

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

ОТЧЕТ
по лабораторной работе №2
по дисциплине «Объектно-ориентированное программирование»

Студентка гр. 4384

Калинина А.В.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

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 интерфейс карточки заклинания. Заклинание должно применяться игроком. На использование заклинания игрок тратит один ход.

Создать класс “руки” игрока, которая содержит все карточки заклинаний, которые игрок может применить в свой ход. Изначально рука игрока содержит только одно случайное заклинание. Реализовать возможность получать новые заклинание игроком, например, тратить очки на покупку или после уничтожения определенного кол-ва врагов. Размер “руки” должен быть ограничен и задается через конструктор.

Реализовать интерфейс заклинанием прямого урона. Это заклинание при использовании должно наносить урон врагу или вражескому зданию, если они находятся в достижимом радиусе. Если в качестве цели не выбран враг или вражеское здание, то заклинание не используется.

Реализовать интерфейс заклинания урона по площади. Это заклинание при использовании в допустимом радиусе наносит урон по области 2 на 2 клетки. Заклинание используется, даже если там нет никого.

Выполнение работы.

Была реализована программа, содержащая все указанные условия лабораторной работы:

- Интерфейс карточки заклинания;
- Класс «руки» игрока;
- Интерфейс заклинания прямого урона;
- Интерфейс заклинания урона по площади;

Архитектура программы.

В программе реализована иерархия классов, соответствующая принципам ООП.

Основные классы:

- главный управляющий класс, координирует игровой цикл
 - ядро игровой логики, обрабатывает правила и механики
 - хранилище состояния игры (поле, игрок, враги)
 - управление игровым полем и позициями объектов
 - пользовательский интерфейс (ввод/вывод)
 - управление игроком, его характеристиками и способностями
 - вражеские персонажи с координатами и поведением
 - базовый класс для всех игровых сущностей
 - отдельная клетка игрового поля
 - рука заклинаний игрока с ограниченным размером
 - система заклинаний (базовый класс и реализации)
 - пошаговая система и обработка действий
- магазин заклинаний

Описание классов.

- Класс Cell

Назначение: Представляет одну клетку игрового поля

Поля класса:

- ❖ content (char) - символ содержимого клетки

Методы класса:

- Cell() - конструктор, инициализирует клетку точкой '.'
- getContent() - возвращает содержимое клетки
- setContent(char) - устанавливает содержимое клетки
- isEmpty() - проверяет пуста ли клетка
- clear() - очищает клетку (устанавливает '.')

2. Класс Enemy

Назначение: Представляет вражеского персонажа

Поля класса:

- ❖ x, y (int) - координаты врага на поле

Методы класса:

- Enemy(int startX, int startY) - конструктор с начальными координатами
- getX(), getY() - возвращают координаты
- move(int newX, int newY) - перемещает врага

Класс Entity

Назначение: Базовый класс для всех игровых сущностей

Поля класса:

- ❖ health (int) - здоровье сущности
- ❖ damage (int) - наносимый урон

Методы класса:

- Entity(int startHealth, int startDamage) - конструктор
- getHealth(), getDamage() - геттеры здоровья и урона
amount) - получение урона
- isAlive() - проверка жива ли сущность

4. Класс GameField

Назначение: Управление игровым полем и позициями

Поля класса:

- ❖ width, height (int) - размеры поля
- ❖ grid (vector<vector<Cell>>) - сетка клеток
- ❖ playerX, playerY (int) - координаты игрока

Методы класса:

- GameField(int w, int h) - конструктор с проверкой размеров

- getWidth(), getHeight() - геттеры размеров
- setPlayerPosition(x, y) - установка позиции игрока
- getPlayerPosition() - получение позиции игрока
- ~~getCellContent(x, y)~~ - получение содержимого клетки
- setCellContent(x, y, content) - установка содержимого клетки
- isEmpty(x, y) - проверка пуста ли клетка
- clearCell(x, y), clearAllCells() - очистка клеток

проверка валидности позиции

Класс GameInterface

Назначение: Пользовательский интерфейс (ввод/вывод)

Классы внутри:

- обработка ввода
- отображение игры

Методы InputHandler:

- getInput() - основной ввод (WASD/C/M/Q)
- getTarget() - координаты для заклинания
- getSpellChoice(maxSpells) - выбор заклинания
- getShopChoice(maxSpells) - выбор в магазине

Методы RenderSystem:

- displayGameState(gameState) - отображение всего состояния
- displayGameInfo(player, x, y) - информация об игроке
- displayField(field) - отображение поля
- displaySpells(hand) - отображение заклинаний
- displayShop(availableSpells, player) - отображение магазина

6. Класс GameLogic

Назначение: Основная игровая логика и координация систем

Поля класса:

- состояние игры

- ❖ turnSystem (TurnSystem) - система ходов
- ❖ actionSystem (ActionSystem) - система действий
- ❖ shopSystem (ShopSystem) - система магазина
- ❖ rewardSystem (RewardSystem) - система наград

Методы класса:

- GameLogic() - конструктор, инициализирует игру
- PlayerAlive() - проверка жив ли игрок
- isPlayerTurn() - проверка чей ход
- checkPlayerVictory() - проверка победы
- getPlayerScore() - получение счета игрока
- processPlayerAction(input, inputHandler) - обработка действия игрока
- processEnemyTurn() - обработка хода врагов
- getGameState() - получение состояния игры

7. Класс GameManager

Назначение: Главный управляющий класс игры

Поля класса:

- ❖ gameLogic (GameLogic) - игровая логика
- ❖ inputHandler (InputHandler) - обработчик ввода
- ❖ renderSystem (RenderSystem) - система отображения
- ❖ gameRunning (bool) - флаг работы игры

Методы класса:

- GameManager() - конструктор
- run() - главный игровой цикл
- handleGameEnd() - обработка завершения игры

Класс GameState

Назначение: Хранение всего состояния игры

Поля класса:

- ❖ field (GameField) - игровое поле

- ❖ player (Player) - игрок
- ❖ enemies (vector<Enemy>) - враги

Методы класса:

- GameState() - конструктор
- getField(), getPlayer(), getEnemies() - геттеры
- isPlayerAlive() - проверка жив ли игрок
- checkPlayerVictory() - проверка победы

