СОДЕРЖАНИЕ

ВВЕДЕНИЕ ..... 5
1 ПОСТАНОВКА ЗАДАЧИ ..... 7
2 ОБЗОР ЛИТЕРАТУРЫ ..... 8
2.1 Обзор методов и алгоритмов решения поставленной задачи ..... 8
2.2 Обзор библиотеки SFML ..... 9
2.3 Обзор фреймворка Qt ..... 9
2.4 Обзор аналогов игры “Пакман” ..... 10
3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ ..... 14
3.1 Структура входных и выходных данных ..... 14
3.2 Разработка диаграммы классов ..... 14
3.3 Описание классов ..... 14
3.3.1 Класс игры ..... 14
3.3.2 Класс сущности ..... 15
3.3.3 Класс игрока ..... 16
3.3.4 Класс врага ..... 17
3.3.5 Класс-обработчик врагов ..... 17
3.3.6 Класс сущностей, которые могут перемещаться ..... 17
3.3.7 Класс сцены ..... 18
3.3.8 Класс уровня ..... 19
3.3.9 Класс худа ..... 19
3.3.10 Класс меню ..... 20
3.3.11 Класс перечисления MoveDirection ..... 20
4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ ..... 22
4.1 Разработка схем алгоритмов ..... 22
4.2 Разработка алгоритмов ..... 22
4.2.1 Разработка алгоритма метода Update() у класса Player ..... 22
4.2.2 Разработка алгоритма метода MainLoop() для класса Game ..... 23
5 РЕЗУЛЬТАТЫ РАБОТЫ ..... 25
ЗАКЛЮЧЕНИЕ ..... 30
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ ..... 31
ПРИЛОЖЕНИЕ А ..... 32
ПРИЛОЖЕНИЕ Б ..... 33
ПРИЛОЖЕНИЕ В ..... 34
ПРИЛОЖЕНИЕ Г ..... 35
ПРИЛОЖЕНИЕ Д ..... 47

# ВВЕДЕНИЕ

Язык программирования C++ был разработан в 1983 году Бьёрном Страуструпом в качестве расширения языка Си. C++ является объектно-ориентированным языком программирования, который позволяет разработчикам создавать мощные и эффективные программы для широкого спектра приложений.

Основными принципами C++ являются абстракция, инкапсуляция, наследование и полиморфизм. Абстракция позволяет скрыть детали реализации от пользователя, что делает код более понятным и удобным для работы. Инкапсуляция предоставляет возможность объединить данные и методы в одном классе и защитить их от несанкционированного доступа. Наследование позволяет создавать новые классы на основе уже существующих, что упрощает процесс разработки. Полиморфизм позволяет использовать объекты разных классов как одинаковые, что упрощает процесс программирования.

C++ имеет множество возможностей для работы с памятью, таких как указатели, ссылки, динамическое выделение памяти и управление памятью. Это позволяет разработчикам создавать эффективные программы, которые используют ресурсы компьютера максимально эффективно.

C++ широко используется в различных областях, таких как разработка операционных систем, игр, приложений для мобильных устройств и веб-приложений. Он также используется в научных и инженерных расчетах, а также в финансовых и торговых системах.

Одним из основных преимуществ C++ является его скорость выполнения программ. Он позволяет создавать быстрые и эффективные программы, что делает его предпочтительным языком программирования для разработки систем реального времени и других задач, где скорость выполнения критически важна.

C++ также известен своей поддержкой многопоточности, что позволяет создавать многопоточные приложения и использовать параллельные вычисления для ускорения работы программы. Это особенно важно для программ, которые работают с большими объемами данных или выполняют сложные вычисления.

Кроме того, C++ имеет богатую стандартную библиотеку, которая содержит множество функций и классов для работы с различными типами данных, файлами, сетью и другими аспектами программирования. Это делает процесс разработки программ более простым и удобным.

Существует множество инструментов и сред разработки, которые поддерживают C++, такие как Visual Studio, Eclipse, Code::Blocks и другие. Они облегчают процесс написания кода и отладки программ.

Одним из недостатков C++ является его сложность и трудность в освоении. В отличие от более простых языков программирования, таких как

Python или JavaScript, C++ требует более глубокого понимания основных концепций программирования и языка Си.

Тем не менее, C++ остается одним из самых популярных языков программирования в мире благодаря своей мощности, скорости и эффективности. Многие крупные компании, такие как Google, Microsoft, Amazon и другие, используют C++ для создания своих продуктов и сервисов.

# 1 ПОСТАНОВКА ЗАДАЧИ

В курсовой работе реализована классическая игра “Пакман”. Основными элементами данной игры являются карта, по которой перемещается игрок и враги; точки, собрав которые полностью уровень считается пройденным; поле, в котором отображается текущий счёт, набранный игроком, а также количество жизней, которые есть у игрока.

Для успешного прохождения уровня у игрока есть 3 попытки, которые отображаются в виде жизней в левом верхнем углу окна. При столкновении с врагом у игрока отнимается одна жизнь, он и враги перемещаются на стартовые позиции, и игрок вновь пытается пройти уровень. В правом верхнем углу окна отображается текущий счет, который увеличивается в зависимости от количества съеденных на карте точек. В случае победы, на экране отображается сообщение о том, что игрок победил, в противном случае – сообщение о том, что он проиграл. В обоих случаях игрок может вернуться в главное меню и либо начать игру заново, либо выйти из игры.

При запуске игры пользователь видит меню, в котором отображаются возможные опции: играть и выйти. В зависимости от выбора игрока будет выполнено соответствующее действие.

Графическая составляющая проекта реализована при помощи использования графической библиотеки SFML.

# 2 ОБЗОР ЛИТЕРАТУРЫ

## 2.1 Обзор методов и алгоритмов решения поставленной задачи

Для реализации проекта создан базовый класс `Entity`, который содержит в себе поля, применимые для всех сущностей, присутствующих в игре. Например, это сущности игрока (класс `Player`) и врагов (класс `Enemy`). Помимо этого, данный класс содержит в себе два абстрактных метода, которые можно перегрузить в зависимости от требуемой задачи для конкретного класса. Помимо этого, создан класс `Imovable`, от которого наследуются классы тех сущностей, которые могут перемещаться. Для удобной работы с врагами создан класс `EnemyHandler`, который отвечает за обработку игровой логики всех врагов, размещенных на карте.

Помимо данных классов, существует класс `Map`, который хранит в себе информацию об уровне, а именно: расположение стен, точек, позиций игрока и врагов. Другими не менее важными классами является классы `Menu` и `HoodElements`. Класс `Menu` отвечает за отображение и возможность взаимодействия с главным меню, а класс `HoodElemets` отвечает за хранение и отображение количества жизней игрока и текущего счёта.

Все вышеперечисленные классы входят в состав класса `Scene`. Он реализован для того, чтобы отображать все элементы проекта. При запуске игры на сцену загружается карта, враги, игрок, а также элементы интерфейса. В дальнейшем происходит обработка логики для каждой отдельной сущности, отрисовка их на сцене и вывод данной сцены в окно приложения.

Сцена хранится в классе `Game`, который является главным и служит для запуска игры. В нем содержатся методы отображения и взаимодействия с меню, начала уровня, вывода сообщения об успешном прохождении уровня или проигрыше.

Немаловажным элементом в разработке игр является главный цикл, который отвечает за обновление игры и игровой логики каждой сущности один раз за кадр. Данный метод также реализован в классе `Game`, логика которого заключается в получении времени, прошедшего с момента отрисовки последнего кадра, обновление логики, отрисовка всех сущностей на сцене и отображение сцены в окне приложения.

Для реализации графической составляющей используется библиотека SMFL. В ней есть типы `Text, Font, Sprite, CircleShape, Rectangle, Texture`. В папке с проектом хранятся изображения сердца, Пакмана и врагов в формате .png, а также шрифт текста, который отображается в окне игры. Все вышеперечисленные изображения и шрифт хранятся в соответствующих переменных и загружаются в проект с использованием соответствующих методов.

## 2.2 Обзор библиотеки SFML

За реализацию графической составляющей игры отвечает библиотека SFML. SFML (Simple and Fast Multimedia Library) – свободная кроссплатформенная мультимедийная библиотека. Представляет собой объектно-ориентированный аналог SDL. SFML содержит ряд модулей для простого программирования игр и мультимедиа приложений. Основные модули, которые будут использоваться для реализации: System, Window, Graphics. Модуль System управляет временем и потоками а так же является обязательным, так как все модули зависят от него. Модуль Window используется для управления окнами и взаимодействия с пользователем. Модуль Graphics служит для отображения графических примитивов и изображений. [1]

## 2.3 Обзор фреймворка Qt

Еще одним вариантом для разработки оконных приложений является фреймворк Qt. Qt – фреймворк для разработки кроссплатформенного программного обеспечения на языке C++. Для многих языков программирования существуют библиотеки, позволяющие использовать преимущества Qt.[2]

Qt позволяет запускать написанное с его помощью программное обеспечение в большинстве современных операционных систем путём простой компиляции программы для каждой системы без изменения исходного изменения исходного кода. Включает в себя все основные классы, которые могут потребоваться при разработке прикладного программного обеспечения, начиная от элементов графического интерфейса и заканчивая классами для работы с сетью, базами данных и XML. Является полностью объектно-ориентированным, расширяемым и поддерживающим технику компонентного программирования.

