

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

ОТЧЕТ
по лабораторной работе №3
по дисциплине «Объектно-ориентированное программирование»
Тема: Реализация консольной игры с использованием ООП.

Студент гр. 4384

Мазеев В.А.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Целью работы является разработка консольной игры с использованием принципов объектно-ориентированного программирования, демонстрирующей взаимодействие классов, инкапсуляцию, наследование.

Задание.

На 6/3/1 баллов:

1. Создать класс(ы) игры, который реализует основной цикл игры, и которому передаются команды от пользователя. Игровой цикл состоит из следующих шагов:
 - a. Начало игры
 - b. Запуск уровня
 - c. Ход игрока. Ход, атака или применение заклинания.
 - d. Ход союзников - если имеются
 - e. Ход врагов
 - f. 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.

2. 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.
3. 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:

4. 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:

5. 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

Выполнение работы.

Выбрано задание на 8 баллов.

В ходе выполнения лабораторной работы была реализована расширенная архитектура игрового приложения на языке C++ с использованием принципов объектно-ориентированного программирования. Основной целью было построение структурированного игрового цикла, внедрение системы уровней, а также разработка полнофункционального механизма сохранения и загрузки игры.

В класс Game была добавлена система фаз, определяемая enum перечислением GamePhase. Игровой цикл был переработан таким образом, чтобы логика игры проходила через последовательность чётко выделенных этапов:

Menu — отображение стартового меню с выбором: «Новая игра», «Загрузить игру», «Выход».

GameStart — инициализация новой игры.

LevelStart — подготовка очередного уровня.

PlayerTurn — выполнение действий игрока (движение, атака ближним/дальним боем, применение заклинаний, сохранение игры).

EnemiesTurn — логика хода противников (перемещение, атаки).

LevelComplete — обработка завершения уровня.

GameOver — экран поражения с возможностью перезапуска или возврата в меню.

Exit — завершение игры

Добавлены функции:

1. processMenu() – обрабатывает главное меню игры. Показывает варианты действий (новая игра, загрузка, выход) и переключает фазу игрового цикла в зависимости от ввода игрока.

2. `processGameStart()` – инициализирует полностью новую игру (уровень = 1, очистка поля, создание игрока/врагов). Переводит игру в фазу начала уровня.
3. `processLevelStart()` – подготавливает начало уровня: очищает экран, выводит номер уровня и ожидает подтверждение игрока. После ожидания переводит фазу в ход игрока.
4. `processPlayerTurn()` – обрабатывает действия игрока в его ход: движение, переключение режима боя, применение заклинаний, сохранение игры. После выполнения действия переводит управление союзникам.
5. `processEnemiesTurn()` – выполняет ходы всех врагов — случайное перемещение и возможную атаку игрока. Если игрок умирает, переключает игру в фазу GameOver.
6. `processLevelComplete()` – вызывается после уничтожения всех врагов. Показывает статистику, увеличивает номер уровня, подготавливает новый уровень и переводит игру в фазу LevelStart.
7. `processGameOver()` – отображает экран окончания игры, показывает статистику и предлагает перезапуск или выход в меню. Изменяет фазу в зависимости от выбора игрока.
8. `initializeNewGame()` – сбрасывает параметры игры, устанавливает уровень 1, обнуляет счётчик убитых врагов и запускает первый уровень.
9. `initializeLevel(int level)` – полностью перестраивает поле для нового уровня, размещает игрока и генерирует количество врагов по формуле $3 + (\text{level} - 1) * 2$.
10. `checkLevelComplete() const` – возвращает true, если на поле больше нет врагов.
11. `checkGameOver() const` – возвращает true, если игрок отсутствует или его здоровье ≤ 0 .

12. `tryGiveNewSpell()` – добавляет игроку новое случайное заклинание (точечное или по области), если рука заклинаний не заполнена. Вызывается после убийства врага.
13. `showStats() const` – выводит характеристики игрока: уровень, здоровье, урон, очки, позицию.
14. `showHand() const` – отображает список заклинаний игрока, их характеристики и количество ячеек.
15. `displayGameScreen() const` – очищает экран, выводит поле, статистику и заклинания.
16. `pause(const std::string& message) const` – пауза для ожидания нажатия Enter пользователем. Используется для отображения сообщений.
17. `saveGame() const` – сооздает структуру `GameState`, сериализует текущее состояние игры и сохраняет его в файл. Возвращает `true` при успехе.
18. `loadGame()` – загружает состояние игры из файла через `GameState`. Восстанавливает поле, игрока, врагов и заклинания. Возвращает `true` при успешной загрузке.
19. `captureGameState() const` – формирует объект `GameState`, заполняя его всеми параметрами игры: состояние игрока, заклинания, враги, стены, уровень и др.
20. `restoreGameState(const GameState& state)` – восстанавливает состояние игры по структуре `GameState`: поле, игрок, враги, параметры уровня и рука заклинаний полностью пересоздаются.

Добавлен новый класс-структура `GameState`, предназначенный для полного сохранения и восстановления состояния игры между сессиями. Этот класс инкапсулирует все данные, необходимые для корректного восстановления игрового процесса: параметры игрока, состояние поля, список врагов, заклинания, стены, а также текущий прогресс по уровням:

1. `saveToFile()` – сериализация состояния игры. Функция выполняет запись состояния игры в файл. Сохраняет все параметры игры.
2. `loadFromFile()` – десериализация состояния. Загружает состояние игры из файла, выполняя последовательное чтение всех блоков. Сначала происходит проверка заголовка, если строка не `"GAME_SAVE_V1"` – сохранение считается несовместимым. Чтение файлов в строго фиксированном порядке. Используется поэлементарное чтение(`file >> value`), затем `file.ignore()` для перехода на следующую строку.
3. `clear()` – сброс состояния. Функция полностью очищает объект `GameState` и инициализирует поля значениями по умолчанию.

Архитектура.

