МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Южно-Уральский государственный университет
(национальный исследовательский университет)»**
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой, д.ф.-м.н.,
профессор

_____ Л.Б. Соколинский

«___»_____ 2024 г.

Разработка игры в жанре «Tactical RPG» на платформе Unity

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ЮУрГУ – 02.03.02.2024.308-319.ВКР

Научный руководитель,
доцент кафедры СП, к.ф.-м.н.
_____ С.У. Турлакова

Автор работы,
студент группы КЭ-401
_____ А.Д. Шестаков

Ученый секретарь
(нормоконтролер)
_____ И.Д. Володченко
«___»_____ 2024 г.

Челябинск, 2024 г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

УТВЕРЖДАЮ
Зав. кафедрой СП
_____ Л.Б. Соколинский
29.01.2024 г.

ЗАДАНИЕ
на выполнение выпускной квалификационной работы бакалавра
студенту группы КЭ-401
Шестакову Артему Денисовичу,
обучающемуся по направлению
02.03.02 «Фундаментальная информатика и информационные технологии»

1. **Тема работы** (утверждена приказом ректора от 22.04.2024 г. № 764-13/12)
Разработка игры в жанре «Tactical RPG» на платформе Unity.
2. **Срок сдачи студентом законченной работы:** 03.06.2024 г.
3. **Исходные данные к работе**
3.1. Документация Unity. [Электронный ресурс] URL: https://docs.unity.com (дата обращения: 13.02.2024 г.).
3.2. Документация по языку C#. [Электронный ресурс] URL: https://learn.microsoft.com/ru-ru/dotnet/csharp/ (дата обращения: 15.02.2024 г.).
4. **Перечень подлежащих разработке вопросов**
4.1. Провести анализ предметной области.
4.2. Спроектировать архитектуру разрабатываемого приложения.
4.3. Реализовать и протестировать разработанное игровое приложение.
5. **Дата выдачи задания:** 29.01.2024 г.

Научный руководитель,
доцент кафедры СП, к.ф.-м.н. С.У. Турлакова

Задание принял к исполнению А.Д. Шестаков

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность

В современном мире игры занимают неотъемлемую часть индустрии развлечений. Игровая индустрия стремительно развивалась последние десятилетия и достигла своего пика в 2020 году [1]. На 2023 год объем рынка составлял 184 миллиарда долларов [2].

Постановка задачи

Целью выпускной квалификационной работы является разработка компьютерной игры в жанре «Tactical RPG» на платформе Unity. Для достижения поставленной цели необходимо решить следующие задачи:

1) провести анализ предметной области и обзор аналогов;
2) спроектировать игровое приложение;
3) реализовать и протестировать итоговое приложение.

Структура и содержание работы

Работа состоит из введения, пяти глав, заключения, списка литературы и приложения. Объем работы составляет 36 страниц, объем списка литературы – 16 источников.

В первой главе описываются особенности игрового жанра и существующих аналогов.

Вторая глава посвящена функциональным и нефункциональным требованиям к системе, построена диаграмма вариантов использования.

В третьей главе описывается общая архитектура системы и составляющие ее компоненты, построена диаграмма компонентов.

В четвертой главе описывается реализация компонентов системы.

В пятой главе описывается проведенные функциональные и юзабилити тестирования и исправленные ошибки.

В приложении приведены листинги системы ходов и системы контроля персонажей.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Описание предметной области

Игры жанра «Tactical RPG», реже называемые тактическими ролевыми играми, сочетают элементы компьютерных ролевых игр и компьютерных тактических игр. Основными элементами подобных игр является наличие тактических сражений, а также ролевой системы.

Тактические сражения представляют собой задачу, в которой игроку предстоит победить противников за счет правильного использования способностей подконтрольных персонажей, а также правильного стратегического планирования. Сражения делятся на пошаговые, где бой разделен на ходы, в момент которых игрок и компьютер поочередно совершают действия, а также сражения в реальном времени, которые игрок способен поставить на паузу, чтобы обдумать стратегию и раздать команды подконтрольным персонажам.

Ролевая система – это набор игровых правил, которые определяют возможности персонажей совершать те или иные действия. В компьютерных ролевых играх игрок не действует самостоятельно, а управляет действиями своего персонажа, поэтому персонаж может иметь способности и возможности, которыми не обладает игрок. Его возможности определяются набором характеристик, которые игрок обычно выбирает сам при создании персонажа.

1.2. Сравнительный анализ аналогов

Для грамотного определения требований к разрабатываемому игровому приложению был проведен анализ компьютерных в жанре «Tactical RPG». Ниже приведен список причин, почему для анализа были выбраны именно эти игровые проекты.

1. Есть пошаговая боевая система.
2. Есть ролевая система с возможностью выбора направления развития персонажей, подконтрольных игроку.

3. Проекты имеют популярность среди игроков и хорошие отзывы.

4. Основной упор в игровом процессе стоит на боях, а не на разговорах и выполнении заданий.

5. Проекты появились не более 10 лет назад.

На рисунке 1 представлен скриншот из игры «Divinity: Original Sin 2». Данный проект стал одним из самых популярных в жанре тактических ролевых игр и получил высокие оценки от критиков.

Рисунок 1 – Игра «Divinity: Original Sin 2»

Главной особенностью как ролевой, так и тактической составляющей игры является взаимодействие с окружением. В игре присутствует система стихий, позволяющая создавать поверхности разного типа, по-разному взаимодействующие между собой, а также влияющие на персонажей, которые на них находятся.

Также очень важно положение персонажа в пространстве – персонаж получает преимущество, если находится выше, чем его противник, а некоторые способности позволяют наносить дополнительный урон, если персонаж находится у противника за спиной.

Благодаря этому игре удается создавать интересные ситуации, для решения которых нужно применять креативность.

На рисунке 2 представлен скриншот из игры «Wasteland 3». Проект является продолжением серии игр, первая из которых вышла еще в 1988 году.

Рисунок 2 – Игра «Wasteland 3»

Игра делает большой упор на боевую составляющую. Сражения интересны за счет наличия большого количества различного оружия, которое можно модифицировать и подстраивать под конкретные нужды, разнообразия противников с уникальными способностями и интересных боевых локаций. Благодаря наличию различных типов урона, часто приходится подбирать тактику под конкретных противников.

Основным недостатком игры является слишком быстрое развитие персонажей игрока, из-за чего бои становятся слишком простыми и не представляют простора для продумывания тактики боя. Также многие игроки жалуются на слишком простой искусственный интеллект противников, что тоже сильно упрощает игру.

На рисунке 3 представлен скриншот из игры «XCOM 2». Игра, хоть и является в первую очередь пошаговой тактикой, имеет в себе ролевые элементы и простую ролевую систему с выбором способностей для подконтрольных персонажей.

Рисунок 3 – Игра «XCOM 2»

XCOM 2 является одной из самых известных пошаговых тактик, в ней есть два основных игровых режима – тактический, в котором игрок должен сражаться с пришельцами отрядом из 4–6 бойцов, и стратегический, в котором игрок занимается менеджментом ресурсов и бойцов. Проект полюбился игрокам за грамотно выстроенную сложность сражений, которая грамотно повышается по мере развития игрока, благодаря чему в них трудно преуспеть, если не принимать правильных тактических решений.