Отличительная особенность — использование метаобъектного компилятора — предварительной системы обработки исходного кода. Расширение возможностей обеспечивается системой плагинов, которые возможно размещать непосредственно в панели визуального редактора. Также существует возможность расширения программной функциональности виджетов, связанной с размещением их на экране, отображением, перерисовкой при изменении размеров окна.

Комплектуется визуальной средой разработки графического интерфейса Qt Designer, позволяющей создавать диалоги и формы в режиме WYSIWYG. В поставке Qt есть Qt Linguist — графическая утилита, позволяющая упростить локализацию и перевод программы на многие языки; и Qt Assistant — справочная система Qt, упрощающая работу с документацией по библиотеке, а также позволяющая создавать кроссплатформенную справку

для разрабатываемого на основе Qt программного обеспечения. Начиная с версии 4.5.0 в комплект включена среда разработки Qt Creator, которая включает редактор кода, справку, графические средства Qt Designer и возможность отладки приложений. Qt Creator может использовать GCC или Microsoft VC++ в качестве компилятора и GDB в качестве отладчика. Для Windows-версий библиотека комплектуется компилятором, заголовочными и объектными файлами MinGW.

Некоторое время библиотека также распространялась ещё в версии Qt/Embedded, предназначенной для применения на встраиваемых и мобильных устройствах, но начиная с середины 2000-х годов она выделена в самостоятельный продукт Qtopia. [3]

Для реализации игры была выбрана библиотека SFML, так как фреймворк Qt предназначен для любого настольного приложения, использующего довольно стандартный графический интерфейс, а SFML позволяет реализовать нестандартный графический интерфейс. Помимо этого, приложение, написанное с использованием SFML, будет быстрее в исполнении, нежели приложение, написанное на Qt.

### 2.4 Обзор аналогов игры “Пакман”

1 “Dig Dug” (1982) – игра, разработанная компанией “MAMCO”. Главной задачей в игре является уничтожение монстров, живущих под землей, раздувая их, или пока они не лопнут, или сбрасывая на них камни. В игре есть два вида врагов: Pookas, круглые красные монстры, которые носят желтые очки, и Fygas, зеленые драконы, которые могут выдыхать пламя.[4]


Рис. 2.4.1 – “Dig Dug”

2 “Gauntlet” (1985) – игра, разработанная “Atari games” и “Tengen”. Игроки (до 4 в аркадной версии) могут выбрать одного из 4 доступных персонажей, каждый из которых имеет свои достоинства и недостатки. К персонажам относятся воин Тор (самый сильный в рукопашном бою), волшебник Мерлин (самая сильная магия), валькирия Тайра (лучшая броня) и эльф Квестор (он самый быстрый).

После выбора персонажа начинается игра на игровом поле, которое представлено несколькими лабиринтами с видом сверху вниз от третьего лица. Целью игрока является поиск обозначенного на каждом уровне «выхода», до которого нужно дотронуться. На каждом уровне можно найти много различных предметов, которые восстанавливают здоровье игрока, открывают двери, дают очки или уничтожают всех врагов на экране[1].

Врагами являются различные фэнтезийные монстры, в том числе призраки, демоны, волшебники и воры. Они попадают на уровень через генераторы, которые можно уничтожать. В игре нет боссов, но самым страшным врагом является Смерть, которая не только высасывает здоровье персонажа, но и крайне трудно уничтожается.


Рис 2.4.2 – “Gauntlet”

3 “Dodge ‘Em” (1980) – еще один продукт “Atari games”. Игрок управляет одной машиной и должен двигаться против часовой стрелки, избегая машин, управляемых компьютером, единственная цель которых - вызвать лобовое столкновение. Каждая проезжая часть лабиринта имеет четыре пробела вверху, внизу, слева и справа от экрана. Игрок может использовать промежутки для смены полосы движения, чтобы собирать другие точки или избегать машин, управляемых компьютером.

Автомобиль игрока может двигаться на двух скоростях, нормальная скорость равна скорости компьютера. -управляемые автомобили, или повышенная (удвоенная) скорость, активируемая нажатием кнопки

контроллера. У машин с компьютерным управлением только одна скорость. Игроки меняют полосу движения, толкая контроллер в нужном направлении, когда их машина находится возле одного из пробелов на проезжей части.

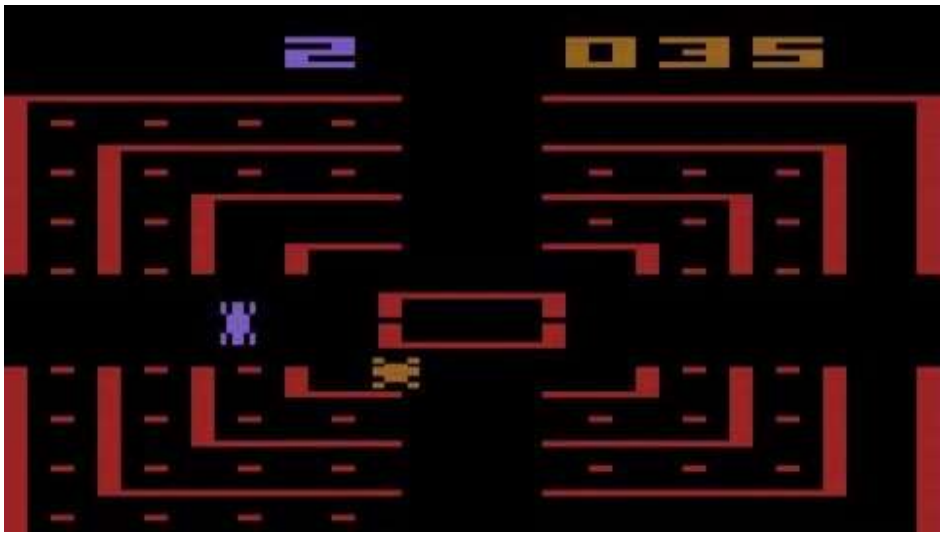

Рис 2.4.3 – “Dodge ‘Em”

4 “RUNRUNRUN” (2017) – это командная игра в жанрах экшены, казуальные и инди, разработанная Zoglu. Сутью игры является взломать банки и забрать как можно больше золотых слитков прежде, чем истечет таймер.

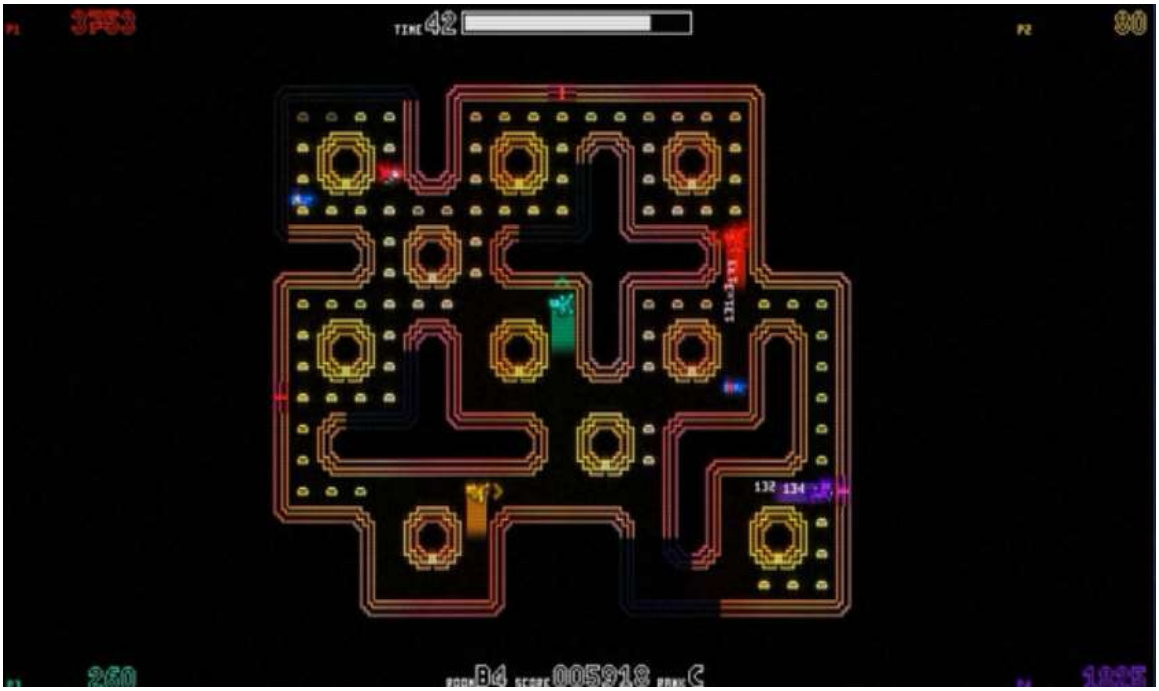

Рис 2.4.4 – “RUNRUNRUN”

5 “Марру” (1984) – компьютерная аркадная игра компании Namco, выпущенная в 1983 году. Представляет собой платформер с главным героем — мышью Маппи — и противниками-кошками. Игра выпускалась под 8-битную консоль Namco Super Pac-Man, поддерживавшую горизонтальную прокрутку экрана.

На каждом из уровней мышь-полицейский Маппи бегает по дому, в котором прячутся кошки — Мяуки и их главарь Горо, и пытается вернуть украденные вещи. В доме шесть этажей (в других версиях — четыре или пять), на разных из них находятся украденные вещи. Чтобы подобрать вещь, к ней нужно просто подбежать. Уровень будет пройден, если Маппи соберёт все вещи. Если Маппи столкнётся с любым из котов, он теряет жизнь.[5]


Рис. 2.4.5 – “Марру”

# 3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе описываются входные и выходные данные программы, диаграмма классов, а также приводится описание используемых классов и их методов.

