МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет	*Кафедра*
вычислительной техники	«Вычислительная техника»

Направление подготовки: 09.03.01 «Информатика и вычислительная техника»

Профиль подготовки: «Программное обеспечение средств вычислительной техники и автоматизированных систем»

БАКАЛАВРСКАЯ РАБОТА

на тему

ПРИЛОЖЕНИЕ НА UNITY ДЛЯ ИНТЕРАКТИВНОГО ИССЛЕДОВАНИЯ АРХИТЕКТУРЫ КОМПЬЮТЕРА

Студент 04.06.2025 Алёшина Анна Васильевна
(подпись, дата) (ФИО полностью)

Руководитель 04.06.25 Токарев А.Н.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр 04.06.25 Бычков А.С.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 28.05.2025 г. № 16)

Заведующий кафедрой _____ М.А. Митрохин
(подпись) (фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 11 июня 2025 № 25)

Секретарь ГЭК _____ Е.И. Гурин
(подпись) (фамилия, инициалы)

Пенза 2025

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет	*Кафедра*
вычислительной техники	«Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
_____М.А. Митрохин
«06» 05 2025 г.

ЗАДАНИЕ
НА БАКАЛАВРСКУЮ РАБОТУ

1. Студент Алёшина Анна Васильевна гр.21ВВП1 факультета вычислительной техники.

2.Тема работы Приложение на Unity для интерактивного исследования архитектуры компьютера.

Тема утверждена приказом ПГУ № 601/0 от " 06 " 05 2025 г.

3. Руководитель бакалаврской работы к.т.н., доцент каф. ВТ ПГУ Токарев А.Н.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

В ходе выполнения выпускной квалификационной работы необходимо разработать приложение на Unity для интерактивного исследования архитектуры компьютера, отвечающий всем требованиям функциональности, дизайна и структуры современного игрового приложения. Главным требованием является понятный интерфейс. Красивый внешний вид. Написание оптимизированных скриптов для игры.

5. Перечень подлежащих разработке вопросов

Требуется произвести анализ предметных областей: изучить существующие игры на Unity про архитектуру компьютера, выделить их достоинства и недостатки. Выбрать средства для разработки, которые подойдут для создания игры. Написать геймплей для игры. Выбрать вспомогательные средства разработки.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Изучение предметной области	32	26.04.2025 – 28.05.2025	
2	Анализ требований	32	29.04.2025 – 30.05.2025	
3	Создание спрайтов, анимаций, графики для игры	30	1.04.2025 – 3.05.2025	
4	Выбор технологий	40	4.04.2025 – 8.05.2025	
5	Реализация скриптов для объектов и персонажей	66	9.04.2025 – 15.05.2025	
6	Разработка логики игры	70	16.04.2025 – 23.05.2025	
7	Оформление пояснительной записки	54	24.04.2025 – 13.05.2025	

Дата выдачи задания "06" 05 2025 г.

Руководитель бакалаврской работы 06.05.25 Токарев А.Н.
(подпись, дата)

Задание к исполнению принял студент 06.05.25 Алёшина А.В.
(подпись, дата)

Работу к защите допустить
Декан факультета 09.06.2025 Фионова Л.Р.
(подпись, дата)

Реферат

Бакалаврская работа содержит 178 листа, 34 рисунков, 7 источников и 3 приложения.

ПРИЛОЖЕНИЕ НА UNITY ДЛЯ ИНТЕРАКТИВНОГО ИССЛЕДОВАНИЯ АРХИТЕКТУРЫ КОМПЬЮТЕРА

Цель работы - разработать в Unity приложение, позволяющее в интерактивной форме изучать архитектуру компьютера.

Для достижения поставленной цели было разработано приложение, отвечающее следующим поставленным задачам: реализация базового передвижения персонажа, разработка анимаций героя, врагов и окружения, создание игровых уровней, разработка системы загадок, реализация механики боя, разработка системы повреждений и здоровья, реализация системы сохранений, реализация пользовательского интерфейса, разработка системы диалогов, разработка звукового сопровождения (SFX и музыка) и его настройка.

