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:

Студентка гр. 4383 _____ Жданова К.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы:

Продолжение разработки игрового приложения. Создание системы взаимодействующих классов для возможности перехода между уровнями, а также возможности сохранять и загружать игру. Построение UML-диаграммы, отображающей структуру классов, их взаимосвязи и архитектурные решения.

Задание.

Лабораторная работа №3

На 6/3/1 баллов:

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

Начало игры

Запуск уровня

Ход игрока. Ход, атака или применение заклинания.

Ход союзников - если имеются

Ход врагов

Ход вражеской базы и башни - если имеются

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

Все взаимодействие должно происходить через классы игры.

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

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

На 8/4/1.5 баллов:

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

На 10/5/2 баллов:

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

Примечания:

Класс игры может знать о игровых сущностях, но не наоборот

При работе с файлом используйте идиому RAII.

Исключения должны обязательно обрабатываться, и программа не должна завершаться

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

Основные теоретические положения.

Исключение в C++ — это механизм обработки ошибок, который позволяет программе отделить обработку нетипичных ситуаций от основного кода.

Обработка исключений в C++ включает в себя использование ключевых слов try, throw и catch для управления ошибками времени выполнения. Блок try заключает код, который может вызвать исключение, а throw используется для его генерации.

Файл с расширением .dat — это общий файл данных, который может содержать любую информацию: текстовую, двоичную или мультимедийную.

reinterpret_cast<char*> — это оператор приведения типов в C++, который низкоуровнево преобразует указатель одного типа в указатель другого типа, например, из int* в char*.

Выполнение работы.

В рамках лабораторной работы была построена UML-диаграмма, отображающая структуру классов, их взаимосвязи и ключевые архитектурные решения. UML-диаграмма представлена на Рисунке 1.

