Ben Vieten, Andrew Sparks, Kyan Chase

Professor King

Programming Languages and Design

4/14/25

Report: Ruby Text-Based Adventure Game Implementation

This program creates a text-based adventure game where players explore rooms, battle enemies, and manage items. The game uses YAML files to save progress and configure settings like player health or enemy types. A terminal interface (built with the `curses` library) displays menus, battles, and events such as finding treasure, solving puzzles or unique events found within each room.

Ruby's features simplified several tasks. Saving and loading data was straightforward with YAML, which converted game objects (like player stats and location) into readable text files. The `method_missing` method made configuration easy by dynamically reading settings (e.g., starting gold) from a YAML file without rigidly defining every property. Modules like `InventoryUtils` kept code organized, allowing items to behave differently (e.g., healing potions vs. weapons) using Ruby's flexible arrays and hashes. Mixed-type data structures streamlined the inventory system. Items are stored as strings (e.g., "Healing Potion"), arrays for stacked items (e.g., ["Fresh Fish", 3]), or hashes for unique effects (e.g., { name: "Ancient Relic", permanent: true }). Ruby's flexibility with arrays allowed these types to coexist without rigid class definitions, making it easy to add or modify items.

Regular expression matching parsed player commands and item usage. For example, the correct_input method uses regex to interpret typos like ‘nroth’ as ‘north’, while item logic matches inputs like ‘use potion’ case-insensitively.

However, some aspects were challenging. The `curses` library required manual control of screen updates, making menus and combat displays time-consuming to build. Tracking changing game states—like player health, inventory updates, or room transitions—led to bugs due to Ruby’s mutable objects. Additionally, ensuring that the terminal pauses after each interaction to ensure the player is aware of what’s happening was difficult. Mainly, because we had to ensure that the ‘tui.pause’ function was called only once by the callee and caller to avoid redundancy. Handling player input was tricky, especially fixing typos (e.g., interpreting "nroth" as "north") using the `levenshtein` gem, which added extra code.

Blocking was another major Ruby feature that we found challenging at first to implement. Blocks are chunks of code that can be passed to methods and executed later using yield, a concept we leveraged to make our input prompts cleaner and more reusable. For example, we applied this idea in our BlockUtils.prompt_loop function to continuously prompt the player until valid input was provided, allowing us to pass a block containing the validation logic. This greatly improved code readability and reduced redundancy.