## 3.1 Структура входных и выходных данных

Изначально карта уровня хранится в виде текстового документа, в котором содержатся символы ‘1’ и ‘.’. ‘1’ обозначают ячейки, в которых находится стенка, а ‘.’ – ячейки, в которых расположены точки, которые есть Пакман. В проекте реализован метод, который считывает символы из файла и заносит их в массив строк, который в дальнейшем является основным носителем карты уровня.

```
111111111111111
1...1..1..1...1
1.1....1....1.1
1...11.1.11...1
111.........111
1.1.1.111.1.1.1
1.............1
1.1111.1.1111.1
1.............1
111.111.111.111
1.1.........1.1
1...1.1.1.1...1
1.111.1.1.111.1
1.............1
111111111111111
```

Рис 3.1.1 – Карта уровня в текстовом документе

## 3.2 Разработка диаграммы классов

Диаграмма классов для курсового проекта приведена в Приложении А

## 3.3 Описание классов

### 3.3.1 Класс игры

Для реализации используется класс `Game`, который отвечает за работу игры. Он хранит в себе загруженные уровни, текстуры, а также выполняет главный игровой цикл, который содержит в себе методы работы с графикой и игровой логикой.

Описание полей класса Game:

`Scene scene` - сцена, на которой отображаются все сущности (враги, игрок, уровень, меню, худ)

`Menu currMenu` - меню

`sf::RenderWindow window` – окно игры, в котором будет отображаться сцена

`vector<Map> loadedMaps` – массив загруженных карт

`unordered_map<std::string, sf::Font> fonts` - массив шрифтов

`list<sf::Texture>` – список загруженных текстур

`static Game* instance` – инстанс игры. Необходим для того, чтобы в любой момент можно было получить копию текущей игры для требуемых целей.

Описание методов класса Game:

`Game()` – конструктор

`~Game()` – деструктор

`void Start()` – запуск игры

`void StartMenu()` – запуск меню

`void StartLevel(Map& map)` – загрузка уровня

`void MainLoop()` – главный цикл отрисовки графики и выполнения игровой логики

`void Loose()` – вывод на экран сообщения о поражении

`void Win()` – вывод на экран сообщения об успешном прохождении уровня

`void Exit()` – Выход из уровня и переход в меню

`Map& loadMap(const std::string& mapFilePath)` – загрузка карты из текстового документа в массив строк

`sf::getOrLoadFont(const std::string& path)` – добавление и получение шрифта

`sf::RenderWindow& getWindow()` – получение окна, в котором отрисовывается игра

`void Update(float deltaTime)` – обработка игровой логики для всех сущностей

`void Redraw(sf::RenderWindow& window)` – отрисовка игры

`Map& getMap()` - получение карты уровня

`Scene& getScene()` - получение текущей сцены

`static Game* getInstance()` - получение указателя на инстанс игры

`sf::Texture& loadTexture(const std::string& path)` - загрузка текстуры в массив текстур

### 3.3.2 Класс сущности

В программе используется базовый абстрактный класс `Entity`, от которого в дальнейшем наследуются классы `Imovable`, `EnemyHandler`. Класс

содержит два виртуальных метода, которые перегружаются под нужды наследуемых классов. Это сделано для того, чтобы можно было удобно работать со всеми сущностями, реализованными в проекте.

Описание полей класса Entity:

`sf::Vector2f position` – координаты сущности на карте

`sf::Sprite entitySprite` – спрайт сущности

Описание методов класса Entity:

`virtual ~Entity()` – деструктор по умолчанию

`virtual void draw(sf::RenderWindow& window)` – метод отрисовки спрайта сущности в окне

`virtual void update(float deltaTime)` – обработка игровой логики для сущности

`void setPosition(sf::Vector2f pos)` – метод, который используется для установки позиции сущности в текущем окне

`sf::Vector2f getPosition()`– метод, который используется для получения позиции сущности

### 3.3.3 Класс игрока

Класс `Player`. Наследуется от класса `Imovable`. Является классом для хранения информации об игроке.

Описание полей класса Player:

`int score` – игровой счет

`int countOfHealth` – количество жизней игрока

Описание методов класса Player:

`Player()` - конструктор по умолчанию

`Player(sf::Sprite& playerSprite)` - конструктор

`void update(float deltaTime) override` - обновление логики игрока

`int getScore()` - получение текущего счета, набранного игроком

`void setScore(int value)` - установка счета игрока

`int getHealth()` - метод, необходимый для получения количества жизней игрока. Используется для отрисовки жизней игрока, а также для проверки на оставшееся количество попыток, доступных для прохождения уровня

`void setHealth(int value)` - метод установки количества жизней игрока

`bool checkCollisionWithEnemies()` - метод проверки столкновения игрока с врагами

`sf::Vector2f getPlayerPosition()` - метод получения позиции игрока

### 3.3.4 Класс врага

Класс `Enemy`. Наследуется от класса `Imovable`. Содержит в себе информацию о начальных позициях врагов, а также массив, хранящий в себе возможные направления движения

Описание полей класса Enemy:

`bool isDirAvailable[4]` — массив возможных направлений передвижения

Описание методов класса Enemy:

`Enemy(sf::Sprite enemySprite)` - конструктор

`void update(float deltaTime) override` – метод обновления логики врага

`void draw(sf::RenderWindow& window) override` – метод отрисовки врага

`void updateDir()` – метод обновления направления движения врага. Вызывается в тех случаях, когда враг находится в ячейке, из которой можно сменить направление

`MoveDirection newDir(float deltaTime)` – выбор нового направления движения

### 3.3.5 Класс-обработчик врагов

Класс `EnemyHandler`. Данный класс хранит в себе массив врагов, а также служит для обработки их игровой логики. Реализован для удобной работы с врагами, так как позволяет обрабатывать сразу четыре сущности.

Описание полей класса EnemyHandelr:

`std::vector<Enemy*>` – массив врагов

Описание методов класса EnemyHandler:

`EnemyHandler(size_t enemiesCount, std::vector<sf::Vector2f>& enemiesPositions, sf::Sprite enemySprite)` – конструктор

`~EnemyHandler()` – диструктор

`void update(float deltaTime) override` – метод обновления логики для всех врагов, содержащихся в массиве

`bool checkCollisionWithPlayer(Player* player)` – проверка столкновения врага с игроком

### 3.3.6 Класс сущностей, которые могут перемещаться

Класс `Imovable`. Данный класс является наследником класса Entity и содержит в себе поля текущего и следующего направлений перемещения сущности, а также скорость перемещения. Является классом-родителем для

классов Player и Enemy. Реализован для удобной работы со всеми сущностями, которые могут перемещаться

Описание полей класса Imovable:

`MoveDirection currDir` – текущее направление движения

`MoveDirection nextDir` – следующее направление движения

`float speed` – скорость движения

`sf::Vector2f size` – размер спрайта. Используется для

Описание методов класса Imovable:

`void update(float deltaTime) override` – обновление логики сущности

`bool checkCollision(sf::Vector2f position)` – проверка на столкновение

`static sf::Vector2f getOffset(MoveDirection dir)` – получение пары чисел, которые определяют направление движения

`void setDir(MoveDirection dir)` – метод установки направления движения

### 3.3.7 Класс сцены

Класс `Scene`. Данный класс содержит в себе все, что происходит на игровой сцене – карта, игрок, враги, худ, а также является основным классом, при отрисовке которого можно видеть графическую составляющую игры.

Описание полей класса Scene:

`std::vector<std::unique_ptr<Entity>> entities` – массив сущностей, которые есть на карте. Является универсальным, так как хранит в себе карту, игрока, врагов, меню и игровой худ.

`Map* map` – карта

Описание методов класса Scene:

`Inline Map& getMap()` – обновление логики сущности

`void update(float deltaTime) override` – метод обновления логики сцены

`void draw(sf::RenderWindow& window) override` – метод отрисовки сцены

`void setMap(Map& map)` – метод добавления уровня на сцену

`Entity* addEntity(std::unique_ptr<Entity>&& entity)` добавление сущности в массив сущностей.

`Void destroyEntity(Entity* entity)` – удаление сущности из массива сущностей

`T* getEntityOfType()` – получение указателя на сущность нужного типа

`Menu& getMenu()` – получение меню

### 3.3.8 Класс уровня

Класс `Map` – класс уровня. Данный класс содержит карту уровня в виде массива строк, хранящих в себе ячейки со стенками и точками.

Описание полей класса Map:

`std::vector<std::string> data` – массив сущностей, которые есть на карте

`std::vector<sf::Vector2f> enemiesPositions` – массив координат врагов

`std::Vector2f playerPosition` – координаты игрока

Описание методов класса Map:

`Map()` – конструктор по умолчанию

`Map(std::stream& stream)` – конструктор

`Inline const std::string& operator[](size_t index)` – перегрузка оператора []

`char getCell(sf::Vector2f position) const` – получение значения, хранящегося в ячейке карты. Используется в проверке на возможность смены направления а также при проверке наличия точки в конкретной ячейке

`void draw(sf::RenderWindow& window)` – отрисовка карты уровня с учетом всех изменений, что происходят на карте

`void setEmemiesPositions()` – установка позиций врагов. Позиции врагов хранятся в классе EnemyHandler

`void setPlayerPosition()` – установка позиции игрока. Позиция игрока хранится в классе Player

`std::vector<sf::Vector2f>& getEnemiesPositions()` – получение позиций врагов

`void setCell(char value, sf::Vector2u pos)` – установка передаваемого значения в клетку карты по координатам

`std::vector<std::string>* getMap()` – получение указателя на массив строк, содержащий в себе карту.

