

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

Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Объектно-ориентированное программирование»

Студент гр. 4384

Овчаренко Я.В.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Изучить принципы объектно-ориентированного программирования.
Написать программу на языке C++, которая будет прототипом пошаговой игры с перемещением персонажа и сражением с врагами.

Задание.

На 6/3/1 баллов:

Создать класс игрока, который должен хранить информацию об игроке (его жизни, урон, очки, и т.д. - студент сам определяет необходимые для работы характеристики). Объект класса игрока должен перемещаться по карте. Если у игрока кончаются жизни, то происходит конец игры.

Создать класс врага, который хранит параметры жизней и урона. Объектами класса врага управляет компьютер. При перемещении, если враг пытается перейти на клетку с игроком, то перемещение не происходит, и игроку наносится урон.

Создать класс квадратного/прямоугольного игрового поля, по которому перемещаются игрок и враги. Игровое поле не должно быть меньше 10 на 10 клеток, и не больше 25 на 25 клеток. Размеры поля задаются через конструктор. Рекомендуется для хранения информации об отдельных клетках поля создать отдельный класс.

Реализовать конструкторы перемещения и копирования для поля, а также соответствующие операторы присваивания с копированием и перемещением (должна происходить глубокая копия).

На 8/4/1.5 баллов:

Реализовать непроходимые клетки на поле. При попытке врагов или игрока перейти на такую клетку, перемещение не происходит. Заполнения поля непроходимыми клетками происходит в момент создания поля.

Добавить возможность для игрока переключаться на ближний или дальний бой с изменением значения наносимого урона. Такое переключение требует один ход.

На 10/5/2 баллов:

Добавить класс вражеского здания. Такое здание размещается на карте, и раз в несколько ходов создает нового врага возле себя. Количество ходов до создания нового врага задается в конструкторе.

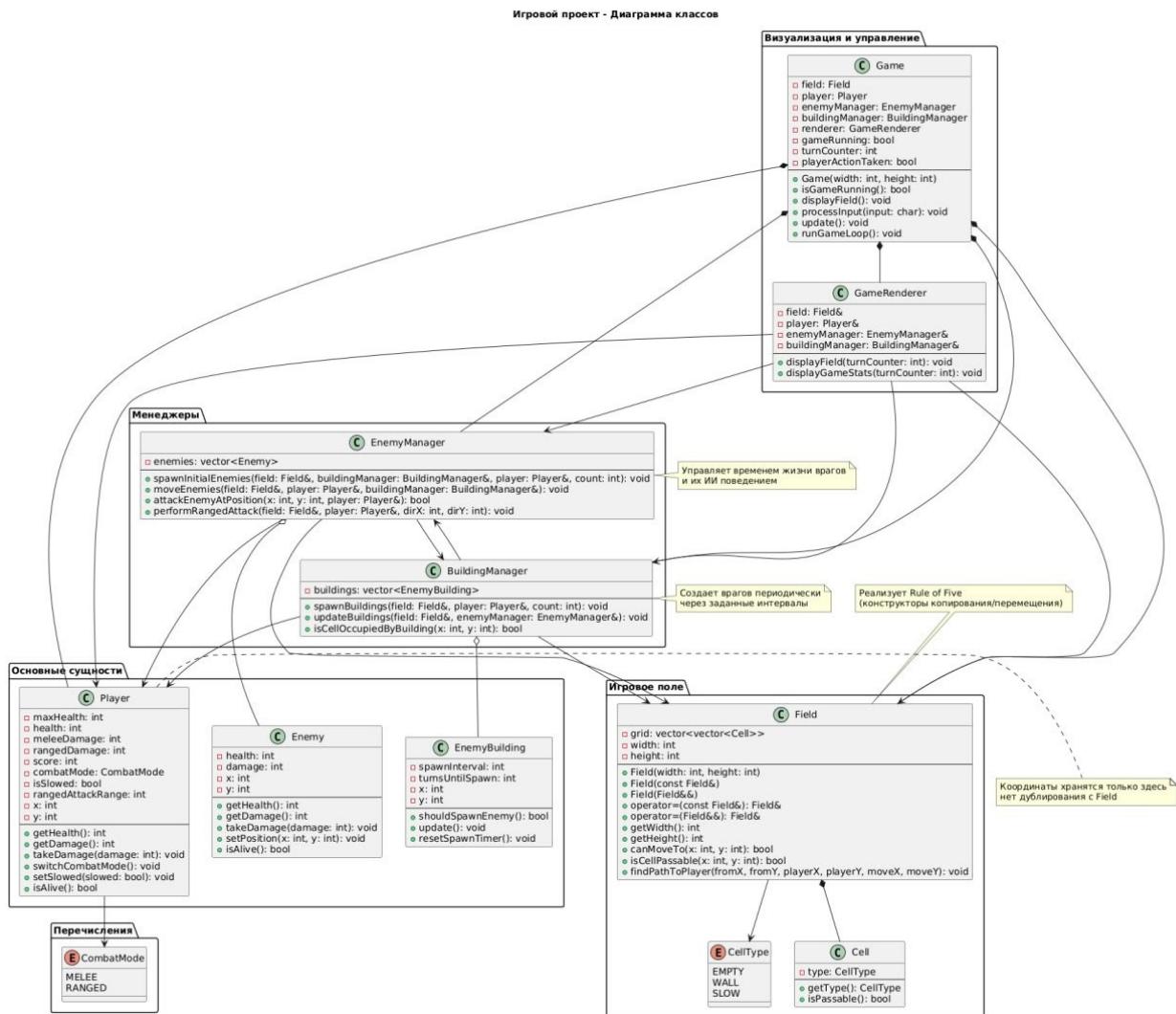
Реализовать замедляющие клетки на поле. Если игрок переходит на такую клетку, то он не может двигаться на следующий ход.