Выбранная архитектура программы была расширена системой сохранения и восстановления состояния игры, основанной на принципах инкапсуляции, разделения ответственности и модульности. Центральным элементом является новый класс-структура `GameState`, который инкапсулирует все параметры, необходимые для корректного восстановления игровой сессии: состояние игрока, коллекцию заклинаний, параметры игрового поля, расположение стен и врагов, а также прогресс по уровням. Благодаря этому логика сохранения полностью отделена от основной игровой механики.

Класс `GameState` реализует функции сериализации и десериализации (`saveToFile()` и `loadFromFile()`), что позволяет преобразовывать внутреннее состояние программы в текстовый формат с версионированием. Такой подход обеспечивает устойчивость к изменениям архитектуры и упрощает последующее расширение формата сохранений. Логика чтения и записи данных изолирована в отдельном модуле, что соответствует принципу единственной ответственности.

Архитектура игрового процесса также была переработана путём введения перечисления GamePhase, описывающего фазы игрового цикла. Теперь класс Game управляет работой программы на основе чётко определённых стадий: меню, начало игры, начало уровня, ход игрока, ходы врагов, завершение уровня и обработка окончания игры. Такой подход позволяет упорядочить выполнение игровых событий, упростить добавление новых фаз и обеспечить детерминированное поведение игрового цикла.

Интеграция системы сохранения выполнена через композицию — объект Game содержит экземпляр GameState и заполняет его данными при сохранении, а при загрузке восстанавливает состояние всех игровых сущностей: игрока, поля, врагов, стены и уровень. Это исключает дублирование данных и создаёт единый центр хранения состояния игры, с которым взаимодействуют остальные компоненты.

Использование блоков try-catch в механизме загрузки/сохранения и явных разделов формата файла обеспечивает надёжность работы и устойчивость к ошибкам ввода-вывода. Такая архитектура делает систему легко масштабируемой: в неё можно без изменения существующих компонентов добавлять новые типы объектов, параметры или игровые механики, просто расширяя структуру GameState и обновляя логику сериализации.

Архитектурные решения обеспечивают слабую связанность модулей, понятную структуру управления игрой и возможность дальнейшего расширения проекта без усложнения основных классов.

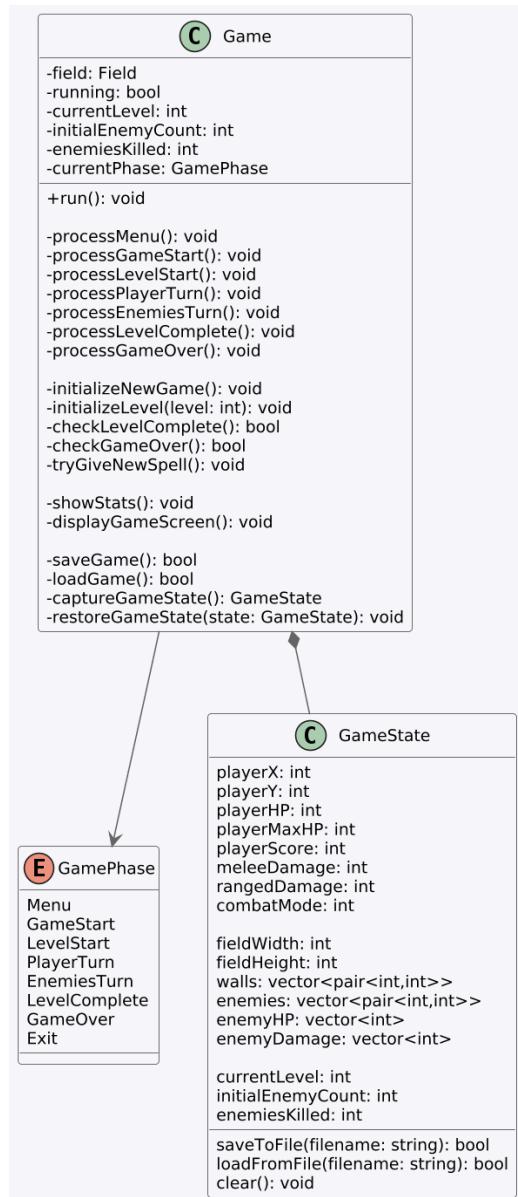


Рис. 1 UML-диаграмма добавленных классов.

Выводы.

В текущей архитектуре GameState выполняет роль снимка (snapshot) состояния игры, фиксируя ключевые данные: позицию и характеристики игрока, состояние поля (стены и враги), а также прогресс по уровню. Класс Game использует GameState для сохранения и восстановления игры, что позволяет игроку продолжать с того же места после выхода или аварийного завершения.

Такое разделение логики игры и хранения состояния повышает модульность кода: Game отвечает за игровой цикл, обработку команд игрока и взаимодействие объектов на поле, а GameState отвечает исключительно за сериализацию и восстановление данных. Это облегчает поддержку и расширение проекта: например, добавление новых типов врагов или изменений в поле не требует изменения механизма сохранения.