В результате получилась игра на Unity, написанная на языке C#, дающая возможность интерактивно исследовать устройство компьютера.

Содержание

Перечень сокращений и обозначений 4
Введение 5
1 Приложение на Unity для интерактивного исследования архитектуры компьютера 6
1.1 Анализ предметной области 6
1.1.1 Описание потребности пользователей 6
1.1.2 Анализ существующих аналогов 7
1.1.3 Постановка задачи 11
1.2 Выбор инструментальных средств 13
1.2.1 Среда разработки Unity 13
1.2.2 Среда разработки кода Visual Studio. Язык программирования C# 15
1.2.3 Графический редактор Paint 3D 17
1.2.4 Генератор фото, видео, музыки YesChat AI 20
1.2.5 Видео редактор Adobe Premiere Pro 22
1.3 Архитектура приложения 24
1.4 Реализация 30
1.4.1 Интерфейс Unity. 30
1.4.2 Создание объекта главного героя в Unity 35
1.4.3 Создание логики управления главным героем 38
1.4.4 Создание новой силы героя 48
1.4.5 Создание босса игры 52
1.4.6 Система уровней 60
1.5 Тестирование 65
2 Безопасность жизнедеятельности 71
Заключение 73
Список используемых источников 74
Приложение А – UML диаграммы 75
Приложение Б – Листинг программы 82
Приложение В - Презентация 181

Перечень сокращений и обозначений

В настоящей работе применяют следующие сокращения и обозначения

ПК - Персональный компьютер

2D (Two-dimensional) - компьютерная двухмерная графика

3D (Three-dimensional) - компьютерная трёхмерная графика

IDE (Integrated Development Enviroment) - интегрированная среда разработки

LCD (Liquid Crystal Display) - жидкокристаллический дисплей, тип плоскопанельного экрана, в котором используется технология модуляции света жидкими кристаллами

LED (Light Emitting Diode) - светоизлучающий диод, полупроводниковый источник света, излучающий свет при прохождении через него электрического тока.

SFX (Sound Effects) - Звуковые эффекты

UI (User Interface) - пользовательский интерфейс

UML (Unified Modeling Language) - унифицированный язык моделирования.

Введение

В индустрии развлечений, особенно в разработке видеоигр и образовательных программ, активно используются технологии, изначально предназначенные для повышения продуктивности и решения задач. В современном мире, где информация льется потоком, а удержать внимание становится все сложнее традиционные методы обучения часто оказываются неэффективными. Именно поэтому образовательные игры приобретают все большую популярность и важность. Игры, в отличие от сухих лекций и учебников, способны захватить внимание, вовлечь в активное взаимодействие и, что самое главное, превратить процесс обучения в увлекательное приключение. Они создают безопасную и мотивирующую среду для экспериментов, где эксперименты, ошибки и получение практического опыта не только допустимы, но и приветствуются. Это позволяет учащимся глубже погружаться в предмет, усваивать информацию более осознанно и прочно, а также развивать критическое мышление и навыки решения проблем.

Именно поэтому для облегчения понимания основ компьютерной архитектуры было решено создать интерактивную игру. активное взаимодействие с игровым миром, решение головоломок и выполнение задач в увлекательном формате позволит пользователям не просто запомнить информацию, но и глубоко понять принципы работы компьютерных систем. В конечном счете, цель данной разработки - доказать, что обучение может быть не только полезным, но и захватывающим, превращая сложное в простое, а скучное – в интересное. Игра – это не просто развлечение, а мощный инструмент обучения.

1 Приложение на Unity для интерактивного исследования архитектуры компьютера

1.1 Анализ предметной области

1.1.1 Описание потребности пользователей

В эпоху цифровых технологий, когда компьютеры стали неотъемлемой частью нашей жизни, понимание их устройства становится все более актуальным. Однако для многих пользователей принципы работы компьютера остаются сложной и малопонятной темой. Интерактивная игра «Коди - Компьютерный доктор» призвана решить эту проблему, предлагая увлекательный способ изучения мира компьютеров. Она позволяет в игровой форме познакомиться с устройством компьютера и принципами его работы.