Выполнение работы.

Была реализована программа, содержащая все указанные в условие лабораторной работы классы и их поля и методы, а именно:

- Класс игрока с возможностью переключения типа атаки;
- Класс врага;
- Класс игрового поля;
- Непроходимый и замедляющий тип клеток;
- Класс вражеского здания.

Архитектура программы.



В программе реализована иерархия классов, соответствующая принципам

ООП.

Основные классы:

- Player – класс игрока
- Enemy – класс врага
- EnemyBuilding – класс вражеского здания
- Cell – класс клетки поля
- Field – класс игрового поля
- EnemyManager – класс управления врагами
- BuildingManager – класс управления зданиями
- GameRenderer – класс отображения игры
- Game – основной класс, управляющий игровым циклом

Дополнительные enum классы:

- CellType – класс типов клеток
- CombatMode – класс режимов боя

Описание классов.

Основные классы:

- Класс Player

Класс содержит характеристики и методы игрока.

Поля класса:

- health, maxHealth – текущее и максимальное здоровье
- meleeDamage, rangedDamage – урон в ближнем и дальнем бою
- score – очки игрока
- combatMode – текущий режим боя
- isSlowed – состояние замедления
- rangedAttackRange – дистанция дальней атаки
- x, y – координаты на поле

Методы класса:

- getDamage – возвращает урон в зависимости от текущего режима боя
- takeDamage – метод получения урона
- setPosition – метод изменения позиции игрока
- switchCombatMode – метод смены типа атаки
- addScore – метод получения очков
- setSlowed – метод установки состояния замедления
- Класс Enemy

Класс содержит характеристики и методы вражеских персонажей.

Поля класса:

- health – здоровье врага
- damage – урон врага
- x, y – координаты на поле

Методы класса:

- takeDamage – метод получения урона
- setPosition – метод изменения позиции
- isAlive – проверка, жив ли враг

- Класс EnemyBuilding

Класс содержит характеристики и методы вражеских зданий.

Поля класса:

- spawnInterval – период спавна врага
- turnsUntilSpawn – счётчик ходов до следующего спавна
- x, y – координаты на поле

Методы класса:

- shouldSpawnEnemy – метод проверки возможности спавна
- update – уменьшение счётчика ходов
- resetSpawnTimer – обнуление счётчика ходов

- Класс Cell

Класс содержит характеристики и методы клеток поля.

Поля класса:

- type – тип клетки

Методы класса:

- getType – получение типа клетки
- setType – метод установки типа клетки
- isPassable – проверка, проходима ли клетка

- Класс Field

Класс содержит характеристики и методы игрового поля, а также методы взаимодействия различных классов с игровым полем.

Поля класса:

- width – ширина поля
- height – высота поля
- grid – двумерный массив клеток поля

Методы класса:

- canMoveTo – метод проверки возможности перемещения в клетку. Проверяет, находится ли клетка в пределах поля и проходима ли она
- canPlaceEntity – метод проверки возможности размещения сущности. Проверяет, что клетка свободна и не занята игроком
- findPathToPlayer – метод вычисления пути к игроку. Определяет направление движения врага к игроку с учетом препятствий
- hasLineOfSight – метод проверки видимости между двумя точками
- isCellPassable – проверка проходимости клетки
- isSlowCell – проверка, является ли клетка замедляющей
- Класс EnemyManager

Класс управления вражескими персонажами.

Поля класса:

- enemies – массив вражеских персонажей

Методы класса:

- spawnInitialEnemies – метод создания начальных врагов. Размещает врагов на поле в случайных позициях, избегая зданий и игрока
- moveEnemies – метод процесса хода врага. Передвигает каждого живого врага в направлении игрока. Если враг пытается перейти на клетку с игроком, то перемещение не происходит, и игроку наносится урон. Если враг пытается перейти на клетку с другим врагом или зданием, перемещение не происходит.
- attackEnemyAtPosition – метод атаки врага. Если игрок пытается перейти на клетку с врагом, то перемещение не происходит, и врагу наносится урон

- `performRangedAttack` – метод дальней атаки. Выпускает снаряд в указанном направлении, нанося урон первому встречному врагу или стене
- `isCellOccupiedByEnemy` – проверка занятости клетки врагом

- Класс `BuildingManager`

Класс управления вражескими зданиями.

Поля класса:

- `buildings` – массив вражеских зданий

Методы класса:

- `spawnBuildings` – метод добавления на поле зданий. Размещает здания на поле в случайных позициях, избегая позиции игрока
- `updateBuildings` – метод процесса хода здания. Проверяет у каждого здания, прошло ли необходимое количество ходов с момента предыдущего спавна. Если да, то пытается спawnить нового врага в соседней клетке. Враги спавнятся вокруг здания, но не внутри его.
- `isCellOccupiedByBuilding` – проверка занятости клетки зданием

- Класс `GameRenderer`

Класс отображения игрового состояния.