ПРИЛОЖЕНИЕ А

ТЕСТИРОВАНИЕ

tests.cpp

```
#include <iostream>
#include <cassert>
#include <memory>
#include <cstdlib>
#include <windows.h>

#include "Cell.h"
#include "Entity.h"
#include "Player.h"
#include "Enemy.h"
#include "Field.h"
#include "Hand.h"
#include "DirectDamageSpell.h"
#include "AreaDamageSpell.h"

namespace {
    void clearField(Field& field) {
        for (int y = 0; y < field.getHeight(); ++y) {
            for (int x = 0; x < field.getWidth(); ++x) {
                field.clearCell(x, y);
            }
        }
    }
}

// ===== Testing Cell =====
void testCell() {
    std::cout << "Testing Cell..." << std::endl;

    Cell cell;

    // Check initial state
    assert(cell.isEmpty() == true);
    assert(cell.isWall() == false);

    // Check type setting
    cell.setType(CellType::Wall);
    assert(cell.isWall() == true);
    assert(cell.isEmpty() == false);

    cell.setType(CellType::Player);
    assert(cell.getType() == CellType::Player);
    assert(cell.isOccupied() == true);

    std::cout << "Cell tests passed!" << std::endl;
}

// ===== Testing Entity =====
void testEntity() {
    std::cout << "Testing Entity..." << std::endl;

    // Create test class for abstract Entity
    class TestEntity : public Entity {
```

```

public:
    TestEntity(int x, int y, int hp) : Entity(x, y, hp) {}
    int getDamage() const override { return 10; }
    char getDisplayChar() const override { return 'T'; }
};

TestEntity entity(5, 5, 100);

// Check position and health
assert(entity.getX() == 5);
assert(entity.getY() == 5);
assert(entity.getHP() == 100);
assert(entity.isAlive() == true);

// Check damage taking
entity.takeDamage(30);
assert(entity.getHP() == 70);

// Check death
entity.takeDamage(100);
assert(entity.getHP() == 0);
assert(entity.isAlive() == false);

std::cout << "Entity tests passed!" << std::endl;
}

// ===== Testing Player =====
void testPlayer() {
    std::cout << "Testing Player..." << std::endl;

    Player player(2, 2, 100, 15, 5);

    // Check initial state
    assert(player.getHP() == 100);
    assert(player.getCombatMode() == CombatMode::Melee);
    assert(player.getDamage() == 15); // Melee mode

    // Check combat mode switching
    player.switchCombatMode();
    assert(player.getCombatMode() == CombatMode::Ranged);
    assert(player.getDamage() == 5); // Ranged mode

    // Check score system
    assert(player.getScore() == 0);
    player.addScore(10);
    assert(player.getScore() == 10);

    // Check display character
    assert(player.getDisplayChar() == 'P');

    std::cout << "Player tests passed!" << std::endl;
}

// ===== Testing Hand =====
void testHand() {
    std::cout << "Testing Hand..." << std::endl;

    std::srand(0); // deterministic first spell

```

```

    Hand hand(3);
    assert(hand.getMaxSize() == 3);
    assert(hand.getSpellCount() == 1);
    assert(!hand.isEmpty());
    assert(!hand.isFull());
    assert(hand.getSpell(0) != nullptr);

    bool added = hand.addSpell(std::make_unique<DirectDamageSpell>(3, 25));
    assert(added);
    assert(hand.getSpellCount() == 2);
    assert(hand.getSpell(1) != nullptr);

    added = hand.addSpell(std::make_unique<AreaDamageSpell>(3, 15));
    assert(added);
    assert(hand.getSpellCount() == 3);
    assert(hand.isFull());

    added = hand.addSpell(std::make_unique<DirectDamageSpell>(3, 10));
    assert(added == false);

    Hand copy = hand;
    assert(copy.getSpellCount() == hand.getSpellCount());
    assert(copy.isFull() == hand.isFull());
    assert(copy.getSpell(0) != nullptr);

    Hand assigned(2);
    assigned = hand;
    assert(assigned.getSpellCount() == hand.getSpellCount());
    assert(assigned.isFull() == hand.isFull());

    std::cout << "Hand tests passed!" << std::endl;
}

void testEnemy() {
    std::cout << "Testing Enemy..." << std::endl;

    Enemy enemy(3, 3, 50, 8);

    assert(enemy.getHP() == 50);
    assert(enemy.getDamage() == 8);
    assert(enemy.isAlive() == true);

    enemy.takeDamage(20);
    assert(enemy.getHP() == 30);
    assert(enemy.isAlive() == true);

    assert(enemy.getDisplayChar() == 'E');

    std::cout << "Enemy tests passed!" << std::endl;
}

void testField() {
    std::cout << "Testing Field..." << std::endl;

    Field field(10, 10);

```

```

        clearField(field);

        assert(field.getWidth() == 10);
        assert(field.getHeight() == 10);

        assert(field.isValidPosition(0, 0) == true);
        assert(field.isValidPosition(9, 9) == true);
        assert(field.isValidPosition(-1, 0) == false);
        assert(field.isValidPosition(10, 10) == false);

        auto player = std::make_unique<Player>(1, 1);
        field.setPlayer(std::move(player));
        assert(field.getPlayer() != nullptr);
        assert(field.hasPlayerAt(1, 1) == true);

        auto enemy = std::make_unique<Enemy>(2, 2);
        field.addEnemy(std::move(enemy));
        assert(field.getEnemies().size() == 1);
        assert(field.hasEnemyAt(2, 2) == true);

        Enemy* enemyPtr = field.getEnemyAt(2, 2);
        assert(enemyPtr != nullptr);
        field.removeEnemy(enemyPtr);
        assert(field.hasEnemyAt(2, 2) == false);

        std::cout << "Field tests passed!" << std::endl;
    }

void testMovement() {
    std::cout << "Testing Movement..." << std::endl;

    Field field(10, 10);
    clearField(field);
    auto player = std::make_unique<Player>(5, 5);
    field.setPlayer(std::move(player));

    assert(field.getPlayer() != nullptr);
    assert(field.getPlayer()->getX() == 5);
    assert(field.getPlayer()->getY() == 5);

    assert(field.getPlayer()->canMove(1, 0, field) == true);
    assert(field.getPlayer()->canMove(-5, 0, field) == false);

    std::cout << "Movement tests passed!" << std::endl;
}

// ===== Testing Direct Damage Spell =====
void testDirectDamageSpell() {
    std::cout << "Testing DirectDamageSpell..." << std::endl;

    Field field(10, 10);
    clearField(field);

    auto player = std::make_unique<Player>(1, 1);
    Player* playerPtr = player.get();
    field.setPlayer(std::move(player));

    DirectDamageSpell spell(3, 20);

