Game Character Design Using OOPs

Table of contents

Abstract

This project aims to develop a sophisticated object-oriented programming framework that leverages fundamental principles of class hierarchy, inheritance, polymorphism, abstraction, encapsulation, dynamic memory allocation with smart pointers, and efficient control structures. The framework will be designed to create and manage simulation systems for diverse domains such as robotics, environmental modelling, or game development.

The utilization of class hierarchy and inheritance will facilitate the organization of diverse entities within the simulation, ensuring a clear and scalable structure. Polymorphism and virtual functions will enable flexibility in behaviour implementation, allowing interchangeable components within the system without compromising functionality.

Abstraction and encapsulation will be employed to hide complex implementation details, providing simplified interfaces for system interaction while maintaining robustness and security. Dynamic memory allocation, complemented by smart pointers, will manage memory efficiently, preventing memory leaks and enhancing system stability.

Additionally, we have incorporated the Standard Template Library (STL) as an integral part of our software development in C++. Utilizing a comprehensive suite of generic algorithms, containers, and iterators, the STL empowers us to build efficient and adaptable solutions. This abstract data structure library plays a pivotal role in our codebase by offering a robust set of tools for manipulating and storing data. By embracing the STL, we ensure code reusability and maintainability, following standardized practices for handling data structures and algorithms, thereby significantly boosting the productivity and reliability of our C++ programs. Also, the project will integrate randomization techniques and optimized control structures to introduce variability and realistic behaviours within the simulated environment. This will enable the creation of dynamic scenarios and diverse outcomes, enhancing the authenticity and adaptability of the simulation system.

The framework's versatility will allow developers to easily extend and customize functionalities while maintaining code integrity and reusability. This project aims to provide a comprehensive and efficient solution for building simulation systems, empowering developers to create complex and realistic simulations across various domains.

Problem Definition

Develop a game character design system that leverages fundamental object-oriented programming (OOP) concepts to handle character details, abilities, and interactions. The system should embody the following:

1. Class Hierarchy and Inheritance:

- Construct a class hierarchy with a base class and derived classes, showcasing appropriate inheritance relationships to facilitate code reuse and specialization.

2. Polymorphism and Virtual Functions:

- Implement polymorphism using virtual functions to enable dynamic behaviour based on the specific instance's type. Show how overridden functions provide specialized behaviour in derived classes.

3. Abstraction and Encapsulation:

- Demonstrate abstraction by defining abstract base classes with pure virtual functions to represent generic entity properties and utilize encapsulation to manage access to member variables, employing appropriate access specifiers to maintain data integrity.

4. STL Integration:

- Utilize STL containers such as vector, list, and map for storing character entities, managing character abilities, and implementing crucial functionalities like randomization and algorithmic operations.

5. Polymorphic Behaviour and Function Templates:

- Design and utilize function templates or polymorphic functions to enable polymorphic behaviour, allowing different character types to engage in battles.