Поля класса:

- `field` – игровое поле
- `player` – игрок
- `enemyManager` – менеджер врагов
- `buildingManager` – менеджер зданий

Методы класса:

- `displayField` – метод отображения поля и статистики. Отображает текущее состояние поля, позиции всех объектов и статистику игрока

- displayGameStats – метод отображения статистики игры
- Класс Game
 - Основной класс, управляющий игровым циклом и координирующий работу всех компонентов.
 - Поля класса:
 - field – игровое поле
 - player – игрок
 - enemyManager – менеджер врагов
 - buildingManager – менеджер зданий
 - renderer – рендерер игры
 - gameRunning – состояние игры
 - turnCounter – счётчик ходов
 - playerActionTaken – флаг совершения действия игроком
 - Методы класса:
 - processInput – метод обработки ввода игрока. Обрабатывает команды движения, смены режима боя, дальней атаки и выхода из игры
 - update – метод обновления игрового состояния. Вызывает обновление врагов и зданий только если игрок совершил действие. Увеличивает счётчик ходов.
 - runGameLoop – метод запуска игрового цикла
 - isGameOver – метод окончания игры. Проверяет здоровье игрока, если оно равно 0, то игра заканчивается.
- Дополнительные enum классы:
 - Класс CellType
 - Значения:
 - EMPTY – пустая клетка
 - WALL – непроходимая клетка
 - SLOW – замедляющая клетка
 - Класс CombatMode

Значения:

- MELEE – ближний бой
- RANGED – дальний бой

Разработанный программный код см. в приложении А.

Выводы.

Была изучена парадигма объектно-ориентированного программирования.

Была реализована программа на языке C++ содержащая основные классы игры с необходимыми полями и методами.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: buildingmanager.h

```
#ifndef BUILDING_MANAGER_H
#define BUILDING_MANAGER_H

#include <vector>
#include "enemybuilding.h"
#include "gametypes.h"

class BuildingManager {
private:
    std::vector<EnemyBuilding> buildings;

public:
    void spawnBuildings(Field& field, Player& player, int count);
    void updateBuildings(Field& field, EnemyManager& enemyManager);
    bool isCellOccupiedByBuilding(int x, int y) const;
    const std::vector<EnemyBuilding>& getBuildings() const { return
buildings; }
};

#endif
```

Название файла: cell.h

```
#ifndef CELL_H
#define CELL_H

enum class CellType {
    EMPTY,
    WALL,
    SLOW
};

class Cell {
private:
    CellType type;

public:
    Cell(CellType cellType = CellType::EMPTY);

    CellType getType() const;
    void setType(CellType cellType);
```

```
    bool isPassable() const;  
};  
  
#endif
```

Название файла: enemy.h

```
#ifndef ENEMY_H  
#define ENEMY_H  
  
class Enemy {  
private:  
    int health;  
    int damage;  
    int x;  
    int y;  
  
public:  
    Enemy(int health = 30, int damage = 10, int x = -1, int y = -1);  
  
    int getHealth() const;  
    int getDamage() const;  
    int getX() const;  
    int getY() const;  
  
    void takeDamage(int damage);  
    void setPosition(int newX, int newY);  
    bool isAlive() const;  
};  
  
#endif
```

Название файла: enemybuilding.h

```
#ifndef ENEMY_BUILDING_H  
#define ENEMY_BUILDING_H  
  
class EnemyBuilding {
```

```

private:
    int spawnInterval;
    int turnsUntilSpawn;
    int x;
    int y;

public:
    EnemyBuilding(int spawnInterval = 5, int x = -1, int y = -1);

    bool shouldSpawnEnemy();
    void update();
    void resetSpawnTimer();
    int getTurnsUntilSpawn() const;
    int getX() const;
    int getY() const;
    void setPosition(int newX, int newY);
};

#endif

```

Название файла: enemymanager.h

```

#ifndef ENEMY_MANAGER_H
#define ENEMY_MANAGER_H

#include <vector>
#include "enemy.h"
#include "gametypes.h"

class EnemyManager {
private:
    std::vector<Enemy> enemies;

public:
    void spawnInitialEnemies(Field& field, BuildingManager& buildingManager, Player& player, int count);
    void moveEnemies(Field& field, Player& player, BuildingManager& buildingManager);
    bool attackEnemyAtPosition(int x, int y, Player& player);
};

```

```

        void performRangedAttack(Field& field, Player& player, int
directionX, int directionY);
        bool isCellOccupiedByEnemy(int x, int y) const;

        // Getters
        const std::vector<Enemy>& getEnemies() const { return
enemies; }
        std::vector<Enemy>& getEnemies() { return enemies; }
};

#endif

```

Название файла: field.h

```

#ifndef FIELD_H
#define FIELD_H

#include <vector>
#include <memory>
#include <utility>
#include "cell.h"

class Field {
private:
    std::vector<std::vector<Cell>> grid;
    int width;
    int height;

public:
    Field(int width, int height);

    // Rule of Five
    Field(const Field& other);
    Field(Field&& other) noexcept;
    Field& operator=(const Field& other);
    Field& operator=(Field&& other) noexcept;
    ~Field() = default;

    int getWidth() const;
}

```

```

        int getHeight() const;
        Cell getCell(int x, int y) const;
        CellType getCellType(int x, int y) const;

        bool canMoveTo(int x, int y) const;

        bool canPlaceEntity(int x, int y, int playerX, int playerY)
const;

        bool isCellPassable(int x, int y) const;
        bool isSlowCell(int x, int y) const;
        bool isValidPosition(int x, int y) const;

        void findPathToPlayer(int fromX, int fromY, int playerX, int
playerY, int& moveX, int& moveY) const;

        bool hasLineOfSight(int fromX, int fromY, int toX, int toY)
const;
    };

#endif

```

Название файла: game.h

```

#ifndef GAME_H
#define GAME_H

#include "field.h"
#include "player.h"
#include "enemymanager.h"
#include "buildingmanager.h"
#include "gamerenderer.h"

class Game {
private:
    Field field;
    Player player;
    EnemyManager enemyManager;
    BuildingManager buildingManager;