### 3.3.9 Класс худа

Класс `HoodElements` – класс худа. Данный класс содержит спрайт жизни игрока, а также текст, который отображает набранный игроком счёт. Необходим для отображения интерфейса.

Описание полей класса HoodElements:

`sf::Sprite heart` – спрайт сердца

`sf::Text scoreText` – объект класса Text, который отображает счет игрока

`sf::Text splashText` – объект класса Text, который отображает сообщение или о победе, или о проигрыше.

Описание методов класса HoodElements:

`void createHood()` – создание худа

`void drawHood(sf::RenderWindow& window, Player player)` – отрисовка худа

`void update() override` – обновление логики интерфейса

`void setSpashText(const std::string& text)` – установка текста сообщения. Используется при отображении окна с текстом о победе или поражении пользователя

`void onEndGame()` – присвоение параметру `isEnd` значения `true`. Параметр isEnd отвечает за текущее состояние проекта и служит флагом для отображаемого контента в окне игры. При значении параметра false в окне отображается уровень, который можно пройти. Присвоение значения true означает, что уровень был пройден, или пользователь не успел собрать все точки и потратил все попытки, которые были даны на прохождение уровня. В этом случае на экране должно быть отображено соответствующее сообщение и при нажатии пробела пользователь должен оказаться в главном меню.

### 3.3.10 Класс меню

Класс `Menu`. Данный класс содержит в себе элементы, которые отображаются в главном меню. Отображается при запуске проекта и позволяет пользователю выбрать пункт. При успешном или не успешном прохождении уровня пользователь также перемещается в главное меню.

Описание полей класса Menu:

`int userChoise` – поле выбора игрока, хранит в себе номер выбранного раздела меню

`sf::Text play` – поле, хранящее в себе название пункта меню PLAY

`sf::Text exit` – поле, хранящее в себе название пункта меню EXIT

`sf::CircleShape pointer` – поле указателя, стоящего напротив текущего выбранного пункта меню

`sf::Font font` - поле шрифта для всех текстовых полей

`sf::Text header` – поле, содержащее в себе название игры “PACMAN”

Описание методов класса Menu:

`void createMenu()` – создание меню

`void update(float deltaTime) override` – метод обновления логики меню

`void draw(sf::RenderWindow& window)` – метод отрисовки меню

### 3.3.11 Класс перечисления MoveDirection

В данном классе перечисляются 4 возможных направления движения, а также ситуация, когда сущность стоит на месте. Необходимость класса объясняется тем, что у тех сущностей, которые могут перемещаться, есть поля, хранящие в себе текущее и следующее направления движения. Они

используются для реализации перемещения сущностей по карте, проверки смены направления движения при нажатии кнопок для игрока или вычислении следующего направления для врагов.

Описание класса MoveDirection:

`None` – сущность стоит на месте

`Left` – сущность двигается влево

`Right` – сущность двигается вправо

`Up` – сущность двигается вверх

`Down` - сущность двигается вниз

# 4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

## 4.1 Разработка схем алгоритмов

Блок-схемы алгоритмов приведены в Приложениях Б, В соответственно

## 4.2 Разработка алгоритмов

### 4.2.1 Разработка алгоритма метода Update() у класса Player

Метод Update() у игрока отвечает за управление персонажем, а также проверяет столкновение игрока с врагами.

Координаты персонажа по x и по y имеют тип float. Плавное перемещение персонажа реализуется следующим образом: скорость перемещения персонажа равна 4.f. Это означает, что за один кадр персонаж может сместиться на 4 пикселя в отрисовываемом окне. Количество кадров в секунду для каждого компьютера может отличаться, что связано с его производительностью. Поэтому, чтобы перемещение персонажа и других сущностей было плавным вне зависимости от устройства, на котором запускается приложение, используется переменная deltaTime, которое хранит в себе значение, равное промежутку времени, прошедшему с момента отрисовки последнего кадра.

Далее высчитывается вектор коэффициентов, которые зависят от направления движения персонажа. Коэффициенты зависят от того, в какую сторону двигается персонаж и, соответственно, как изменяется одна из двух его координат. Например, если игрок двигается влево, то его координата по y остается неизменной, а по x уменьшается, т.е. смещение по x – отрицательное. Тогда вектор коэффициентов имеет вид {-1, 0}.

После этого высчитывается значение следующего предполагаемого положения игрока. Оно вычисляется по формуле текущая координата + вектор коэффициентов * скорость * deltaTime. Если предполагаемая позиция возможна, то есть нет стены или врага, то персонаж совершает перемещение.

Персонаж перемещается по карте при помощи нажатия стрелок. Во время передвижения нужно учитывать некоторые факторы. Первый из них – возможность смены направления движения. Это напрямую зависит от того, есть ли стенка в том направлении, которое игрок выбрал следующим. При этом есть возможность планировать повороты заранее, если до смены направления прошло менее 0.3 секунды и возможность повернуть есть. Это позволяет сделать управление удобным, а перемещение более плавным и ровным относительно отображения иконки персонажа.

Второй фактор – наличие точки в ячейке, в которой находится игрок. Поедание точек напрямую влияет на игровой счет, поэтому при смене ячейки необходимо проверить, есть ли в ней точка. При положительном ответе

необходимо увеличить счет игрока на 10 очков, а также внести изменение в карту уровня, заменив символ точки на карте пробелом.

Третий фактор – столкновение с врагом. В этой ситуации необходимо уменьшить количество жизней игрока на 1, установить игрока и врагов на их стартовые позиции.

Опишем данный алгоритм по шагам:

Шаг 1. Начало.

Шаг 2. Для всех четырех возможных направлений движения проверяем, нажата ли соответствующая кнопка. Если была нажата кнопка управления, то переходим к Шагу 3. Если нет – к Шагу 4.

Шаг 3. Переменной nextDir присваиваем значение следующего направления.

Шаг 4. Переменной nextDir присваивается значение None.

Шаг 5. Создаем указатель на карту и присваиваем ему значение посредством вызова метода getMap() для инстанса игры.

Шаг 6. Создаем переменную playerPos и присваиваем ей значение посредством преобразования координат игрока к типу unsigned int.

Шаг 7. Вызываем метод CheckCollisionWithEnemies() для проверки столкновения игрока с врагами.

Шаг 8. В случае, если игрок столкнулся с врагом, переходим к Шагу 9. Если нет – к Шагу 12.

Шаг 9. Уменьшаем количество жизней игрока на 1.

Шаг 10. Присваиваем игроку стартовую позицию.

Шаг 11. Присваиваем переменной currDir значение None.

Шаг 12. Проверяем значение, которое хранится в клетке с координатами, равными преобразованным координатам игрока

Шаг 13. Если в клетке хранится символ ‘.’, увеличиваем значение счета игрока на 10.

Шаг 14. Меняем значение клетки на символ ‘ ’(пробел).

Шаг 15. Вызываем метод Update() для класса Imovable.

Шаг 16. Конец.

### 4.2.2 Разработка алгоритма метода MainLoop() для класса Game

Метод MainLoop() отвечает за обработку и отрисовку одного кадра.

Данный метод играет большую роль в визуализации проекта, так как при наличии графики в приложении её необходимо отрисовывать в окне. Логика данного метода следующая: сначала необходимо получить время, прошедшее с момента отрисовки последнего кадра. Оно нужно для того, чтобы перемещение игрового персонажа и врагов было плавным. Затем нужно очистить окно от прошлого кадра, выполнить метод update(), который отвечает за обновление состояния каждой сущности, которая присутствует на сцене, потом отрисовать сцену и вывести её в окно проекта. Условием завершения

выполнения программы служит нажатая красная кнопка в правом верхнем углу окна приложения.

Опишем данный алгоритм по шагам:

Шаг 1. Начало.

Шаг 2. Создаём переменную deltaTime типа Clock.

Шаг 3 .Создаём переменную time типа float.

Шаг 4. Запуск цикла while, внутри которого происходит обработка игровой логики и отрисовка одного кадра.

Шаг 5. Присваиваем переменной time значение, равное времени, прошедшему с момента отрисовки последнего кадра

Шаг 6. Очищаем окно, в котором отображается игра.

Шаг 7. Обновляем игровую логику.

Шаг 8. Отрисовываем новый кадр.

Шаг 9. Выводим новый кадр в окно игры.

Шаг 10. Создаем переменную event типа Event.

Шаг 11. Запускаем цикл, внутри которого происходит проверка, нажата ли кнопка закрытия окна.

Шаг 12. Если кнопка нажата, закрываем окно.

Шаг 13. Конец.