Первыми, кому необходима эта игра, являются школьники и студенты. В эпоху, когда информатика прочно вошла в школьную программу, «Коди - Компьютерный доктор» становится живой и увлекательной альтернативой скучным учебникам и лекциям. Игра позволяет получить базовые знания о компьютерах, их устройстве и функционировании в легкой, запоминающейся форме.

Новички и начинающие пользователи - еще одна важная целевая аудитория. Зачастую, страх перед поломкой компьютера, боязнь сделать что-то не так, парализует и отталкивает от полноценного использования технологий. «Коди - Компьютерный доктор» поможет разобраться в основах работы компьютера. Преодолевая страх перед техникой и получая практический опыт, новички начинают понимать, как компьютер взаимодействует с окружающей средой.

Люди, стремящиеся углубить свои знания о компьютерах, также найдут в игре ценный ресурс. «Коди - Компьютерный доктор» предлагает погрузиться в изучение внутренней структуры компьютера, его

архитектуры и особенностей работы различных компонентов. Предоставляя возможность изучать компьютер в безопасной и игровой среде, дарит более глубокое понимание информационных технологий и хорошо подходит для общего развития в области информатики.

Нельзя забывать и о преподавателях и учителях информатики. Для них «Коди - Компьютерный доктор» – это хороший вспомогательный инструмент, способный повысить вовлеченность учеников и облегчить процесс обучения. Использование игры на уроках позволяет сделать занятия более интерактивными и наглядными, а значит, эффективнее донести знания до учеников.

«Коди - Компьютерный доктор» – это не просто игра, это интерактивный учебник, призванный сделать мир информационных технологий более понятным и доступным для всех. Это путешествие в мир электронных глубин, которое не только развлекает, но и обучает, подготавливая к успешной жизни в цифровом веке.

1.1.2 Анализ существующих аналогов

Human Resource Machine.

(https://tomorrowcorporation.com/humanresourcemachine).

Human Resource Machine (рисунок 1) – это головоломка, в которой игроку предстоит программировать маленького офисного работника, чтобы выполнять простые задачи, используя визуальный язык программирования, напоминающий ассемблер. Игрок должен оптимизировать код для повышения эффективности и снижения количества шагов. Геймплей игры изображен на рисунке 2.

Human Resource Machine и «Коди - Компьютерный Доктор» предлагают разные подходы к обучению сложным концепциям. Первая больше подходит для тех, кто хочет изучить основы программирования, в то время как «Коди - Компьютерный Доктор» ориентирована на тех, кто хочет понять, как работает аппаратное обеспечение компьютера.

Уникальность «Коди - Компьютерный Доктор» заключается в сочетании игрового процесса и объяснения принципов работы аппаратного обеспечения на доступном и увлекательном уровне, чего не хватает Human Resource Machine, сосредоточенной на абстрактных концепциях программирования.

