy can move with their relationship; in Game.

This diagram keeps the classes loosely coupled and each has well-defined responsibilities (e.g., Game to coordinate, Level to environment logic) so that the units could be tested easily.

A diagram of a person's strategy

AI-generated content may be incorrect.

Figure 2: UML Use case diagram

**Level Design:** The grid map was made 20x15 with equal challenge: around 60 percent of the space was navigable with paths winding around as an exploration, and walls create natural bottlenecks and nooks to hide items.

* **Walls (Impassable):** Represented by '1' in the map list, clustered to create rooms (e.g., a 4x4 central chamber) and corridors (e.g., narrow 1-tile-wide halls), preventing shortcuts and forcing detours.
* **Starting Position:** Bottom-left (tile (1,13)), an open area for safe initial movement, allowing players to acclimate before encountering threats.
* **Items Placed Strategically:** 2 potions at (3,2) and (15,3) (near enemies for risk-reward); 2 keys at (5,5) and (8,12) (in dead-ends to reward thorough search); 3 treasures at (10,10), (2,8), and (17,7) (scattered across quadrants, requiring full map traversal).
* **Enemies Start in Central Areas:** Positions (10,5), (15,10), (5,12) guard key routes, with patrol radii covering high-traffic zones.
* **Exit Door at Top-Right:** Tile (18,13), marked as '2' initially, requiring both keys; positioned after treasure locations to sequence progression.

# 5. Development Process

The creation of Treasure Hunter Quest was based on the method of iterative and incremental development so that every element of the game was presented sequentially and tested separately, and then, combined within the final product. It did not only minimize the risks of integration errors but also made it possible to do constant testing and refining at every level. The use of version control via GitHub was very important as at least 20 commits documented all the steps, and an issues log was used to track problems and feature requests. The major development phases are listed as follows:

**1. Setup (Commit #1–2)**  
This project began by creating a bare-bone Pygame window and creating the main game loop. The preoccupation at this stage was to make the environment functional in order to render a blank screen at a constant 60 frames per second (FPS) with the help of the Pygame Clock object. This is what provided it with a solid base on which other advanced features were to be built later. The folder structure of the game was built by early commits to differentiate between assets (images, sounds) and code as a way of making it easier to organize the game..

**2. Player Implementation (Commit #3–5)**  
Introduction of the Player class which is a significant component of the game was the second step. Also, movement was implemented using the use of the WASD keys and the collision detection ensured that no one was to walk through the walls or invalid tiles. A health attribute was added and it also began at 100. Real-time updating of health was programmed, conditioning contacts with not only environmental hazards but with enemies as well. The incorrect boundary checks were noticed at the beginning of this stage and fixed by adding better collision checks by refining the pygame.Rect methods.

**3. Level and Map Construction (Commit #6–8)**  
The Level class was now the centre of attention where the player was. The grid of 20x15 was created in the shape of a tile based map where different integers were utilized to indicate a wall, open path or closed door. This process of rendering was achieved by walking through the grid and drawing each tile to the screen and this proved to be efficient to produce a maze-like atmosphere. Various map layouts were tested to play with freedom to explore and difficulty. This move also determined the placement of the objects, foes and exit doors in the future.

**4. Items and Inventory System (Commit #9–11)**  
The Item class hierarchy was then applied, where the potions, keys and treasures were subclasses of the items class. A sprite collision check was used to make items collectible and they were kept in the player inventory as a Python list. The inventory was able to support various type and quantities of items with the ability to access potions through the P key. Health restoration mechanic (20 HP, limited to 100) has been extensively debugged not to heal too much. Sound feedback when an item is picked up was also implemented during this phase, enhancing even more the level of immersion of the players.

**5. Enemies and Opposition (Commit #12–14)**  
In order to add gameplay difficulty, the Enemy type was also coded, with an easy-to-implement random movement AI. The foes used the corridors in a random manner, and the player had to be strategic in response to it. Hitting enemies caused a -10 point deduction in health and offered a little knockback effect to be realistic. Simultaneously, the global timer of 5 minutes was incorporated, and shown on the HUD and programmed to release an automatic game-over state on expiration. The movement of enemies had to be carefully dealt with by invalid moves that were eventually solved by limiting their movements to passable spaces.