## 5 РЕЗУЛЬТАТЫ РАБОТЫ

При запуске проекта игрока встречает главное меню, в котором он путем нажатия стрелочек Вверх и Вниз может выбрать один из предложенных пунктов меню. Меню изображено на рисунке 5.1


Рис 5.1 – Главное меню

Выбор соответствующего пункта меню осуществляется путем нажатия кнопки Enter. В случае, если игрок выберет пункт Exit, выполнение программы завершится и окно Pacman будет закрыто. Если игрок выберет пункт Play, то запустится уровень. Окно игры изображено на рисунке 5.2

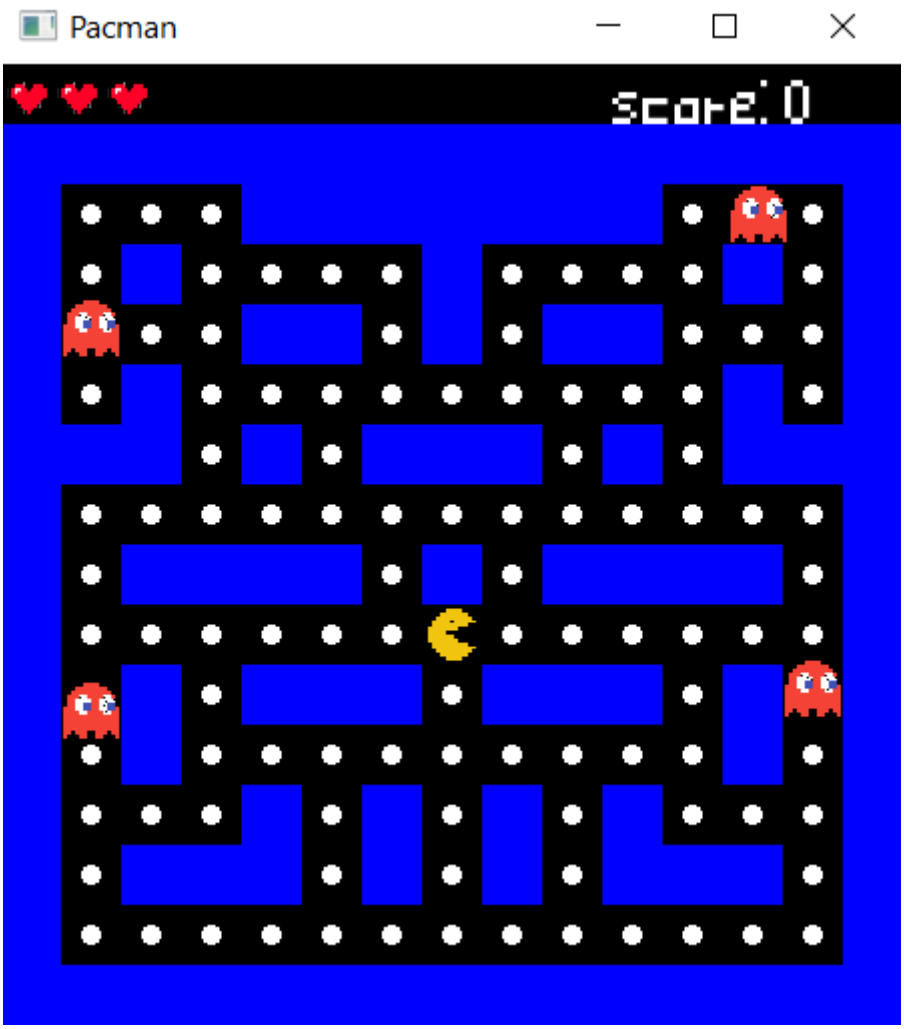


Рис. 5.2 – Уровень

В случае, если игрок столкнется с врагом, его количество жизней уменьшится, а персонаж и враги будут помещены на их стартовые позиции. Окно игры после столкновения с врагом изображено на рисунке 5.3

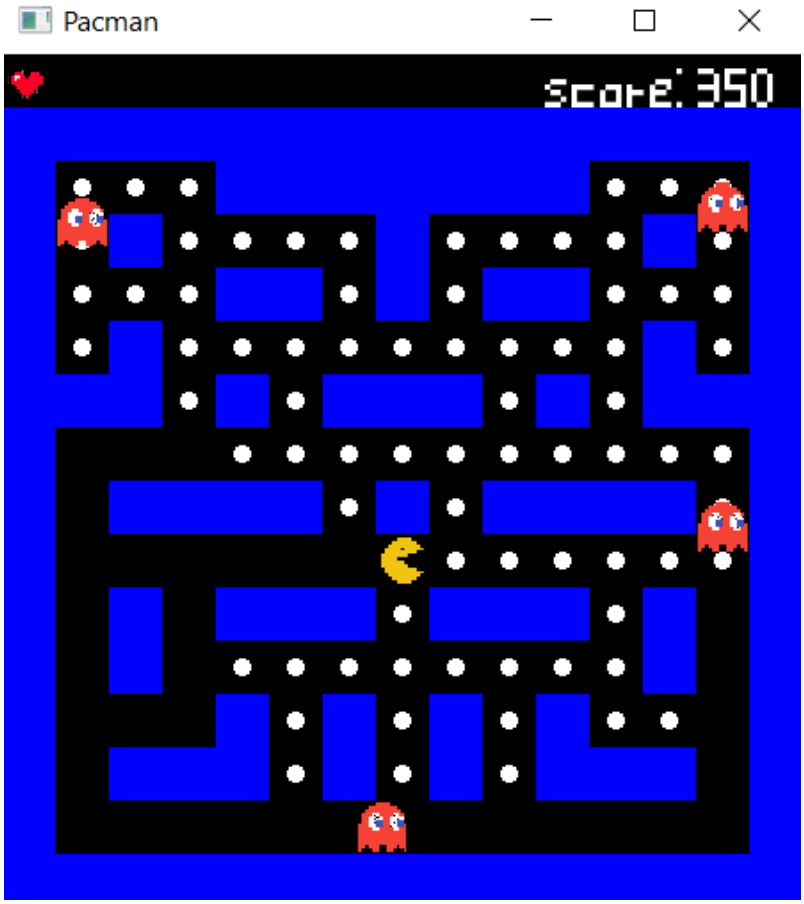


Рис. 5.3 – Состояние уровня после двух столкновений

При успешном прохождении уровня на экран выводится соответствующее сообщение. После нажатия клавиши Пробел пользователь сможет вернуться в главное меню. Окно с сообщением о победе изображено на рисунке 5.4

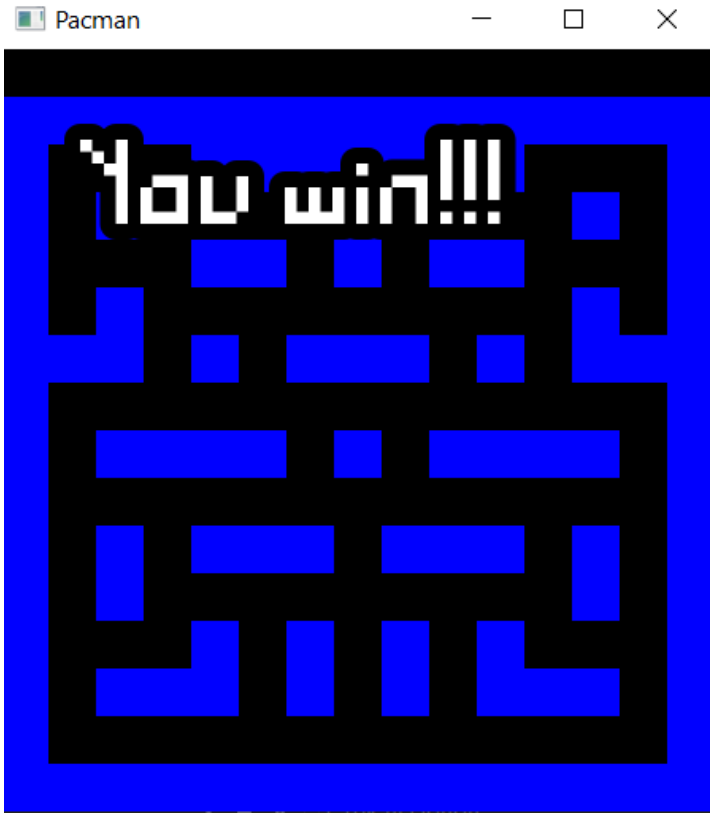


Рис. 5.4 – Сообщение об успешном прохождении уровня

Если игроку не удастся пройти уровень, то на экран будет выведено сообщение о том, что пользователь проиграл. При нажатии клавиши пробела пользователь будет возвращен в главное меню. Окно с сообщением о поражении изображено на рисунке 5.5

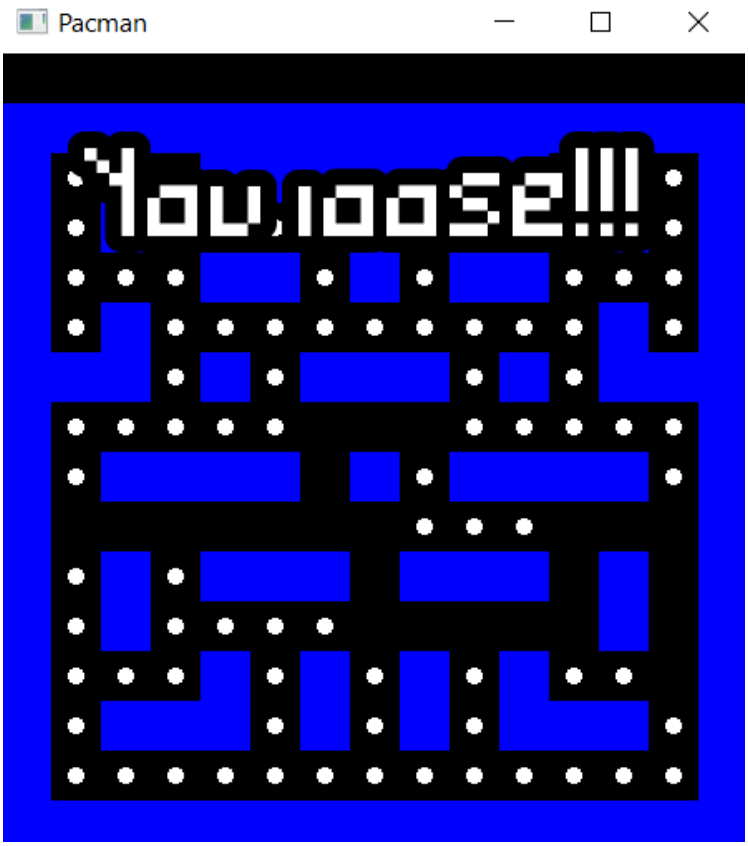


Рис 5.5 – Сообщение о проигрыше

# ЗАКЛЮЧЕНИЕ

В процессе выполнения курсовой работы были закреплены знания основ программирования и алгоритмизации. Были приобретены знания в объектно-ориентированном программировании. Также я познакомился с созданием полноценных проектов при помощи использования библиотеки SFML и убедился, что она подходит для создания несложных игр и проектов с удобным интерфейсом.