Класс Player

Назначение: Управление игроком и его способностями

Поля класса:

- ❖ score (int) - счет игрока
- ❖ mana, maxMana (int) - текущая и максимальная мана
- ❖ spellHand (SpellHand) - рука заклинаний

Методы класса:

- Player() - конструктор, создает случайное стартовое заклинание
- getScore(), getMana(), getMaxMana() - геттеры
- getSpellHand() - доступ к руке заклинаний
- addScore(points) - добавление очков
- addMana(amount), useMana(amount) - управление маной
- restoreMana() - полное восстановление маны
- castSpell(index, targetX, targetY, field, enemies) - применение заклинания

Класс RewardShop

Назначение: Система наград и магазина

Класс внутри:

- магазин заклинаний

Поля ShopSystem:

- availableSpells (vector<unique_ptr<Spell>>) - доступные заклинания

Методы ShopSystem:

- buySpell(player, index) - покупка заклинания
- interactWithPlayer(player, inputHandler) - взаимодействие с магазином
- getAvailableSpells() - получение доступных заклинаний

Класс SpellHand

Назначение: Управление заклинаниями игрока

Поля класса:

- ❖ spells (vector<unique_ptr<Spell>>) - заклинания в руке
- ❖ maxSize (int) - максимальный размер руки

Методы класса:

- SpellHand(int size) - конструктор с заданным размером
- addSpell(spell) - добавление заклинания
- removeSpell(index) - удаление заклинания
- castSpell(index, target, field, enemies, player) - применение заклинания
- getSpellCount(), getMaxSize() - геттеры
- isFull() - проверка заполненности
- displaySpells() - отображение заклинаний

Назначение: Базовая система заклинаний

Классы внутри:

- цель для заклинания

- абстрактный класс заклинания
- заклинание прямого урона
- заклинание урона по площади

Поля SpellTarget:

- ❖ targetX, targetY, casterX, casterY (int) - координаты

Методы Spell:

- cast(target, field, enemies, player) - абстрактный метод применения
- getDescription(), clone() - абстрактные методы
- getName(), getManaCost(), getRange() - геттеры

Поля DirectDamageSpell:

- ❖ damage (int) - урон

Поля AreaDamageSpell:

- ❖ damage (int) - урон по области

Класс TurnAction

Назначение: Управление пошаговой системой и действиями

Классы внутри:

- управление очередностью ходов
- обработка действий

Поля TurnSystem:

- ❖ playerTurn (bool) - флаг хода игрока

Методы TurnSystem:

- isPlayerTurn() - проверка чей ход
- endPlayerTurn(), endEnemyTurn() - завершение ходов

Методы ActionSystem:

- processPlayerMove(gameState, newX, newY) - обработка движения игрока
- processSpellCast(gameState, inputHandler) - обработка заклинаний
- processShopInteraction(gameState, shopSystem, inputHandler) - магазин

- processEnemyMoves(gameState) - обработка ходов врагов

Назначение: Файл констант и перечислений игры

Содержание:

Перечисления (enum class):

- типы заклинаний:

DIRECT_DAMAGE - прямое поражение (0)

AREA_DAMAGE - урон по площади (1)

- направления движения:

EAST - восток (0)

WEST - запад (1)

SOUTH - юг (2)

NORTH - север (3)

- Пространство имен CellContent - символы содержимого клеток:

- пустая клетка

PLAYER = 'P' - игрок

ENEMY = 'E' - враг

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Cell.cpp

```
#include "Cell.h"

Cell::Cell() : content('.') {
}

char Cell::getContent() const {
    return content;
}

void Cell::setContent(char newContent) {
    content = newContent;
}

bool Cell::isEmpty() const {
    return content == '.';
}

void Cell::clear() {
    content = '.';
}
```

Cell.h

```
#ifndef CELL_H
#define CELL_H

class Cell {
private:
    char content;

public:
    Cell();

    char getContent() const;
    void setContent(char newContent);
    bool isEmpty() const;
    void clear();
};

#endif
```

Enemy.cpp

```
#include "Enemy.h"

Enemy::Enemy(int startX, int startY)
    : Entity(30, 5), x(startX), y(startY) {
}

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

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

```

        return y;
    }

void Enemy::move(int newX, int newY) {
    x = newX;
    y = newY;
}

Enemy.h

#ifndef ENEMY_H
#define ENEMY_H

#include "Entity.h"

class Enemy : public Entity {
private:
    int x, y; // Координаты остаются - врагов много, они независимы от поля

public:
    Enemy(int startX, int startY);

    int getX() const;
    int getY() const;
    void move(int newX, int newY);
};

#endif

Entity.cpp
#include "Entity.h"

Entity::Entity(int startHealth, int startDamage)
    : health(startHealth), damage(startDamage) {
}

int Entity::getHealth() const {
    return health;
}

int Entity::getDamage() const {
    return damage;
}

void Entity::takeDamage(int amount) {
    health -= amount;
    if (health < 0) {
        health = 0;
    }
}

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

```

```

Entity.h
#ifndef ENTITY_H
#define ENTITY_H

class Entity {
protected:
    int health;
    int damage;

public:
    Entity(int startHealth, int startDamage);
    virtual ~Entity() = default;

    int getHealth() const;
    int getDamage() const;
    void takeDamage(int amount);
    bool isAlive() const;
};

#endif

```

```

GameField.cpp
#include "GameField.h"
#include <stdexcept>
#include <utility>

GameField::GameField(int w, int h) : width(w), height(h), playerX(0),
playerY(0) {
    if (w < 10 || w > 25 || h < 10 || h > 25) {
        throw std::invalid_argument("Field size must be between 10x10
and 25x25");
    }

    grid.resize(height);
    for (int i = 0; i < height; i++) {
        grid[i].resize(width);
    }
}

GameField::~GameField() {
}

GameField::GameField(const GameField& other)
    : width(other.width), height(other.height),
    playerX(other.playerX), playerY(other.playerY), grid(other.grid) {
}

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

```

```

GameField& GameField::operator=(const GameField& other) {
    if (this != &other) {
        GameField temp(other);
        swap(*this, temp);
    }
    return *this;
}