```

```

GameRenderer renderer;
bool gameRunning;
int turnCounter;
bool playerActionTaken;

void spawnPlayer();
void processCellEffects(int x, int y);
void displayHelp() const;

public:
    Game(int fieldWidth, int fieldHeight);

    bool isGameRunning() const;
    void displayField() const;
    void processInput(char input);
    void update();
    void runGameLoop();
};

#endif

```

Название файла: gamerenderer.h

```

#ifndef GAME_RENDERERTER_H
#define GAME_RENDERERTER_H

#include "field.h"
#include "player.h"
#include "enemymanager.h"
#include "buildingmanager.h"

class GameRenderer {
public:
    GameRenderer(Field& field, Player& player, EnemyManager&
enemyManager, BuildingManager& buildingManager);

    void displayField(int turnCounter) const;
    void displayGameStats(int turnCounter) const;

```

```
private:
    Field& field;
    Player& player;
    EnemyManager& enemyManager;
    BuildingManager& buildingManager;
};

#endif
```

Название файла: gametypes.h

```
#ifndef GAME_TYPES_H
#define GAME_TYPES_H

class Field;
class Player;
class EnemyManager;
class BuildingManager;

#endif
```

Название файла: player.h

```
#ifndef PLAYER_H
#define PLAYER_H

enum class CombatMode {
    MELEE,
    RANGED
};

class Player {
private:
    int maxHealth;
    int health;
    int meleeDamage;
    int rangedDamage;
    int score;
    CombatMode combatMode;
    bool isSlowed;
```

```

        int rangedAttackRange;

        // Новые поля - координаты игрока
        int x;
        int y;

public:
    Player(int health = 100, int meleeDamage = 15, int rangedDamage
= 8, int rangedRange = 3);

        // Геттеры и сеттеры для координат
        int getX() const;
        int getY() const;
        void setPosition(int newX, int newY);

        // Остальные методы
        int getHealth() const;
        int getMaxHealth() const;
        int getDamage() const;
        int getScore() const;
        CombatMode getCombatMode() const;
        bool getIsSlowed() const;
        int getRangedAttackRange() const;

        void takeDamage(int damage);
        void addScore(int points);
        void switchCombatMode();
        void setSlowed(bool slowed);
        void heal(int amount);

        bool isAlive() const;
};

#endif

```

Название файла: buildingmanager.cpp

```

#include "buildingmanager.h"
#include "field.h"

```

```

#include "enemymanager.h"
#include "player.h"
#include <iostream>
#include <random>

void BuildingManager::spawnBuildings(Field& field, Player& player,
int count) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> disX(1, field.getWidth() - 2);
    std::uniform_int_distribution<> disY(1, field.getHeight() - 2);
    std::uniform_int_distribution<> disInterval(3, 8);

    int playerX = player.getX();
    int playerY = player.getY();

    for (int i = 0; i < count; ++i) {
        int x, y;
        do {
            x = disX(gen);
            y = disY(gen);
        } while (!field.canPlaceEntity(x, y, playerX, playerY));

        buildings.push_back(EnemyBuilding(disInterval(gen), x, y));
    }
}

void BuildingManager::updateBuildings(Field& field, EnemyManager&
enemyManager) {
    for (auto& building : buildings) {
        building.update();

        if (building.shouldSpawnEnemy()) {
            // Try to spawn enemy in adjacent cells (not in the
            building itself)
            const std::vector<std::pair<int, int>> directions = {
                {0, 1}, {1, 0}, {0, -1}, {-1, 0},
                {1, 1}, {1, -1}, {-1, 1}, {-1, -1}
            };
        }
    }
}

```

```

        std::vector<std::pair<int, int>>
possibleSpawnPositions;

        // Find all valid spawn positions around the building
        int buildingX = building.getX();
        int buildingY = building.getY();

        for (const auto& dir : directions) {
            int newX = buildingX + dir.first;
            int newY = buildingY + dir.second;

            // Check if position is valid and not occupied
            if (field.isValidPosition(newX, newY) &&
                field.isCellPassable(newX, newY) &&
                !enemyManager.isCellOccupiedByEnemy(newX, newY)
&&
                !isCellOccupiedByBuilding(newX, newY)) {
                possibleSpawnPositions.emplace_back(newX,
newY);
            }
        }

        // Spawn enemy at a random valid position if available
        if (!possibleSpawnPositions.empty()) {
            std::random_device rd;
            std::mt19937 gen(rd());
            std::uniform_int_distribution<> dis(0,
possibleSpawnPositions.size() - 1);

            int randomIndex = dis(gen);
            int spawnX = possibleSpawnPositions[randomIndex].first;
            int spawnY = possibleSpawnPositions[randomIndex].second;

            enemyManager.getEnemies().push_back(Enemy(30, 10,
spawnX, spawnY));
        }
    }
}

```

```

        building.resetSpawnTimer(); // Сбрасываем таймер
только после успешного спавна

    std::cout << "Building at (" << buildingX << ", "
<< buildingY
                << ") spawned new enemy at (" << spawnX
<< ", " << spawnY << ")\n";
    } else {
        // Если нет свободных клеток, все равно сбрасываем
таймер, чтобы не спавнить каждый ход
        building.resetSpawnTimer();
        std::cout << "Building at (" << buildingX << ", "
<< buildingY
                << ") tried to spawn enemy but no free
space available\n";
    }
}

bool BuildingManager::isCellOccupiedByBuilding(int x, int y) const
{
    for (const auto& building : buildings) {
        if (building.getX() == x && building.getY() == y) {
            return true;
        }
    }
    return false;
}