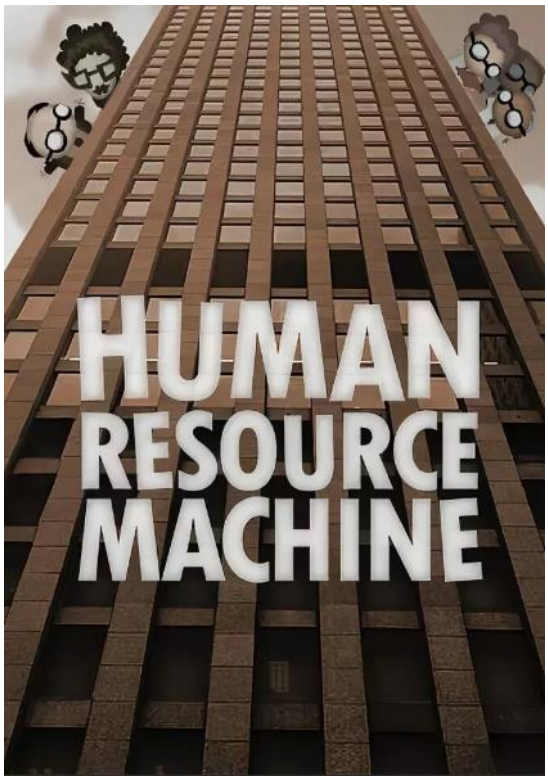


Рисунок 1 – Логотип игры Human Resource Machine.

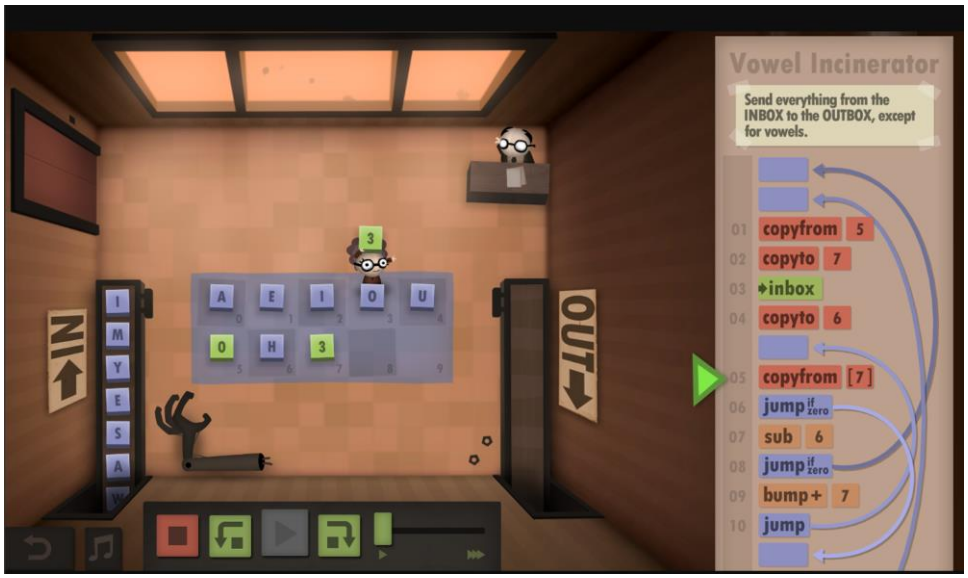


Рисунок 2 – Изображение из игры Human Resource Machine.

PC Building Simulator.

(https://store.steampowered.com/app/621060/PC_Building_Simulator/).

Одним из популярных симуляторов сборки компьютеров является PC Building Simulator (рисунок 3). Эта игра позволяет игрокам виртуально собирать и настраивать компьютеры, используя лицензированные компоненты реальных производителей. PC Building Simulator предлагает реалистичный опыт сборки, позволяя изучить различные компоненты, их совместимость и производительность. Игра может быть полезна для тех, кто хочет научиться собирать компьютеры на практике. Однако, PC Building Simulator, в основном, фокусируется на физической сборке и не объясняет принципы работы отдельных компонентов на более глубоком уровне.

Рисунок 3 – Логотип игры PC Building Simulator.

Scratch.

(https://scratch.mit.edu/).

Другой интересный пример - визуальная среда программирования Scratch, представленная на рисунках 4, 5. Эта платформа позволяет детям и начинающим программистам создавать собственные интерактивные истории, игры и анимации, используя блочное программирование. Scratch отлично подходит для развития логического мышления и знакомства с

основными концепциями программирования. Однако, не затрагивает тему аппаратного обеспечения и не дает представления о том, как работают компоненты компьютера.