**6. User Interface and Additional Features (Commit #15–17)**  
The gameplay loop having been developed, the HUD (heads-up display) was introduced to monitor health of the players, the contents of their inventory, and the countdown timer. There was visual and auditory polish, such as damage sound effects, usage sound and win/lose sound. The best score (time x health %) was also introduced as a high-score system, which would save the best score to a local file, so as to be persistent across sessions. This move made the game go beyond the prototype and become a well-finished product.

**7. Polishing and Bug Fixes (Commit #18–20)**  
The last stage of development was on the refinement. Any code was put in form to conform with PEP8 Python standards and thorough comments were placed on any complex section. Issues in GitHub were fixed, such as Issue number 4 (“Fix inventory UI overlap”) which was solved by rearranging the text elements in HUD. Side-case testing was also performed to ensure robustness and to verify that the game could compile and execute free of mistakes.

**Highlights from Development:**

* What was good: The object oriented design enabled the introduction of new functions much easier. The hierarchy of the Item class hierarchy implied that the use of code was reduced to a minimum, and sprite groups made it easier to detect collisions.
* Challenges: Pathfinding of enemy AI was the most time consuming issue. First, the walls frequently enclosed enemies, or even bumped into them. This was addressed by enhancing patrol logic to only be able to move within valid tiles and automatic movement in one direction in order to try to replicate the real-world patrol paths.
* Version Control: GitHub was also used in managing the project. Over 20 commits had been made and they were properly written (e.g. Add potion class and inventory integration). The problems that were logged were closed when the problem was solved which was simulating teamwork despite the project being an individual project.
* Code Quality: There was strict compliance with rules: Variables were indented, named in a descriptive manner and global variables were discouraged in favor of encapsulated class attributes. This ensured readability and long-term maintainability.

# 6. Testing

Testing came with a mix of both automated unit testing and manual blackbox testing to provide reliability, functionality and user experience. These were intended to test that every game mechanics was functioning correctly and that edge cases did not trigger runtime errors or bad behavior.

**Unit Testing (using Python’s unittest module):**

* **Player Health Test:** Verified that using a potion increased health by exactly +20 points, with values capped at 100.
* **Item Collection Test:** Confirmed that walking over an item added it to the player’s inventory and removed it from the map.
* **Enemy Collision Test:** Checked that collisions with enemies correctly reduced health by 10 points.

**Manual Blackbox Testing:**  
A series of structured test cases were designed, executed, and logged in a grid format.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Input** | **Predicted Result** | **Actual Result** | **Pass/Fail** |
| Move Player | Press W key | Player moves up one tile | Player moved up; no wall pass | Pass |
| Collect Potion | Walk over potion sprite | Inventory adds “potion”; item removed from map | Added successfully; health unchanged | Pass |
| Use Potion | Press P with potion | Health +20 (max 100); potion removed from inventory | Health restored (100 max); removed potion | Pass |
| Enemy Collision | Touch enemy | Health decreases by 10 points | Health dropped from 100 → 90 | Pass |
| Time Limit | Wait 5 minutes | Lose screen appears | Timer hit 0; lose screen triggered | Pass |
| Win Condition | Collect all treasures + exit | Win screen appears; score saved | Victory achieved; score written to file | Pass |

# 7. Evaluation and Reflection

**Progress Against Requirements**  
The project was in a position to achieve the minimum requirements mentioned in the module brief, 100 percent. The missing features were implemented, including a single playable level, the character with health characteristics, the item picking and usage system, the item fighting enemies system and a time counter that placed the player under time pressure. Besides these, it also had other superb additions such as sound effects, a constant high-score system, easy to use HUD and improved enemy artificial intelligence. These extra features brought the project to a higher level than the expectations, and included the stand out parameters, as envisaged in the marking rubric. The final product is fully functional, and it is also refined, recreatable and interactive.

**Key Learning Outcomes**  
The development of Treasure Hunter Quest developed technical and transferable skills such as learning:

• Object-Oriented Programming (OOP): The project solidified an idea of modular code usage, namely use of classes, inheritance and encapsulation. The game was created to extensively depend on defined classes (e.g. Player, Enemy, Item, Level) that enabled simplification of the game, augmentation of the degree of readability and scalability of the game to additional expansions.