```

Название файла: cell.cpp

```

#include "cell.h"

Cell::Cell(CellType cellType) : type(cellType) {}

CellType Cell::getType() const {
    return type;
}

```

```
void Cell::setType(CellType cellType) {
    type = cellType;
}
```

```
bool Cell::isPassable() const {
    return type != CellType::WALL;
}
```

Название файла: enemy.cpp

```
#include "enemy.h"
#include <algorithm>

Enemy::Enemy(int health, int damage, int x, int y)
    : health(health), damage(damage), x(x), y(y) {}
```

```
int Enemy::getHealth() const {
    return health;
}
```

```
int Enemy::getDamage() const {
    return damage;
}
```

```
int Enemy::getX() const {
    return x;
}
```

```
int Enemy::getY() const {
    return y;
}
```

```
void Enemy::takeDamage(int damage) {
    health = std::max(0, health - damage);
}
```

```
void Enemy::setPosition(int newX, int newY) {
    x = newX;
```

```
    y = newY;
}

bool Enemy::isAlive() const {
    return health > 0;
}
```

Название файла: enemybuilding.cpp

```
#include "enemybuilding.h"
```

```
EnemyBuilding::EnemyBuilding(int spawnInterval, int x, int y)
    : spawnInterval(spawnInterval), turnsUntilSpawn(0), x(x), y(y)
{
}

bool EnemyBuilding::shouldSpawnEnemy() {
    return turnsUntilSpawn <= 0;
}

void EnemyBuilding::update() {
    if (turnsUntilSpawn > 0) {
        turnsUntilSpawn--;
    }
}

void EnemyBuilding::resetSpawnTimer() {
    turnsUntilSpawn = spawnInterval;
}

int EnemyBuilding::getTurnsUntilSpawn() const {
    return turnsUntilSpawn;
}

int EnemyBuilding::getX() const {
    return x;
}

int EnemyBuilding::getY() const {
```

```

        return y;
    }

void EnemyBuilding::setPosition(int newX, int newY) {
    x = newX;
    y = newY;
}

```

Название файла: enemymanager.cpp

```

#include "enemymanager.h"
#include "field.h"
#include "player.h"
#include "buildingmanager.h"
#include <iostream>
#include <random>

void EnemyManager::spawnInitialEnemies(Field& field,
BuildingManager& buildingManager, Player& player, int count) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> disX(1, field.getWidth() - 2);
    std::uniform_int_distribution<> disY(1, field.getHeight() - 2);

    int playerX = player.getX();
    int playerY = player.getY();

    for (int i = 0; i < count; ++i) {
        int x, y;
        int attempts = 0;
        do {
            x = disX(gen);
            y = disY(gen);
            attempts++;
            if (attempts > 50) {
                std::cout << "Warning: Difficulty finding spawn
position for enemy " << i << "\n";
                break;
            }
        }
    }
}

```

```

        } while (!field.canPlaceEntity(x, y, playerX, playerY) ||
buildingManager.isCellOccupiedByBuilding(x, y));

    enemies.push_back(Enemy(30, 10, x, y));
}

std::cout << "Spawned " << enemies.size() << " initial enemies.
\n";
}

// Остальные методы EnemyManager остаются без изменений
void EnemyManager::moveEnemies(Field& field, Player& player,
BuildingManager& buildingManager) {
    int playerX = player.getX();
    int playerY = player.getY();

    int movedEnemies = 0;

    for (auto& enemy : enemies) {
        if (!enemy.isAlive()) continue;

        int currentX = enemy.getX();
        int currentY = enemy.getY();
        int newX = currentX;
        int newY = currentY;

        // Передаем координаты игрока в findPathToPlayer
        field.findPathToPlayer(currentX, currentY, playerX,
playerY, newX, newY);

        // Check if enemy is trying to move to player's cell
        if (newX == playerX && newY == playerY) {
            player.takeDamage(enemy.getDamage());
            std::cout << "Enemy attacks you for " <<
enemy.getDamage()
            << " damage! Your HP: " << player.getHealth()
<< "\n";
        }
    }
}

```

```

        // Check if enemy is trying to move to another enemy's cell
        or building
                else if (isCellOccupiedByEnemy(newX, newY) ||
buildingManager.isCellOccupiedByBuilding(newX, newY)) {
                    // Don't move - cell is occupied by another enemy or
building
                }
                else if (field.isCellPassable(newX, newY)) {
                    enemy.setPosition(newX, newY);
                    movedEnemies++;
                }
            }

        if (movedEnemies > 0) {
            std::cout << movedEnemies << " enemies moved this turn.\n";
        }
    }

    bool EnemyManager::attackEnemyAtPosition(int x, int y, Player&
player) {
        for (auto& enemy : enemies) {
            if (enemy.isAlive() && enemy.getX() == x && enemy.getY() ==
y) {
                enemy.takeDamage(player.getDamage());
                std::cout << "You attack enemy for " <<
player.getDamage()
                << " damage. Enemy HP: " << enemy.getHealth()
                << "\n";
                if (!enemy.isAlive()) {
                    player.addScore(10);
                    std::cout << "Enemy defeated! +10 score. Total: "
                << player.getScore() << "\n";
                }
                return true;
            }
        }
        return false;
    }
}