Рисунок 4 – Логотип игры Scratch

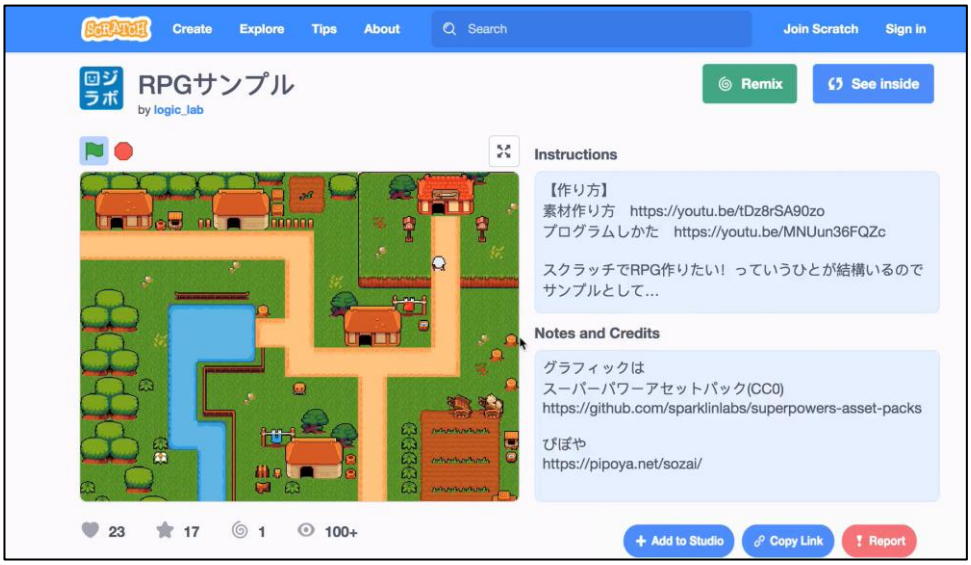


Рисунок 5 – Геймплей Scratch

Таким образом, хотя PC Building Simulator предлагает практический опыт сборки компьютеров, а Scratch развивает навыки программирования, ни один из этих инструментов не фокусируется на объяснении принципов работы аппаратного обеспечения на доступном и увлекательном уровне. Именно в этом заключается уникальность «Коди - Компьютерный Доктор». Эта игра предлагает не просто сборку или программирование, а погружает игрока в мир электронных компонентов, объясняя, как они работают и

взаимодействуют друг с другом. «Коди - Компьютерный Доктор» – это инновационный ответ на потребность в углубленном изучении комплектующих, предлагающий новый подход к изучению принципов работы ПК.

1.1.3 Постановка задачи

В рамках реализации игрового проекта необходимо разработать ряд ключевых функциональных возможностей, обеспечивающих полноценное взаимодействие игрока с создаваемым миром. Первоочередной задачей является реализация системы настроек, позволяющей игроку адаптировать параметры игры под свои предпочтения. В частности, необходимо обеспечить возможность включения и выключения звуковых эффектов, предоставить инструменты для управления общим уровнем громкости, чтобы звук не был чрезмерно навязчивым или, наоборот, слишком тихим, а также реализовать функцию включения и выключения музыкального сопровождения, создающего атмосферу игрового мира.

Следующим важным этапом является реализация игрового процесса, который позволит игроку непосредственно взаимодействовать с созданным миром и выполнять поставленные задачи. Необходимо реализовать систему передвижения, обеспечивающую перемещение персонажа по игровому миру. Важной частью управления является взаимодействие с миром, включающее в себя возможность подбора предметов, активации различных объектов и ведения диалогов с неигровыми персонажами. Наконец, необходимо реализовать боевую систему, позволяющую персонажу наносить удары врагам.

Игроку предстоит исследовать мир, состоящий из четырёх уникальных уровней, каждый из которых представляет собой отдельное испытание с собственной неповторимой задачей и игровой механикой. Уровни должны быть спроектированы таким образом, чтобы последовательно раскрывать

возможности игрового мира и постепенно усложнять задачи, требуя от игрока адаптации и освоения новых умений.