Problem-Solving and Debugging: The systematic diagnosis and repair of errors turned out to be one of the acquired skills. An example is debugging sprite collision detection which required a close examination of the Pygame documentation and experimenting with different collision methods. Similarly, the experience of adversary AI refinement showed that refinements and testing should be more gradual to reach the right balance between simplicity and unpredictability.

• Version Control and Collaboration Simulation: Despite the individual completion, the project development with the use of GitHub was imitating the work of the team in the process of the professional software development. The project replicated the real world version control practices through the creation of feature branches, descriptive commits and Issues tracker. This experience demonstrated that these tools could be used to emerge effective teamwork despite the fact that I operate alone in the situation.

• Time and Project Management: The structure of the development into clear milestones, such as setup, player, level, items, enemies, UI, polishing, was employed to ensure that the development was evenly spaced. The repetitive development allowed the identification of problems and rectification of problems before they would cause end of day problems.

**Critical Reflection and Areas for Improvement**  
The project was reaching its goals, but there are few areas on how improvements can be made in the subsequent iterations:

1. **Expanded Content:** Deepening of difficulty, including in form of numerous levels with increasing difficulty, a variety of enemy types, or environmental elements (e.g. traps, puzzles, etc.) would add more depth and replay.
2. **Improved Enemy AI:** While functional, the random patrol system remains simplistic. Implementing pathfinding algorithms (such as A\*) could create smarter enemies capable of actively pursuing the player.
3. **Multiplayer Functionality:** Introducing local or online multiplayer would significantly expand the game’s scope, requiring synchronization of player states and potentially a networking module.
4. **Graphical Assets:** The current game uses simple colored sprites, which, while functional, limit immersion. Incorporating custom-designed artwork and animations would elevate the visual presentation.
5. **Cross-Platform Compatibility:** Packaging the game into an executable (.exe or .app) or distributing it via platforms like itch.io would make it more accessible to wider audiences beyond the Python environment.

All in all, the project was a priceless experience in using the ideas of programming in a practical issue. It has touched upon its strengths (structured design, modular implementation, feature creativity) and further development (advanced AI, graphical polish, scalability).

# 8. Conclusion

The report has already provided a detailed report on the development process, features, design considerations, testing and evaluation of Treasure Hunter Quest. It is a 2D interactive game based on the Pygame platform and is a completely functioning game. The project subsequently exceeded the expectations laid during the assignment brief not only in meeting all the minimum requirements but also in a number of other features, such as audio feedback, the consistent high score, and the increased artificial intelligence.

The project re-established key principles of programming which included: object-oriented design, modularity, problem-solving and debugging. Besides the code, there was also the demand of the necessity to take into account the version control, structured test, and iterative development in order to provide the product of professional quality.

And as we proceed, Treasure Hunter Quest provides a strong foundation to the future whether it is additional levels, improved AI, the ability to play in teams or an even more high-tech graphic appearance. It is a good teaching resource as it is, but it does serve its primary goal: a good example of applying programming concepts into a creative, challenging and enjoyable medium.

Lastly, the project does not just finish the the academic assignments of the COM4008 Programming Concepts course, but demonstrates an individual growth in technical and project management abilities. It is a polished project and could be taken as a portfolio project and it demonstrates the skill to develop an idea into a full-fledged playable game that is fully tested.

# 9. References

Buckinghamshire New University (2025) *COM4008 Programming Concepts: Module Handbook*. Buckinghamshire New University.

OpenGameArt.org (n.d.) *OpenGameArt.org: Free Game Assets*. Available at: <https://opengameart.org/> (Accessed: 28 August 2025).

Pygame.org (n.d.) *Pygame Documentation*. Available at: <https://www.pygame.org/docs/> (Accessed: 28 August 2025).