В игре нельзя создавать своих персонажей, но можно развивать подконтрольных персонажей, выбирать для них различные способности и снаряжение.

Основным недостатком игры являются шансы попадания в игре. Даже стреляя в противника вплотную, персонаж способен промахнуться, часто подобное происходит в самый ответственный момент, решающий исход битвы.

1.3. Обоснование выбора средств реализации

Для реализации будет использоваться игровой движок Unity [3]. Он является одним из самых распространенных на данный момент игровых

движков с низким порогом входа из-за большого количества обучающих материалов.

Движок поддерживает множество современных игровых платформ и большое количество расширений, ускоряющих процесс разработки игрового приложения.

Среди недостатков Unity можно выделить сложность в разработке крупных игровых приложений в большой команде разработчиков, большой вес созданных игр и проблемы с производительностью игровых приложений [4].

Но данные минусы несущественные для поставленной цели, более того, на Unity было создано множество ролевых тактических игр как от небольших независимых игровых студий, так и от куда более известных разработчиков: Pillars of Eternity, Pathfinder, а также Wasteland 3, приведенный в анализе аналогичных проектов. Такая популярность у разработчиков жанра говорит о том, что Unity является хорошим решением для создания тактических ролевых игр.

Вывод по первой главе

В результате анализа предметной области были рассмотрены аналоги, проанализированы их сильные стороны и недостатки. Было решено использовать движок Unity для разработки игрового приложения.

2. АНАЛИЗ ТРЕБОВАНИЙ К ПРОЕКТИРУЕМОЙ СИСТЕМЕ

2.1. Функциональные и нефункциональные требования

Функциональные требования – это требования того, как должна вести себя система. Они определяют, что система должна делать, чтобы удовлетворить потребности или ожидания пользователя без учета ограничений, связанных с ее реализацией.

Функциональные требования к проектируемой игре приведены ниже.

1. Игра должна иметь пошаговый боевой режим на клеточном поле.
2. Должна быть реализована ролевая система, в условиях которой игрок мог бы создавать своих персонажей.
3. Должна быть реализована возможность совершать различные действия во время хода игрока.
4. Должен быть реализован искусственный интеллект противников.
5. Должна быть реализована система боя, в которой эффективность тех или иных действий персонажа игрока зависит от его характеристик.
6. Должна быть реализована система перезапуска игры при потере всех жизней.

Нефункциональные требования – требования, которые определяют качественные характеристики проектируемого игрового приложения, они приведены ниже.

1. Игра должна быть разработана на платформе Unity.
2. Система должна быть написана на языке программирования C#.
3. Система должна работать на операционной системе Windows 10.

2.2. Диаграмма вариантов использования

Для проектирования системы был использован язык графического описания UML. В соответствии с требованиями была построена диаграмма вариантов использования, отражающая взаимодействие пользователя с функциями разрабатываемого игрового приложения, она представлена на рисунке 4.

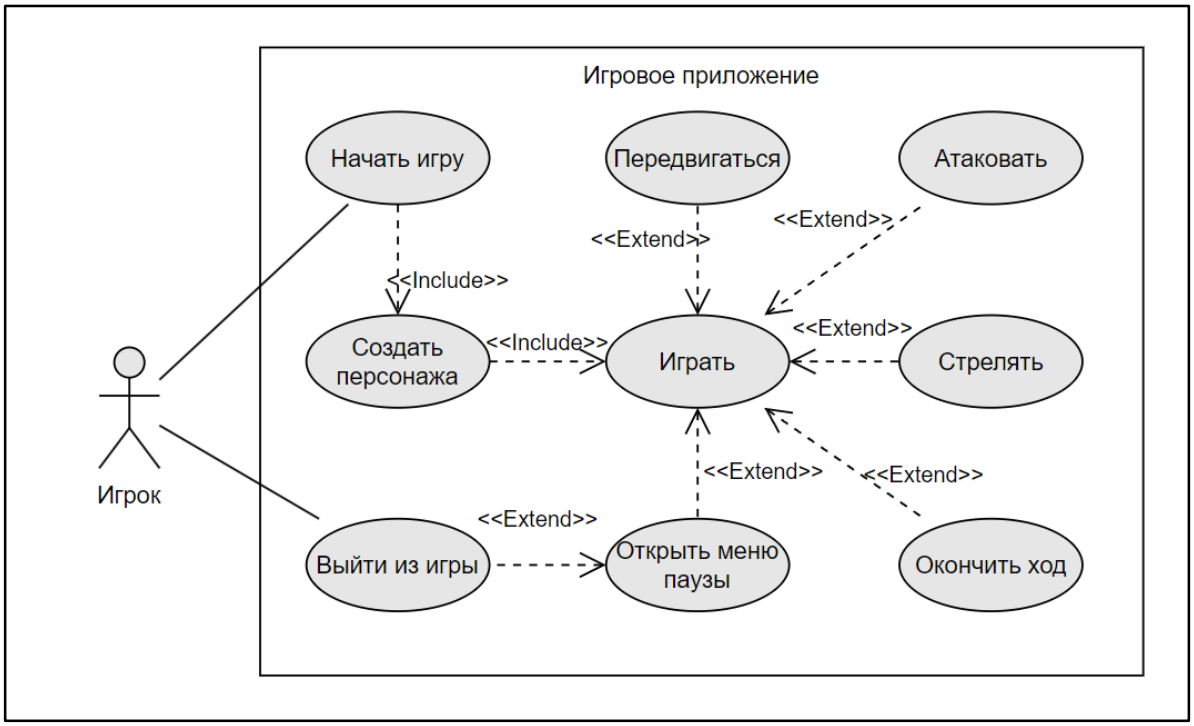

Рисунок 4 – Диаграмма вариантов использования

Единственным актером, представленным на диаграмме является игрок. Перед началом игры игрок должен создать своего персонажа. После этого запускается уровень, который игрок должен пройти до конца. В ходе игры игрок может передвигаться, стрелять и атаковать противников. Помимо этого, игрок в любой момент может открыть меню паузы и выйти из игры.

Вывод по второй главе

Во второй главе были определены как функциональные, так и нефункциональные требования. На основе требований была построена диаграмма вариантов использования, определены основные актеры, взаимодействующие с системой, а также краткое описание вариантов использования.

3. АРХИТЕКТУРА СИСТЕМЫ

Архитектура системы состоит из сцены главного меню, сцены создания персонажа и сцены игрового уровня.

Во время запуска игры загружается главное меню, оно состоит из заднего фона, названия и пользовательского интерфейса, состоящего из набора кнопок. После нажатия на кнопки «Начать игру» приложение перейдет в сцену создания персонажа. После создания персонажа запустится сцена игрового уровня.

Сцена игрового уровня позволяет игроку перейти в меню паузы нажатием кнопки «Esc» на клавиатуре. В этом меню есть кнопка «Продолжить», закрывающее меню паузы и кнопка «Выйти», которая выключит приложение.