```

```

        field.clearCell(2, 1);
        auto enemy1 = std::make_unique<Enemy>(2, 1, 30, 5);
        field.addEnemy(std::move(enemy1));

        bool castSuccess = spell.cast(playerPtr, 2, 1, field);
        assert(castSuccess == true);
        Enemy* enemyAfter = field.getEnemyAt(2, 1);
        assert(enemyAfter != nullptr);
        assert(enemyAfter->getHP() == 10);
        assert(playerPtr->getScore() == 0);

        bool castNoTarget = spell.cast(playerPtr, 3, 1, field);
        assert(castNoTarget == false);

        bool castOutOfRange = spell.cast(playerPtr, 9, 9, field);
        assert(castOutOfRange == false);

        field.clearCell(3, 2);
        auto enemy2 = std::make_unique<Enemy>(3, 2, 10, 5);
        field.addEnemy(std::move(enemy2));
        Enemy* enemyToDie = field.getEnemyAt(3, 2);
        assert(enemyToDie != nullptr);
        int previousScore = playerPtr->getScore();

        bool castKill = spell.cast(playerPtr, 3, 2, field);
        assert(castKill == true);
        assert(field.getEnemyAt(3, 2) == nullptr);
        assert(playerPtr->getScore() == previousScore + 10);

        std::cout << "DirectDamageSpell tests passed!" << std::endl;
    }

// ===== Testing Area Damage Spell =====
void testAreaDamageSpell() {
    std::cout << "Testing AreaDamageSpell..." << std::endl;

    Field field(10, 10);
    clearField(field);

    auto player = std::make_unique<Player>(2, 2);
    Player* playerPtr = player.get();
    field.setPlayer(std::move(player));

    AreaDamageSpell spell(4, 15);

    field.clearCell(3, 3);
    field.clearCell(4, 3);
    field.clearCell(3, 4);
    field.clearCell(4, 4);

    auto enemy1 = std::make_unique<Enemy>(3, 3, 25, 5);
    auto enemy2 = std::make_unique<Enemy>(4, 4, 25, 5);
    field.addEnemy(std::move(enemy1));
    field.addEnemy(std::move(enemy2));

    bool castSuccess = spell.cast(playerPtr, 3, 3, field);
    assert(castSuccess == true);
}

```

```

        Enemy* enemyA = field.getEnemyAt(3, 3);
        Enemy* enemyB = field.getEnemyAt(4, 4);
        assert(enemyA != nullptr);
        assert(enemyB != nullptr);
        assert(enemyA->getHP() == 10);
        assert(enemyB->getHP() == 10);

        bool castEmptyArea = spell.cast(playerPtr, 7, 7, field);
        assert(castEmptyArea == true);

        std::cout << "AreaDamageSpell tests passed!" << std::endl;
    }

// ===== Testing Player Casting via Hand =====
void testPlayerCasting() {
    std::cout << "Testing Player casting spells..." << std::endl;

    Field field(10, 10);
    clearField(field);

    auto player = std::make_unique<Player>(1, 1);
    Player* playerPtr = player.get();
    field.setPlayer(std::move(player));

    Hand& hand = playerPtr->getHand();
    int initialCount = hand.getSpellCount();

    bool added = hand.addSpell(std::make_unique<DirectDamageSpell>(3,
35));
    assert(added == true);
    int spellIndex = hand.getSpellCount() - 1;

    field.clearCell(2, 1);
    auto enemy = std::make_unique<Enemy>(2, 1, 25, 5);
    field.addEnemy(std::move(enemy));

    bool castSuccess = playerPtr->castSpell(spellIndex, 2, 1, field);
    assert(castSuccess == true);
    assert(field.getEnemyAt(2, 1) == nullptr);
    assert(playerPtr->getScore() >= 10);

    bool invalidIndex = playerPtr->castSpell(999, 2, 1, field);
    assert(invalidIndex == false);

    bool emptyTarget = playerPtr->castSpell(spellIndex, 3, 1, field);
    assert(emptyTarget == false);

    // Ensure original random spells remain accessible
    assert(hand.getSpellCount() >= initialCount);

    std::cout << "Player casting tests passed!" << std::endl;
}

int main() {
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);
    std::cout << "==== Starting Simple Game Tests ===" << std::endl;
}

```

```

        std::cout << "Running basic functionality tests...\n" <<
std::endl;

    try {
        testCell();
        testEntity();
        testPlayer();
        testEnemy();
        testHand();
        testField();
        testMovement();
        testDirectDamageSpell();
        testAreaDamageSpell();
        testPlayerCasting();

        std::cout << "\n==== ALL TESTS PASSED! ===" << std::endl;
        std::cout << "Basic game functionality is working correctly!"
<< std::endl;

    } catch (const std::exception& e) {
        std::cerr << "\nTest failed: " << e.what() << std::endl;
        return 1;
    } catch (...) {
        std::cerr << "\nUnknown test failure" << std::endl;
        return 1;
    }

    return 0;
}

```

```
PS C:\Users\vladm\OneDrive\Desktop\LETI\OOP\lb1> ./tests.exe
==== Starting Simple Game Tests ====
Running basic functionality tests...

Testing Cell...
Cell tests passed!
Testing Entity...
Entity tests passed!
Testing Player...
Игрок переключился в Ranged fight
Player tests passed!
Testing Enemy...
Enemy tests passed!
Testing Hand...
Hand tests passed!
Testing Field...
Field tests passed!
Testing Movement...
Movement tests passed!
Testing DirectDamageSpell...
Заклинание прямого урона нанесло 20 урона врагу в позиции (2, 1)!
В указанной позиции нет врага или вражеского здания!
Цель вне радиуса действия заклинания!
Заклинание прямого урона нанесло 20 урона врагу в позиции (3, 2)!
Враг убит заклинанием! +10 очков
DirectDamageSpell tests passed!
Testing AreaDamageSpell...
Заклинание урона по области нанесло 15 урона в области 2x2 начиная с (3, 3)!
Затронуто врагов: 2
Заклинание урона по области нанесло 15 урона в области 2x2 начиная с (7, 7)!
В области нет врагов, но заклинание все равно использовано.
AreaDamageSpell tests passed!
Testing Player casting spells...
Заклинание прямого урона нанесло 35 урона врагу в позиции (2, 1)!
Враг убит заклинанием! +10 очков
Неверный индекс заклинания!
В указанной позиции нет врага или вражеского здания!
Player casting tests passed!

==== ALL TESTS PASSED! ====
Basic game functionality is working correctly!
```

Рис. 2 Результаты тестирования.

```
==== ИГРА ====
1. Новая игра
2. Загрузить игру
3. Выход
Выберите действие: |
```

Рис. 3 Начало игры.

```
==== УРОВЕНЬ 1 ПРОЙДЕН! ====
Уровень: 1 | Здоровье: 90 | Урон: 15 | Очки: 30 | Позиция: (8, 9)
Нажмите Enter для продолжения...
```

Рис. 5 Прохождение уровня.