GameField& GameField::operator=(GameField&& other) noexcept {
    if (this != &other) {
        width = other.width;
        height = other.height;
        playerX = other.playerX;
        playerY = other.playerY;
        grid = std::move(other.grid);

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

void swap(GameField& first, GameField& second) noexcept {
    using std::swap;
    swap(first.width, second.width);
    swap(first.height, second.height);
    swap(first.playerX, second.playerX);
    swap(first.playerY, second.playerY);
    swap(first.grid, second.grid);
}

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

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

void GameField::setPlayerPosition(int x, int y) {
    if (isValidPosition(x, y)) {
        playerX = x;
        playerY = y;
    }
}

std::pair<int, int> GameField::getPlayerPosition() const {
    return { playerX, playerY };
}

bool GameField::movePlayer(int newX, int newY) {
    if (!isValidPosition(newX, newY)) {
        return false;
    }
    if (!isCellEmpty(newX, newY)) {

```

```

        return false;
    }
    playerX = newX;
    playerY = newY;
    return true;
}

char GameField::getCellContent(int x, int y) const {
    if (!isValidPosition(x, y)) return '.';
    return grid[y][x].getContent();
}

void GameField::setCellContent(int x, int y, char content) {
    if (isValidPosition(x, y)) {
        grid[y][x].setContent(content);
    }
}

bool GameField::isCellEmpty(int x, int y) const {
    if (!isValidPosition(x, y)) return true;
    return grid[y][x].isEmpty();
}

void GameField::clearCell(int x, int y) {
    if (isValidPosition(x, y)) {
        grid[y][x].clear();
    }
}

void GameField::clearAllCells() {
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            grid[y][x].clear();
        }
    }
}

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

```

```

GameField.h
#ifndef GAMEFIELD_H
#define GAMEFIELD_H

#include "Cell.h"
#include <vector>
#include <utility>

class GameField {
private:
    int width;
    int height;
    std::vector<std::vector<Cell>> grid;
    int playerX, playerY;

public:

```

```

GameField(int w, int h);
~GameField();

GameField(const GameField& other);
GameField(GameField&& other) noexcept;
GameField& operator=(const GameField& other);
GameField& operator=(GameField&& other) noexcept;
friend void swap(GameField& first, GameField& second) noexcept;

int getWidth() const;
int getHeight() const;
void setPlayerPosition(int x, int y);
std::pair<int, int> getPlayerPosition() const;
bool movePlayer(int newX, int newY);
char getCellContent(int x, int y) const;
void setCellContent(int x, int y, char content);
bool isCellEmpty(int x, int y) const;
void clearCell(int x, int y);
void clearAllCells();
bool isValidPosition(int x, int y) const;
};

#endif

GameLogic.cpp
#include "GameLogic.h"
#include "GameConstants.h"
#include "GameInterface.h"
#include <iostream>

GameLogic::GameLogic()
    : gameState(), turnSystem(), actionSystem(), shopSystem(),
rewardSystem() {
    gameState.getField().setPlayerPosition(0, 0);
    gameState.getEnemies().push_back(Enemy(5, 5));
    gameState.getEnemies().push_back(Enemy(10, 10));
}

bool GameLogic::isPlayerAlive() const {
    return gameState.getPlayer().isAlive();
}

bool GameLogic::isPlayerTurn() const {
    return turnSystem.isPlayerTurn();
}

bool GameLogic::checkPlayerVictory() const {
    for (const auto& enemy : gameState.getEnemies()) {
        if (enemy.isAlive()) {
            return false;
        }
    }
    return true;
}

int GameLogic::getPlayerScore() const {
    return gameState.getPlayer().getScore();
}

```

```

}

bool GameLogic::processPlayerAction(char input, InputHandler&
inputHandler) {
    bool actionSuccess = false;

    if (input == 'c' || input == 'C') {
        actionSuccess = actionSystem.processSpellCast(gameState,
inputHandler);
    }
    else if (input == 'm' || input == 'M') {
        actionSuccess = actionSystem.processShopInteraction(gameState,
shopSystem, inputHandler);
    }
    else {
        auto playerPos = gameState.getField().getPlayerPosition();
        int newX = playerPos.first;
        int newY = playerPos.second;

        switch (input) {
            case 'w': case 'W': newY--; break;
            case 's': case 'S': newY++; break;
            case 'a': case 'A': newX--; break;
            case 'd': case 'D': newX++; break;
            default: return false;
        }

        actionSuccess = actionSystem.processPlayerMove(gameState, newX,
newY);
    }

    if (actionSuccess) {
        turnSystem.endPlayerTurn();
    }
    return actionSuccess;
}

void GameLogic::processEnemyTurn() {
    actionSystem.processEnemyMoves(gameState);
    turnSystem.endEnemyTurn();
}

GameState& GameLogic::getGameState() {
    return gameState;
}

```

GameLogic.h

```

#ifndef GAMELOGIC_H
#define GAMELOGIC_H

#include "GameState.h"
#include "TurnAction.h"
#include "RewardShop.h"

```

```

class InputHandler;

class GameLogic {
private:
    GameState gameState;
    TurnSystem turnSystem;
    ActionSystem actionSystem;
    ShopSystem shopSystem;
    RewardSystem rewardSystem;

public:
    GameLogic();

    bool isPlayerAlive() const;
    bool isPlayerTurn() const;
    bool checkPlayerVictory() const;
    int getPlayerScore() const;

    bool processPlayerAction(char input, InputHandler& inputHandler);
    void processEnemyTurn();

    GameState& getGameState();
};

#endif

```

```

GameManager.cpp
#include "GameManager.h"
#include "GameConstants.h"
#include <iostream>

GameManager::GameManager()
    : gameLogic(), inputHandler(), renderSystem(), gameRunning(true) {
}

void GameManager::run() {
    std::cout << "Game Started! Use WASD to move, C to cast spell, M for
shop, Q to quit" << std::endl;

    while (gameRunning && gameLogic.isPlayerAlive()) {
        renderSystem.displayGameState(gameLogic.getGameState());

        if (gameLogic.checkPlayerVictory()) {
            std::cout << "Player victory! All enemies are destroyed!"
<< std::endl;
            std::cout << "Final score: " << gameLogic.getPlayerScore()
<< std::endl;
            gameRunning = false;
            break;
        }

        if (gameLogic.isPlayerTurn()) {
            std::cout << "=== YOUR TURN ===" << std::endl;
            std::cout << "Move (WASD), Cast spell (C), Shop (M), Quit
(Q): ";

```