```

```

void EnemyManager::performRangedAttack(Field& field, Player&
player, int directionX, int directionY) {
    int playerX = player.getX();
    int playerY = player.getY();
    int range = player.getRangedAttackRange();

    int targetX = playerX;
    int targetY = playerY;

    for (int i = 0; i < range; ++i) {
        targetX += directionX;
        targetY += directionY;

        if (!field.isValidPosition(targetX, targetY)) break;

        for (auto& enemy : enemies) {
            if (enemy.isAlive() && enemy.getX() == targetX &&
            enemy.getY() == targetY) {
                enemy.takeDamage(player.getDamage());
                std::cout << "You ranged attack enemy at (" <<
targetX << ", " << targetY
                << ") for " << player.getDamage() << "
damage. Enemy HP: " << enemy.getHealth() << "\n";

                if (!enemy.isAlive()) {
                    player.addScore(10);
                    std::cout << "Enemy defeated! +10 score. Total:
" << player.getScore() << "\n";
                }
            }
        }
    }

    if (!field.isCellPassable(targetX, targetY)) {
        std::cout << "Your attack hit a wall at (" << targetX
<< ", " << targetY << ")\n";
        break;
    }
}

```

```

    }

    std::cout << "Ranged attack: no enemy hit within range.\n";
}

bool EnemyManager::isCellOccupiedByEnemy(int x, int y) const {
    for (const auto& enemy : enemies) {
        if (enemy.isAlive() && enemy.getX() == x && enemy.getY() ==
y) {
            return true;
        }
    }
    return false;
}

```

Название файла: field.cpp

```

#include "field.h"
#include <random>
#include <chrono>
#include <algorithm>
#include <cmath>

Field::Field(int width, int height)
: width(std::max(10, std::min(25, width))),  

height(std::max(10, std::min(25, height))) {

    grid.resize(height, std::vector<Cell>(width));

    // Initialize with random walls and slow cells
    std::mt19937  

gen(std::chrono::system_clock::now().time_since_epoch().count());
    std::uniform_real_distribution<> dis(0.0, 1.0);

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            double randomValue = dis(gen);
            if (randomValue < 0.15) { // 15% chance for wall
                grid[y][x] = Cell(CellType::WALL);
            }
        }
    }
}

```

```

        } else if (randomValue < 0.25) { // 10% chance for slow
cell
            grid[y][x] = Cell(CellType::SLOW);
        }
    }
}

// Walls on map edges
for (int y = 0; y < height; ++y) {
    grid[y][0] = Cell(CellType::WALL);
    grid[y][width - 1] = Cell(CellType::WALL);
}
for (int x = 0; x < width; ++x) {
    grid[0][x] = Cell(CellType::WALL);
    grid[height - 1][x] = Cell(CellType::WALL);
}
}

// Конструкторы и операторы остаются без изменений
Field::Field(const Field& other)
    : width(other.width), height(other.height), grid(other.grid) {}

Field::Field(Field&& other) noexcept
    : width(other.width),     height(other.height),
grid(std::move(other.grid)) {
    other.width = 0;
    other.height = 0;
}

Field& Field::operator=(const Field& other) {
    if (this != &other) {
        width = other.width;
        height = other.height;
        grid = other.grid;
    }
    return *this;
}

Field& Field::operator=(Field&& other) noexcept {

```

```

        if (this != &other) {
            width = other.width;
            height = other.height;
            grid = std::move(other.grid);

            other.width = 0;
            other.height = 0;
        }
        return *this;
    }

bool Field::isValidPosition(int x, int y) const {
    return x >= 0 && x < width && y >= 0 && y < height;
}

int Field::getWidth() const {
    return width;
}

int Field::getHeight() const {
    return height;
}

Cell Field::getCell(int x, int y) const {
    if (!isValidPosition(x, y)) {
        return Cell(CellType::WALL);
    }
    return grid[y][x];
}

CellType Field::getCellType(int x, int y) const {
    if (!isValidPosition(x, y)) {
        return CellType::WALL;
    }
    return grid[y][x].getType();
}

// Новый метод вместо movePlayer
bool Field::canMoveTo(int x, int y) const {

```

```

        return isValidPosition(x, y) && isCellPassable(x, y);
    }

    // Обновленный метод - теперь принимает координаты игрока
    bool Field::canPlaceEntity(int x, int y, int playerX, int playerY)
const {
    return isValidPosition(x, y) && isCellPassable(x, y) &&
        !(x == playerX && y == playerY);
}

bool Field::isCellPassable(int x, int y) const {
    if (!isValidPosition(x, y)) {
        return false;
    }
    return grid[y][x].isPassable();
}

bool Field::isSlowCell(int x, int y) const {
    return isValidPosition(x, y) && grid[y][x].getType() ==
CellType::SLOW;
}

// Обновленный метод - теперь принимает координаты игрока
void Field::findPathToPlayer(int fromX, int fromY, int playerX, int
playerY, int& moveX, int& moveY) const {
    moveX = fromX;
    moveY = fromY;

    // Simple AI: prioritize movement towards player
    // Try to move in the direction that reduces distance to player
    if (std::abs(playerX - fromX) > std::abs(playerY - fromY)) {
        // Prioritize horizontal movement
        if (playerX > fromX && isCellPassable(fromX + 1, fromY)) {
            moveX = fromX + 1;
        } else if (playerX < fromX && isCellPassable(fromX - 1,
fromY)) {
            moveX = fromX - 1;
        } else if (playerY > fromY && isCellPassable(fromX, fromY +
1)) {