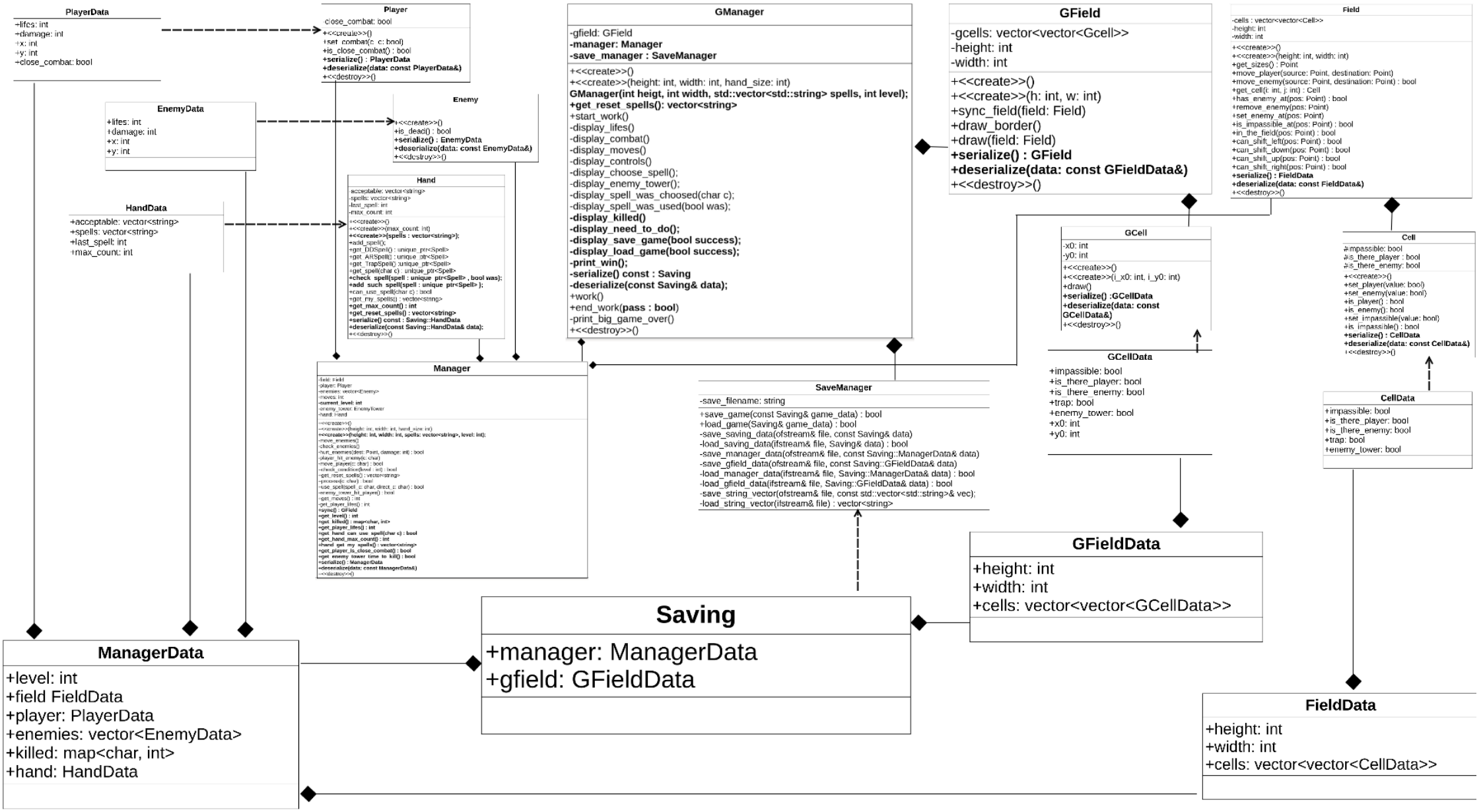

Рисунок 1. UML-диаграмма.

Основной цикл игры состоит из:

Начало игры, запуск уровня, ход игрока, ход врагов, ход вражеской башни.

Основной цикл игры реализовывают классы Game, GManager и Manager.

GManager — это один уровень. То есть цикл игры от запуска до выполнения одного из условий : игрок умер или условие прохождения уровня было выполнено.

Game в методе void process() создает очередной уровень с необходимыми параметрами (теми, которые изменяются от уровня к уровню, а именно - само значение текущего уровня, размеры поля и список имеющихся заклинаний)

GManager отвечает за обработку вводимых символов и за отрисовку уровня. Возможные опции и символы, которые за них отвечают:

h,j,k,l,w,a,s,d,c — работает Manager

x — использовать заклинание

m — сохраниться

n — загрузить последнее сохранение

q — выйти из игры

Работа класса Manager заключается в последовательности: ход игрока (в зависимости от введенного символа удар по врагу, движение или смена боя), ход врагов и ход вражеской башни. Также каждое определенное количество ходов создаются новые враги и в руку добавляются заклинания.

В игре 3 уровня. Условие прохождения уровня проверяется каждый ход после последовательности-части игрового цикла.

Сохранение и загрузка игры.

Для реализации возможности сохранять игру был написан класс Saving, с числом вложенных классов — классов с данными, которые нужно сохранить.

Запись идет в бинарный файл .dat. Для этого нужно перевести все данные в вид, который можно будет записать в файл.

Saving отвечает только за хранение данных. Его поля — классы ManagerData, GfieldData. Для каждого класса, который нужно было запомнить — это Field,

Player и так далее, были написаны соответствующие классы *Data, а в сами классы добавлены методы *Data serialize() и void deserialize(*Data) , для создания объекта с данными и наоборот, обновления данных в соответствии с полученным объектом. .
Но Saving только хранит данные и является, по сути, кастомным типом данных.
Реализацией сохранения и загрузки игры занимается класс SaveManager.
Его главные методы — save_game(Saving& data) и load_game(Saving& data), где в первом случае мы используем полученные с помощью метода GManager.serialize() данные, чтобы их записать в файл, а во втором случае — загружаем в data данные, чтобы потом, используя ях, сделать GManager.deserialize(data).
SaveManager записывает все поля всех полей построчно в файл, предварительно представляя их в char* для возможной записи.
SaveManager является полем GManager.
Делать можно только одно сохранение. При загрузке игры загружается имеющееся сохранение.

Были добавлены исправления к Лабораторной работе №2. Необходимо было исправить в методе Manager.use_spell(char spell_c, char direct_c) проверку на «тип» заклинания и унифицировать использование заклинание. Проверки на тип были убраны

На данный момент игровое приложение имеет следующий вид, представленный на Рисунке 2.

Рисунок 2. Вид игрового приложения

Выводы:

Была продолжена разработка консольного игрового приложения на основе системы взаимодействующих классов для реализации возможности перехода между уровнями, а также сохранения и загрузки игры. Была построена UML-диаграмма, отображающая структуру новодобавленных классов, их взаимосвязи и архитектурные решения.