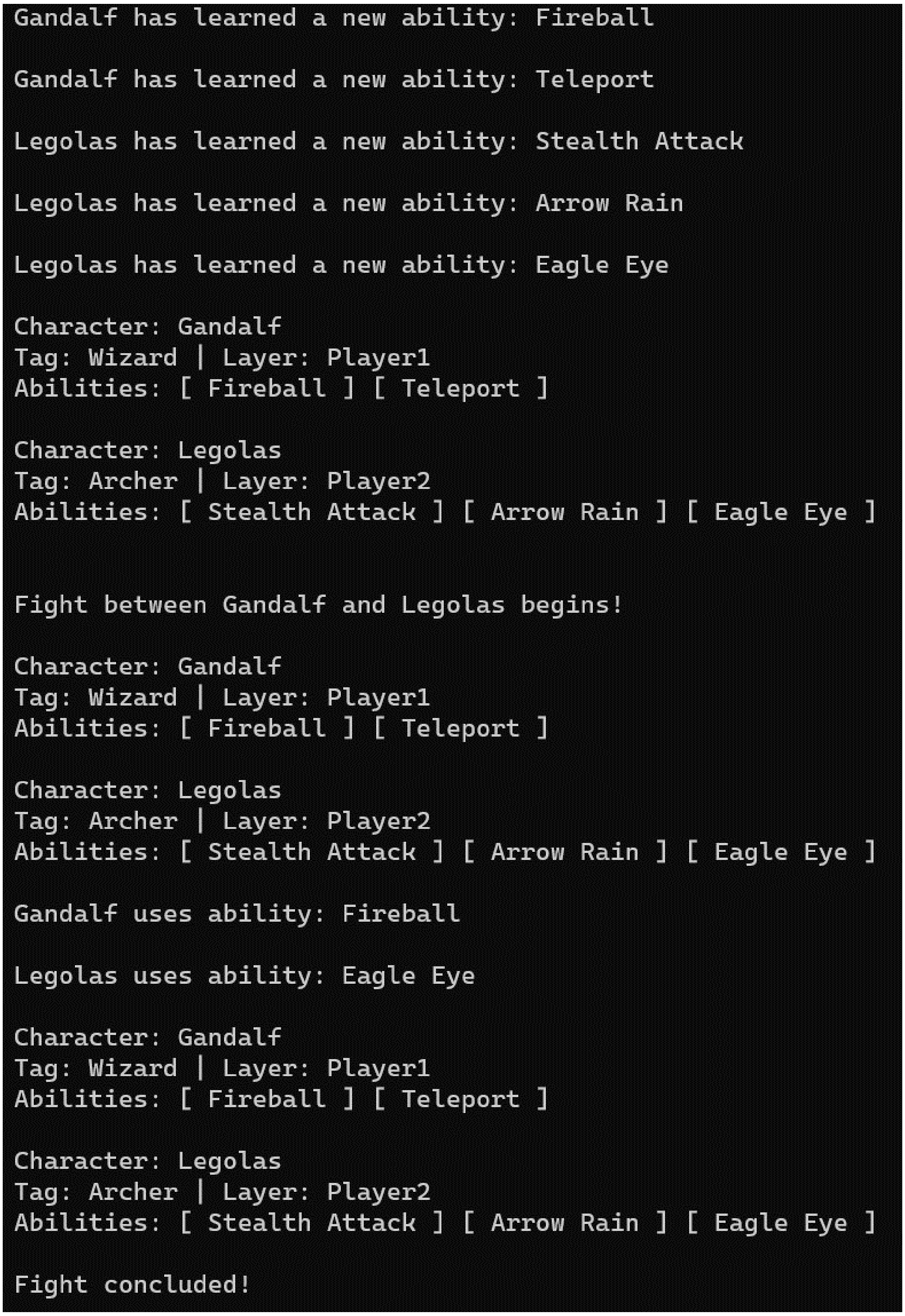
Introduction

The project "Game Character Design Using OOPs" embodies a captivating gaming system rooted in the fundamental principles of object-oriented programming (OOP). Its design aims to orchestrate exhilarating clashes between mage and rogue characters, showcasing the seamless application of inheritance, polymorphism, encapsulation, and abstraction within a dynamic gaming environment. At its core, this project meticulously constructs a sophisticated ecosystem of characters, abilities, and combat mechanics. Each facet of the system is intricately crafted to illustrate the power of class hierarchies and inheritance. A foundational Entity class serves as the bedrock, extending its essence into specialized entities like CharacterType, Abilities, and GameCharacter, paving the way for distinct identities and functionalities.

Polymorphism breathes life into the characters, granting them the ability to display unique behaviors and wield distinct abilities through the use of virtual functions, allowing subclasses like Mage and Rogue to dynamically redefine and extend base class functionalities. This STL-enhanced design harnesses the power of containers such as lists and vectors to manage character abilities efficiently, while maps facilitate tagging and layering mechanisms, adding depth to character classification. Meanwhile, abstraction takes center stage, laying the groundwork through abstract base classes, offering a blueprint for generic entity properties, leveraging the versatility of STL containers to manage complex data structures. Encapsulation ensures data integrity by meticulously controlling access to entity attributes, fostering a resilient and secure system architecture, while smart pointers like unique_ptr<GameCharacter> enable efficient memory management, ensuring optimal allocation and protection against leaks, fortifying stability within the system. The integration of randomization techniques, possibly utilizing the <random> header, coupled with precise control structures, orchestrates thrilling battles where abilities are strategically selected from STL containers, influencing health deductions and character advancements in a dynamic and engaging manner.

In essence, "Game Character Design Using OOPs" stands as a testament to the harmonious fusion of OOP principles within a gaming environment. It elegantly demonstrates the prowess of OOP concepts, culminating in a captivating and expansible gaming experience, inviting players into an immersive world teeming with the clashes of mage and rogue battles.

Sample Run



```
Gandalf has learned a new ability: Fireball

Gandalf has learned a new ability: Teleport

Legolas has learned a new ability: Stealth Attack

Legolas has learned a new ability: Arrow Rain

Legolas has learned a new ability: Eagle Eye

Character: Gandalf
Tag: Wizard | Layer: Player1
Abilities: [ Fireball ] [ Teleport ]

Character: Legolas
Tag: Archer | Layer: Player2
Abilities: [ Stealth Attack ] [ Arrow Rain ] [ Eagle Eye ]


Fight between Gandalf and Legolas begins!

Character: Gandalf
Tag: Wizard | Layer: Player1
Abilities: [ Fireball ] [ Teleport ]

Character: Legolas
Tag: Archer | Layer: Player2
Abilities: [ Stealth Attack ] [ Arrow Rain ] [ Eagle Eye ]

Gandalf uses ability: Fireball

Legolas uses ability: Eagle Eye

Character: Gandalf
Tag: Wizard | Layer: Player1
Abilities: [ Fireball ] [ Teleport ]

Character: Legolas
Tag: Archer | Layer: Player2
Abilities: [ Stealth Attack ] [ Arrow Rain ] [ Eagle Eye ]

Fight concluded!
```

Understanding the Code

Libraries and Constants

- Libraries: These are imported libraries such as ‘iostream’, ‘string’, ‘list’, ‘vector’, ‘memory’, ‘random’, and ‘ctime’, providing various functionalities like I/O operations, data structures, memory management, and random number generation.

