Horrified

Advanced Programming

Professor:

Morteza YosefSanati

Repository on GitHub

Seyed Ali Hosseini - 40312358009   Seyed Ehsan Tousi - 40312358027

Contents

Introduction

This report documents the structure and functionality of a modular C++ game system designed around cooperative gameplay, strategic decision-making, and object-oriented design principles. The project incorporates multiple subsystems, each responsible for distinct roles such as character management, enemy behavior, map navigation, item handling, and game progression. The architecture follows a clear separation of concerns, allowing each class to perform a specialized task while contributing to the overall game logic.
Core elements such as heroes, villagers, monsters, and map components are encapsulated using polymorphism and composition, enabling extensibility and maintainability. The game loop coordinates sequential phases—hero actions followed by monster responses—enforced through a robust state management system. Utility classes like TUI facilitate user interaction, while TaskBoard, MonsterManager, and ItemBag ensure coherent rule enforcement and dynamic challenge scaling.
This report presents each class individually, highlighting its responsibilities, internal logic, and interactions with other components in the system. The analysis is based on the current implementation and may evolve as gameplay mechanics are refined or expanded.

game

The Game class acts as the central controller for the entire gameplay experience. It encapsulates the initialization of all core systems—such as heroes, monsters, map, villagers, items, task board, and interface—and coordinates the main event loop. The play() method manages user input, phase transitions, and win/loss condition checks.
During setup, it determines starting players, assigns roles, and randomly positions monsters based on predefined regions. Each turn consists of a structured hero phase, allowing movement, guidance, item interaction, task progress, and perk usage; followed by a monster phase, in which enemies act through the MonsterManager. The logic enforces victory conditions based on monster defeat and terror thresholds.
The class integrates tightly with UI components and encapsulates nearly all game logic within a single loop, making it an ideal place for global flow control. As the project evolves, minor updates or refactors (e.g. adding new heroes, expanding monsters, introducing game modes) are naturally accommodated through this orchestration point. It serves as the architectural backbone of the system.

TUI

The TUI class implements the text-based interface used to manage and display gameplay information in the terminal. It provides structured output for core game elements such as heroes, monsters, items, and game phases. Methods like showWelcomeScreen(), showPhase(), and showTerrorLevel() deliver visual cues to the player regarding progress and current conditions.
The class supports command interaction through askForCommand() and contextual input via askForNumber() and askForLocationName(), allowing flexible user-driven gameplay. It formats character and item data using functions like showHeroInfo(), showMonsterInfo(), and showItemsOnBoard(), integrating data from related modules (Map, ItemBag, etc.) for accurate presentation.

Additional methods like showFullGameUI() and showMapWithHeroInfo() combine visual elements into synchronized layouts, showing map data alongside hero status. The UI also adapts to system type (Windows or Unix) via clearScreen() for a consistent user experience. Though still under development, the class establishes a modular and expandable structure for non-graphical game interaction.

Location

The Location class models a distinct point on the game map. Each location contains a unique name, a list of neighboring connections for movement logic, characters currently present at that site, and a collection of items available there. Neighbor additions are validated to avoid duplication or self-referencing, ensuring proper map structure. Items and characters can be dynamically added or removed, with checks in place to maintain integrity. The class also offers functions for querying its current state and managing inventory via operations like clearItems(). It serves as the primary unit for navigation, interaction, and state representation within the game environment.

Map

The Map class defines the spatial layout and movement logic of the game. It maintains a collection of Location objects mapped by name and enables construction of the game world by adding locations and setting neighbor relationships. Core functionalities include retrieving specific locations, finding the location with the highest item count, and computing path-based distances using breadth-first search. The method findCloserLocation() helps determine directional movement by choosing the optimal neighbor that brings a character closer to a target. These operations support strategic navigation and gameplay mechanics across the game board.

Town

The Town class models a location on the game map. Each town holds a name, a list of neighboring towns, and a collection of characters currently present. Neighboring relationships are maintained bidirectionally to support pathfinding and movement logic. The class provides safe interfaces for retrieving its name, neighbors, and inhabitants, while enforcing constraints when adding or removing characters to prevent duplication or invalid operations. It also ensures that each neighbor is unique and valid, contributing to consistent map topology. This component plays a foundational role in representing the game world's spatial structure.

