MIA's RPG System
Design & Programming Documentation

Developer: Antonius Torode

Version – Version 0.001
Latest update: July 13, 2025

This document provides documentation for an RPG system designed within Multiple Integrated Applications (MIA), a program created for personal use. Herein, "MIA" refers to "Multiple Integrated Applications." This acronym is used exclusively for this program and was coined by its creator. The RPG system within MIA is built to support flexible, expandable gameplay mechanics, allowing continuous addition of features and adaptability for various gaming scenarios and purposes.

This document is continuously under development.
Most Current Revision Date: July 13, 2025

Torode, A.
MIA's RPG Docs.
2025.
Includes Source Code and References

Contents

1 Introduction **1**

2 Design **2**
2.1 Goals 2
2.2 Configuration 4
2.2.1 Vision 4
2.2.2 Configuration Design 6

This page intentionally left blank.
(Yes, this is a contradiction)

Chapter 1

Introduction

Chapter 2

Design

This chapter will cover design ideas for the MIA RPG system. This chapter will be considered a living document which may change very often. Ideas, the purpose for the ideas, and different solutions or considerations for design will be discussed.

2.1 Goals

The following goals define the core vision for the MIA RPG system, ensuring it is modular, scalable, and simple to extend for MMO development and game creation tools. Each goal includes its purpose and design considerations to guide implementation.

1. **Fully Configurable Framework**
    - **Purpose**: Enable dynamic addition or removal of game elements (e.g., vitals, currencies, items, abilities) via configuration files, minimizing code changes and supporting tools like a game editor similar to Warcraft 3's.
    - **Considerations/Solutions**:
        - Use JSON/YAML files (e.g., `vitals.json`, `items.json`) to define all game elements, including properties like name, type, and behavior.
        - Implement registries (e.g., `VitalsRegistry`, `CurrencyRegistry`) to load and manage configurations at runtime.
        - Support runtime updates via an API or scripting (e.g., Lua) for editor-driven additions.
        - Example: Add a "stamina" vital via the editor, updating `vitals.json` and reflecting in-game without recompiling.
2. **Editor-Friendly Design**
    - **Purpose**: Facilitate a game creation editor by exposing all configurable elements (vitals, items, loot tables) in a user-friendly format, enabling non-programmers to build levels, quests, or mechanics.
    - **Considerations/Solutions**:
        - Develop a GUI editor that reads/writes config files, with templates for common elements (e.g., health vital, gold currency).
        - Provide scripting support (e.g., Lua) for custom behaviors, accessible via editor UI.
        - Include validation tools to ensure config integrity (e.g., valid ranges for vital max/min).
        - Example: Editor allows dragging a "sword" item into a creature's loot table, updating `loot.json`.
3. **Interconnected Gameplay Elements**
    - **Purpose**: Create a rich, MMO-like experience where elements like vitals, items, and environments interact dynamically (e.g., temperature vital affected by climate zones, loot tables yielding currencies/items).

- **Considerations/Solutions**:
  - Use an Entity-Component-System (ECS) to manage entities (players, NPCs, zones) with components like vitals, inventory, or stats.
  - Define relationships in config files (e.g., `climate.json` links zones to temperature effects).
  - Implement event-driven interactions (e.g., `onEnterZone` adjusts temperature vital).
  - Example: A "frozen tundra" zone reduces the "temperature" vital, countered by equipping a "fur cloak" item.

4. **Scalable MMO Infrastructure**

- **Purpose**: Support MMO-scale features like large player counts, persistent worlds, and networked data access for vitals, inventories, and quests.
- **Considerations/Solutions**:
  - Design a server-client architecture with a database backend (e.g., SQLite) for persistent storage of player data.
  - Use efficient data structures (e.g., `std::unordered_map`) for fast access to vitals/currencies.
  - Enable networked updates via events (e.g., `onVitalChanged` syncs health across clients).
  - Example: Player's "gold" currency updates in real-time across server and clients after looting.

5. **Simplicity for Expansion**

- **Purpose**: Ensure new elements (e.g., vitals, items) can be added via configs or editor without code changes, maintaining simplicity for developers and modders.
- **Considerations/Solutions**:
  - Centralize element management in registries with generic APIs (e.g., `addVital(name, config)`).
  - Use data-binding to link configs to game systems (e.g., UI, combat) automatically.
  - Support modding via external config files or scripts, validated at load time.
  - Example: Adding a "morale" vital via editor updates UI and gameplay without altering codebase.

6. **Comprehensive Core Mechanics**

- **Purpose**: Cover all essential RPG mechanics (combat, progression, crafting, exploration) with configurable parameters to support diverse MMO features.
- **Considerations/Solutions**:
  - Define core systems (e.g., combat, inventory, questing) in config files with customizable rules (e.g., damage formulas, drop rates).
  - Support loot tables linking items, currencies, and probabilities (e.g., `loot.json` for creature drops).
  - Include environmental effects (e.g., climate impacting vitals like temperature or thirst).
  - Example: A "dragon" loot table includes "gold" currency and "dragon scale" item, with configurable drop chances.

7. **Robust and User-Friendly UI**

- **Purpose**: Provide a dynamic UI that reflects configurable elements and supports MMO-scale interactions (e.g., trading, party management).
- **Considerations/Solutions**:
  - Use a UI config file (e.g., `ui_layout.json`) to define displays for vitals, inventories, etc.
  - Implement data-binding to update UI automatically when elements change (e.g., new vital added).
  - Support customizable HUDs via editor for player-specific layouts.
  - Example: A "health" progress bar auto-updates when a new vital is added via `vitals.json`.

