Riddles & Ruins - Complete Documentation

Table of Contents

Game Documentation

[Content from README.md follows]

Riddles & Ruins

Game Overview

"Riddles & Ruins" is an engaging text-based adventure game where players navigate through a series of dungeons, solving riddles and engaging in combat encounters. The game emphasizes strategic decision-making, resource management, and player choice through branching paths.

Game Mechanics

Dungeon Paths

Players must choose one of two possible paths to complete the game:

- Path 1: Dungeons [1 → 3 → 5 → 6]
- Path 2: Dungeons [1 → 2 → 4 → 6]

Quest System

Each dungeon contains specific quests that must be completed to progress. There are two types of quests:

1. Riddle Quests

- Players are presented with a riddle to solve
- Rewards:
  - Correct answer: Receive a Health Potion (+20 HP when used)
  - Skip option: -20 HP penalty

2. Combat Quests

Combat is probability-based with three difficulty levels:

Level 1

- Win: -5 HP + Battle Potion reward

- Lose: -10 HP
- Skip: -20 HP penalty

Level 2

- Win: -10 HP + Battle Potion reward
- Lose: -20 HP
- Skip: -20 HP penalty

Level 3

- Win: -15 HP + Battle Potion reward
- Lose: -30 HP
- Skip: -30 HP penalty

Items

1. Health Potion
    - Restores 20 HP
    - Obtained by solving riddles
    - Can be used at any time
2. Battle Potion
    - Provides complete protection during combat encounters
    - When active, prevents all health point loss
    - Protects against combat outcomes (win/loss) and skip penalties
    - Can be used at any time during the game
    - Effect lasts for one combat encounter

Technical Implementation

Project Structure

The game is built using a modular, object-oriented approach with the following key components:

```
adventure-game/
├── src/
│   ├── main.py           # Game entry point
│   ├── game.py           # Core game loop
│   ├── player.py         # Player state management
│   ├── quest.py          # Quest tracking system
│   ├── dungeons/         # Dungeon implementations
│   ├── encounters/       # Combat and puzzle systems
│   ├── items/           # Inventory management
│   └── utils/           # Helper functions
```

Key Classes

- `Player`: Manages player state, inventory, and progress
- `Game`: Controls game flow and dungeon progression
- `Quest`: Handles quest objectives and completion
- `Dungeon`: Base class for dungeon implementations
- `Inventory`: Manages items and usage

Development Journey

1. Initial Structure
    - Basic project skeleton created using GitHub Copilot
    - Core files and basic functionality established
2. Enhancements
    - Added modular design patterns
    - Implemented inheritance for dungeons
    - Created comprehensive quest system
    - Added inventory management
    - Implemented branching paths
3. Optimizations
    - Improved path progression logic
    - Enhanced combat system
    - Added robust error handling
    - Implemented save/load functionality

Strategic Elements

Player Choices

1. Path Selection
    - Two distinct paths offer different challenges
    - Each path requires unique strategies
2. Combat Decisions
    - Fight: Risk vs. reward mechanics
    - Skip: Guaranteed penalty but avoid uncertainty
3. Resource Management
    - Health points management
    - Strategic use of health potions
    - Balancing risk in combat

Tips for Players

1. Health Management

    - Keep track of your HP
    - Save health potions for critical moments
    - Consider skip penalties vs. combat risks

2. Path Strategy

    - Plan your route before starting
    - Each path has different difficulty curves
    - Consider your playstyle (combat vs. puzzles)

3. Combat Tips

    - Higher levels have higher risks and rewards
    - Use battle potions wisely
    - Skip if health is critically low

Development Notes

- Modular design allows easy addition of new dungeons
- Quest system supports multiple objective types
- Combat system uses probability-based outcomes
- Inventory system handles item management
- Save system tracks progress across sessions

Testing Framework

Unit Test Architecture

The project implements a comprehensive testing suite covering all major components:

1. Player Testing (`test_player.py`)

    - Health system validation
    - Inventory management
    - Dungeon progression tracking
    - Stage completion logic

2. Game Testing (`test_game.py`)

    - Game initialization
    - Path validation
    - Dungeon progression
    - Game completion states
    - Invalid state handling

3. Quest Testing (`test_quest.py`)

    - Quest creation and tracking
    - Objective completion
    - Progress monitoring
    - Multi-objective quests