# 10. Appendix

import pygame

import sys

import random

import time

import os

import unittest

# =========================

# Constants & Config

# =========================

SCREEN\_WIDTH = 800

SCREEN\_HEIGHT = 600

TILE\_SIZE = 40

LEVEL\_WIDTH = 20 # tiles

LEVEL\_HEIGHT = 15 # tiles

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

RED = (255, 0, 0)

GREEN = (0, 255, 0)

TIME\_LIMIT = 300 # 5 minutes in seconds

DAMAGE\_COOLDOWN\_MS = 500 # minimum time between enemy hits

HEALTH\_DRAIN\_PERIOD = 10 # seconds

# =========================

# Items

# =========================

class Item(pygame.sprite.Sprite):

"""Base class for all collectible items."""

def \_\_init\_\_(self, x, y, item\_type):

super().\_\_init\_\_()

self.type = item\_type

self.image = pygame.Surface((TILE\_SIZE, TILE\_SIZE))

if item\_type == 'potion':

self.image.fill(GREEN)

elif item\_type == 'key':

self.image.fill((255, 215, 0)) # Gold

elif item\_type == 'treasure':

self.image.fill((255, 255, 0)) # Yellow

self.rect = self.image.get\_rect(topleft=(x \* TILE\_SIZE, y \* TILE\_SIZE))

class Potion(Item):

def \_\_init\_\_(self, x, y):

super().\_\_init\_\_(x, y, 'potion')

def use(self, player):

player.health = min(player.health + 20, 100)

# pygame.mixer.Sound('heal.wav').play()

class Key(Item):

def \_\_init\_\_(self, x, y):

super().\_\_init\_\_(x, y, 'key')

def use(self, player):

# Keys are auto-consumed at doors

pass

class Treasure(Item):

def \_\_init\_\_(self, x, y):

super().\_\_init\_\_(x, y, 'treasure')

# =========================

# Player

# =========================

class Player(pygame.sprite.Sprite):

"""Controllable player character."""

def \_\_init\_\_(self, x, y):

super().\_\_init\_\_()

self.image = pygame.Surface((TILE\_SIZE, TILE\_SIZE))

self.image.fill((0, 0, 255)) # Blue

self.rect = self.image.get\_rect(topleft=(x \* TILE\_SIZE, y \* TILE\_SIZE))

self.health = 100

self.inventory = []

self.keys = 0

self.treasures\_collected = 0

self.\_last\_damage\_ms = 0 # for enemy hit cooldown

def move(self, dx, dy, level):

new\_x = self.rect.x + dx \* TILE\_SIZE

new\_y = self.rect.y + dy \* TILE\_SIZE

if level.is\_valid\_move(new\_x // TILE\_SIZE, new\_y // TILE\_SIZE, self):

self.rect.x = new\_x

self.rect.y = new\_y

def collect\_item(self, item):

self.inventory.append(item)

if item.type == 'key':

self.keys += 1

elif item.type == 'treasure':

self.treasures\_collected += 1

item.kill() # remove from world

# pygame.mixer.Sound('collect.wav').play()

def use\_item(self, item\_type):

for item in self.inventory:

if item.type == item\_type:

if item\_type == 'potion':

item.use(self)

# keys are auto-used at doors, but allow manual removal if needed

self.inventory.remove(item)

return True

return False

def take\_damage(self, amount):

now = pygame.time.get\_ticks()

if now - self.\_last\_damage\_ms < DAMAGE\_COOLDOWN\_MS:

return # still in cooldown

self.\_last\_damage\_ms = now

self.health = max(self.health - amount, 0)

# pygame.mixer.Sound('hurt.wav').play()

# =========================

# Enemy

# =========================

class Enemy(pygame.sprite.Sprite):

"""Simple patrolling enemy with random movement."""

def \_\_init\_\_(self, x, y):

super().\_\_init\_\_()

self.image = pygame.Surface((TILE\_SIZE, TILE\_SIZE))

self.image.fill(RED)

self.rect = self.image.get\_rect(topleft=(x \* TILE\_SIZE, y \* TILE\_SIZE))

def patrol(self, level):

directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

dx, dy = random.choice(directions)

new\_x = self.rect.x + dx \* TILE\_SIZE

new\_y = self.rect.y + dy \* TILE\_SIZE

if level.is\_valid\_move(new\_x // TILE\_SIZE, new\_y // TILE\_SIZE, self):

self.rect.x = new\_x

self.rect.y = new\_y

# =========================

# Level

# =========================

class Level:

"""Tile-based map with walls (1) and locked door (2)."""

def \_\_init\_\_(self):