Важным этапом разработки является реализация запуска игры и выхода из неё, обеспечивающая игроку возможность завершить игровой сеанс в любой момент. Функция выхода из приложения должна обеспечивать корректное завершение всех процессов, чтобы избежать потери данных и других проблем. Кроме того, необходимо реализовать систему сохранения, позволяющую игроку продолжить игру, не теряя достигнутый прогресс. Диаграмма вариантов использования представленна на рисунке 6.

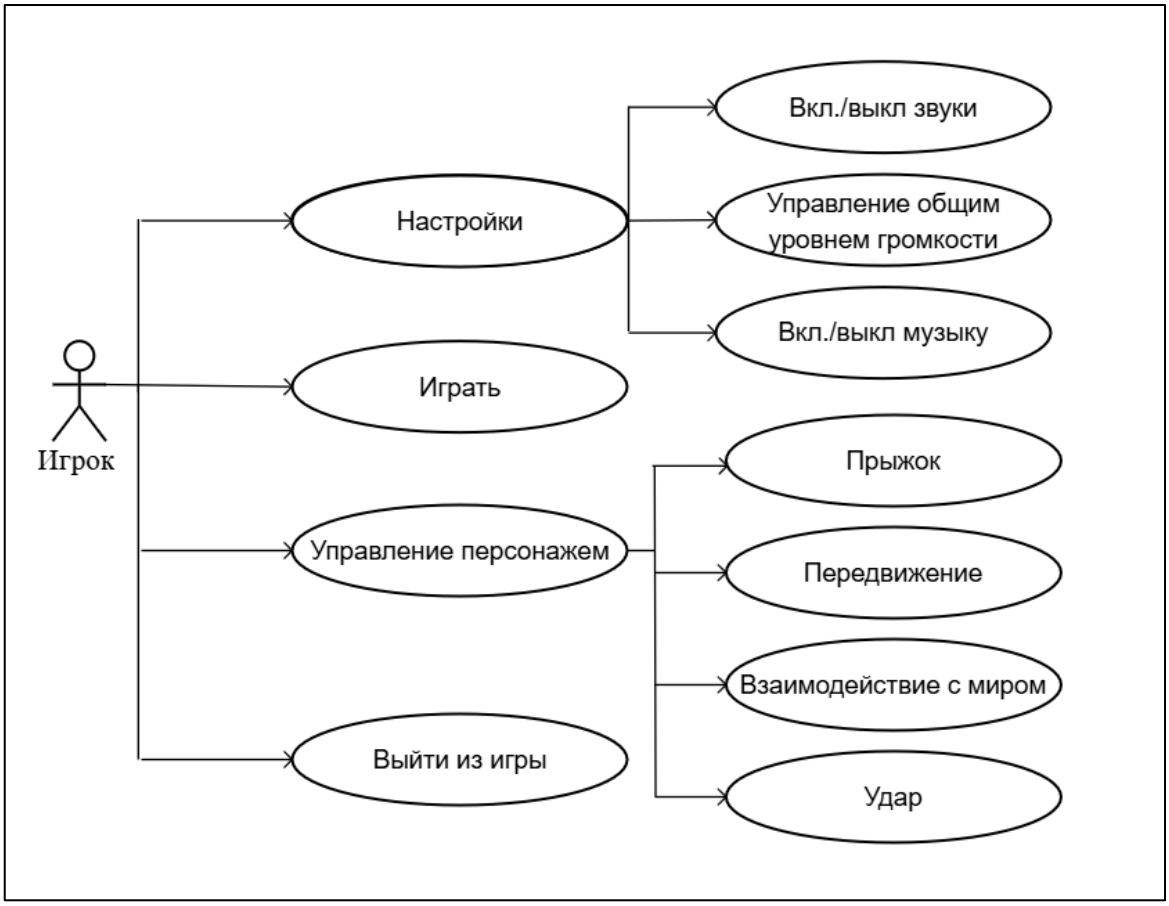


Рисунок 6 – UML – диаграмма вариантов использования.