4. Inventory Testing (`test_inventory.py`)
    - Item addition/removal
    - Item usage mechanics
    - Inventory display
    - Item effect validation

Test Execution

Run tests using pytest with these commands:

```
# Full test suite
python -m pytest tests/

# Specific component
python -m pytest tests/test_game.py

# Individual test
python -m pytest tests/test_game.py::test_game_dungeon_progression

# Verbose output
python -m pytest -v tests/

# With print statements
python -m pytest -s tests/
```

Test Coverage

The test suite validates:

- Valid path combinations ([1,3,5,6] and [1,2,4,6])
- Player health and inventory mechanics
- Quest completion scenarios
- Game state transitions
- Error handling and edge cases
- Combat and riddle outcomes
- Item usage effects

Test Design Principles

1. Isolation: Each test focuses on a specific feature
2. Independence: Tests can run in any order
3. Repeatability: Consistent results across runs
4. Clarity: Clear test names and documentation
5. Coverage: Tests both success and failure cases

Future Enhancements

- Additional dungeon paths

- More quest types
- Enhanced combat mechanics
- New items and rewards
- Achievement system
- Character classes
- Difficulty settings

Development with AI

[Content from WORKING_WITH_AI.md follows]

Initial Project Structure Generation

GitHub Copilot successfully created the basic structure of the adventure game project, generating:

- Core game files (`main.py`, `game.py`, `player.py`, `quest.py`)
- Location-based files (`location.py` and location folder with 3 location implementations)
- Basic interconnections between components

Strengths and Limitations

What Worked Well

1. Basic Project Setup
    - Quick generation of essential files
    - Basic implementation of game mechanics
    - Initial interconnection of components
    - Random location selection mechanism
2. Code Optimization
    - Helped optimize path progression logic
    - Improved dungeon navigation system
    - Enhanced player state management
    - Assisted with documentation generation
3. Documentation
    - Generated comprehensive README.md
    - Created project documentation structure
    - Saved significant manual documentation time
4. Testing Framework
    - Generated complete test suite structure
    - Created test cases for all major components
    - Implemented test fixtures and assertions
    - Significantly reduced testing development time

    - Only minor adjustments needed for:
        - Import path corrections
        - Additional edge cases
        - More comprehensive assertions
        - Test scenario coverage

Areas Needing Human Input

1. Architecture Decisions
    - Inheritance patterns for locations not implemented initially
    - Needed manual intervention for proper modularization
    - Required restructuring for better code reusability
2. Core Functionality
    - Basic random location system needed enhancement
    - Combat and quest systems required refinement
    - Inventory system needed more sophisticated implementation
3. Code Organization
    - Initial location system lacked inheritance structure
    - Needed better separation of concerns
    - Required manual restructuring for maintainability

Iterative Development with Copilot

1. Initial Generation
    - Basic file structure
    - Simple location system
    - Basic game loop
2. Human-Guided Improvements
    - Added proper inheritance for locations
    - Enhanced modularization
    - Implemented core gameplay features
3. Final Refinements
    - Optimized path progression
    - Enhanced documentation
    - Improved code organization

Lessons Learned

1. AI Strengths
    - Quick project bootstrapping
    - Basic code generation

    - Documentation assistance
    - Code optimization suggestions

2. Human Input Needed For

    - Architecture decisions
    - Design patterns implementation
    - Complex game mechanics
    - Code organization

3. Best Practices

    - Start with clear requirements
    - Review and refine AI-generated code
    - Guide AI with specific context
    - Iterate on generated solutions

Time Savings

- Rapid initial project setup
- Quick documentation generation
- Efficient code optimization
- Reduced boilerplate coding
- Substantial testing time reduction:
    - Automated test suite generation
    - Pre-built test scenarios
    - Test fixtures and helper functions
    - Common test patterns implemented
    - Only minimal adjustments needed

Conclusion

While GitHub Copilot provided an excellent starting point and saved considerable development time, human expertise was crucial for:

- Implementing proper design patterns
- Ensuring code modularity and reusability
- Refining core gameplay mechanics
- Creating a more sophisticated structure

The combination of AI assistance and human oversight resulted in a more robust and maintainable codebase than either could have achieved alone.