3.1. Диаграмма классов.

Одной из главных архитектурных задач во время проектирования игрового приложения на платформе Unity является решение проблемы внедрения зависимостей. Различные объекты должны взаимодействовать друг с другом и передавать сообщения, для этого нужно определять зависимость одних объектов от других.

Основными паттернами в проектировании, решающими эту проблему, являются: Одиночка (Singleton), Сервис локатор (Service Locator) и Внедрение зависимостей (Dependency Injection) [5].

Для проектирования данной системы было решено использовать паттерн «Сервис локатор», он является частным случаем паттерна «Одиночка». Преимущество его заключается в том, что он представляет собой куда более централизованную систему, в отличии от использования нескольких реализаций паттерна «Одиночка», а также предполагает получение зависимостей через интерфейс. В то же время «Сервис локатор» реализуется значительно проще, чем «Внедрение зависимостей».

«Сервис локатор» представляет собой класс, который имеет только один статичный экземпляр в системе, а также предоставляет глобальную точку доступа к этому экземпляру. Он хранит ссылки на объекты классов сервисов, функционалом которых пользуются остальные объекты системы [6].

На рисунке 5 представлена диаграмма классов реализации паттерна сервис локатор.

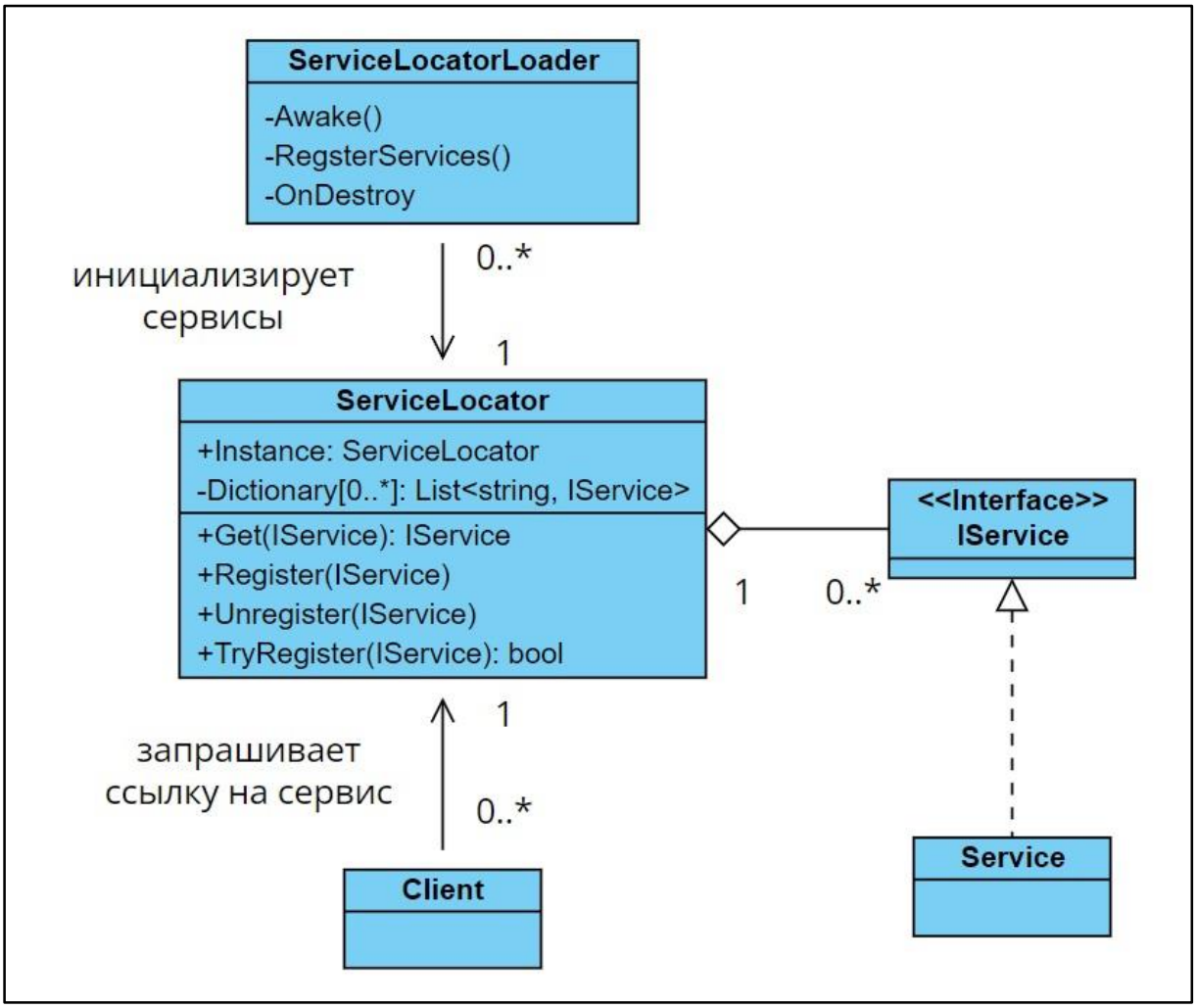

Рисунок 5 – Диаграмма классов реализации паттерна сервис локатор

`ServiceLocatorLoader` является классом, которой инициализирует классы сервисы в `ServiceLocator`. Класс `ServiceLocator` является реализацией паттерна, поле Instance является единственным публичным статическим объектом этого же класса, что позволяет обращаться к нему из любой точки программы. Методы `Register` и `TryRegister` регистрируют сервисы,

`Unregister` удаляет ссылку на класс сервис из словаря, а метод `Get` возвращает ссылку на объект класса сервиса.

Классы `Client` и `Service` в диаграме являются примером. На месте `Client` может находится любой другой класс системы, если тот будет пользоваться методами класса `ServiceLocator`. `Service` же представляет собой класс сервис, который реализует интерфейс `IService`. Описание нужных для системы классов сервисов представлено далее.

1. `UnitActionSystem` представляет собой систему контроля действий, выбираемых игроком. Содержит набор методов для выбора, выполнения, ожидания и завершения действий персонажа игрока.

2. `GridSystemVisual` отвечает за отображение игрового поля. Содержит набор методов для изменения цвета конкретных ячеек, а также их отключения.

3. `SceneGrid` реализует игровое поле. Содержит набор методов для получения информации об игровом поле и объектах, расположенных на нем.

4. `Pathfinding` реализует систему поиска наикратчайшего пути от одной клетки до другой.

5. `TurnSystem` реализует систему ходов, содержит метод для смены хода.

6. `UnitManager` контроллирует активных персонажей на сцене. Содержит набор методов для получения списка активных персонажей.

7. `GameStateController` контроллирует текущее состоянии игры, находится ли она на паузе.

8. `AudioManager` является реализацией звуковой системы в игре, отвечает за проигрывание звуков игры, пользовательского интерфейса и музыки.

9. `SceneController` отвечает за переключение между сценами игры.

3.2. Макеты пользовательского интерфейса.