```

        char input = inputHandler.getGameInput();

        if (input == 'q' || input == 'Q') break;

        if (!gameLogic.processPlayerAction(input, inputHandler)) {
            std::cout << "Action failed! Try again." << std::endl;
        }
    }
    else {
        std::cout << "=== ENEMY TURN ===" << std::endl;
        gameLogic.processEnemyTurn();

        if (!gameLogic.isPlayerAlive()) {
            std::cout << "GAME OVER! Your score: " <<
gameLogic.getPlayerScore() << std::endl;
            gameRunning = false;
        }
    }

    handleGameEnd();
}

void GameManager::handleGameEnd() const {
    if (gameLogic.isPlayerAlive() && !gameLogic.checkPlayerVictory()) {
        std::cout << "Game finished. Final score: " <<
gameLogic.getPlayerScore() << std::endl;
    }
}

```

```

GameManager.h
#ifndef GAMEMANAGER_H
#define GAMEMANAGER_H

#include "GameLogic.h"
#include "GameInterface.h"

class GameManager {
private:
    GameLogic gameLogic;
    InputHandler inputHandler;
    RenderSystem renderSystem;
    bool gameRunning;

    void handleGameEnd() const;

public:
    GameManager();
    void run();
};

#endif

```

```

GameInterface.cpp
#include "GameInterface.h"
#include "GameState.h"
#include "GameField.h"
#include "Player.h"
#include "SpellHand.h"
#include "SpellSystem.h"
#include "GameConstants.h"
#include <iostream>

char InputHandler::getGameInput() {
    char input;
    std::cin >> input;
    return input;
}

std::pair<int, int> InputHandler::getSpellTarget() {
    int targetX, targetY;
    std::cout << "Enter target coordinates (x y): ";
    std::cin >> targetX >> targetY;
    return { targetX, targetY };
}

int InputHandler::getSpellChoice(int maxSpells) {
    int choice;
    std::cout << "Choose spell (1-" << maxSpells << "): ";
    std::cin >> choice;
    return choice;
}

int InputHandler::getShopChoice(int maxSpells) {
    int choice;
    std::cout << "Choose spell to buy (1-" << maxSpells << ") or 0 to
cancel: ";
    std::cin >> choice;
    return choice;
}

void RenderSystem::displayGameState(const GameState& gameState) const {
    const auto& field = gameState.getField();
    const auto& player = gameState.getPlayer();
    const auto& enemies = gameState.getEnemies();
    GameField& nonConstField = const_cast<GameField&>(field);
    nonConstField.clearAllCells();

    for (const auto& enemy : enemies) {
        if (enemy.isAlive()) {
            nonConstField.setCellContent(enemy.getX(), enemy.getY(),
CellContent::ENEMY);
        }
    }

    auto playerPos = field.getPlayerPosition();
    nonConstField.setCellContent(playerPos.first, playerPos.second,
CellContent::PLAYER);
    displayGameInfo(player, playerPos.first, playerPos.second);
    displayField(field);
    displaySpells(player.getSpellHand());
}

```

```

}

void RenderSystem::displayGameInfo(const Player& player, int playerX,
int playerY) const {
    std::cout << "Health: " << player.getHealth();
    std::cout << " | Mana: " << player.getMana() << "/" <<
player.getMaxMana();
    std::cout << " | Score: " << player.getScore();
    std::cout << " | Position: (" << playerX << ", " << playerY << ")"
<< std::endl;
}

void RenderSystem::displayField(const GameField& field) const {
    for (int y = 0; y < field.getHeight(); y++) {
        for (int x = 0; x < field.getWidth(); x++) {
            std::cout << field.getCellContent(x, y) << " ";
        }
        std::cout << std::endl;
    }
}

void RenderSystem::displaySpells(const SpellHand& hand) const {
    hand.displaySpells();
}

void RenderSystem::displayShop(const
std::vector<std::unique_ptr<Spell>>& availableSpells,
const Player& player) const {
    std::cout << "=== SPELL SHOP ===" << std::endl;
    std::cout << "Your score: " << player.getScore() << " points" <<
std::endl;
    std::cout << "Free spell slots: "
<< (player.getSpellHand().getMaxSize()
player.getSpellHand().getSpellCount()) -
<< std::endl;
    std::cout << "Available spells:" << std::endl;

    for (int i = 0; i < availableSpells.size(); i++) {
        int cost = availableSpells[i]->getManaCost(); // Теперь это цена
в очках
        // Убираем дублирование - показываем только описание и цену
        std::cout << i + 1 << ". " <<
availableSpells[i]->getDescription()
<< " - Cost: " << cost << " points" << std::endl;
    }
}

GameInterface.h
#ifndef GAMEINTERFACE_H
#define GAMEINTERFACE_H

#include <utility>
#include <vector>
#include <memory>

class GameState;
class Player;
class SpellHand;

```

```

class Spell;
class GameField;

class InputHandler {
public:
    char getGameInput();
    std::pair<int, int> getSpellTarget();
    int getSpellChoice(int maxSpells);
    int getShopChoice(int maxSpells);
};

class RenderSystem {
public:
    void displayGameState(const GameState& gameState) const;
    void displayGameInfo(const Player& player, int playerX, int playerY)
const;
    void displayField(const GameField& field) const;
    void displaySpells(const SpellHand& hand) const;
    void displayShop(const std::vector<std::unique_ptr<Spell>>&
availableSpells,
const Player& player) const;
};

#endif

```

Main.cpp

```

#include "GameManager.h" // Включение главного класса управления игрой

// Главная функция программы - точка входа
int main() {
    GameManager game; // Создание объекта управления игрой
    game.run();        // Запуск главного игрового цикла
    return 0;          // Успешное завершение программы
}

```

Player.cpp

```

#include "Player.h"
#include "SpellSystem.h"
#include "GameField.h"
#include "Enemy.h"
#include "GameConstants.h"
#include <iostream>
#include <memory>
#include <cstdlib>
#include <ctime>