ПРИЛОЖЕНИЕ В

ИСХОДНЫЙ КОД

Game.cpp

```
#include "Game.h"
#include "DirectDamageSpell.h"
#include "AreaDamageSpell.h"
#include "Spell.h"
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <limits>
#include <algorithm>

const std::string Game::SAVE_FILE = "game_save.dat";

Game::Game() : field(10, 10), running(true), currentLevel(1),
               initialEnemyCount(0), enemiesKilled(0),
               currentPhase(GamePhase::Menu) {
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);
    std::srand(static_cast<unsigned int>(std::time(nullptr)));
}

void Game::run() {
    while (running && currentPhase != GamePhase::Exit) {
        switch (currentPhase) {
            case GamePhase::Menu:
                processMenu();
                break;
            case GamePhase::GameStart:
                processGameStart();
                break;
            case GamePhase::LevelStart:
                processLevelStart();
                break;
            case GamePhase::PlayerTurn:
                processPlayerTurn();
                break;
            case GamePhase::AlliesTurn:
                processAlliesTurn();
                break;
            case GamePhase::EnemiesTurn:
                processEnemiesTurn();
                break;
            case GamePhase::EnemyBaseTurn:
                processEnemyBaseTurn();
                break;
            case GamePhase::LevelComplete:
                processLevelComplete();
                break;
            case GamePhase::GameOver:
                processGameOver();
                break;
            default:
                running = false;
                break;
        }
    }
}
```

```

        }
    }

void Game::processMenu() {
    system("cls");
    std::cout << "==== ИГРА ===" << std::endl;
    std::cout << "1. Новая игра" << std::endl;
    std::cout << "2. Загрузить игру" << std::endl;
    std::cout << "3. Выход" << std::endl;
    std::cout << "Выберите действие: ";

    int choice;
    std::cin >> choice;
    std::cin.ignore();

    switch (choice) {
        case 1:
            currentPhase = GamePhase::GameStart;
            break;
        case 2:
            if (loadGame()) {
                currentPhase = GamePhase::LevelStart;
            } else {
                pause("Не удалось загрузить игру. Нажмите Enter для
продолжения...");
            }
            break;
        case 3:
            currentPhase = GamePhase::Exit;
            break;
        default:
            pause("Неверный выбор. Нажмите Enter для
продолжения...");
            break;
    }
}

void Game::processGameStart() {
    initializeNewGame();
    currentPhase = GamePhase::LevelStart;
}

void Game::processLevelStart() {
    system("cls");
    std::cout << "==== УРОВЕНЬ " << currentLevel << " ===" <<
std::endl;
    pause("Нажмите Enter, чтобы начать уровень...");
    currentPhase = GamePhase::PlayerTurn;
}

void Game::processPlayerTurn() {
    if (!field.getPlayer() || !field.getPlayer()->isAlive()) {
        currentPhase = GamePhase::GameOver;
        return;
    }

    if (checkLevelComplete())

```

```

        currentPhase = GamePhase::LevelComplete;
        return;
    }

    displayGameScreen();

    std::cout << "Команды: W/A/S/D - движение, M - переключить режим,
С - заклинание, V - сохранить, Q - выход\n";
    std::cout << "Ваш ход: ";

    char command;
    std::cin >> command;
    std::cin.ignore();

    int dx = 0, dy = 0;
    bool turnUsed = false;

    switch (std::tolower(command)) {
        case 'w': dy = -1; break;
        case 's': dy = 1; break;
        case 'a': dx = -1; break;
        case 'd': dx = 1; break;
        case 'm':
            if (field.getPlayer()) {
                field.getPlayer()->switchCombatMode();
            }
            return; // Don't use turn for mode switch
        case 'c':
            if (!field.getPlayer()) break;

            int spellIndex, targetX, targetY;
            std::cout << "Введите индекс заклинания (0-"
(field.getPlayer()->getHand().getSpellCount() - 1) << "): ";
            std::cin >> spellIndex;
            std::cout << "Введите координату X цели: ";
            std::cin >> targetX;
            std::cout << "Введите координату Y цели: ";
            std::cin >> targetY;
            std::cin.ignore();

            if (field.getPlayer()->castSpell(spellIndex, targetX,
targetY, field)) {
                turnUsed = true;
                pause();
            } else {
                pause("Заклинание не применено. Нажмите Enter для
продолжения...");
            }
            break;
        }
        case 'v':
            if (saveGame()) {
                pause("Игра сохранена! Нажмите Enter для
продолжения...");
            } else {
                pause("Ошибка сохранения! Нажмите Enter для
продолжения...");
            }
    }
}

```

```

        return; // Don't use turn for saving
    }
    case 'q':
        currentPhase = GamePhase::Menu;
        return;
    default:
        pause("Неверная команда! Нажмите Enter для
продолжения...");
        return;
    }

    if ((dx != 0 || dy != 0) && field.getPlayer() && !turnUsed) {
        field.getPlayer()->move(dx, dy, field);
        turnUsed = true;
    }

    // Check if enemy was killed and give new spell
    int currentEnemyCount =
static_cast<int>(field.getEnemies().size());
    int newKills = initialEnemyCount - currentEnemyCount -
enemiesKilled;
    if (newKills > 0) {
        enemiesKilled += newKills;
        tryGiveNewSpell();
    }

    if (turnUsed) {
        currentPhase = GamePhase::AlliesTurn;
    }
}

void Game::processAlliesTurn() {
    // Placeholder for future allies implementation
    // For now, just move to enemies turn
    currentPhase = GamePhase::EnemiesTurn;
}

void Game::processEnemiesTurn() {
    for (const auto& enemy : field.getEnemies()) {
        int dx = (std::rand() % 3) - 1; // -1, 0, 1
        int dy = (std::rand() % 3) - 1;

        if (dx != 0 || dy != 0) {
            enemy->move(dx, dy, field);
        }

        if (!field.getPlayer() || !field.getPlayer()->isAlive()) {
            currentPhase = GamePhase::GameOver;
            return;
        }
    }

    currentPhase = GamePhase::EnemyBaseTurn;
}

void Game::processEnemyBaseTurn() {
    // Placeholder for future enemy bases/towers implementation
    // For now, just move back to player turn
}