Для игрового приложения было разработано 2 макета пользовательского интерфейса. Первый показывает вид интерфейса во время игрового процесса, он продемонстрирован на рисунке 6.

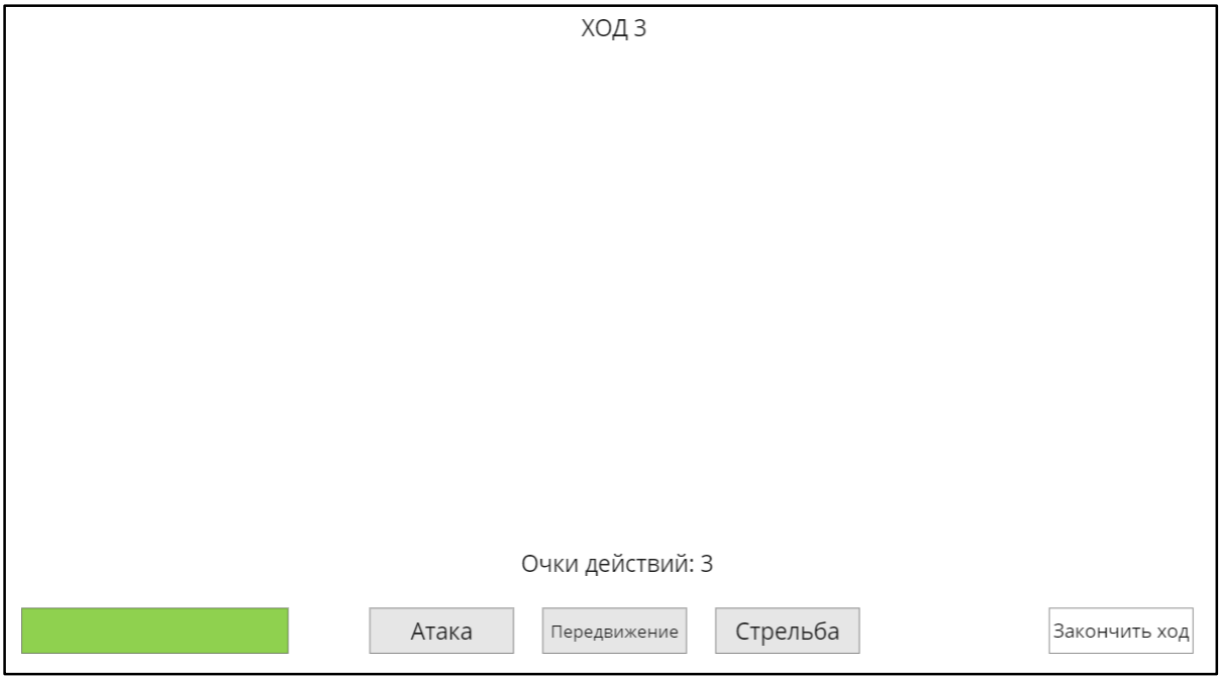

Рисунок 6 – Макет интерфейса игрового процесса

Вверху по центру расположен счетчик ходов, который показывает номер текущего хода. В нижнем левом углу расположена шкала здоровья, котороя уменьшается при уменьшении количества очков здоровья у персонажа. В центре расположены кнопки действий, при нажатии на них игрок сможет совершить соответствующие действия. Над ними расположен счетчик оставшихся действий за ход. В левом нижнем углу расположена кнопка завершения хода.

Следующий макет используется для представления пользовательского интерфейся окна создания персонажа, на котором игрок будет распределять характеристики для своего персонажа. Макет представлен на рисунке 7.

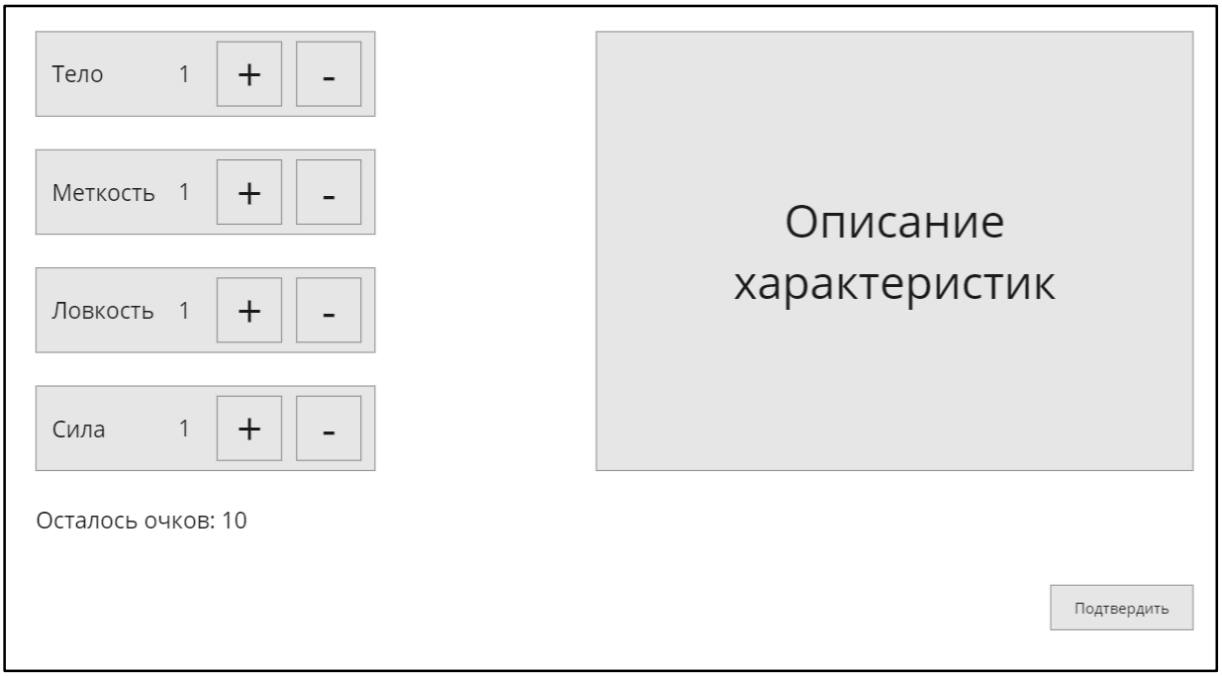

Рисунок 7 – Макет интерфейса создания персонажа

В левой части экрана расположены характеристики, у каждой есть кнопка увеличения и у меньшения, а также числовое значение. Под списком написано количество оставшихся очков для распределения. В правой части экрана расположено описание характеристик, которое будет появлятся при наведении на одну из характеристик. В правом нижнем углу расположена кнопка, подтверждающее создание персонажа. Закончить создание персонажа можно будет только тогда, когда все очки распределены по характеристикам.

Вывод по третьей главе

В третьей главе была спроектирована архетиктура приложения, а также приведено ее краткое описание. Для описания структурного аспекта системы была построена диаграмма компонентов.