Рекомендуемые требования к аппаратной части:

- процессор Четырехъядерный процессор с частотой 3 ГГц;
- оперативная память 4 ГБ;
- видеокарта дискретная видеокарта;

- место на диске 1 ГБ;

- операционная система Windows 10.

1.2 Выбор инструментальных средств

1.2.1 Среда разработки Unity

Unity – это мощная кроссплатформенная среда разработки, предоставляющая полный инструментарий для создания 2D игр. Она позволяет создавать игры, которые могут быть развернуты на различных платформах, включая PC, мобильные устройства (iOS и Android), веб-браузеры и другие, что является критически важным для достижения широкой аудитории. При разработке игр на Unity, полезным руководством является: «Как создать игру на Unity» [1].

Существует ряд альтернативных 2D движков, таких как GameMaker Studio 2, Godot Engine и Cocos2d-x. GameMaker Studio 2, безусловно, прост в освоении и хорошо подходит для небольших проектов, но может быть ограничен в своей гибкости и масштабируемости для более сложных игр. Godot Engine является мощным и бесплатным движком с открытым исходным кодом, но его сообщество пока еще не настолько развито, как у Unity, что может создавать трудности при поиске решений и поддержки. Cocos2d-x, хоть и предлагает высокую производительность, требует более глубоких знаний программирования и может быть сложнее в освоении.

Возможность визуальной компоновки сцены, использования префабов для повторного использования элементов и настраиваемый редактор значительно создания 2D графики и анимации. Sprite Editor и Animation Window предоставляют удобные средства для работы с 2D ассетами. Поддержка Sprite Atlas позволяет оптимизировать текстуры и повысить производительность игры. Широкие возможности анимации, включая skeletal animation, позволяют создавать живые и выразительные 2D персонажи. Другие движки могут предлагать схожие инструменты, но часто уступают Unity в удобстве

использования, интеграции с ассетами и возможностях оптимизации.

На рисунке 7 представлено окно Unity.