```

```

        currentPhase = GamePhase::PlayerTurn;
    }

void Game::processLevelComplete() {
    system("cls");
    field.draw();
    std::cout << "\n==== УРОВЕНЬ " << currentLevel << " ПРОЙДЕН! ==="
<< std::endl;
    showStats();

    currentLevel++;
    pause("Нажмите Enter для продолжения...");

    // Initialize next level
    initializeLevel(currentLevel);
    currentPhase = GamePhase::LevelStart;
}

void Game::processGameOver() {
    system("cls");
    field.draw();
    std::cout << "\n==== ИГРА ОКОНЧЕНА ===" << std::endl;
    showStats();

    std::cout << "\n1. Начать заново" << std::endl;
    std::cout << "2. Выход в меню" << std::endl;
    std::cout << "Выберите действие: ";

    int choice;
    std::cin >> choice;
    std::cin.ignore();

    if (choice == 1) {
        currentPhase = GamePhase::GameStart;
    } else {
        currentPhase = GamePhase::Menu;
    }
}

void Game::initializeNewGame() {
    currentLevel = 1;
    enemiesKilled = 0;
    initializeLevel(currentLevel);
}

void Game::initializeLevel(int level) {
    // Clear field
    for (int y = 0; y < field.getHeight(); ++y) {
        for (int x = 0; x < field.getWidth(); ++x) {
            field.clearCell(x, y);
        }
    }

    // Initialize walls
    for (int i = 0; i < field.getHeight(); i++) {
        for (int j = 0; j < field.getWidth(); j++) {
            if (std::rand() % 10 == 0) {

```

```

        // Walls are handled by Field's initializeWalls, but
we need to clear first
    }
}
}

// Create new field with walls
field = Field(field.getWidth(), field.getHeight());

// Place player
auto player = std::make_unique<Player>(1, 1);
field.setPlayer(std::move(player));

// Place enemies (more enemies per level)
int enemyCount = 3 + (level - 1) * 2;
initialEnemyCount = enemyCount;
int width = field.getWidth();
int height = field.getHeight();

for (int i = 0; i < enemyCount; i++) {
    int x, y;
    int attempts = 0;
    do {
        x = std::rand() % width;
        y = std::rand() % height;
        attempts++;
    } while ((!field.isEmpty(x, y) ||
              (x == field.getPlayer()->getX() && y ==
field.getPlayer()->getY())) &&
              attempts < 100);

    if (attempts < 100) {
        field.addEnemy(std::make_unique<Enemy>(x, y));
    }
}

enemiesKilled = 0;
}

bool Game::checkLevelComplete() const {
    return field.getEnemies().empty();
}

bool Game::checkGameOver() const {
    return !field.getPlayer() || !field.getPlayer()->isAlive();
}

void Game::tryGiveNewSpell() {
    if (!field.getPlayer()) return;

    Hand& hand = field.getPlayer()->getHand();
    if (hand.isFull()) {
        std::cout << "Рука заклинаний полна! Новое заклинание не
получено.\n";
        return;
    }

    std::unique_ptr<Spell> newSpell;

```

```

int spellType = std::rand() % 2;

if (spellType == 0) {
    newSpell = std::make_unique<DirectDamageSpell>(3, 20);
} else {
    newSpell = std::make_unique<AreaDamageSpell>(3, 15);
}

if (hand.addSpell(std::move(newSpell))) {
    std::cout << "Вы получили новое заклинание: "
        << hand.getSpell(hand.getSpellCount() -
1)->getName() << "!\n";
    pause();
}
}

void Game::showStats() const {
    if (field.getPlayer()) {
        std::cout << "\nУровень: " << currentLevel
            << " | Здоровье: " << field.getPlayer()->getHP()
            << " | Урон: " << field.getPlayer()->getDamage()
            << " | Очки: " << field.getPlayer()->getScore()
            << " | Позиция: (" << field.getPlayer()->getX() <<
", " << field.getPlayer()->getY() << ")\n";
    }
}

void Game::showHand() const {
    if (field.getPlayer()) {
        const Hand& hand = field.getPlayer()->getHand();
        std::cout << "\n==== Рука заклинаний (" <<
hand.getSpellCount()
            << "/" << hand.getMaxSize() << ") ===\n";
        for (int i = 0; i < hand.getSpellCount(); i++) {
            Spell* spell = hand.getSpell(i);
            if (spell) {
                std::cout << "[" << i << "] " << spell->getName()
                    << " - " << spell->getDescription()
                    << " (Урон: " << spell->getDamage()
                    << ", Радиус: " << spell->getRadius() <<
")\n";
            }
        }
        std::cout << "=====\\n";
    }
}

void Game::displayGameScreen() const {
    system("cls");
    field.draw();
    showStats();
    showHand();
}

void Game::pause(const std::string& message) const {
    std::cout << message;
    std::cin.ignore();
    std::cin.get();
}