4. РЕАЛИЗАЦИЯ

4.1. Реализация паттерна «Сервис локатор»

Для реализации паттерна созданы классы `ServiceLocatorLoader`, `ServiceLocator`, а также интерфейс `IService`.

`ServiceLocatorLoader` отвечает за инициализацию всех классов сервисов. Он содержит сериализованые поля для каждого сервиса, это позволяет устанавливать нужные экземпляры классов сервисов в редакторе Unity. Компонент `ServiceLocatorLoader` продемонстрирован на рисунке 8.

Рисунок 8 – Компонент `ServiceLocatorLoader`

Класс `ServiceLocator` имеет публичное статическое поле, хранящее ссылку на единственный экземпляр, который инициализируется в методе `Initialize`. Также `ServiceLocator` имеет словарь, хранящий ссылки на объекты классов сервисов `services`. Для регистрации сервиса используется метод `Register`, который добавляет в словарь ссылку на передаваемый объект. Класс `ServiceLocator` представлен в листинге 1.

Листинг 1 – Класс `ServiceLocator`

```
public class ServiceLocator
{
    public static ServiceLocator Instance { get; private set; }
    private ServiceLocator()
    public static void Initialize()
    {
        if (Instance != null)
        {
            Debug.LogError("ServiceLocator Instance already exists - " +
Instance);
```

```
            return;
        }
        Instance = new ServiceLocator();
    }
    private readonly Dictionary<string, IService> services = new
Dictionary<string, IService>();
    public T Get<T>() where T : IService
    {
        string key = typeof(T).Name;
        if (!services.ContainsKey(key))
        {
            Debug.LogError($"{key} is not registered.");
            throw new InvalidOperationException();
        }
        return (T)services[key];
    }
    public void Register<T>(T service) where T : IService
    {
        string key = typeof(T).Name;
        if (services.ContainsKey(key))
        {
            Debug.LogError(
                $"Attempted to register service of type {key} which is
already registered.");
            return;
        }
        services.Add(key, service);
    }
    public void Unregister<T>() where T : IService
    {
        string key = typeof(T).Name;
        if (!services.ContainsKey(key))
        {
            Debug.LogError(
                $"Attempted to unregister service of type {key} which is
not registered/");
            return;
        }
        services.Remove(key);
    }
}
```

`ServiceLocator` взаимодействует только с пустым интерфейсом `IService`, который использует как обобщение для методов получения и регистрации сервисов [7]. Таким образом любой пользователь сервиса способен запросить у экземпляра `ServiceLocator` ссылку на экземпляр конкретного класса.

4.2. Реализация системы ходов

Боевая система в игре представляет собой пошаговое сражение, в процессе которого игрок должен победить противников. Каждый ход у игрока имеется набор действий, за использование которого тратятся очки действий.

Когда очки действий заканчиваются, игрок больше не может совершать действия и должен закончить ход.

Для реализации пошагового боя был реализован класс `TurnSystem`, который нужен для контроля ходов. Данный класс является сервисом [8]. Реализация интерфейса `IService` позволит классам, которые пользуются функционалом системы ходов получать данные, когда им это нужно. Реализация системы пошагового боя представлена в листинге 1 приложения.

Для того, чтобы игровые объекты могли получать информацию о смене хода, было создано событие `OnTurnChanged`, которое вызывается во время смены хода [9].

Также был реализован класс `TurnSystemUI`, отвечающий за поведение пользовательского интерфейса системы ходов. Вверху игрового окна расположен счетчик, показывающий номер идущего хода, а также кнопка «Завершить ход», которая завершает ход игрока.

4.3. Реализация ролевой системы

Ролевая система контролирует способность и эффективность персонажей в совершении того или иного действия. Для создание ролевой системы был реализован класс `CharacterStats` с вложенным классом `Stat`.

Класс `Stat` представляет собой сущность характеристики, он хранит ее значение, на основе которого будет определяться эффективность того или иного действия, а также ее тип. Класс `Stat` представлен в листинге 2.

Листинг 2 – Класс `Stat`

```
[Serializable]
public class Stat
{
    public enum Type
    {
        Body,
        Power,
        Dexterity,
        Aim
    }
    [SerializeField] private int value;
    [SerializeField] private Type type;
    public int GetValue()
    {
```

```
        return value;
    }
    public void SetValue(int value)
    {
        this.value = value;
    }
    public new Type GetType()
    {
        return type;
    }

}
```

Класс `CharacterStats` определяет характеристики персонажа, он хранит список объектов вложенного класса `Stats`, а также имеет методы `GetStatValue`, который возвращает значение характеристики по заданному типу, а также `GetStatRoll`, который производит проверку характеристики, путем суммирования случайно сгенерированного числа в диапазоне от 1 до 10 и значения характеристики. Класс `CharacterStats` представлен на листинге 3.

Листинг 3 – Класс `CharacterStats`

```
public class CharacterStats : MonoBehaviour
{
    [SerializeField] private List<Stat> stats;
    public void SetCharacterStats(int bodyValue, int powerValue, int
dexterityValue, int aimValue)
    {
        foreach (Stat stat in stats)
        {
            switch (stat.GetType())
            {
                case Stat.Type.Body:
                    {
                        stat.SetValue(bodyValue);
                        break;
                    }
                case Stat.Type.Power:
                    {
                        stat.SetValue(powerValue);
                        break;
                    }
                case Stat.Type.Dexterity:
                    {
                        stat.SetValue(dexterityValue);
                        break;
                    }
                case Stat.Type.Aim:
                    {
                        stat.SetValue(aimValue);
                        break;
                    }
            }
        }
```

```
    }
    public int GetStatValue(Stat.Type type)
    {
        foreach (Stat stat in stats)
        {
            if (stat.GetType() == type) return stat.GetValue();
        }
        Debug.LogError($"There is no stat with type {type}");
        return 0;
    }
    public int GetStatRoll(Stat.Type type)
    {
        foreach (Stat stat in stats)
        {
            if (stat.GetType() == type)
                return UnityEngine.Random.Range(1, 10) + stat.GetValue();
        }
        Debug.LogError($"There is no stat with type {type}");
        return 0;
    }
}
```

Оба класса сериализуются в движке Unity таким образом, чтобы в окне редактора можно было задавать все значения характеристик для конкретного персонажа, что позволяет создавать различные вариации персонажей с разными характеристиками и сохранять их, как префабы [10]. Префабом называется шаблон игрового объекта в движке Unity, который хранит все компоненты, параметры и свойства объекта [11].

Пример распределения характеристик в редакторе представлен на рисунке 9.