```

```

        moveY = fromY + 1;
    } else if (playerY < fromY && isCellPassable(fromX, fromY -
1)) {
            moveY = fromY - 1;
        }
    } else {
        // Prioritize vertical movement
        if (playerY > fromY && isCellPassable(fromX, fromY + 1)) {
            moveY = fromY + 1;
        } else if (playerY < fromY && isCellPassable(fromX, fromY -
1)) {
            moveY = fromY - 1;
        } else if (playerX > fromX && isCellPassable(fromX + 1,
fromY)) {
            moveX = fromX + 1;
        } else if (playerX < fromX && isCellPassable(fromX - 1,
fromY)) {
            moveX = fromX - 1;
        }
    }
}

bool Field::hasLineOfSight(int fromX, int fromY, int toX, int toY)
const {
    // Bresenham's line algorithm for checking line of sight
    int dx = std::abs(toX - fromX);
    int dy = std::abs(toY - fromY);
    int sx = (fromX < toX) ? 1 : -1;
    int sy = (fromY < toY) ? 1 : -1;
    int err = dx - dy;

    int currentX = fromX;
    int currentY = fromY;

    while (currentX != toX || currentY != toY) {
        // Skip the starting position
        if (currentX != fromX || currentY != fromY) {
            if (!isCellPassable(currentX, currentY)) {
                return false;
            }
        }
        currentX += sx;
        currentY += sy;
        if (err > 0) {
            err -= dy;
        } else {
            err += dx;
        }
    }
}

```

```

        }

    }

    int e2 = 2 * err;
    if (e2 > -dy) {
        err -= dy;
        currentX += sx;
    }
    if (e2 < dx) {
        err += dx;
        currentY += sy;
    }
}

return true;
}

```

Название файла: game.cpp

```

#include "game.h"
#include <iostream>
#include <random>

Game::Game(int fieldWidth, int fieldHeight)
: field(fieldWidth, fieldHeight),
player(100, 15, 8, 5),
renderer(field, player, enemyManager, buildingManager),
gameRunning(true),
turnCounter(0),
playerActionTaken(false) {

spawnPlayer();
enemyManager.spawnInitialEnemies(field, buildingManager,
player, 3);
buildingManager.spawnBuildings(field, player, 2);

std::cout << "Game initialized with " <<
buildingManager.getBuildings().size()

```

```

        << " buildings that will spawn enemies periodically.
\n";
}

void Game::spawnPlayer() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> disX(1, field.getWidth() - 2);
    std::uniform_int_distribution<> disY(1, field.getHeight() - 2);

    int x, y;
    do {
        x = disX(gen);
        y = disY(gen);
        } while (!field.isCellPassable(x, y) ||
buildingManager.isCellOccupiedByBuilding(x, y));

    player.setPosition(x, y);
}

void Game::processCellEffects(int x, int y) {
    if (field.isSlowCell(x, y)) {
        player.setSlowed(true);
        std::cout << "You stepped on a slow cell! You'll miss next
turn.\n";
    }
}

bool Game::isGameRunning() const {
    return gameRunning && player.isAlive();
}

void Game::displayHelp() const {
    std::cout << "Controls:\n";
    std::cout << " w/a/s/d - move\n";
    std::cout << " m - switch combat mode (costs turn)\n";
    std::cout << " f - ranged attack (only in ranged mode)\n";
    std::cout << " h - show this help\n";
    std::cout << " q - quit\n";
}

```

```

        std::cout << "Current mode: " << (player.getCombatMode() == CombatMode::MELEE ? "Melee" : "Ranged") << "\n";
        std::cout << "Ranged attack: use f then direction (w/a/s/d) to attack in that direction\n";
    }

void Game::displayField() const {
    renderer.displayField(turnCounter);
}

void Game::processInput(char input) {
    playerActionTaken = false;

    int x = player.getX();
    int y = player.getY();
    int newX = x;
    int newY = y;

    // Determine movement direction
    switch (input) {
        case 'w': case 'W': newY = y - 1; break;
        case 's': case 'S': newY = y + 1; break;
        case 'a': case 'A': newX = x - 1; break;
        case 'd': case 'D': newX = x + 1; break;
        default: break;
    }

    // If player is slowed, they can only switch mode or do ranged attack, not move
    if (player.getIsSlowed()) {
        if (input == 'w' || input == 'a' || input == 's' || input == 'd' ||
            input == 'W' || input == 'A' || input == 'S' || input == 'D') {
            std::cout << "You are slowed and cannot move this turn!
\n";
            player.setSlowed(false);
            playerActionTaken = true;
            return;
        }
    }
}

```

```

        }

    }

    // Handle movement and combat
    switch (input) {
        case 'w': case 'W':
        case 's': case 'S':
        case 'a': case 'A':
        case 'd': case 'D':
            if (enemyManager.attackEnemyAtPosition(newX, newY,
player)) {
                playerActionTaken = true;
            } else if
(buildingManager.isCellOccupiedByBuilding(newX, newY)) {
                std::cout << "Cannot move there - building in the
way!\n";
            } else if (field.canMoveTo(newX, newY)) {
                player.setPosition(newX, newY);
                processCellEffects(newX, newY);
                playerActionTaken = true;
            } else {
                std::cout << "Cannot move there!\n";
            }
            break;

        case 'm': case 'M':
            player.switchCombatMode();
            std::cout << "Switched to "
                << (player.getCombatMode() ==
CombatMode::MELEE ? "Melee" : "Ranged")
                << " mode (turn consumed)\n";
            playerActionTaken = true;
            break;

        case 'f': case 'F':
            if (player.getCombatMode() == CombatMode::RANGED) {
                std::cout << "Enter direction for ranged attack
(w/a/s/d): ";
                char direction;