Item

The Item class represents in-game objects that heroes can pick up and use. Each item has a name, color (Red, Blue, or Yellow), and a power value. It also tracks its location on the map and can be reassigned to different places. The static method colorToString() converts item color enums into readable text for display or logging. Items are categorized for strategic use based on their color and effect.

ItemBag

The ItemBag class handles item pool management. It stores both initial and current items, seeds a random number generator, shuffles the collection, and distributes items across the map during setup or refills. It ensures variety and balance by avoiding duplicate active instances and reintroducing unused ones. The method drawRandomItem() selects an item, assigns it to its location, and updates internal tracking. This class supports dynamic item flow and gameplay diversity.

Hero

The Hero class models a player-controlled character in the game, equipped with actions, location, inventory, and special abilities. Each hero starts with a customizable number of actions and can interact with game elements such as villagers, items, perk cards, and monsters. Key behaviors include moving across the map, guiding villagers, picking up items, advancing quest progress, and attempting to defeat enemies. Perk cards enhance hero capabilities and are fully integrated into their action flow. Item management allows for strategic combat and puzzle-solving. The class also supports polymorphism via a pure virtual specialAction() method, allowing for custom behavior in subclasses. Game control and turn progression revolve around each hero's state and decisions.

Mayor

The Mayor subclass represents a hero with high action capacity (5 actions per turn) but no unique ability. Its constructor sets the name and starting location, while the specialAction() function is intentionally unimplemented, throwing an exception if called. This design emphasizes standard gameplay behavior and strategic flexibility rather than special mechanics.

Archeologist

The Archeologist subclass enables a hero to collect items from neighboring locations. Its specialAction() scans all adjacent areas, lists available items, and lets the player select and retrieve them. This ability enhances item accessibility, supporting faster preparation for defeating monsters or completing tasks. The archeologist starts with 4 actions and focuses on exploration and resource gathering.

Villager

The Villager class represents non-playable residents that players must guide to safety. Each villager has a name and a location on the map. Movement can occur either through player guidance or monster actions, with validation against neighbor connections. Upon reaching their predefined safe zone, the villager exits the game and may reward the guiding hero with a random perk card. Safety checks are performed per character based on fixed location logic. Error handling ensures robustness for invalid transitions or missing references.

VillagerManager

The VillagerManager class oversees all villager instances. It uses a hash map to store and retrieve villagers by name and offers access to the full list for rendering or logic traversal. It supports adding new villagers with assigned locations and provides safe lookup functionality for other game systems.

Monster

The Monster class serves as a base for enemy characters on the map. Each monster has a name and current position, and can move toward nearby non-monster characters using shortest-path logic. The method moveToNearestCharacter() calculates a valid route and updates its position. In combat, attack() determines whether to target a hero or villager, handles item usage for defense, applies consequences (such as sending wounded heroes to the hospital), and increases the terror level on successful strikes. The power() method is abstract, allowing each derived monster to define its unique ability. The class is designed to interact deeply with map state, hero behavior, and global threat metrics.

Dracula

Dracula The Dracula class inherits from Monster and features a special ability called “Dark Charm.” This power forces a hero to teleport directly to Dracula’s location if they aren’t already there, repositioning them on the map and potentially exposing them to danger. The action manipulates hero movement without causing direct harm, enhancing control and tactical pressure.

Invisibleman

InvisibleMan The InvisibleMan class adds stealth and surprise to gameplay. Its “Stalk Unseen” ability allows it to instantly kill a villager present at its location, raising the terror level. Additionally, it includes moveTowardsVillager() which searches for the nearest unprotected civilian and navigates toward them in calculated steps. This design creates suspense and makes the Invisible Man a persistent off-screen threat.

FrenzyMarker

The FrenzyMarker class controls the current frenzied monster in gameplay. It keeps an ordered list of monster references and designates one as frenzied, enabling special behaviors or prioritization during the monster phase. The marker initially targets Dracula, and its advance() method cycles the frenzied state through available monsters, accounting for their presence on the board. If Dracula is not active (i.e. has no location), the second monster is disregarded and Dracula is reassigned. This class ensures dynamic behavior and tension escalation by rotating the frenzy state logically.

