

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №3
по дисциплине «ООП»
Тема: Разработка игровой системы с тактическими боями.

Студент гр. 4384

Преподаватель

Стукалкин М. М.

Жангиров Т.Р.

Санкт-Петербург

2025

Цель.

Разработать игровую систему, реализующую механику пошаговых тактических боев с управлением персонажем, врагами и специальными сооружениями. Построить архитектуру с учетом принципов ООП и расширяемости

Задание.

На 6/3/1 баллов:

Создать класс(ы) игры, который реализует основной цикл игры, и которому передаются команды от пользователя. Игровой цикл состоит из следующих шагов:

Начало игры

Запуск уровня

Ход игрока. Ход, атака или применение заклинания.

Ход союзников - если имеются

Ход врагов

Ход вражеской базы и башни - если имеются

Условие прохождения уровня студент определяет самостоятельно. Если игрок проигрывает, то игроку должно предлагаться начать заново игру, либо выйти из программы.

Все взаимодействие должно происходить через классы игры.

Реализовать систему сохранения и загрузки игры. Пользователь должен иметь возможность сохранить игру в любой момент. Пользователь должен иметь возможность загружаться при запуске программы (или выбрать новую), либо во время игры. Сохранения должны оставаться в консистентном состоянии между запусками игры.

Добавить обработку исключительных ситуаций для загрузки/сохранения, например, невозможность записать в файл, нельзя загрузиться так как файл не существует или в нем некорректные данные.

На 8/4/1.5 баллов:

Реализовать переход на следующий уровень, после прохождения уровня. При переходе на следующий уровень создается новое поле другого размера с более сильными врагами. При переходе на следующий уровень, значение жизни игрока восстанавливается, и половина его карточек заклинаний случайным образом удаляется.

На 10/5/2 баллов:

Реализовать прокачку игрока при переходе между уровнями. Пользователь может улучшить характеристики игрока или улучшить заклинание (что и как улучшать определяет студент). Для этого нужно расширить игровой цикл.

Примечания:

Класс игры может знать о игровых сущностях, но не наоборот

При работе с файлом используйте идиому RAII.

Исключения должны обязательно обрабатываться, и программа не должна завершаться

Исключения должны быть информативными (содержать информацию о том, что и где произошло), на разные виды исключительных ситуаций должны быть свои исключения

Архитектурные решения и обоснование.

1. Класс Game.

Атрибуты:

1. GameArea* gamearea = nullptr – указатель на игровое поле.
2. Player* player = nullptr – указатель на игрока.
3. std::vector<Enemy*> enemies. std::vector<Summon*> summons, std::vector<Building*> buildings – указатели на вектора врагов, союзников и зданий соответственно.
4. int current_level = 1 – текущий уровень.
5. bool game_running = true – флаг, который отвечает на вопрос идет игра или нет.

Методы:

1. `void start_new_game()` - Запускает новую игру (сброс всего состояния).
2. `void init_level()` – создает игровой уровень. Создает самого игрока, если еще не был создан, а также заново генерирует все вражеские объекты, увеличивая их характеристики, в зависимости от уровня.
3. `void player_turn()` – метод определяющий ход игрока. Выводит возможные команды пользователю и обрабатывает их.
4. `void allies_turn()`, `void enemies_turn()`, `buildings_turn()` – методы определяющие ходы для остальных объектов.
5. `bool check_win_condition()` – проверяет уничтожены ли все вражеские башни и противники, если да, то возвращает `true`, что будет знаком для перехода на следующий уровень.
6. `bool check_lose_condition()` – проверяет здоровье игрока, если здоровье меньше 0, то игрок проиграл и возвращается `true`.
7. `void check_allies_on_area` – ищет всех союзников на игровом поле, и если найдены те, что еще не были добавлены, то добавляет. Обоснование: все остальные объекты на поле размещаются самим классом `Game` и он про всех сам все знает. Объекты класса `Summon` размещаются при помощи заклинания, которое кастует `Player` и нужно как то передать информацию `Game` о появившемся союзнике.
8. `void upgrade_player()` – метод, который улучшает персонажа после прохождения уровня. Можно увеличить количество здоровья на 20, количество урона на 10 и увеличить вместимость руки.
9. `void next_level()` – метод для перехода на следующий уровень. Очищает половину заклинаний игрока, предлагает улучшение и создает новый уровень.
10. `void game_loop()` – определяет игровой цикл: выводит поле, проверяет, победил или проиграл игрок, делает ходы всех объектов на поле.
11. `void handle_game_over()` – метод, который запускается в случае поражение игрока и предлагает начать новую игру или загрузить сохранение.