```

```

}

bool Game::saveGame() const {
    GameState state = captureGameState();
    return state.saveToFile(SAVE_FILE);
}

bool Game::loadGame() {
    GameState state;
    if (!state.loadFromFile(SAVE_FILE)) {
        return false;
    }
    restoreGameState(state);
    return true;
}

GameState Game::captureGameState() const {
    GameState state;

    if (field.getPlayer()) {
        Player* player = field.getPlayer();
        state.playerX = player->getX();
        state.playerY = player->getY();
        state.playerHP = player->getHP();
        state.playerMaxHP = player->getMaxHP();
        state.playerScore = player->getScore();
        state.meleeDamage = player->getMeleeDamage();
        state.rangedDamage = player->getRangedDamage();
        state.combatMode = (player->getCombatMode() ==
CombatMode::Melee) ? 0 : 1;

        const Hand& hand = player->getHand();
        state.handMaxSize = hand.getMaxSize();
        state.spells.clear();
        for (int i = 0; i < hand.getSpellCount(); ++i) {
            Spell* spell = hand.getSpell(i);
            if (spell) {
                GameState::SpellData spellData;
                // Determine spell type by name or use RTTI
                std::string name = spell->getName();
                if (name == "Прямой урон") {
                    spellData.type = 0;
                } else {
                    spellData.type = 1;
                }
                spellData.radius = spell->getRadius();
                spellData.damage = spell->getDamage();
                state.spells.push_back(spellData);
            }
        }
    }

    state.fieldWidth = field.getWidth();
    state.fieldHeight = field.getHeight();

    // Save walls
    state.walls.clear();
    for (int y = 0; y < field.getHeight(); ++y) {

```

```

        for (int x = 0; x < field.getWidth(); ++x) {
            if (field.isWall(x, y)) {
                state.walls.push_back({x, y});
            }
        }
    }

    // Save enemies
    state.enemies.clear();
    state.enemyHP.clear();
    state.enemyDamage.clear();
    for (const auto& enemy : field.getEnemies()) {
        state.enemies.push_back({enemy->getX(), enemy->getY()});
        state.enemyHP.push_back(enemy->getHP());
        state.enemyDamage.push_back(enemy->getDamage());
    }

    state.currentLevel = currentLevel;
    state.initialEnemyCount = initialEnemyCount;
    state.enemiesKilled = enemiesKilled;

    return state;
}

void Game::restoreGameState(const GameState& state) {
    currentLevel = state.currentLevel;
    initialEnemyCount = state.initialEnemyCount;
    enemiesKilled = state.enemiesKilled;

    // Recreate field without walls first
    field = Field(state.fieldWidth, state.fieldHeight);

    // Clear all cells first
    for (int y = 0; y < field.getHeight(); ++y) {
        for (int x = 0; x < field.getWidth(); ++x) {
            field.clearCell(x, y);
        }
    }

    // Restore walls
    for (const auto& wall : state.walls) {
        field.setWall(wall.first, wall.second);
    }

    // Restore player
    auto player = std::make_unique<Player>(state.playerX,
state.playerY,
                                             state.playerMaxHP,
state.meleeDamage, state.rangedDamage);
    player->setPosition(state.playerX, state.playerY);
    player->setHP(state.playerHP);
    player->setScore(state.playerScore);

    if (state.combatMode == 1 && player->getCombatMode() ==
CombatMode::Melee) {
        player->switchCombatMode();
    }
}

```

```

// Restore hand
Hand& hand = player->getHand();
// Clear existing spells and add saved ones
while (hand.getSpellCount() > 0) {
    hand.removeSpell(0);
}

for (const auto& spellData : state.spells) {
    std::unique_ptr<Spell> spell;
    if (spellData.type == 0) {
        spell = std::make_unique<DirectDamageSpell>(spellData.radius,
                                                spellData.damage);
    } else {
        spell = std::make_unique<AreaDamageSpell>(spellData.radius,
                                                spellData.damage);
    }
    hand.addSpell(std::move(spell));
}

field.setPlayer(std::move(player));

// Restore enemies
for (size_t i = 0; i < state.enemies.size(); ++i) {
    auto enemy = std::make_unique<Enemy>(state.enemies[i].first,
                                           state.enemies[i].second,
                                           state.enemyHP[i],
                                           state.enemyDamage[i]);
    field.addEnemy(std::move(enemy));
}
}
}

```

Game.h

```

#pragma once
#include <iostream>
#include <windows.h>
#include <cstdlib>
#include <ctime>
#include <memory>
#include <string>

#include "Field.h"
#include "Player.h"
#include "Enemy.h"
#include "GameState.h"

enum class GamePhase {
    Menu,
    GameStart,
    LevelStart,
    PlayerTurn,
    EnemiesTurn,
    AlliesTurn, //d
}
```

```

        EnemyBaseTurn, //d
        LevelComplete,
        GameOver,
        Exit
    };

    class Game {
private:
    Field field;
    bool running;
    int currentLevel;
    int initialEnemyCount;
    int enemiesKilled;
    GamePhase currentPhase;
    static const std::string SAVE_FILE;

    // Game loop phases
    void processMenu();
    void processGameStart();
    void processLevelStart();
    void processPlayerTurn();
    void processAlliesTurn(); // Placeholder for future allies
    void processEnemiesTurn();
        void processEnemyBaseTurn(); // Placeholder for future
bases/towers
    void processLevelComplete();
    void processGameOver();

    // Game state management
    void initializeNewGame();
    void initializeLevel(int level);
    bool checkLevelComplete() const;
    bool checkGameOver() const;
    void tryGiveNewSpell();

    // UI
    void showStats() const;
    void showHand() const;
    void displayGameScreen() const;

    // Save/Load
    bool saveGame() const;
    bool loadGame();
    GameState captureGameState() const;
    void restoreGameState(const GameState& state);

    // Helper methods
        void pause(const std::string& message = "Press Enter to
continue...") const;

public:
    Game();
    ~Game() = default;

    Game(const Game& other) = delete;
    Game& operator=(const Game& other) = delete;
    Game(Game&& other) = delete;
    Game& operator=(Game&& other) = delete;