Player::Player()
    : Entity(100, 10), score(0), mana(50), maxMana(50), spellHand(3) {
    std::srand(std::time(nullptr));
    auto randomSpell = createRandomStarterSpell();
    spellHand.addSpell(std::move(randomSpell));
}

```

```

std::unique_ptr<Spell> Player::createRandomStarterSpell() const {
    int randomType = std::rand() % 2;
    if (randomType == 0) {
        return std::make_unique<DirectDamageSpell>("Starter Fireball",
8, 2, 15);
    }
    else {
        return std::make_unique<AreaDamageSpell>("Starter Fire Storm",
12, 1, 8);
    }
}

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

int Player::getMana() const {
    return mana;
}

int Player::getMaxMana() const {
    return maxMana;
}

SpellHand& Player::getSpellHand() {
    return spellHand;
}

const SpellHand& Player::getSpellHand() const {
    return spellHand;
}

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

void Player::addMana(int amount) {
    mana += amount;
    if (mana > maxMana) {
        mana = maxMana;
    }
}

bool Player::useMana(int amount) {
    if (mana >= amount) {
        mana -= amount;
        return true;
    }
    return false;
}

void Player::restoreMana() {
    mana = maxMana;
}

bool Player::castSpell(int spellIndex, int targetX, int targetY,
    GameField& field, std::vector<Enemy>& enemies) {

```

```

        std::string spellName = spellHand.getSpellName(spellIndex);
        if (spellName.empty()) {
            std::cout << "Invalid spell index!" << std::endl;
            return false;
        }
        int manaCost = spellHand.getSpellManaCost(spellIndex);
        if (!useMana(manaCost)) {
            std::cout << "Not enough mana! Required: " << manaCost
                << ", available: " << mana << std::endl;
            return false;
        }
        std::cout << "Casting " << spellName << "..." << std::endl;
        auto playerPos = field.getPlayerPosition();
        SpellTarget target(targetX, targetY, playerPos.first,
playerPos.second);
        return spellHand.castSpell(spellIndex, target, field, enemies,
*this);
    }

Player.h
#ifndef PLAYER_H
#define PLAYER_H

#include "Entity.h"
#include "SpellHand.h"
#include <memory>

class GameField;
class Enemy;

class Player : public Entity {
private:
    int score;
    int mana;
    int maxMana;
    SpellHand spellHand;

    std::unique_ptr<Spell> createRandomStarterSpell() const;

public:
    Player();

    // Убраны методы перемещения - координаты только в GameField
    int getScore() const;
    int getMana() const;
    int getMaxMana() const;
    SpellHand& getSpellHand();
    const SpellHand& getSpellHand() const;

    void addScore(int points);
    void addMana(int amount);
    bool useMana(int amount);
    void restoreMana();

    bool castSpell(int spellIndex, int targetX, int targetY,
        GameField& field, std::vector<Enemy>& enemies);
};

```

```
#endif
```

```
GameState.cpp
```

```
#include "GameState.h"
```

```
GameState::GameState()  
    : field(15, 15), player() {  
}
```

```
GameField& GameState::getField() {  
    return field;  
}
```

```
Player& GameState::getPlayer() {  
    return player;  
}
```

```
std::vector<Enemy>& GameState::getEnemies() {  
    return enemies;  
}
```

```
const GameField& GameState::getField() const {  
    return field;  
}
```

```
const Player& GameState::getPlayer() const {  
    return player;  
}
```

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

```
bool GameState::isPlayerAlive() const {  
    return player.isAlive();  
}
```

```
bool GameState::checkPlayerVictory() const {  
    for (const auto& enemy : enemies) {  
        if (enemy.isAlive()) {  
            return false;  
        }  
    }  
    return true;  
}
```

```
GameState.h
```

```
#ifndef GAMESTATE_H
```

```
#define GAMESTATE_H
```

```
#include "GameField.h"
```

```
#include "Player.h"
```

```

#include "Enemy.h"
#include <vector>

class GameState {
private:
    GameField field;
    Player player;
    std::vector<Enemy> enemies;

public:
    GameState();

    GameField& getField();
    Player& getPlayer();
    std::vector<Enemy>& getEnemies();

    const GameField& getField() const;
    const Player& getPlayer() const;
    const std::vector<Enemy>& getEnemies() const;

    bool isPlayerAlive() const;
    bool checkPlayerVictory() const;
};

#endif

```

RewardShop.cpp

```

#include "RewardShop.h"
#include "GameConstants.h"
#include <iostream>
#include <cstdlib>
#include <ctime>

RewardSystem::RewardSystem() : enemiesKilled(0) {
    std::srand(std::time(nullptr));
}

void RewardSystem::onEnemyKilled(Player& player) {
    enemiesKilled++;
    std::cout << "Enemies killed: " << enemiesKilled << std::endl;
}

void RewardSystem::giveVictoryReward(Player& player) const {
    std::cout << "Congratulations! You won!" << std::endl;
}

void RewardSystem::resetKillCounter() {
    enemiesKilled = 0;
}

int RewardSystem::getEnemiesKilled() const {
    return enemiesKilled;
}

ShopSystem::ShopSystem() {
    // Цена = стоимость в очках (убираем зависимость от маны)
}

```

```

        availableSpells.push_back(std::make_unique<DirectDamageSpell>("Ice
Arrow", 40, 4, 20));
        availableSpells.push_back(std::make_unique<AreaDamageSpell>("Ice
Storm", 90, 3, 12));
    }

    std::unique_ptr<Spell> ShopSystem::createSpellCopy(Spell* baseSpell)
    const {
        return baseSpell->clone();
    }

    int ShopSystem::calculateCost(Spell* spell) const {
        return spell->getManaCost(); // Теперь это цена в очках
    }

    bool ShopSystem::buySpell(Player& player, int spellIndex) const {
        if (spellIndex < 0 || spellIndex >= availableSpells.size()) {
            std::cout << "Invalid spell index!" << std::endl;
            return false;
        }