В процессе разработки игры “Пакман” были реализованы алгоритмы обновления игровой логики, отрисовки кадров, проверка всевозможных коллизий и другие. Как упоминалось раннее, моя игра является неполной версией оригинальной игры “Пакман”. Однако она открыта для модернизации. Например, можно создать новый уровень, добавить режим игры “Все или ничего” и т.д.

Игра оказалась достаточно простой в создании, имеет удобный интерфейс.

На основе полученных знаний при создании игры можно будет создать проекты, где нужно будет отрисовывать графику, нажимать кнопки обрабатывать необходимую логику. Такими примерами могут быть приложение для ведения заметок, ежедневник, календарь.

Создание игры было выполнено на ОС Windows 10, используя среду разработки Visual Studio 2022.

Код программы приведен в приложении Г.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] SFML [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.sfml-dev.org/ – Дата доступа: 11.12.2023

[2] Qt Creator [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.qt.io/product/development-tools/ – Дата доступа: 11.12.2023

[3] Qt [Электронный ресурс]. – Электронные данные. – Режим доступа: https://ru.wikipedia.org/wiki/Qt – Дата доступа: 11.12.2023

[4] Игры, похожие на Pacman [Электронный ресурс]. – Электронные данные. – Режим доступа: https://ag.ru/games/pac-man/suggestions - Дата доступа: 11.12.2023

[5] Mappy [Электронный ресурс]. – Электронные данные. – Режим доступа: https://ru.wikipedia.org/wiki/Mappy - Дата доступа: 11.12.2023

**ПРИЛОЖЕНИЕ А**

(*обязательное*)

Диаграмма классов

**ПРИЛОЖЕНИЕ Б**

(*обязательное*)

Схема алгоритма Update()

**ПРИЛОЖЕНИЕ В**

(*обязательное*)

Схема алгоритма MainLoop()

## ПРИЛОЖЕНИЕ Г
(*обязательное*)
Код программы

Файл Enemy.cpp:

```
#pragma once
#include "Enemy.h"
#include "Game.h"
#include <ctime>
#define POS_LEFT 0
#define POS_UP 2
#define POS_RIGHT 1
#define POS_DOWN 3
Imovable::MoveDirection Enemy::newDir(float deltaTime) {
	std::vector<MoveDirection> newDir;
	sf::Vector2f expVector = position;
	int currIndex = 0;
	Map& map = Game::getInstance()->getMap();
	expVector.x += deltaTime * speed;
	if (map.getCell(expVector) != '1') {
		newDir.push_back(MoveDirection::DOWN);
		expVector = position; }
	if (expVector.x >= deltaTime * speed) {
		expVector.x -= deltaTime * speed;
		if (map.getCell(expVector) != '1') {
			newDir.push_back(MoveDirection::UP);
			expVector = position; } }
	expVector.y += deltaTime * speed;
	if (map.getCell(expVector) != '1') {
		newDir.push_back(MoveDirection::LEFT);
		expVector = position; }
	if (expVector.y >= deltaTime * speed) {
		expVector.y -= deltaTime * speed;
		if (map.getCell(expVector) != '1') {
			newDir.push_back(MoveDirection::RIGHT); } }
	return newDir[rand() % newDir.size()]; }
void Enemy::updateDir() {
	currDir = newDir(1 / speed); }
void Enemy::update(float deltaTime) {
	Map& map = Game::getInstance()->getMap();
	static float hold = 2.f;
	bool isDirAvailNow[4];
	std::vector<MoveDirection> avail;
	for (int i = 0; i < 4; ++i) {
		MoveDirection dir = static_cast<MoveDirection>(i + 1);
		isDirAvailNow[i] = map.getCell(position + getOffset(dir)) == '1';
		if (isDirAvailNow[i] == false && isDirAvailable[i] !=
isDirAvailNow[i])
		avail.push_back(dir);
		isDirAvailable[i] = isDirAvailNow[i]; }
	hold -= deltaTime;
	if (hold <= 0.f && nextDir == MoveDirection::NONE && avail.empty() ==
false) {
		nextDir = avail[rand() % avail.size()];
		hold = 2.f; }
	else if (checkCollision(position + (getOffset(currDir) * speed *
deltaTime))) {
		nextDir = newDir(deltaTime); }
	Imovable::update(deltaTime); }
```

```
void Enemy::draw(sf::RenderWindow& window) {
	Entity::draw(window); }
```

Файл Enemy.h:

```
#pragma once
#include "IMovable.h"
class Enemy : public Imovable {
private:
	bool isDirAvailable[4] = { true };
public:
	Enemy(sf::Sprite enemySprite) {
		entitySprite = enemySprite; }
	void update(float deltaTime) override;
	void draw(sf::RenderWindow& window) override;
	void updateDir();
	MoveDirection newDir(float deltaTime);
};
```

Файл EnemyHandler.cpp:

```
#include "EnemyHandler.h"
#include "Game.h"
#include "Player.h"
EnemyHandler::EnemyHandler(size_t enemiesCount, std::vector<sf::Vector2f>&
enemiesPositions, sf::Sprite enemySprite) {
	Scene& scene = Game::getInstance()->getScene();
	for (int i = 0; i < enemiesCount; ++i) {
		Enemy* enemy =
reinterpret_cast<Enemy*>(scene.addEntity(std::make_unique<Enemy>(enemySprite)
));
		enemy->setPosition(enemiesPositions[i]);
		enemy->updateDir();
		enemies.push_back(enemy); } }
EnemyHandler::~EnemyHandler() {
	for (auto enemy : enemies)
		Game::getInstance()->getScene().destroyEntity(enemy); }
void EnemyHandler::update(float deltaTime) {    }
bool EnemyHandler::checkCollisionWithPlayer(Player* player){
	Map map = Game::getInstance()->getMap();
	std::vector<sf::Vector2f> enPos = map.getEnemiesPositions();
	sf::Vector2f playerPos = player->getPosition();
	sf::Rect<float> playerRect(playerPos - sf::Vector2f{ 0.5f,0.5f },
sf::Vector2f{ 1,1 });
	for (auto& enemy : enemies) {
		sf::Rect<float> enemyRect(enemy->getPosition() -
sf::Vector2f{0.5f,0.5f}, sf::Vector2f{1,1});
		if (enemyRect.intersects(playerRect)) {
		for (int i = 0; i < 4; i++) {
		enemies[i]->setPosition(enPos[i]); }
		return true; } }
	return false; }
```

Файл EnemyHandler.h:

```
#pragma once
#include "Entity.h"
#include "Enemy.h"
class Player;
class EnemyHandler : public Entity {
private:
	std::vector<Enemy*> enemies;
public:
```

```
	EnemyHandler(size_t enemiesCount, std::vector<sf::Vector2f>&
enemiesPositions, sf::Sprite enemySprite);
	~EnemyHandler();
	void update(float deltaTime) override;
	bool checkCollisionWithPlayer(Player* player);
};
```

Файл Entity.cpp:

```
#pragma once
#include "Entity.h"
void Entity::draw(sf::RenderWindow& window) {
	entitySprite.setPosition(position.x * 30, (position.y + 1) * 30);
	window.draw(entitySprite); }
void Entity::setPosition(sf::Vector2f pos) {
	position = pos; }
sf::Vector2f Entity::getPosition() {
	return position; }
```

Файл Entity.h:

```
#pragma once
#include <iostream>
#include "SFML\Graphics.hpp"
#include "Map.h"
class Entity {
protected:
	sf::Vector2f position;
	sf::Sprite entitySprite;
public:
	virtual ~Entity() = default;
	virtual void draw(sf::RenderWindow& window);
	virtual void update(float deltaTime) = 0;
	void setPosition(sf::Vector2f pos);
	sf::Vector2f getPosition();
};
```