12. `void save_current_game()` – метод, который извлекает из игры состояния всех объектов в объекте класса `GameState` и после сохраняет все в файл при помощи `SaveManager::save_game`.
13. `void load_saved_game` – метод, который позволяет загрузить сохранение.
14. `void run()` – метод, который запускает саму игру.

Обоснование: класс `Game` отдельно отвечает за всю логику игрового процесса, инкапсулируя ее в себе. Также он нигде не передается объектам игрового процесса, не нарушая условия задания.

2. Структура `GameState`.

Назначение: это структура данных, предназначенная для хранения полного состояния игры в момент сохранения или загрузки.

Хранит внутри себя:

1. Основные параметры уровня: `int current_level` — номер текущего уровня, `int area_width, area_height` — размеры игрового поля
2. Карта типов клеток: `std::vector<std::vector<int>> cell_types` — двумерный массив с информацией о типе каждой клетки (например, 0=BASIC, 1=BLOCKED, 2=SLOW). Это необходимо для точного восстановления игрового поля и эффектов, связанных с клетками.
3. Игрок: статые игрока (`double player_health, player_damage, player_experience`), `Coords player_coords` — текущие координаты игрока, `int player_hand_size` — размер "руки" (количество слотов для заклинаний).
4. Враги: структура `EnemyData` содержит: здоровье, урон, координаты каждого врага. `std::vector<EnemyData> enemies` — массив всех врагов на карте.
5. Союзники: структура `SummonData`: здоровье, урон, координаты `std::vector<SummonData> summons` — все действующие союзники.

6. Вражеские здания: структура `BuildingData`: тип, здоровье, координаты, счётчик спауна. `std::vector<BuildingData> buildings` — здания на поле (башни, базы и т.д.

Обоснование: явное разделение (все важные аспекты игрового процесса собраны в одной структуре — легко пройти для сохранения/загрузки). Масштабируемость: добавить новый тип объекта на поле легко — достаточно добавить новую структуру и массив.

Класс SaveManager.

Назначение: это сервисный класс, отвечающий за сохранение и загрузку состояния игры в структуру `GameState` и записи/чтения из файла.

Методы:

1. `static void save_game(GameState& state, const std::string& filename = "savegame.txt")` — сохраняет игру посредством записывания всех данных из `GameState` в файл `savegame.txt`.
2. `GameState load_game(const std::string& filename)` — загружает игровое сохранение посредством считывания данных из файла и создания объекта класса `GameState`, куда все заносится.

За счет чего достигается идиома RAII: благодаря классу `FileHandler`, который используется внутри всех методов `SaveManager`.

Что делает `FileHandler`: в конструкторе открывает файл, в деструкторе закрывает файл, если он открыт. Если что-то пошло не так, то бросается исключение.

4. Обработка исключений при работе с файлом. (SaveLoadException.h)

Реализация: создан родительский класс для всех исключений `SaveLoadException`, который наследуется от `runtime_error`,

1. `FileOpenException`: Наследует от `SaveLoadException`. Хранит имя файла `filename`. Выбрасывается, если файл не получился открыть на запись или чтение:

2. `FileWriteException`: Наследует от `SaveLoadException`. Хранит имя файла. Выбрасывается, если нельзя завершить запись или возникает проблема при записи.
3. `FileReadException`: Наследует от `SaveLoadException`. Хранит имя файла. Выбрасывается при ошибке чтения (например, если файл закончился раньше или возникла системная ошибка).
4. `CorruptedDataException`: Наследует от `SaveLoadException`. Конструктор принимает строку с деталями. Выбрасывается, если структура файла не совпадает с ожидаемой: формат, некорректные значения, неверное количество, типы данных.

Обоснование: Отделить причины ошибок: разный `catch` — разная стратегия восстановления или сообщения. Передать подробное сообщение пользователю: в каждом исключении содержится имя файла или деталь сбоя.

