Knights Of Eldoria

Abstract

The ***KnightsOfEldoria*** project is a Python-based simulation that emulates a strategic world of hunters, knights, treasures, and hideouts on a 2D grid. The simulation features a turn-based interface where players guide hunters to locate and collect treasures while avoiding knights. The primary goal was to create an interactive, modular system utilizing Python's object-oriented principles, with support for autonomous agents, AI decision-making, real-time statistics, and robust testing mechanisms. This project demonstrates a flexible foundation for future AI-based strategy games or simulations.

1. Introduction

The ***KnightsOfEldoria*** project explores strategic agent movement, decision-making, and competition within a simulated virtual environment. The simulation consists of various entities, including treasure hunters, knights, hideouts, and treasures, each with distinct roles and logic. The project showcases Python's object-oriented capabilities and grid-based simulation mechanics, providing a foundation for the development of more complex strategy games.

Key objectives include

- Coordinating multiple agents with autonomous behavior.
- Designing realistic game mechanics and AI interactions.
- Ensuring real-time updates and handling conflicts in a dynamic world.

The project also serves as an exploration into modular game architecture, making it possible to expand with graphical user interfaces (GUIs), reinforcement learning models, or multiplayer functionality.

2. System Design & Architecture

The *KnightsOfEldoria* simulation is built on a modular Python-based system that models a 2D grid world where agents interact. The structure is organized to support independent testing, ease of extension, and intuitive updates.

2.1 Architecture Overview

CopyEdit

- KnightsOfEldoria/
- ├── src/
- │ ├── main.py # Entry point
- │ ├── simulation.py # Simulation engine
- │ ├── grid.py # Grid handling
- │ ├── hunter.py # Hunter logic
- │ ├── knight.py # Knight logic
- │ ├── treasure.py # Treasure handling
- │ ├── hideout.py # Hideout logic
- │ ├── utils.py # Utility functions
- │ └── ai_agents.py # AI decision-making

2.2 Class Design

- **Grid**: Manages the 2D grid layout and cell content.
- **Hunter**: Represents player-controlled agents with abilities like movement, resting, and treasure collection.
- **Knight**: AI-controlled agents that patrol the grid and pose threats to hunters.
- **Treasure**: Items placed on the grid with values, which hunters attempt to collect.

- **Hideout**: Safe zones where hunters can store collected treasures.

2.3 Flow Diagram (Simplified)

CopyEdit

- [Start] → [Initialize Grid] → [Spawn Entities] → [Game Loop]
- → [Hunter Actions] → [Knight Actions] → [Treasure Update] → [Render Map] → [Check End Conditions] → [Repeat or End]

3. Implementation Details

3.1 Simulation Loop (main.py)

The main entry point of the simulation is responsible for managing user input or automated actions. The loop progresses step-by-step, where the player can interact with the simulation by moving hunters, switching agents, or invoking AI behavior.

3.2 Agent Behavior (hunter.py / knight.py)

- **Hunters**: Each hunter has skills (speed, stealth) and stamina, influencing their actions. Movement costs stamina, and hunters must rest when low. Hunters are also capable of collecting treasures.

- **Knights**: These AI-controlled agents are designed to patrol the map. They reduce the stamina of hunters upon collision, thus creating challenges.

3.3 Treasure Collection

Hunters must navigate to treasure-adjacent cells. If they have sufficient stamina, they can collect treasures, updating their score and removing the treasure from the grid.

3.4 Movement Conflicts

Only one entity can occupy a cell at any time. If multiple entities attempt to occupy the same space, the simulation handles the conflict by displaying an appropriate message, such as "space occupied."

3.5 AI Agents (ai_agents.py)

The AI decision-making system provides agents with basic strategies like pathfinding to the nearest treasure or resting when stamina is low.

4. Testing & Validation

The system is thoroughly tested to ensure reliability across different modules. Testing is performed using **pytest**, and unit tests are located in the tests/ folder.

4.1 Key Tests

- **test_treasure.py**: Ensures correct spawning and value assignment of treasures.
- **test_simulation.py**: Verifies correct game loop operation and state transitions.
- **test_knight.py**: Validates knight patrol and attack behavior.
- **test_hunter.py**: Confirms correct movement, resting, and stamina handling.
- **test_hideout.py**: Checks functionality of hideouts for treasure storage.

Sample test output for hunter movement:

- Choose action: move right
- ✅ Hunter moved to (12, 15)
- 💰 Treasure collected!

5. Software Artefact Description

5.1 Key Files

- **main.py**: Starts the simulation and processes user inputs.

- **simulation.py**: Contains the game loop and turn-based mechanics.
- **grid.py**: Responsible for rendering and updating the 2D grid.
- **hunter.py**: Defines the logic for the hunter agents.
- **treasure.py**: Manages treasure placement and collection mechanics.
- **ai_agents.py**: Implements basic AI decision-making for knights and hunters.

5.2 How to Run

To run the simulation:

- Install dependencies:

  nginx
  CopyEdit
  ```
  pip install -r requirements.txt
  ```
1.
- Start the simulation.

  css
  CopyEdit
  ```
  python src/main.py
  ```
2.

5.3 Features

- Multiple hunters with skill-based abilities.
- Dynamic, emoji-rendered 2D grid map.
- Basic AI-driven knight agents.
- Hideouts for treasure storage.

6. Evaluation

6.1 What Worked Well

- **Grid Rendering**: The use of emojis to represent the grid made visualization both engaging and intuitive.
- **Modular Structure**: Each component is independently testable and easy to extend.
- **AI Logic**: Basic AI decisions (e.g., pathfinding) functioned well, adding a layer of challenge to the game.

6.2 Challenges

- Synchronizing multiple hunters working toward the same goal.
- Handling scenarios where multiple agents are adjacent to a treasure.
- Balancing stamina consumption and treasure collection logic.

6.3 Performance

The simulation performed well, even with up to 5 hunters and 5 knights on a 30x30 grid. The game loop maintained consistent frame rates and responsive input handling.

6.4 Limitations

- **No GUI**: The simulation is limited to a command-line interface (CLI).
- **Basic AI**: AI behavior relies on greedy strategies and could be expanded.
- **Static Treasures**: Treasures do not decay or respawn, limiting map dynamism.

7. Future Enhancements

- **Graphical User Interface (GUI)**: Implement a GUI using libraries like Tkinter or PyQt for a more interactive experience.
- **Reinforcement Learning**: Integrate machine learning to improve hunter decision-making and make the AI more adaptive.
- **Dynamic Treasures**: Introduce treasure decay or respawning to keep the simulation dynamic.
- **Additional Features**: Add traps, fog-of-war, weather conditions, and multiplayer support.

8. Conclusion

The *KnightsOfEldoria* project successfully applies Python's object-oriented design to create a functional and engaging 2D grid-based simulation. Despite its basic AI and limited features, it lays a solid foundation for future expansions. The modular architecture and comprehensive testing demonstrate a robust system capable of supporting further development in AI-based strategy games.

9. References

- Python Official Documentation: https://docs.python.org/
- PEP8 Guidelines: https://peps.python.org/pep-0008/
- Pytest Documentation: https://docs.pytest.org/
- Emojipedia: https://emojipedia.org/
- StackOverflow discussions on Python game loops and grid logic