        Spell* baseSpell = availableSpells[spellIndex].get();
        int cost = calculateCost(baseSpell);

        if (player.getScore() < cost) {
            std::cout << "Not enough points! Need " << cost << ", but you
have " << player.getScore() << std::endl;
            return false;
        }

        if (player.getSpellHand().isFull()) {
            std::cout << "Your hand is full! Cannot buy more spells." <<
std::endl;
            return false;
        }

        auto newSpell = createSpellCopy(baseSpell);
        if (newSpell) {
            player.addScore(-cost);
            player.getSpellHand().addSpell(std::move(newSpell));
            std::cout << "Bought " << baseSpell->getName() << " for " <<
cost << " points!" << std::endl;
            return true;
        }

        return false;
    }

    const std::vector<std::unique_ptr<Spell>>&
ShopSystem::getAvailableSpells() const {
        return availableSpells;
    }

    bool ShopSystem::interactWithPlayer(Player& player, InputHandler&
inputHandler) const {
        // Проверяем, есть ли свободные слоты
        if (player.getSpellHand().isFull()) {

```

```

        std::cout << "Your hand is full! Cannot buy more spells." <<
std::endl;
        return false;
    }

    // Проверяем, есть ли доступные заклинания
    if (availableSpells.empty()) {
        std::cout << "No spells available in the shop!" << std::endl;
        return false;
    }

    // Показываем магазин
    std::cout << "=== SPELL SHOP ===" << std::endl;
    std::cout << "Your score: " << player.getScore() << " points" <<
std::endl;
    std::cout << "Free spell slots: "
        << (player.getSpellHand().getMaxSize() -
player.getSpellHand().getSpellCount())
        << "/" << player.getSpellHand().getMaxSize() << std::endl;
    std::cout << "Available spells:" << std::endl;

    for (int i = 0; i < availableSpells.size(); i++) {
        int cost = calculateCost(availableSpells[i].get());
        std::cout << i + 1 << ". " <<
availableSpells[i]->getDescription()
        << " - Cost: " << cost << " points" << std::endl;
    }

    // Получаем выбор игрока
    int choice = inputHandler.getShopChoice(availableSpells.size());

    if (choice == 0) {
        std::cout << "Shop interaction cancelled." << std::endl;
        return true;
    }

    return processPlayerChoice(player, choice - 1);
}

bool ShopSystem::processPlayerChoice(Player& player, int spellIndex)
const {
    return buySpell(player, spellIndex);
}

```

RewardShop.h

```

#ifndef REWARDSHOP_H
#define REWARDSHOP_H

#include "Player.h"
#include "SpellSystem.h"
#include "GameInterface.h" // ДОБАВИТЬ ЭТО!
#include <vector>
#include <memory>

class RewardSystem {
private:
    int enemiesKilled;

```

```

public:
    RewardSystem();
    void onEnemyKilled(Player& player);
    void giveVictoryReward(Player& player) const;
    void resetKillCounter();
    int getEnemiesKilled() const;
};

class ShopSystem {
private:
    std::vector<std::unique_ptr<Spell>> availableSpells;
    std::unique_ptr<Spell> createSpellCopy(Spell* baseSpell) const;
    bool processPlayerChoice(Player& player, int choice) const;

public:
    ShopSystem();
    bool buySpell(Player& player, int spellIndex) const;
    const std::vector<std::unique_ptr<Spell>>& getAvailableSpells()
const;
    int calculateCost(Spell* spell) const;
    bool interactWithPlayer(Player& player, InputHandler& inputHandler)
const;
};

#endif

```

SpellHand.cpp

```

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

```

```

SpellHand::SpellHand(int size) : maxSize(size) {
}

```

```

bool SpellHand::addSpell(std::unique_ptr<Spell> spell) {
    if (spells.size() >= maxSize) {
        std::cout << "Hand is full! Cannot add new spell." << std::endl;
        return false;
    }
    spells.push_back(std::move(spell));
    std::cout << "Added spell: " << spells.back()->getName() <<
std::endl;
    return true;
}

```

```

bool SpellHand::removeSpell(int index) {
    if (index < 0 || index >= spells.size()) {
        return false;
    }
    spells.erase(spells.begin() + index);
    return true;
}

```

```

void SpellHand::clearHand() {
    spells.clear();
}

```

```

bool SpellHand::castSpell(int index, const SpellTarget& target,

```

```

        GameField& field, std::vector<Enemy>& enemies, Player& player) const
    {
        if (index < 0 || index >= spells.size()) {
            return false;
        }
        return spells[index]->cast(target, field, enemies, player);
    }

    std::string SpellHand::getSpellName(int index) const {
        if (index < 0 || index >= spells.size()) {
            return "";
        }
        return spells[index]->getName();
    }

    std::string SpellHand::getSpellDescription(int index) const {
        if (index < 0 || index >= spells.size()) {
            return "";
        }
        return spells[index]->getDescription();
    }

    int SpellHand::getSpellManaCost(int index) const {
        if (index < 0 || index >= spells.size()) {
            return 0;
        }
        return spells[index]->getManaCost();
    }

    int SpellHand::getSpellCount() const {
        return spells.size();
    }

    int SpellHand::getMaxSize() const {
        return maxSize;
    }

    bool SpellHand::isFull() const {
        return spells.size() >= maxSize;
    }

    void SpellHand::displaySpells() const {
        std::cout << "=== Spells in hand ===" << std::endl;
        for (int i = 0; i < spells.size(); i++) {
            std::cout << i + 1 << ". " << spells[i]->getDescription() <<
std::endl;
        }
        std::cout << "Free slots: " << (maxSize - spells.size()) << "/" <<
maxSize << std::endl;
    }

SpellHand.h
#ifndef SPELLHAND_H
#define SPELLHAND_H

#include "SpellSystem.h"
#include <vector>
#include <memory>

```

```

class SpellHand {
private:
    std::vector<std::unique_ptr<Spell>> spells;
    int maxSize;

public:
    SpellHand(int size);

    bool addSpell(std::unique_ptr<Spell> spell);
    bool removeSpell(int index);
    void clearHand();

    bool castSpell(int index, const SpellTarget& target,
        GameField& field, std::vector<Enemy>& enemies, Player& player)
const;

    std::string getSpellName(int index) const;
    std::string getSpellDescription(int index) const;
    int getSpellManaCost(int index) const;
    int getSpellCount() const;
    int getMaxSize() const;
    bool isFull() const;
    void displaySpells() const;
};