MonsterManager

The MonsterManager class coordinates the behavior of monsters during their active phase. It holds a set of MonsterCard`s, provides card drawing and shuffling utilities, and executes full monster logic through the `MonsterPhase() method. This includes spawning items, triggering events, handling strikes, updating terror level, and managing interactions with heroes and villagers. It also ensures frenzy mechanics are respected. The method moveVillagersCloserToSafePlaces() facilitates automatic movement toward safety during relevant events, making this class a central orchestrator of enemy logic and board tension.

MonsterCard

The MonsterCard class defines enemy actions during the monster phase. Each card holds a name, item count, event description, and a list of strike instructions that specify which monster attacks, how far it moves, and how many dice it rolls. This structure allows for flexible monster behaviors and precise game event execution.
monster turns throughout gameplay.

PerkCard

The PerkCard class represents special abilities available to heroes during gameplay. Each card is tied to a PerkType such as Hurry or Repel, and includes a descriptive effect. Cards are categorized and identified through the perkTypeToString() method, supporting clean terminal display and logic handling. The class is simple and designed for portability across game components.

PerkDeck

The PerkDeck class manages a randomized stack of PerkCard`s. It generates default cards based on game rules, shuffles them with a time-seeded `mt19937, and provides methods to draw, inspect, or check availability. Cards are drawn one at a time, with error handling for empty deck cases. This class ensures dynamic gameplay by controlling the flow of hero abilities.

TerrorTracker

The TerrorTracker class manages the city's fear level during gameplay. It starts from 0 and can increase up to 5, triggering game loss if maxed out. The increase() method raises the terror level without exceeding the cap. It's a simple yet crucial class used for monitoring progression and threat intensity throughout the game.

Dice

The Dice class simulates monster-phase dice rolls using a random generator. It returns one of three outcomes—Power, Strike, or Empty—based on a random value. Only 0 and 5 lead to special effects, making the roll logic simple but effective. The faceToString() method converts these outcomes into symbols ("!", "*", or blank) for TUI display. It's a lightweight and standalone component.

TaskBoard

The TaskBoard class tracks mission objectives tied to defeating Dracula and the Invisible Man. For Dracula, it manages destruction progress for four specific coffin locations, each requiring six points of strength. Once all coffins are destroyed, players can target Dracula directly. The class also records accumulated attack strength against him until he is officially defeated.
For the Invisible Man, the board monitors delivery of clues at five locations. Once all clues are delivered, players can begin attacking him, requiring nine points of red strength to defeat. Task completion is monitored with boolean flags and strength counters, while status-check methods provide live progress feedback for UI or reporting purposes. This class centralizes all win-condition logic related to boss monsters.

Challenges

Object Management: Handling lifetimes of shared objects like heroes and monsters without memory leaks.

Input Buffer Issue: Sometimes user input was misread due to leftover characters from previous cin operations.

Pathfinding: Designing monster movement to target villagers accurately across linked locations.

Hero-Villager Interaction: Synchronizing guidance events with perk rewards while avoiding tight coupling.

Text-Based UI: Presenting dynamic game states clearly in terminal output with limited space.

Game Phase Control: Coordinating hero and monster phases under multiple win/lose conditions.

Monster Frenzy Rotation: Ensuring frenzy state advances safely when monsters are defeated or inactive.

Solutions

Object Management: Used shared_ptr throughout to ensure safe ownership and cleanup.
Pathfinding: Applied BFS algorithm for efficient monster navigation across locations.

Hero-Villager Interaction: Centralized villager logic in VillagerManager to decouple responsibilities.

Text-Based UI: Split UI methods into modular functions to improve clarity and reusability.

Game Phase Control: Structured the play() method with distinct blocks and clear phase transitions.

Input Buffer Issue: Used cin.ignore(numeric_limits<streamsize>::max(), '\n') to discard leftover input. This ensures that any remaining characters (e.g. newline or unintended input) are cleared before the next cin, preventing logic errors and repeated prompts.

Monster Frenzy Rotation: Advanced frenzy state using indexed lists and null-checks for robustness.

Resources

1. Stack Overflow
2. Chatgpt
3. YouTube videos
4. Cpp reference
5. Opensource website