# 0: open, 1: wall, 2: locked door

self.map = [

[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],

[1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],

[1,0,1,0,1,0,1,1,1,1,1,1,1,1,0,1,1,1,0,1],

[1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1],

[1,0,1,1,1,1,1,1,1,0,1,1,0,1,1,1,0,1,0,1],

[1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,1],

[1,1,1,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1],

[1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,1],

[1,0,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1],

[1,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1],

[1,0,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1,1,0,1],

[1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,1],

[1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,0,1,0,1],

[1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,1], # 2 = door

[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]

]

self.exit\_pos = (18, 13) # door position

def is\_valid\_move(self, x, y, entity):

if 0 <= x < LEVEL\_WIDTH and 0 <= y < LEVEL\_HEIGHT:

tile = self.map[y][x]

if tile == 1:

return False # Wall

if tile == 2: # Door

# Player with a key unlocks and proceeds

if isinstance(entity, Player) and entity.keys > 0:

entity.keys -= 1

self.map[y][x] = 0 # Unlock door

return True

return False

return True

return False

def draw(self, screen):

for y in range(LEVEL\_HEIGHT):

for x in range(LEVEL\_WIDTH):

if self.map[y][x] == 1:

pygame.draw.rect(

screen, (100, 100, 100),

(x \* TILE\_SIZE, y \* TILE\_SIZE, TILE\_SIZE, TILE\_SIZE)

)

elif self.map[y][x] == 2:

pygame.draw.rect(

screen, (139, 69, 19),

(x \* TILE\_SIZE, y \* TILE\_SIZE, TILE\_SIZE, TILE\_SIZE)

)

# =========================

# Game

# =========================

class Game:

"""Main game orchestrator."""

def \_\_init\_\_(self):

pygame.init()

self.screen = pygame.display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT))

pygame.display.set\_caption("Treasure Hunter Quest")

self.clock = pygame.time.Clock()

self.font = pygame.font.SysFont(None, 30)

self.running = True

self.state = "playing" # playing | win | lose

self.start\_time = time.time()

self.\_last\_health\_drain\_at = self.start\_time

self.level = Level()

# Start near bottom-left for safer spawn or (1,1) if you prefer:

# self.player = Player(1, 13)

self.player = Player(1, 1)

self.all\_sprites = pygame.sprite.Group(self.player)

# Items

self.items = pygame.sprite.Group(

Potion(3, 2), Key(5, 5), Treasure(10, 10),

Potion(15, 3), Key(8, 12), Treasure(2, 8), Treasure(17, 7)

)

# Enemies

self.enemies = pygame.sprite.Group(

Enemy(10, 5), Enemy(15, 10), Enemy(5, 12)

)

self.all\_sprites.add(self.items, self.enemies)

# High score file

self.high\_score\_file = "highscore.txt"

if not os.path.exists(self.high\_score\_file):

with open(self.high\_score\_file, 'w') as f:

f.write("0")

# ---------------------

# Main loop

# ---------------------

def run(self):

while self.running:

self.clock.tick(60)

self.handle\_events()

if self.state == "playing":

self.update()

self.render()

pygame.quit()

sys.exit()

# ---------------------

# Event handling

# ---------------------

def handle\_events(self):

for event in pygame.event.get():

if event.type == pygame.QUIT:

self.running = False

if event.type == pygame.KEYDOWN:

if self.state == "playing":

if event.key == pygame.K\_w:

self.player.move(0, -1, self.level)

elif event.key == pygame.K\_s:

self.player.move(0, 1, self.level)

elif event.key == pygame.K\_a:

self.player.move(-1, 0, self.level)

elif event.key == pygame.K\_d:

self.player.move(1, 0, self.level)

elif event.key == pygame.K\_p:

self.player.use\_item('potion')

if self.state in ("win", "lose") and event.key == pygame.K\_r:

self.\_\_init\_\_() # Restart

# ---------------------

# Update logic

# ---------------------

def update(self):

# Enemy movement (randomly, to slow the AI)

if random.randint(1, 10) == 1:

for enemy in self.enemies:

enemy.patrol(self.level)

# Item collection

item\_hits = pygame.sprite.spritecollide(self.player, self.items, False)

for item in item\_hits:

self.player.collect\_item(item)