#endif

```

SpellSystem.cpp

```

#include "SpellSystem.h"
#include "GameField.h"
#include "Enemy.h"
#include "Player.h"
#include <iostream>
#include <cmath>

```

```

SpellTarget::SpellTarget(int tX, int tY, int cX, int cY)
    : targetX(tX), targetY(tY), casterX(cX), casterY(cY) {
}

```

```

Spell::Spell(const std::string& spellName, int cost, int spellRange)
    : name(spellName), manaCost(cost), range(spellRange) {
}

```

```

std::string Spell::getName() const {
    return name;
}

```

```

int Spell::getManaCost() const {
    return manaCost;
}

```

```

int Spell::getRange() const {
    return range;
}

```

```

DirectDamageSpell::DirectDamageSpell(const std::string& name, int cost,
int spellRange, int spellDamage)
    : Spell(name, cost, spellRange), damage(spellDamage) {
}

bool DirectDamageSpell::cast(const SpellTarget& target, GameField&
field,
    std::vector<Enemy>& enemies, Player& player) {
    int distance = std::abs(target.targetX - target.casterX) +
std::abs(target.targetY - target.casterY);
    if (distance > range) {
        std::cout << "Target is too far! Distance: " << distance << ",
max: " << range << std::endl;
        return false;
    }

    if (!field.isValidPosition(target.targetX, target.targetY)) {
        std::cout << "Invalid target position!" << std::endl;
        return false;
    }

    bool enemyFound = false;
    for (auto& enemy : enemies) {
        if (enemy.isAlive() && enemy.getX() == target.targetX &&
enemy.getY() == target.targetY) {
            enemy.takeDamage(damage);
            std::cout << name << " deals " << damage << " damage to
enemy!" << std::endl;
            if (!enemy.isAlive()) {
                std::cout << "Enemy defeated!" << std::endl;
                player.addScore(10);
            }
            enemyFound = true;
            break;
        }
    }

    if (!enemyFound) {
        std::cout << "No enemy on target cell! Spell failed." <<
std::endl;
        return false;
    }
    return true;
}

std::string DirectDamageSpell::getDescription() const {
    return name + " - damage: " + std::to_string(damage) + ", range: "
+ std::to_string(range);
}

std::unique_ptr<Spell> DirectDamageSpell::clone() const {
    return std::make_unique<DirectDamageSpell>(name, manaCost, range,
damage);
}

AreaDamageSpell::AreaDamageSpell(const std::string& name, int cost, int
spellRange, int spellDamage)
    : Spell(name, cost, spellRange), damage(spellDamage) {
}

```

```

}

bool AreaDamageSpell::cast(const SpellTarget& target, GameField& field,
    std::vector<Enemy>& enemies, Player& player) {
    int distance = std::abs(target.targetX - target.casterX) +
std::abs(target.targetY - target.casterY);
    if (distance > range) {
        std::cout << "Target is too far! Distance: " << distance << ",
max: " << range << std::endl;
        return false;
    }

    if (!field.isValidPosition(target.targetX, target.targetY)) {
        std::cout << "Invalid target position!" << std::endl;
        return false;
    }

    std::cout << name << " hits 2x2 area!" << std::endl;
    int enemiesHit = 0;

    for (int y = target.targetY; y <= target.targetY + 1; y++) {
        for (int x = target.targetX; x <= target.targetX + 1; x++) {
            if (field.isValidPosition(x, y)) {
                for (auto& enemy : enemies) {
                    if (enemy.isAlive() && enemy.getX() == x &&
enemy.getY() == y) {
                        enemy.takeDamage(damage);
                        std::cout << "Enemy at (" << x << ", " << y <<
") takes " << damage << " damage!" << std::endl;
                        if (!enemy.isAlive()) {
                            std::cout << "Enemy defeated!" << std::endl;
                            player.addScore(10);
                        }
                        enemiesHit++;
                    }
                }
            }
        }
    }

    if (enemiesHit == 0) {
        std::cout << "No enemies found in the affected area." <<
std::endl;
    }
    return true;
}

std::string AreaDamageSpell::getDescription() const {
    // УБРАТЬ areaSize ИЗ ОПИСАНИЯ
    return name + " - area damage: " + std::to_string(damage) + " (2x2),
range: " + std::to_string(range);
}

std::unique_ptr<Spell> AreaDamageSpell::clone() const {
    return std::make_unique<AreaDamageSpell>(name, manaCost, range,
damage);
}

```

SpellSystem.h

```
#ifndef SPELLSYSTEM_H
#define SPELLSYSTEM_H
```

```
#include <string>
#include <memory>
#include <vector>
```

```
class GameField;
class Enemy;
class Player;
```

```
class SpellTarget {
public:
    int targetX;
    int targetY;
    int casterX;
    int casterY;

    SpellTarget(int tX, int tY, int cX, int cY);
};
```

```
class Spell {
protected:
    std::string name;
    int manaCost;
    int range;

public:
    Spell(const std::string& spellName, int cost, int spellRange);
    virtual ~Spell() = default;

    virtual bool cast(const SpellTarget& target, GameField& field,
        std::vector<Enemy>& enemies, Player& player) = 0;

    std::string getName() const;
    int getManaCost() const;
    int getRange() const;
    virtual std::string getDescription() const = 0;
    virtual std::unique_ptr<Spell> clone() const = 0;
};
```

```
class DirectDamageSpell : public Spell {
private:
    int damage;

public:
    DirectDamageSpell(const std::string& name, int cost, int spellRange,
        int spellDamage);

    bool cast(const SpellTarget& target, GameField& field,
        std::vector<Enemy>& enemies, Player& player) override;
    std::string getDescription() const override;
    std::unique_ptr<Spell> clone() const override;
};
```

```
class AreaDamageSpell : public Spell {
```

```

private:
    int damage;

public:
    AreaDamageSpell(const std::string& name, int cost, int spellRange,
int spellDamage);

    bool cast(const SpellTarget& target, GameField& field,
        std::vector<Enemy>& enemies, Player& player) override;
    std::string getDescription() const override;
    std::unique_ptr<Spell> clone() const override;
};