Файл Game.cpp:

```
#pragma once
#include "Game.h"
#include "Player.h"
#include "Map.h"
#include "EnemyHandler.h"
#include <Windows.h>
#include "HoodElements.h"
#include "Menu.h"
Game* Game::instance = nullptr;
void Game::Redraw(sf::RenderWindow& window) {
	scene.draw(window); }
void Game::Start() {
	StartMenu(); }
void Game::StartMenu() {
	scene = Scene();
	scene.addEntity(std::make_unique<Menu>());
	scene.getEntityOfType<Menu>()->createMenu(); }
void Game::StartLevel(Map& map) {
	scene = Scene();
	sf::Texture& playerIcon = loadTexture("kkk.png");
	sf::Sprite playerSprite;
	playerSprite.setTexture(playerIcon);
	playerSprite.setScale(0.025, 0.025);
	playerSprite.setOrigin(playerIcon.getSize().x * 0.5f,
playerIcon.getSize().y * 0.5f);
```

```
	scene.setMap(map);
	scene.addEntity(std::make_unique<Player>(playerSprite));
	scene.addEntity(std::make_unique<HoodElements>());
	scene.getEntityOfType<HoodElements>()-
>createHood(*scene.getEntityOfType<Player>());
	sf::Texture& enemyIcon = loadTexture("jjj.png");
	sf::Sprite enemySprite;
	enemySprite.setTexture(enemyIcon);
	enemySprite.setScale(0.03, 0.03);
	enemySprite.setOrigin(enemyIcon.getSize().x * 0.5f,
enemyIcon.getSize().y * 0.5f);
	std::srand(std::time(nullptr));
	scene.addEntity(std::make_unique<EnemyHandler>(4,
map.getEnemiesPositions(), enemySprite)); }
void Game::Loose() {
	HoodElements* hood = scene.getEntityOfType<HoodElements>();
	scene.destroyEntity(scene.getEntityOfType<Player>());
	scene.destroyEntity(scene.getEntityOfType<EnemyHandler>());
	hood->setSplashText("You loose!!!");
	hood->onEndGame(); }
void Game::Win() {
	HoodElements* hood = scene.getEntityOfType<HoodElements>();
	scene.destroyEntity(scene.getEntityOfType<Player>());
	scene.destroyEntity(scene.getEntityOfType<EnemyHandler>());
	hood->setSplashText("You win!!!");
	hood->onEndGame(); }
void Game::MainLoop() {
	sf::Clock deltaTime;
	float time = 0;
	while (window.isOpen()) {
		float timer = 0;
		time = deltaTime.restart().asSeconds();
		window.clear();
		Update(time);
		Redraw(window);
		window.display();
		sf::Event event;
		while (window.pollEvent(event)) {
		if (event.type == sf::Event::Closed) {
		window.close(); } } } }
void Game::Exit() {
	window.close(); }
sf::RenderWindow& Game::getWindow() {
	return this->window; }
void Game::Update(float deltaTime) {
	scene.update(deltaTime); }
Map& Game::getMap() {
	return scene.getMap(); }
Map& Game::loadMap(const std::string& mapFilePath) {
	std::ifstream file(mapFilePath);
	if (file.is_open() == false)
		throw std::runtime_error("Can't load map");
	loadedMaps.push_back(Map(file));
	return loadedMaps.back(); }
Scene& Game::getScene() {
	return scene; }
Game* Game::getInstance() {
	return instance; }
sf::Texture& Game::loadTexture(const std::string& path){
	loadedTextures.push_back(sf::Texture());
	loadedTextures.back().loadFromFile(path);
	return loadedTextures.back(); }
```

```
sf::Font& Game::getOrLoadFont(const std::string& path) {
	if (fonts.count(path))
		return fonts.at(path);
	fonts.insert({ path, sf::Font() });
	fonts.at(path).loadFromFile(path);
	return fonts.at(path); }
```

Файл Game.h:

```
#pragma once
#include <iostream>
#include <unordered_map>
#include <list>
#include "Player.h"
#include "Map.h"
#include "Enemy.h"
#include "Scene.h"
#include "Menu.h"
class Game{
private:
	Scene scene;
	Menu currMenu;
	sf::RenderWindow window;
	std::vector<Map> loadedMaps;
	std::unordered_map<std::string, sf::Font> fonts;
	std::list<sf::Texture> loadedTextures;
	static Game* instance;
public:
	Game() : window(sf::VideoMode(450, 480), "Pacman") { instance = this; };
	void Start();
	void StartMenu();
	void StartLevel(Map& map);
	void Loose();
	void Win();
	void MainLoop();
	void Exit();
	sf::RenderWindow& getWindow();
	void Update(float deltaTime);
	void Redraw(sf::RenderWindow& render);
	Map& getMap();
	Map& loadMap(const std::string& mapFilePath);
	Scene& getScene();
	static Game* getInstance();
	sf::Texture& loadTexture(const std::string& path);
	sf::Font& getOrLoadFont(const std::string& path);
};
```

Файл HoodElements.cpp:

```
#include "HoodElements.h"
#include "Game.h"
void HoodElements::createHood(Player& player) {
	heart.setTexture(Game::getInstance()->loadTexture("heart.png"));
	heart.setScale(0.04f, 0.04f);
	scoreText.setFont(Game::getInstance()-
>getOrLoadFont("pixel_perfect.ttf"));
	scoreText.setCharacterSize(25);
	scoreText.setPosition(305, 5);
	splashText.setCharacterSize(60);
	splashText.setFillColor(sf::Color::White);
	splashText.setOutlineThickness(10.f);
	splashText.setFont(Game::getInstance()-
>getOrLoadFont("pixel_perfect.ttf"));
```

```
	splashText.setPosition(50, 50);
	this->player = &player; }
void HoodElements::draw(sf::RenderWindow& window) {
	if (isEnd) {
		window.draw(splashText);
		return; }
	for (int i = 0; i < player->getHealth(); i++) {
		heart.setPosition(i * 25, 5);
		window.draw(heart); }
	std::string scoreMessage = { "score: " + std::to_string(player-
>getScore()) };
	scoreText.setString(scoreMessage);
	window.draw(scoreText); }
void HoodElements::update(float deltaTime) {
	if (isEnd == false)
		return;
	if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space)) {
		Game::getInstance()->StartMenu(); } }
void HoodElements::setSplashText(const std::string& text) {
	splashText.setString(text); }
void HoodElements::onEndGame() {
	isEnd = true; }
```

Файл HoodElements.h:

```
#pragma once
#include "Player.h"
class HoodElements : public Entity {
	sf::Sprite heart;
	sf::Text scoreText;
	sf::Text splashText;
	Player* player;
	bool isEnd = false;
public:
	void createHood(Player& player);
	void update(float deltaTime) override;
	void draw(sf::RenderWindow& window) override;
	void setSplashText(const std::string& text);
	void onEndGame();
};
```

Файл iMovable.cpp:

```
#include "IMovable.h"
#include "Game.h"
bool isCloseEnoughToHalf(float value) {
	return std::abs((value - (unsigned int)value) - 0.5f) < 0.05f; }
sf::Vector2f Imovable::getOffset(MoveDirection dir) {
	switch (dir) {
	case(MoveDirection::LEFT):
		return { -1, 0 };
		break;
	case(MoveDirection::RIGHT):
		return { 1, 0 };
		break;
	case(MoveDirection::UP):
		return { 0, -1 };
		break;
	case(MoveDirection::DOWN):
		return { 0, 1 };
		break;
	default:
		return { 0, 0 };
```

```
            break; } }
void Imovable::update(float deltaTime) {
      static constexpr const float timerDefault = 0.3f;
      static float timer = timerDefault;
      if (nextDir != MoveDirection::NONE && timer > 0) {
            timer -= deltaTime;
            sf::Vector2f offset = getOffset(nextDir) * speed * deltaTime;
            sf::Vector2f tempPos = position;
            if (isCloseEnoughToHalf(tempPos.x))
                  tempPos.x = (unsigned int)position.x + 0.5f;
            if (isCloseEnoughToHalf(tempPos.y))
                  tempPos.y = (unsigned int)position.y + 0.5f;
            if (checkCollision(tempPos + offset) == false) {
                  currDir = nextDir;
                  nextDir = MoveDirection::NONE; } }
      else {
            timer = timerDefault;
            nextDir = MoveDirection::NONE; }
      sf::Vector2f offset = getOffset(currDir) * speed * deltaTime;
      sf::Vector2f expectedPosion = position + offset;
      if (checkCollision(expectedPosion)) {
            position.x = (unsigned int)position.x + 0.5f;
            position.y = (unsigned int)position.y + 0.5f; }
      else {
            position = expectedPosion; } }
float roundToUpper(float value){
      float fraction = value - (int)value;
      return fraction == 0 ? value : (unsigned int)(value + 1); }
void Imovable::setDir(MoveDirection dir) {
      nextDir = dir; }
bool Imovable::checkCollision(sf::Vector2f position) {
      Map& map = Game::getInstance()->getMap();
      sf::Vector2f pointLeftUp, pointRightDown, pointLeftDown, pointRightUp;
      pointLeftUp.x = position.x - size.x / 2;
      pointLeftUp.y = position.y - size.y / 2;
      if (map.getCell(pointLeftUp) == '1')
            return true;
      pointRightDown.x = roundToUpper(position.x + size.x / 2) - 1;
      pointRightDown.y = roundToUpper(position.y + size.y / 2) - 1;
      if (map.getCell(pointRightDown) == '1')
            return true;
      pointLeftDown.x = pointLeftUp.x;
      pointLeftDown.y = pointRightDown.y;
      if (map.getCell(pointLeftDown) == '1')
            return true;
      pointRightUp.x = pointRightDown.x;
      pointRightUp.y = pointLeftUp.y;
      if (map.getCell(pointRightUp) == '1')
            return true;
      return false;     }
```

Файл IMovable.h:

```
#pragma once
#include "Entity.h"
#include "Map.h"
class Imovable : public Entity {
protected:
      enum class MoveDirection {
            NONE,
            LEFT,
            RIGHT,
            UP,
```

```
            DOWN
      };
      MoveDirection currDir = MoveDirection::NONE;
      MoveDirection nextDir = MoveDirection::NONE;
      float speed = 4.f;
      sf::Vector2f size = { 1, 1 };
public:
      void update(float deltaTime) override;
      bool checkCollision(sf::Vector2f position);
      static sf::Vector2f getOffset(MoveDirection dir);
      void setDir(MoveDirection dir);
};
```