```

```

        std::cin >> direction;

        int dirX = 0, dirY = 0;
        switch (direction) {
            case 'w': case 'W': dirY = -1; break;
            case 's': case 'S': dirY = 1; break;
            case 'a': case 'A': dirX = -1; break;
            case 'd': case 'D': dirX = 1; break;
            default:
                std::cout << "Invalid direction!\n";
                return;
        }

        enemyManager.performRangedAttack(field, player,
dirX, dirY);
        playerActionTaken = true;
    } else {
        std::cout << "You can only use ranged attacks in
ranged mode!\n";
    }
    break;

    case 'q': case 'Q':
        gameRunning = false;
        std::cout << "Game ended by player.\n";
        break;

    case 'h': case 'H':
        displayHelp();
        break;

    default:
        std::cout << "Unknown command. Press 'h' for help.\n";
        break;
    }
}

void Game::update() {
    if (playerActionTaken) {

```

```

        enemyManager.moveEnemies(field, player, buildingManager);
        buildingManager.updateBuildings(field, enemyManager);
        turnCounter++;

        std::cout << "Turn " << turnCounter << " completed.
Buildings updated.\n";
    }

    if (!player.isAlive()) {
        gameRunning = false;
        std::cout << "Game Over! Final Score: " <<
player.getScore() << "\n";
    }
}

void Game::runGameLoop() {
    std::cout << "==== Enhanced Lab Game with Ranged Combat ===\n";
    std::cout << "Buildings will spawn enemies every few turns.\n";
    displayHelp();

    while (isGameRunning()) {
        displayField();

        char input;
        std::cout << "Enter command: ";
        std::cin >> input;

        processInput(input);
        update();
    }
}

```

Название файла: gamerenderer.cpp

```
#include "gamerenderer.h"
#include <iostream>
```

```
GameRenderer::GameRenderer(Field& field, Player& player,
```

```

        EnemyManager& enemyManager,
BuildingManager& buildingManager)
    : field(field), player(player), enemyManager(enemyManager),
buildingManager(buildingManager) {}

void GameRenderer::displayGameStats(int turnCounter) const {
    std::cout << "\nTurn " << turnCounter;
    std::cout << " | HP: " << player.getHealth() << "/" <<
player.getMaxHealth();
    std::cout << " | Score: " << player.getScore();
    std::cout << " | Mode: " << (player.getCombatMode() ==
CombatMode::MELEE ? "Melee" : "Ranged");
    std::cout << " | Slowed: " << (player.getIsSlowed() ? "Yes" :
"No");
    std::cout << " | Enemies: " <<
enemyManager.getEnemies().size();
    std::cout << "\n";
}

void GameRenderer::displayField(int turnCounter) const {
    displayGameStats(turnCounter);

    int playerX = player.getX();
    int playerY = player.getY();

    for (int y = 0; y < field.getHeight(); ++y) {
        for (int x = 0; x < field.getWidth(); ++x) {
            if (x == playerX && y == playerY) {
                std::cout << "P ";
            } else {
                bool hasEnemy = false;
                for (const auto& enemy : enemyManager.getEnemies())
{
                    if (enemy.getX() == x && enemy.getY() == y &&
enemy.isAlive()) {
                        std::cout << "E ";
                        hasEnemy = true;
                        break;
                    }
                }
            }
        }
    }
}

```

```

        }
        if (hasEnemy) continue;

        bool hasBuilding = false;
                for (const auto& building :
buildingManager.getBuildings()) {
            if (building.getX() == x && building.getY() ==
y) {
                std::cout << "B ";
                hasBuilding = true;
                break;
            }
        }
        if (hasBuilding) continue;

        switch (field.getCellType(x, y)) {
            case CellType::WALL: std::cout << "# "; break;
            case CellType::SLOW: std::cout << "~ "; break;
            case CellType::EMPTY: std::cout << ". "; break;
        }
    }
    std::cout << "\n";
}
}

```

Название файла: main.cpp

```

#include <iostream>
#include "game.h"

int main() {
    int width, height;

    std::cout << "Enter field width (10-25): ";
    std::cin >> width;

    std::cout << "Enter field height (10-25): ";
    std::cin >> height;

```

```
    Game game(width, height);
    game.runGameLoop();

    return 0;
}
```

Название файла: player.cpp

```
#include "player.h"
#include <algorithm>

Player::Player(int health, int meleeDamage, int rangedDamage, int
rangedRange)
    : maxHealth(health), health(health), meleeDamage(meleeDamage),
      rangedDamage(rangedDamage), score(0),
combatMode(CombatMode::MELEE),
      isSlowed(false), rangedAttackRange(rangedRange), x(-1), y(-1)
{}

// Новые методы для работы с координатами
int Player::getX() const {
    return x;
}

int Player::getY() const {
    return y;
}

void Player::setPosition(int newX, int newY) {
    x = newX;
    y = newY;
}

// Остальные методы остаются без изменений
int Player::getHealth() const {
    return health;
}
```

```

int Player::getMaxHealth() const {
    return maxHealth;
}

int Player::getDamage() const {
    return (combatMode == CombatMode::MELEE) ? meleeDamage : rangedDamage;
}

int Player::getScore() const {
    return score;
}

CombatMode Player::getCombatMode() const {
    return combatMode;
}

bool Player::getIsSlowed() const {
    return isSlowed;
}

int Player::getRangedAttackRange() const {
    return rangedAttackRange;
}

void Player::takeDamage(int damage) {
    health = std::max(0, health - damage);
}

void Player::addScore(int points) {
    score += points;
}

void Player::switchCombatMode() {
    combatMode = (combatMode == CombatMode::MELEE) ? CombatMode::RANGED : CombatMode::MELEE;
}

void Player::setSlowed(bool slowed) {

```

```
    isSlowed = slowed;
}

void Player::heal(int amount) {
    health = std::min(maxHealth, health + amount);
}

bool Player::isAlive() const {
    return health > 0;
}
```