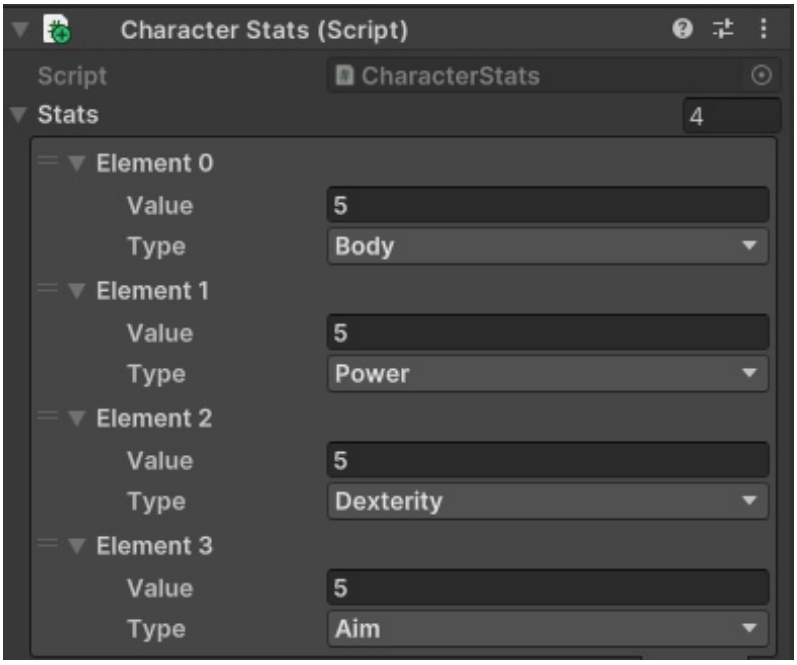

Рисунок 9 – Распределение характеристик в редакторе

Всего было создано 3 префаба противников. С упором на ближний бой, стрельбу и универсальный. Значения характеристик противников представлены в таблице 1.

Таблица 1 – Значение характеристик противников

Название префаба	Сила	Ловкость	Меткость	Тело
UnitEnemy	2	2	2	2
UnitEnemy_Accurate	2	3	2	2
UnitEnemy_Strong	3	2	3	1

4.4. Реализация поиска пути

Для грамотного передвижения персонажей по игровому полю был реализован алгоритм поиска наикратчайшего пути из одной ячейки поля в другую.

Для реализации поиска пути был выбран A* алгоритм [12]. Он работает на основе оценки стоимости пути до точки назначения. Эта стоимость вычисляется из суммы значений стоимости достижения рассматриваемой вершины из начальной и стоимости эвристической оценки расстояния от рассматриваемой вершины к конечной.

Алгоритм последовательно проходит по клеткам игрового поля от той, где находится персонаж игрока до точки назначения до тех пор, пока не дойдет до нее. После чего алгоритм вернет путь из клеток с наименьшей стоимостью пути.

Значение стоимости просчитывается из вектора расстояния начальной и конечной точки. Для передвижения от одной позиции до другой по прямой установлена стоимость, равная 10, по диагонали – 14.

Просчет оценки стоимости пути представлен на листинге 4.

Листинг 4 – Метод `CalculateCost`

```
public int CalculateCost(GridPosition startPosition, GridPosition
endPosition)
    {
        GridPosition distance = startPosition - endPosition;
        int xDiff = Mathf.Abs(distance.x);
        int zDiff = Mathf.Abs(distance.z);
        int remaining = Mathf.Abs(xDiff - zDiff);
        return DIAGONAL_COST * Mathf.Min(xDiff, zDiff) + STRAIGHT_COST *
remaining;
    }
```

Пример работы реализованного алгоритма в игровом приложении продемонстрирован на рисунке 10.

Рисунок 10 – Пример работы A* алгоритма в игре.

4.5. Реализация искусственного интеллекта противников

Так как персонаж игрока и противники представляют собой одну сущность, было решено использовать для их реализации один класс `Unit`. Он содержит методы, контролирующие действия, совершаемые персонажем, а также имеет параметр `isEnemy`, который определяет, является ли персонаж подконтрольным игроку или системе.

Для контроля над персонажами, был создан класс `UnitManager`, которой хранит список из всех персонажей на уровне. При появлении на сцене нового персонажа или его смерти вызывается событие в классе `Unit`, на которое подписан `UnitManager`. Таким образом класс хранит актуальную информацию о персонажах на сцене. Реализация системы контроля персонажей представлена в листинге 2 приложения.

Для реализации искусственного интеллекта противников было решено использовать систему, основанную на полезности [13].

В данной системе у каждого агента есть набор возможных действий. Совершаемое действие выбирается на основе значения полезности этого действия, которое устанавливается в некотором диапазоне в зависимости от окружающих факторов.

Плюсы данной системы заключаются в простоте определения поведения противников и легкой расширяемости их возможностей, при создании новых действий. Изменения одной только величины параметров полезности может поменять тактику поведения противника. Это также позволяет создать систему, при которой персонажи с разными характеристиками будут вести себя наиболее эффективно, в зависимости от величины этих характеристик.

Параметры полезности, для возможных действий противников, предоставлены в таблице 2.

Таблица 2 – Величина полезности действий

Действие	**Вычисление полезности**
ShootAction	20, умноженное на значение характеристики «Меткость», если есть доступная цель.
MoveAction	Количество возможных для атаки целей на ячейке, умноженное на 10 и на значение характеристики «Тело».
AttackAction	20, умноженное на значение характеристики «Сила».

Таким образом, противник будет атаковать доступную цель, если у него есть такая возможность. А передвигаться он будет в ту ячейку поля, при занятии которой, у него будет как можно больше доступных целей.

Чем выше будет значение «Силы» и «Тела» противника, тем более вероятней, что он будет атаковать в ближнем бою. Если же он будет более метким, то и больше вероятность, что он будет атаковать издалека. Это позволяет создавать разные типы противников с разными характеристиками, не меняя систему поведения.

Для реализации данной системы был создан класс `EnemyAI`, контролирующий поведение противников, а также `EnemyAIAction`, хранящий значение полезности действия и координату ячейки игрового поля, на котором будет совершаться действие.

4.6. Реализация боевой системы

Для создания возможности атаки одних персонажей другими, был написан класс `ShootAction`. Игроку представляется на выбор для атаки позиции, на которых находятся противники, в пределах определенной дистанции.

Для реализации нанесения урона персонажам был написан класс `HealthSystem`, хранящий количество единиц здоровья. Когда здоровье персонажа опускается ниже нуля, вызывается событие `OnDead`. Реализация нанесения урона представлена в листинге 5.

Листинг 5 – Класс `HealthSystem`

```
public class HealthSystem : MonoBehaviour
{
    public event EventHandler OnDead;

    [SerializeField] private int health = 10;

    public void Damage(int damageAmount)
    {
        health -= damageAmount;

        if (health <= 0)
        {
            Die();
        }
        Debug.Log(health);
    }

    private void Die()
    {
        OnDead?.Invoke(this, EventArgs.Empty);
    }
```

4.7. Реализация системы звуков

Для воспроизведения звуков в Unity используется компонент Audio Source [14, 15]. Для возможности воспроизведения разных звуков для разных целей, был написан класс `Sound`, который хранит ссылку на файл с

определенным звуком, а также его названия. Были созданы три объекта с компонентами Audio Source для звуков пользовательского интерфейса, звуковых эффектов и музыки. Компонент `AudioManager` представлен на рисунке 11.