# Enemy collisions (with cooldown inside take\_damage)

enemy\_hits = pygame.sprite.spritecollide(self.player, self.enemies, False)

if enemy\_hits:

self.player.take\_damage(10)

# Health drain over time (once every HEALTH\_DRAIN\_PERIOD seconds)

now = time.time()

if now - self.\_last\_health\_drain\_at >= HEALTH\_DRAIN\_PERIOD and self.state == "playing":

self.player.take\_damage(1)

self.\_last\_health\_drain\_at += HEALTH\_DRAIN\_PERIOD

# Time check

elapsed = now - self.start\_time

if elapsed > TIME\_LIMIT:

self.state = "lose"

# Win check: have all treasures AND stand on exit tile (after unlocking)

if (self.player.treasures\_collected == 3 and

(self.player.rect.x // TILE\_SIZE, self.player.rect.y // TILE\_SIZE) == self.level.exit\_pos):

self.state = "win"

score = max(0, int((TIME\_LIMIT - elapsed) \* (self.player.health / 100)))

self.save\_high\_score(score)

if self.player.health <= 0:

self.state = "lose"

# ---------------------

# Render

# ---------------------

def render(self):

self.screen.fill(BLACK)

self.level.draw(self.screen)

self.all\_sprites.draw(self.screen)

# UI

health\_text = self.font.render(f"Health: {self.player.health}", True, WHITE)

self.screen.blit(health\_text, (10, 10))

time\_left = max(0, TIME\_LIMIT - int(time.time() - self.start\_time))

time\_text = self.font.render(f"Time: {time\_left}", True, WHITE)

self.screen.blit(time\_text, (10, 40))

inv\_counts = {

'potion': sum(1 for i in self.player.inventory if i.type == 'potion'),

'key': self.player.keys, # keys might be in inv or counted; we show count

}

inv\_text = self.font.render(

f"Inventory: {len(self.player.inventory)} (Potions: {inv\_counts['potion']}, Keys: {inv\_counts['key']})",

True, WHITE

)

self.screen.blit(inv\_text, (10, 70))

treasures\_text = self.font.render(f"Treasures: {self.player.treasures\_collected}/3", True, WHITE)

self.screen.blit(treasures\_text, (10, 100))

if self.state == "win":

win\_text = self.font.render("You Win! Press R to Restart", True, GREEN)

self.screen.blit(win\_text, (SCREEN\_WIDTH//2 - 170, SCREEN\_HEIGHT//2))

elif self.state == "lose":

lose\_text = self.font.render("Game Over! Press R to Restart", True, RED)

self.screen.blit(lose\_text, (SCREEN\_WIDTH//2 - 190, SCREEN\_HEIGHT//2))

pygame.display.flip()

# ---------------------

# High score

# ---------------------

def save\_high\_score(self, score):

try:

with open(self.high\_score\_file, 'r') as f:

current = int(f.read().strip() or "0")

except (ValueError, FileNotFoundError):

current = 0

if score > current:

with open(self.high\_score\_file, 'w') as f:

f.write(str(score))

# =========================

# Unit Tests

# =========================

class TestGame(unittest.TestCase):

def setUp(self):

# Optional: headless mode for CI

# os.environ["SDL\_VIDEODRIVER"] = "dummy"

self.game = Game()

self.player = self.game.player

def test\_health\_increase(self):

self.player.health = 60

potion = Potion(0, 0)

self.player.collect\_item(potion)

self.player.use\_item('potion')

self.assertEqual(self.player.health, 80)

def test\_damage(self):

self.player.health = 100

# Reset cooldown to allow immediate damage

self.player.\_last\_damage\_ms = -DAMAGE\_COOLDOWN\_MS

self.player.take\_damage(10)

self.assertEqual(self.player.health, 90)

def test\_collection(self):

item = Item(0, 0, 'treasure')

initial\_len = len(self.player.inventory)

self.player.collect\_item(item)

self.assertEqual(len(self.player.inventory), initial\_len + 1)

self.assertEqual(self.player.treasures\_collected, 1)

if \_\_name\_\_ == "\_\_main\_\_":

# Uncomment to run tests directly:

# unittest.main()

game = Game()

game.run()