- Constants: ‘MAX_ABILITIES’ is a defined constant used to limit the maximum number of abilities a character can possess.

Class Definitions

- ‘Entity’ Class: Represents a generic game entity with a name and a function to display entity information.

- ‘CharacterType’ Class: Represents character types with tags and layers, inheriting from ‘Entity’ and overriding the ‘displayInfo’ function.

- ‘Abilities’ Class: Represents character abilities with names and powers.

- ‘GameCharacter’ Class: Represents game characters inheriting from ‘CharacterType’. Manages character attributes like health, level, abilities list, and functions for level-up, displaying character info, and using abilities.

- ‘Mage’ and ‘Rogue’ Classes: Specialized character types inheriting from ‘GameCharacter’ with overridden functions to use abilities.

Function Templates and Main Function

- ‘fightPlayers’ Template Function: Simulates a battle between two players, deducting health based on abilities used and incrementing levels accordingly.

- ‘main’ Function: The entry point of the program. Creates instances of ‘Mage’ and ‘Rogue’, adds abilities to them, stores them in a vector of smart pointers, and simulates a battle between the characters.

Logic Flow

- Character Creation: Instances of ‘Mage’ and ‘Rogue’ are created with specific names, tags, and layers. Abilities are added to each character.

- Display Information: Character information is displayed before and after the battle.

- Battle Simulation: A battle is simulated between the characters, involving ability usage, health deduction, level-up based on the outcome, and final character information display.

Let us explore the implementation of several object-oriented programming (OOP) concepts, including inheritance, polymorphism, encapsulation, and others, within the provided code:

Inheritance

- Usage:

1. Single Inheritance:

   - Example: `class CharacterType : public Entity`

   - Explanation: `CharacterType` inherits from a single base class `Entity`. It directly inherits from one base class.

2. Multilevel and Multiple Inheritance:

   - Example: `class GameCharacter : public CharacterType, public Abilities`

   - Explanation: `GameCharacter` inherits from `CharacterType`, which itself inherits from `Entity`. This creates a chain of inheritance where a derived class inherits from another derived class. Also, it inherits both the CharacterType and Abilities classes. So, it can be catergorized as multiple inheritance too.

3. Hierarchical and Hybrid Inheritance:

   - Example: `class Mage : public GameCharacter` and `class Rogue : public GameCharacter`

   - Explanation: Both `Mage` and `Rogue` classes inherit from the same base class `GameCharacter`. This creates a hierarchy where multiple classes inherit from a single base class. Also, the GameCharacter inherits from 2 classes, so the Mage and Rogue classes can also be categorized into hybrid classes.

Polymorphism

- Usage: Polymorphism is exhibited via virtual functions like 'displayInfo' and 'useAbility' which are overridden in subclasses ('CharacterType', 'Mage', 'Rogue'). This allows the same function name to exhibit different behaviours based on the object type.

- Example: ‘displayInfo’ function is overridden in ‘CharacterType’, ‘Mage’, and ‘Rogue’ classes to display specific information for each character type.

Encapsulation

- Usage: Encapsulation ensures data integrity and controlled access to member variables. It's applied by setting member variables as ‘protected’ or ‘private’ and accessing them via public member functions (‘addAbility’, ‘levelUp’, etc.).

- Example: ‘characterAbilities’ is a ‘protected’ member of ‘GameCharacter’, managed through the ‘addAbility’ function to restrict direct access.

Abstraction

- Usage: Abstraction is manifested by defining abstract base classes (‘Entity’, ‘Abilities’) with pure virtual functions (‘virtual void displayInfo()’), providing a blueprint for generic entity properties.

- Example: ‘Entity’ class contains the pure virtual function ‘displayInfo’, creating a blueprint for all entities to have a display function.

STL Containers

- Usage:

> vector is used to store instances of GameCharacter as smart pointers (unique_ptr), ensuring dynamic memory management and allowing flexible storage of character objects. list is employed within the GameCharacter class to store character abilities.

> list is used to store the character abilities of the GameCharacter.
> map is used to store the content / values of player tag, player layer, player health and player level.

- Example: vector<unique_ptr<GameCharacter>> characters, list<Abilities> characterAbilities

Function Template Definition

- Usage: The fightPlayers function template is designed to simulate battles between different types of game characters, accommodating any pair of players for engaging battles without knowing their specific types at compile time.

Conclusion

In conclusion, the "Game Character Design Using OOPs" project stands as a testament to the harmonious fusion of object-oriented programming (OOP) principles within a gaming environment. By intricately weaving together inheritance, polymorphism, encapsulation, and abstraction, this project showcases a sophisticated and extensible framework for character design and interactions. Through the seamless integration of OOP concepts, it delivers a captivating gaming experience, offering dynamic battles, distinct character behaviors, and flexible system scalability. Ultimately, this project underscores the power and elegance of OOP in crafting immersive and engaging game character design paradigms.