Рисунок 11 – Компонент `AudioManager` в редакторе

За проигрывание звуков отвечает класс `AudioManager`, который имеет методы `PlayUISound`, `PlaySFX`, `PlayEnvSound`, каждый из которых отвечает за отдельный тип проигрываемых звуков. Так как `AudioManager` является классом сервисом, к нему может обратиться любой объект, чтобы в нужный момент прозвучал тот или иной звук. Метод `PlayUISound` представлен в листинге 6. Методы `PlaySFX` и `PlayEnvSound` реализованы аналогичным образом.

Листинг 6 – Метод `PlayUISound`

```
public void PlayUISound(string name)
    {
        Sound sound = uiSounds.Find(x => x.Name == name);

        if (sound != null)
        {
            UISource.clip = sound.Clip;
            UISource.Play();
        }
        else
        {
            Debug.LogError("Sound not found");
        }
    }
```

4.8. Реализация сцен и пользовательского интерфейса

Игровое приложение состоит из трех сцен: MainMenu, CharacterCreation и Level.

На сцене MainMenu расположено главное меню, в нем представлены кнопка начала игры и выхода из игры.

Сцена CharacterCreation представляет собой окно создания персонажа. На ней расположены характеристики персонажа, их описание, которое отображается при наведении мышкой на характеристику, а также кнопки для увеличения или уменьшения характеристик. Всего на распределение дается 21 очко характеристик, их количество также указывается на экране. Создать персонажа можно только тогда, когда все очки распределены по характеристикам. Также существует ограничение, из-за которого значение любой характеристики находится в диапазоне от 1 до 9.

После создания персонажа и нажатия кнопки «Подтвердить», значения характеристик персонажа сериализуются в `PlayerPrefs` [16]. После чего загружается сцена Level, в котором и происходит основной игровой процесс.

При нажатии игроком клавиши «Escape», откроется меню паузы, а игровой процесс приостановится. Если персонаж игрока погибнет, откроется меню проигрыша, из которого можно перезапустить уровень и попробовать сыграть заново.

4.9. Реализация графики игры

Для реализации графики использовались бесплатные ассеты с Unity Asset Store. Среди них RPG/FPS Game Assets for PC/Mobile (Industrial Set v3.0) для создания окружения, Low Poly Soliders Demo для персонажей. Скриншот игры представлен на рисунке 12.

Рисунок 12 – Скриншот внешнего вида игры

Вывод по четвертой главе

В четвертой главе была описана реализация основных компонентов системы и ее архитектуры в соответствии с поставленными к системе требованиями. Всего в процессе разработки было реализовано 47 классов и написано 7484 строки кода.

5. ТЕСТИРОВАНИЕ

5.1. Функциональное тестирование

В ходе функционального тестировании система проверялась на соответствие представленным функциональным требованиям. Результат тестирования игровых механик представлен в таблице 3, результат тестирования пользовательского интерфейса представлен в таблице 4.

Таблица 3 – Тестирование игровых механик

№	Название	Действия	Ожидаемый результат	Тест пройден?
1	Построение игровой сетки	Запуск сцены игрового процесса.	Появление игровой сетки.	Да
2	Ход игрока	Нажатие на кнопку «Завершить ход».	Ход перейдет компьютеру, в течении которого игрок не сможет выполнять действия	Да
3	Ход противников	Нажатие на кнопку «Завершить ход».	Противники на уровне поочередно совершают действия, после чего ход возвращается к игроку.	Да
4	Стрельба	Нажатие на кнопку Стрельба, нажатие на противника.	Игровая сетка выделит доступную цель, персонаж произведет выстрел в цель.	Да
5	Ближний бой	Нажатие на кнопку Атака, нажатие на противника.	Игровая сетка выделит доступную цель, персонаж произведет атаку ближнего боя.	Да
6	Движение	Нажатие на кнопку Движение, нажатие на доступную для передвижения клетку.	Игровая сетка выделит доступные для передвижения клетки, персонаж дойдет до этой клетки.	Да
7	Проигрыш	Потеря всех очков здоровья.	Откроется окно проигрыша.	Да
8	Поиск пути	Нажатие на кнопку Движение, нажатие на доступную для передвижения клетку, находящуюся за преградой.	Персонаж корректно обходит препятствия на пути к цели самым коротким путем.	Да
9	Искусственный интеллект	Завершение хода игроком.	Противники корректно себя ведут при встрече с игроком, приближаются к нему и атакуют.	Да
10	Управление камерой	Нажатие клавиш, отвечающих за изменение положения камеры.	Камера корректно изменяет свое положение, в соответствии с нажимаемыми клавишами.	Да

Таблица 4 – Тестирование интерфейса.

№	Название	Действия	Ожидаемый результат	Тест пройден?
1	Начало игры	Нажать на кнопку «Начать игру» в главном меню.	Откроется окно создание персонажа.	Да
2	Изменение характеристик	Нажатие на кнопку «+» или «-» у одной из характеристик.	Характеристика изменит свое значение на 1, в соответствии с нажатой кнопкой. Изменится оставшееся количество очков.	Да
3	Ограничение на характеристику	Нажатие на кнопку «+», для характеристики, значение которой равно 9.	Значение характеристики не изменится.	Да
4	Создание персонажа	Нажатие на кнопку «Подтвердить» в окне создания персонажа.	Загрузится сцена игрового процесса.	Да
5	Меню паузы	Нажатие на кнопку «Escape» во время игрового процесса.	Игра приостановится, откроется меню паузы.	Да
6	Полоска здоровья	Получение урона.	Полоска здоровья уменьшится в соответствии с полученным уроном.	Да
7	Очки действий	Использование действий.	Количество очков действий изменится.	Да
8	Интерфейс действий	Нажать на одну из кнопок действий.	Кнопка подсветится, игровая сетка покажет доступные позиции.	Да
9	Смена хода	Нажать на кнопку «Закончить ход».	Счетчик ходов увеличится, количество очков действий обновится к началу следующего хода.	Да
10	Выход из игры	Нажать на кнопку «Выйти».	Игровое приложение завершает свою работу.	Да
11	Перезапуск уровня	Нажать на кнопку «Начать заново» после поражения.	Игровое приложение перезапускает сцену игрового процесса.	Да

5.2. Юзабилити тестирование

В юзабилити тестировании участвовало 7 человек, тестирование проводилось единожды. Юзабилити тестирование нужно для проверки работоспособности игровых механик, правильного определения сложности игры и

поиска незамеченных ошибок и недочетов во время функционального тестирования. Далее проведен список изменений, примененных в ходе юзабилити тестирования.

1. В некоторых местах игрового уровня строилась сетка внутри игровых объектов, из-за чего игрок мог проходить сквозь них, а также выходить за пределы уровня. Решением проблемы стало добавление компонента MeshCollider на подобные объекты.

2. При попытке увеличить характеристику выше 9 уменьшалось общее количество очков на распределение. Проблема была исправлена.

3. Противники представляли слишком мало угрозы, всвязи с чем, их характеристики были немного увеличены.

4. Персонажи имели всего 2 очка действия за ход, что показалось недостаточным большенству игроков. Количество очков действий было увеличено до трех.