#endif

```

TurnAction.cpp

```

#include "TurnAction.h"
#include "GameConstants.h"
#include <iostream>
#include <cstdlib>

```

```

TurnSystem::TurnSystem() : playerTurn(true) {
}

```

```

bool TurnSystem::isPlayerTurn() const {
    return playerTurn;
}

```

```

void TurnSystem::endPlayerTurn() {
    playerTurn = false;
}

```

```

void TurnSystem::endEnemyTurn() {
    playerTurn = true;
}

```

```

void TurnSystem::reset() {
    playerTurn = true;
}

```

```

bool ActionSystem::processPlayerMove(GameState& gameState, int newX, int
newY) {
    auto& field = gameState.getField();
    auto& player = gameState.getPlayer();
    auto& enemies = gameState.getEnemies();

    if (!field.isValidPosition(newX, newY)) {
        std::cout << "Cannot move there - out of bounds!" << std::endl;
        return false;
    }

    // Проверка атаки врага
    for (auto& enemy : enemies) {
        if (enemy.isAlive() && enemy.getX() == newX && enemy.getY() ==
newY) {
            enemy.takeDamage(player.getDamage());
        }
    }
}

```

```

        std::cout << "You attacked enemy! Enemy health: " <<
enemy.getHealth() << std::endl;
        if (!enemy.isAlive()) {
            player.addScore(10);
            std::cout << "Enemy defeated! +10 points" << std::endl;
        }
        return true;
    }

    // Обычное движение
    if (field.movePlayer(newX, newY)) {
        player.addScore(1);
        player.addMana(5);
        return true;
    }

    return false;
}

bool ActionSystem::processSpellCast(GameState& gameState, InputHandler&
inputHandler) {
    auto& player = gameState.getPlayer();

    if (player.getSpellHand().getSpellCount() == 0) {
        std::cout << "You have no spells in hand!" << std::endl;
        return false;
    }

    int spellChoice =
inputHandler.getSpellChoice(player.getSpellHand().getSpellCount());

    if (spellChoice < 1 || spellChoice >
player.getSpellHand().getSpellCount()) {
        std::cout << "Invalid spell choice!" << std::endl;
        return false;
    }

    std::pair<int, int> target = inputHandler.getSpellTarget();

    return player.castSpell(spellChoice - 1, target.first,
target.second,
        gameState.getField(), gameState.getEnemies());
}

bool ActionSystem::processShopInteraction(GameState& gameState,
ShopSystem& shopSystem,
    InputHandler& inputHandler) {

    return shopSystem.interactWithPlayer(gameState.getPlayer(),
inputHandler);
}

void ActionSystem::processEnemyMoves(GameState& gameState) {
    auto& field = gameState.getField();
    auto& player = gameState.getPlayer();
    auto& enemies = gameState.getEnemies();

```

```

auto playerPos = field.getPlayerPosition();
int playerX = playerPos.first;
int playerY = playerPos.second;

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

    int randomNumber = std::rand() % 4;
    int oldX = enemy.getX();
    int oldY = enemy.getY();
    int newX = oldX;
    int newY = oldY;

    switch (randomNumber) {
    case 0: newX++; break; // EAST
    case 1: newX--; break; // WEST
    case 2: newY++; break; // SOUTH
    case 3: newY--; break; // NORTH
    }

    if (field.isValidPosition(newX, newY)) {
        bool canMove = true;

        if (newX == playerX && newY == playerY) {
            player.takeDamage(enemy.getDamage());
            std::cout << "Enemy attacked you! Lost " <<
enemy.getDamage() << " health!" << std::endl;
            canMove = false;
        }

        for (auto& otherEnemy : enemies) {
            if (&enemy != &otherEnemy && otherEnemy.isAlive() &&
                otherEnemy.getX() == newX && otherEnemy.getY() ==
newY) {
                canMove = false;
                break;
            }
        }

        if (canMove && field.isCellEmpty(newX, newY)) {
            enemy.move(newX, newY);
        }
    }
}

```

TurnAction.h

```

#ifndef TURNACTION_H
#define TURNACTION_H

#include "GameState.h"
#include "GameInterface.h"
#include "RewardShop.h"

class TurnSystem {
private:
    bool playerTurn;

```

```

public:
    TurnSystem();
    bool isPlayerTurn() const;
    void endPlayerTurn();
    void endEnemyTurn();
    void reset();
};

class ActionSystem {
public:
    bool processPlayerMove(GameState& gameState, int newX, int newY);
    bool processSpellCast(GameState& gameState, InputHandler&
inputHandler);
    bool processShopInteraction(GameState& gameState, ShopSystem&
shopSystem,
    InputHandler& inputHandler);
    void processEnemyMoves(GameState& gameState);
};

#endif

```

```

GameConstants.h
#ifndef GAMECONSTANTS_H
#define GAMECONSTANTS_H

enum class SpellType {
    DIRECT_DAMAGE = 0,
    AREA_DAMAGE = 1
};

enum class Direction {
    EAST = 0,
    WEST = 1,
    SOUTH = 2,
    NORTH = 3
};

namespace CellContent {
    constexpr char EMPTY = '.';
    constexpr char PLAYER = 'P';
    constexpr char ENEMY = 'E';
}

#endif

```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

```
Game Started! Use WASD to move, Q to quit
Health: 100 | Score: 0 | Position: (0,0)
P . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . E . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
E . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
Move (WASD):
```

1. Запуск игры


```

Health: 100 | Score: 1 | Position: (1,0)
. P . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . E . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. E . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
Move (WASD): s
Health: 100 | Score: 2 | Position: (1,1)
. . . . . . . . . . . . . . .
. P . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . E . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. E . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
Move (WASD): |

```

3. Перемещение вниз

A 15x15 grid of dots on a black background. The letters 'P' and 'E' are formed by removing dots in a specific pattern. 'P' is located at row 5, column 3. 'E' is located at row 5, column 5.

You attacked enemy! Enemy health: 20

A 15x15 grid of dots. The letters 'P E' are placed on the 10th row, 4th and 5th columns. The letter 'E' is placed on the 12th row, 4th column.

1. *Journal of Management Studies*, 1996, 33, 1, 1-14.