Файл main.cpp:

```
#include <iostream>
#include "Game.h"
int main() {
      Game pacman;
      pacman.Start();
      pacman.MainLoop();
      return 0; }
```

Файл Map.cpp:

```
#include "Map.h"
Map::Map(std::istream& stream) {
      std::string tempLine;
      while (std::getline(stream, tempLine)) {
            data.push_back(tempLine); }
      setEnemiesPositions(); }
static float GetNearestToHalf(float value) {
      return std::abs((value - (unsigned int)value) - 0.5f); }
void Map::setEnemiesPositions() {
      enemiesPositions.resize(4);
      enemiesPositions[0] = { 1.5f, 1.5f };
      enemiesPositions[1] = { 13.5f, 1.5f };
      enemiesPositions[2] = { 1.5f, 13.5f };
      enemiesPositions[3] = { 13.5f, 13.5f }; }
std::vector<sf::Vector2f>& Map::getEnemiesPositions() {
      return this->enemiesPositions; }
char Map::getCell(sf::Vector2f position) const {
      sf::Vector2u iPos = { (unsigned int)position.x, (unsigned int)position.y
};
      if (iPos.y >= data.size())
            return true;
      if (iPos.x >= data[iPos.y].size())
            return true;
      return data[iPos.y][iPos.x]; }
void Map::setPlayerPosition(sf::Vector2f player_Position) {
      playerPosition = player_Position; }
void Map::draw(sf::RenderWindow& window) {
      sf::RectangleShape brick(sf::Vector2f(30, 30));
      sf::CircleShape point(5.f);
      brick.setFillColor(sf::Color::Blue);
      point.setFillColor(sf::Color::White);
      for (int i = 0; i < 15; i++) {
            for (int j = 0; j < 15; j++) {
            brick.setPosition(j * 30, 30 + i * 30);
            point.setPosition(j * 30 + 10, i * 30 + 40);
            if (data[i][j] == '1' && data[i][j] != ' ') {
            window.draw(brick); }
            else if (data[i][j] == '.' && data[i][j] != ' ') {
```

```
            window.draw(point); } } } }
void Map::setCell(char value, sf::Vector2u pos) {
      data[pos.y][pos.x] = value; }
std::vector<std::string>* Map::getMap() {
      return &data; }
```

Файл Map.h:

```
#pragma once
#include <iostream>
#include <SFML/Graphics.hpp>
#include <time.h>
#include <fstream>
class Map {
private:
      std::vector<std::string> data;
      std::vector<sf::Vector2f> enemiesPositions;
      sf::Vector2f playerPosition;
public:
      Map() = default;
      Map(std::istream& stream);
      inline const std::string& operator[](size_t index) const {
            return data[index]; }
      char getCell(sf::Vector2f position) const;
      void draw(sf::RenderWindow& window);
      void setEnemiesPositions();
      void setPlayerPosition(sf::Vector2f player_Position);
      std::vector<sf::Vector2f>& getEnemiesPositions();
      void setCell(char value, sf::Vector2u pos);
      std::vector<std::string>* getMap();
};
```

Файл Menu.cpp:

```
#include "Menu.h"
#include "Game.h"
void Menu::draw(sf::RenderWindow& window) {
      pointer.setPosition(145, 215 + userChoise * 40);
      window.draw(header);
      window.draw(pointer);
      window.draw(play);
      window.draw(exit); }
void Menu::createMenu() {
      font.loadFromFile("pixel_perfect.ttf");
      play.setFont(font);
      std::string playMessage = { "PLAY" };
      play.setString(playMessage);
      play.setCharacterSize(35);
      play.setPosition(180, 200);
      exit.setFont(font);
      std::string exitMessage = { "EXIT" };
      exit.setString(exitMessage);
      exit.setCharacterSize(35);
      exit.setPosition(180, 280);
      header.setFont(font);
      std::string headerMessage("PACMAN");
      header.setString(headerMessage);
      header.setCharacterSize(90);
      header.setPosition(50, 50);
      pointer.setRadius(5.f);
      pointer.setFillColor(sf::Color::White); }
void Menu::update(float deltaTime) {
      if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up) && userChoise > 0) {
```

```
            userChoise--; }
      if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down) && userChoise < 2) {
            userChoise++; }
      if (sf::Keyboard::isKeyPressed(sf::Keyboard::Enter)) {
            if (userChoise == 0) {
            Game::getInstance()->StartLevel(Game::getInstance()-
>loadMap("lvl1.txt")); }
            else {
            Game::getInstance()->Exit(); } } }
```

Файл Menu.h:

```
#pragma once
#include <iostream>
#include <SFML/Graphics.hpp>
#include "Entity.h"
class Menu : public Entity{
protected:
      int userChoise = 0;
      sf::Text play;
      sf::Text exit;
      sf::CircleShape pointer;
      sf::Font font;
      sf::Text header;
public:
      void createMenu();
      void update(float deltaTime) override;
      void draw(sf::RenderWindow& window) override;
};
```

Файл Player.cpp:

```
#pragma once
#include "Player.h"
#include "Game.h"
#include "SFML/Graphics/Text.hpp"
#include "EnemyHandler.h"
#include "Map.h"
void Player::update(float deltaTime) {
      if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left)) {
            nextDir = MoveDirection::LEFT; }
      if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right)) {
            nextDir = MoveDirection::RIGHT; }
      if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up)) {
            nextDir = MoveDirection::UP; }
      if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down)) {
            nextDir = MoveDirection::DOWN; }
      std::vector<std::string>* currMap = Game::getInstance()-
>getMap().getMap();
      sf::Vector2u playerPos = { (unsigned int)position.x, (unsigned
int)position.y };
      if (checkCollisionWithEnemies()) {
            countOfHealth--;
            currDir = MoveDirection::NONE;
            sf::Vector2f newPos = { 7.5, 8.5 };
            setPosition(newPos);
            if (countOfHealth == 0)
            Game::getInstance()->Loose();
            return; }
      if ((*currMap)[playerPos.y][playerPos.x] == '.') {
            setScore(10);
            (*currMap)[playerPos.y][playerPos.x] = ' ';
            if (score == 1110) {
```

```
            Game::getInstance()->Win();
            return; } }
      Imovable::update(deltaTime); }
void Player::setHealth(int value) {
      countOfHealth = value; }
int Player::getHealth() {
      return this->countOfHealth; }
void Player::setScore(int value) {
      score += value; }
int Player::getScore(){
      return this->score; }
sf::Vector2f Player::getPlayerPosition() {
      return position; }
bool Player::checkCollisionWithEnemies() {
      Scene& scene = Game::getInstance()->getScene();
      EnemyHandler* handler = scene.getEntityOfType<EnemyHandler>();
      return handler->checkCollisionWithPlayer(this); }
```

Файл Player.h:

```
#pragma once
#include "IMovable.h"
#include "EnemyHandler.h"
class Player : public Imovable {
protected:
      int score = 0;
      int countOfHealth = 3;
public:
      Player() = default;
      Player(sf::Sprite& playerSptire) {
            entitySprite = playerSptire;
            position = { 7.5, 8.5 }; }
      void update(float deltaTime) override;
      int getScore();
      void setScore(int value);
      int getHealth();
      void setHealth(int value);
      bool checkCollisionWithEnemies();
      sf::Vector2f getPlayerPosition();
};
```

Файл Scene.cpp:

```
#include "Scene.h"
void Scene::update(float deltaTime) {
      for (int i = 0; i < entities.size(); ++i)
            if (entities[i] != nullptr)
            entities[i]->update(deltaTime); }
void Scene::draw(sf::RenderWindow& window) {
      if (map)
            map->draw(window);
      for (auto& entity : entities)
            if (entity != nullptr)
            entity->draw(window); }
void Scene::setMap(Map& map) {
      this->map = &map; }
Entity* Scene::addEntity(std::unique_ptr<Entity>&& entity) {
      entities.push_back(std::move(entity));
      return entities.back().get(); }
std::vector<std::unique_ptr<Entity>>& Scene::getEntities() {
      return entities; }
void Scene::destroyEntity(Entity* entity) {
      for (auto& ent : entities) {
```

```
            if (ent.get() == entity) {
            std::remove(entities.begin(), entities.end(), ent);
            break; } } }
```

Файл Scene.h:

```
#pragma once
#include "Entity.h"
#include "Map.h"
#include <SFML/Graphics.hpp>
#include "HoodElements.h"
#include "Menu.h"
class Scene {
private:
      std::vector<std::unique_ptr<Entity>> entities;
      Map* map;
public:
      inline Map& getMap() {
            return *map; }
      void update(float deltaTime);
      void draw(sf::RenderWindow& window);
      void setMap(Map& map);
      Entity* addEntity(std::unique_ptr<Entity>&& entity);
      std::vector<std::unique_ptr<Entity>>& getEntities();
      void destroyEntity(Entity* entity);
      template<typename T>
      T* getEntityOfType() {
            for (auto& entity : entities) {
            if (dynamic_cast<T*>(entity.get()) != nullptr) {
            return reinterpret_cast<T*>(entity.get()); } }
            return nullptr; }
      Menu& getMenu();
};
```

**ПРИЛОЖЕНИЕ Д**
(обязательное)
Ведомость документов