```

```
    void run();
};
```

GameState.cpp

```
#include "GameState.h"
#include <iostream>
#include <sstream>
#include <stdexcept>

bool GameState::saveToFile(const std::string& filename) const {
    try {
        std::ofstream file(filename, std::ios::binary);
        if (!file.is_open()) {
            std::cerr << "Ошибка: Не удалось открыть файл для записи:"
" << filename << std::endl;
            return false;
        }

        // Write header
        file << "GAME_SAVE_V1\n";

        // Write player state
        file << "PLAYER\n";
        file << playerX << " " << playerY << " " << playerHP << " "
<< playerMaxHP << " "
                << playerScore << " " << meleeDamage << " " <<
rangedDamage << " " << combatMode << "\n";

        // Write hand
        file << "HAND\n";
        file << handMaxSize << " " << spells.size() << "\n";
        for (const auto& spell : spells) {
            file << spell.type << " " << spell.radius << " " <<
spell.damage << "\n";
        }

        // Write field
        file << "FIELD\n";
        file << fieldWidth << " " << fieldHeight << "\n";
        file << walls.size() << "\n";
        for (const auto& wall : walls) {
            file << wall.first << " " << wall.second << "\n";
        }

        // Write enemies
        file << "ENEMIES\n";
        file << enemies.size() << "\n";
        for (size_t i = 0; i < enemies.size(); ++i) {
            file << enemies[i].first << " " << enemies[i].second << "
"
                << enemyHP[i] << " " << enemyDamage[i] << "\n";
        }
    }
```

```

// Write game state
file << "GAME\n";
    file << currentLevel << " " << initialEnemyCount << " " <<
enemiesKilled << "\n";

file << "END\n";

if (file.fail()) {
    std::cerr << "Ошибка: Ошибка записи в файл: " << filename
<< std::endl;
    return false;
}

file.close();
return true;
} catch (const std::exception& e) {
    std::cerr << "Ошибка при сохранении: " << e.what() <<
std::endl;
    return false;
}
}

bool GameState::loadFromFile(const std::string& filename) {
try {
    std::ifstream file(filename, std::ios::binary);
    if (!file.is_open()) {
        std::cerr << "Ошибка: Не удалось открыть файл для чтения:
" << filename << std::endl;
        return false;
    }

    clear();

    std::string line;
    std::getline(file, line);
    if (line != "GAME_SAVE_V1") {
        std::cerr << "Ошибка: Неверный формат файла сохранения"
<< std::endl;
        return false;
    }

    // Read player
    std::getline(file, line);
    if (line != "PLAYER") {
        std::cerr << "Ошибка: Ожидался раздел PLAYER" <<
std::endl;
        return false;
    }
    file >> playerX >> playerY >> playerHP >> playerMaxHP
        >> playerScore >> meleeDamage >> rangedDamage >>
combatMode;
    file.ignore();

    // Read hand
    std::getline(file, line);
    if (line != "HAND") {
        std::cerr << "Ошибка: Ожидался раздел HAND" << std::endl;
        return false;
    }
}

```

```

    }

    int spellCount;
    file >> handMaxSize >> spellCount;
    file.ignore();
    spells.resize(spellCount);
    for (int i = 0; i < spellCount; ++i) {
        file >> spells[i].type >> spells[i].radius >>
    spells[i].damage;
        file.ignore();
    }

    // Read field
    std::getline(file, line);
    if (line != "FIELD") {
        std::cerr << "Ошибка: Ожидался раздел FIELD" <<
    std::endl;
        return false;
    }
    file >> fieldWidth >> fieldHeight;
    file.ignore();
    int wallCount;
    file >> wallCount;
    file.ignore();
    walls.resize(wallCount);
    for (int i = 0; i < wallCount; ++i) {
        file >> walls[i].first >> walls[i].second;
        file.ignore();
    }

    // Read enemies
    std::getline(file, line);
    if (line != "ENEMIES") {
        std::cerr << "Ошибка: Ожидался раздел ENEMIES" <<
    std::endl;
        return false;
    }
    int enemyCount;
    file >> enemyCount;
    file.ignore();
    enemies.resize(enemyCount);
    enemyHP.resize(enemyCount);
    enemyDamage.resize(enemyCount);
    for (int i = 0; i < enemyCount; ++i) {
        file >> enemies[i].first >> enemies[i].second >>
    enemyHP[i] >> enemyDamage[i];
        file.ignore();
    }

    // Read game state
    std::getline(file, line);
    if (line != "GAME") {
        std::cerr << "Ошибка: Ожидался раздел GAME" << std::endl;
        return false;
    }
    file >> currentLevel >> initialEnemyCount >> enemiesKilled;
    file.ignore();

    std::getline(file, line);

```

```

        if (line != "END") {
            std::cerr << "Ошибка: Файл сохранения поврежден" <<
std::endl;
            return false;
        }

        if (file.fail()) {
            std::cerr << "Ошибка: Ошибка чтения файла: " << filename
<< std::endl;
            return false;
        }

        file.close();
        return true;
    } catch (const std::exception& e) {
        std::cerr << "Ошибка при загрузке: " << e.what() <<
std::endl;
        return false;
    }
}

void GameState::clear() {
    playerX = playerY = 0;
    playerHP = playerMaxHP = 100;
    playerScore = 0;
    meleeDamage = 15;
    rangedDamage = 5;
    combatMode = 0;
    handMaxSize = 5;
    spells.clear();
    fieldWidth = fieldHeight = 10;
    walls.clear();
    enemies.clear();
    enemyHP.clear();
    enemyDamage.clear();
    currentLevel = 1;
    initialEnemyCount = 0;
    enemiesKilled = 0;
}

```

GameState.h

```

#pragma once
#include <string>
#include <vector>
#include <fstream>

// Forward declarations
class Game;
class Field;
class Player;
class Hand;

struct GameState {

```

```

// Player state
int playerX, playerY;
int playerHP, playerMaxHP;
int playerScore;
int meleeDamage, rangedDamage;
int combatMode; // 0 = Melee, 1 = Ranged
int handMaxSize;

// Hand spells (type, radius, damage)
struct SpellData {
    int type; // 0 = DirectDamage, 1 = AreaDamage
    int radius;
    int damage;
};
std::vector<SpellData> spells;

// Field state
int fieldWidth, fieldHeight;
std::vector<std::pair<int, int>> walls; // (x, y) positions
std::vector<std::pair<int, int>> enemies; // (x, y) positions
std::vector<int> enemyHP; // HP for each enemy
std::vector<int> enemyDamage; // Damage for each enemy

// Game state
int currentLevel;
int initialEnemyCount;
int enemiesKilled;

// Serialization
bool saveToFile(const std::string& filename) const;
bool loadFromFile(const std::string& filename);

// Helper methods
void clear();
};

```