2.2 Configuration

This section details the configuration strategy for the MIA RPG system, focusing on simplicity for testing, scalability for MMO deployment, and support for a game creation editor. Configuration is designed to allow dynamic addition and removal of game elements (e.g., vitals, items, loot tables) via JSON files during development, with a compressed format for release and robust server-side validation to prevent tampering.

2.2.1 Vision

The configuration system uses JSON files in code folders to define example items, vitals, loot tables, and other objects for testing and development. These will be compressed into a single, MPQ-like format for game releases, with tools to view and edit the archive, supporting editor-driven game creation. Server-side database features will store character-specific data and save states, with consideration for including item definitions to prevent tampering. A file-based "saveToFile" and "loadFromFile" system will support player data storage during testing. These files can even be stored locally for the release game, and hased values can be checked against the server-side saves to detect tampering and restore changed or damaged files.

The configuration system will be structured to support the following objectives, ensuring simplicity, editor compatibility, and MMO-scale robustness.

1. **JSON-Based Configuration for Testing**
   - **Purpose**: Enable rapid iteration during development by using human-readable JSON files to define game elements, facilitating testing and editor integration.
   - **Design**:
     - Store JSON files in relevant folders (e.g., `data/vitals/vitals.json`, `data/items/items.json`, `data/loot/loot.json`).
     - Example `vitals.json`:

```
[
  {"id": "health", "name": "Health", "current": 100, "max": 100, "min": 0},
  {"id": "mana", "name": "Mana", "current": 50, "max": 100, "min": 0}
]
```

     - Example `items.json`:

```
[
  {"id": "sword_001", "name": "Iron Sword", "type": "weapon", "damage": 10, "tradeable": true},
  {"id": "potion_001", "name": "Health Potion", "type": "consumable",
   "effect": {"vital": "health", "value": 20}}
]
```

     - Load files into registries (e.g., `VitalsRegistry`, `ItemRegistry`) using a JSON parser (e.g., nlohmann/json in C++).
     - Include validation for required fields (e.g., `id`, `max`) and default values for optional fields (e.g., `min=0`).
2. **Compressed Format for Release**
   - **Purpose**: Optimize game releases by bundling JSON configurations into a single, compressed archive (similar to WoW's MPQ) to reduce file count and improve load times.
   - **Design**:
     - Use a compression library (e.g., libzip) to create an archive (e.g., `game_data.mia`) containing all JSON files.
     - Include a manifest file (e.g., `manifest.json`) listing contents for quick access:

```
{
  "files": ["vitals.json", "items.json", "loot.json"]
}
```

- Develop tools (e.g., C++ or Python-based editor) to extract, view, and edit the archive, supporting the game creation editor.
- Example: Editor extracts `items.json`, allows adding a new item, and recompresses into `game_data.mia`.

3. **Server-Side Database with Tamper Checks**

- **Purpose**: Store character-specific data and item definitions server-side to ensure persistence and prevent tampering in MMO environments.
- **Design**:
  - Use a database (e.g., SQLite for testing, scalable DBMS for release) to store player data (e.g., vitals, inventory) and static item definitions.
  - Example `items` table:

```
CREATE TABLE items (
    id VARCHAR(50) PRIMARY KEY,
    name VARCHAR(100),
    type VARCHAR(50),
    properties JSON
);
```

  - Sync `items.json` with the `items` table during release to validate client actions (e.g., item usage) against server data.
  - Example: Client sends `use_item("potion_001")`, server checks `properties` (e.g., heals 20 health) and applies the effect.
  - Use checksums (e.g., SHA256) on the compressed archive to detect client-side tampering.

4. **File-Based Player Data Storage for Testing**

- **Purpose**: Provide a simple, file-based system to save and load player data during testing, mirroring future database functionality.
- **Design**:
  - Store player state in `player.json`:

```
{
  "player_id": "player_001",
  "name": "Hero",
  "vitals": [
    {"id": "health", "current": 80, "max": 100, "min": 0},
    {"id": "mana", "current": 30, "max": 100, "min": 0}
  ],
  "inventory": [
    {"item_id": "sword_001", "quantity": 1},
    {"item_id": "potion_001", "quantity": 3}
  ],
  "currencies": [
    {"currency_id": "gold", "amount": 500}
  ],
  "position": {"x": 100, "y": 200}
}
```

  - Implement `saveToFile` and `loadFromFile` in a `Player` class using nlohmann/json.
  - Example:

```
void Player::saveToFile(const std::string& path) const;
bool Player::loadFromFile(const std::string& path);
```

- Handle errors with defaults (e.g., new player with 100 health) and log issues to a debug console.
- Design fields to match future database schema for seamless migration.

2.2.2 Configuration Design

The configuration system will begin with defining various objects within a JSON configuration file. Each object will contain a unique ID, name, and description which will serve as an identifier for the object itself. The remaining fields will depend on the object's purpose and function. In order to have multiple defined types within the JSON files, an object type will be declared as well, to separate the various types.

```
{
    "OBJECTTYPE":
    [
    {
        "id": 1,
        "name": "object name 1",
        "description": "object description 1",
        // Type specific values...
    },
    {
        "id": 2,
        "name": "object name 2",
        "description": "object description 2",
        // Type specific values...
    },
    // Other values...
    ]
}
```

These object types will have associated data classes which match the data fields to store each type. These types will define the main objects of the RPG system and will need continual referencing to gather their data and use their values. Since these are configurable, they are not deterministic (except for their structure) at compile time. For this reason, a registry will be designed for each type which will load in all of the configurable values defined in the JSON files and store them in a unique singleton instance. This will provide a single location to gather this information.

The various types all will potentially store in different ways. For example, a vital may store some current value, as well as some min or max, whereas a currency may just be a single value. A separate storage class for each type will need designed which defines the stored values associated with it.