5. Персонажи слишком долго передвигались по игровому полю. Скорость перемещения персонажей была увеличена.

6. Урон в ближнем бою зависил от характеристики «Меткость», а не характеристики «Сила», как изначально было задумано. Проблема была исправлена.

Вывод по пятой главе

В пятой главе было описано тестирование разработанной системы. В процессе тестирования были исправлены ошибки и недочеты системы.

ЗАКЛЮЧЕНИЕ

В рамках данной работы было разработано игровое приложение в жанре «Tactical RPG» на платформе Unity.

В процессе разработки были решены следующие задачи.

1. Произведен анализ предметной деятельности и аналогичных проектов.

2. Произведен анализ функциональных и нефункциональных требований к разрабатываемому игровому приложению.

3. Спроектирована архитектура системы, описан применяющийся паттерн, визуализированы макеты интерфейса.

4. Реализована спроектированная архитектура игрового приложения, в соответствии с поставленными требованиями.

5. Протестировано приложение с последующим исправлением выявленных ошибок.

В ходе проделанной работы были получены теоретические и практические знания в разработке игр на платформе Unity.

Дальнейшим направлением развития проекта будет доработка существующих механик и увеличение количества игрового контента, с последующей публикацией проекта на платформе itch.io в рамках личного портфолио.

ЛИТЕРАТУРА

1. Игровая индустрия за 2020 год в цифрах – инфографика от GamesIndustry.biz. [Электронный ресурс] URL: https://dtf.ru/gameindustry/294571-igrovaya-industriya-za-2020-god-v-cifrah-infografika-ot-gamesindustrybiz (дата обращения: 09.02.2024 г.).

2. Финансовый успех Hogwarts Legacy и большое внимание к Starfield – игровая индустрия за 2023 год в цифрах. [Электронный ресурс] URL: https://dtf.ru/gameindustry/294571-igrovaya-industriya-za-2020-god-v-c (дата обращения: 09.02.2024 г.).

3. Усков М. А. Обзор преимуществ и недостатков игровых движков. Обоснование выбора инструментов и технологий разработки клиентской части игровых приложений. // Глобус: технические науки. – 2020. – №5 – (36). – С. 6–10.

4. Карелова Р.А., Коробейников П.С. Контроль над проектом на Unity: частые проблемы начинающих разработчиков и пути их решения. // МНИЖ. – 2020. – №5–1 – (95). – С. 40–45.

5. Уваров А.Н. Инверсия управления и внедрение зависимостей // Символ науки. – 2016. – №10 – 1 – С. 28–32.

6. Implementing Service Locator in Unity. [Электронный ресурс] URL: https://briantria.com/ioc-service-locator-unity/ (дата обращения: 16.03.2024 г.).

7. Универсальные классы и методы. [Электронный ресурс] URL: https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/generics (дата обращения: 16.03.2024 г.).

8. Кошелев О.В. Шаблоны проектирования в программировании. // Научный журнал. – 2018. – №4 (27) – С. 43–49.

9. Система событий (EventSystem). [Электронный ресурс] URL: https://docs.unity.cn/ru/2018.4/Manual/EventSystem.html (дата обращения: 19.03.2024 г.).

10. Serializable. [Электронный ресурс] URL: https://docs.unity3d.com/ScriptReference/Serializable.html (дата обращения: 19.03.2024 г.).

11. Префабы (Prefabs). [Электронный ресурс] URL: https://docs.unity3d.com/ru/530/Manual/Prefabs.html (дата обращения: 19.03.2024 г.).

12. Авдеев В.С. Метод поиска оптимального маршрута на графе // Альманах «Крым». – 2022. – №30. – С. 19–28.

13. Донских А.К., Барабанов В.Ф., Гребенникова Н.И., Белых М.А. Обзор архитектуры систем управления интеллектом на основе полезности и дерева поведения. // Вестник ВГТУ. – 2021. – №2. – С. 36–41.

14. Introduction to components. [Электронный ресурс] URL: https://docs.unity3d.com/Manual/Components.html (дата обращения: 19.03.2024 г.).

15. Audio Source. [Электронный ресурс] URL: https://docs.unity3d.com/Manual/class-AudioSource.html (дата обращения: 19.03.2024 г.).

16. PlayerPrefs. [Электронный ресурс] URL: https://docs.unity3d.com/ScriptReference/PlayerPrefs.html (дата обращения: 19.03.2024 г.).

ПРИЛОЖЕНИЕ. Система ходов и контроля персонажей

В листинге 1 представлена реализация системы ходов.

Листинг 2 – Система ходов

```
public class TurnSystem : MonoBehaviour
{
    public static TurnSystem Instance { get; private set; }

    public event EventHandler OnTurnChanged;

    private int turnNumber = 1;
    private bool isPlayerTurn = true;

    private void Awake()
    {
        if (Instance != null)
        {
            Debug.LogError("There's more than one TurnSystem " + transform
+ " - " + Instance);
            Destroy(gameObject);
            return;
        }

        Instance = this;
    }

    public void NextTurn()
    {
        turnNumber++;
        isPlayerTurn = !isPlayerTurn;

        OnTurnChanged?.Invoke(this, EventArgs.Empty);
    }

    public int GetTurnNumber()
    {
        return turnNumber;
    }

    public bool IsPlayerTurn()
    {
        return isPlayerTurn;
    }
}
```

В листинге 2 представлена реализация система контроля персонажей.

Листинг 2 – Система контроля персонажей

```
public class UnitManager : MonoBehaviour
{
    public static UnitManager Instance { get; private set; }
    private List<Unit> unitList;
    private List<Unit> friendlyUnitList;
    private List<Unit> enemyUnitList;

    private void Awake()
     {
        if (Instance != null)
        {
```

```
            Debug.LogError("There's more than one UnitManager " + transform
+ " - " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;
        unitList = new List<Unit>();
        friendlyUnitList  = new List<Unit>();
        enemyUnitList = new List<Unit>();
    }
    private void Start()
    {
        Unit.OnAnyUnitSpawned += Unit_OnAnyUnitSpawned;
        Unit.OnAnyUnitDead += Unit_OnAnyUnitDead;
    }

    private void Unit_OnAnyUnitSpawned(object sender, EventArgs e)
    {
        Unit unit = sender as Unit;
        Debug.Log(unit + " spawned");
        unitList.Add(unit);
        if (unit.IsEnemy())
        {
            enemyUnitList.Add(unit);
        }
        else
        {
            friendlyUnitList.Add(unit);
        }
     }
    private void Unit_OnAnyUnitDead(object sender, EventArgs e)
    {
        Unit unit = sender as Unit;

        unitList.Remove(unit);
        if (unit.IsEnemy())
        {
            enemyUnitList.Remove(unit);
        }
        else
        {
            friendlyUnitList.Remove(unit);
        }
    }

    public List<Unit> GetUnitList()
    {
        return unitList;
    }
    public List<Unit> GetFriendlyUnitList()
    {
        return friendlyUnitList;
    }
    public List<Unit> GetEnemyUnitList()
    {
        return enemyUnitList;
    }
```