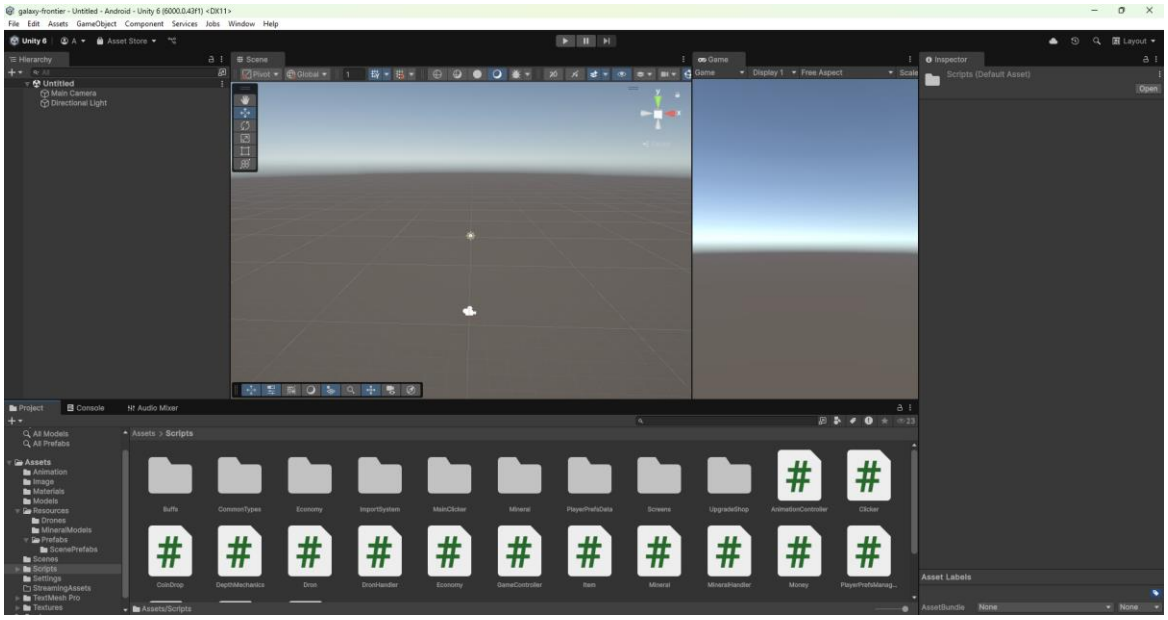


Рисунок 7 – Меню Unity

Огромное и активное сообщество Unity предоставляет бесценную поддержку и ресурсы. В случае возникновения вопросов или проблем, всегда можно обратиться к форумам [2], онлайн-курсам, туториалам и asset store для поиска решений и вдохновения. Asset Store предлагает широкий выбор готовых ассетов, включая спрайты, звуки, инструменты и плагины, которые могут значительно ускорить процесс разработки и сэкономить время. Обширность сообщества Unity и доступность готовых ассетов значительно сокращают время разработки.

Встроенный физический движок Unity, хотя и в первую очередь ориентирован на 3D, также прекрасно подходит для создания реалистичной физики в 2D играх. Благодаря компонентам вроде Rigidbody2D и Collider2D можно легко реализовать физические взаимодействия между объектами, такие как столкновения и гравитацию что добавляет глубины и реалистичности игровому процессу. Некоторые 2D движки имеют специализированные физические движки, но универсальность Unity

позволяет использовать существующие знания и опыт 3D-разработки для создания 2D игр.

Unity поддерживает язык программирования C#, который является мощным и гибким языком, идеально подходящим для разработки игровой логики и механик. C# позволяет писать чистый, структурированный и масштабируемый код, который легко поддерживать и расширять в будущем. Благодаря обширной документации и большому количеству обучающих материалов по C# и Unity, можно быстро освоить необходимые навыки и решать сложные задачи. В то время как некоторые движки используют собственные скриптовые языки, C# является широко распространенным и востребованным языком, что упрощает интеграцию с другими системами.

Выбор Unity для разработки 2D игры обусловлен сочетанием мощных инструментов, гибкого интерфейса, поддержки C#, активного сообщества и кроссплатформенности. Эти факторы позволяют эффективно разрабатывать, тестировать и развертывать игру на различных платформах, достигая широкой аудитории и реализуя любые творческие замыслы. Unity – это оптимальный выбор для создания качественной и увлекательной 2D игры. Официальная документация Unity подробно описывает все возможности кроссплатформенной разработки [3].

1.2.2 Среда разработки кода Visual Studio. Язык программирования C#

Выбор C# в качестве языка программирования и Visual Studio в качестве IDE для разработки 2D игры в Unity был обусловлен сочетанием их мощных возможностей, удобства использования и тесной интеграции с Unity, что позволяет значительно повысить эффективность разработки и качество конечного продукта.

C# – это объектно-ориентированный язык программирования, разработанный Microsoft, который является де-факто стандартом для разработки в Unity. В отличие от JavaScript (другой язык, поддерживаемый

Unity), C# предлагает строгую типизацию, что позволяет выявлять ошибки на этапе компиляции, а не во время выполнения, что значительно снижает количество багов и повышает стабильность игры. Синтаксис C# чистый и лаконичный, что упрощает чтение и понимание кода. Кроме того, C# обеспечивает более высокую производительность, чем JavaScript, что критически важно для создания сложных игровых механик и визуальных эффектов. Для изучения C# была использованна книга «Head First C#» Эндрю Стиллмана и Дженнифер Грин [4].

Есть и альтернативные варианты языка программирования. К примеру, можно рассмотреть Boo или UnityScript (JavaScript), однако они уступают C# по производительности, поддержке и доступности библиотек. C# – это более зрелый и универсальный язык, который широко используется в индустрии разработки программного обеспечения.

Visual Studio – это мощная и полнофункциональная IDE (интегрированная среда разработки), разработанная Microsoft, которая обеспечивает исключительную поддержку для C# и Unity представлена на рисунке 8.

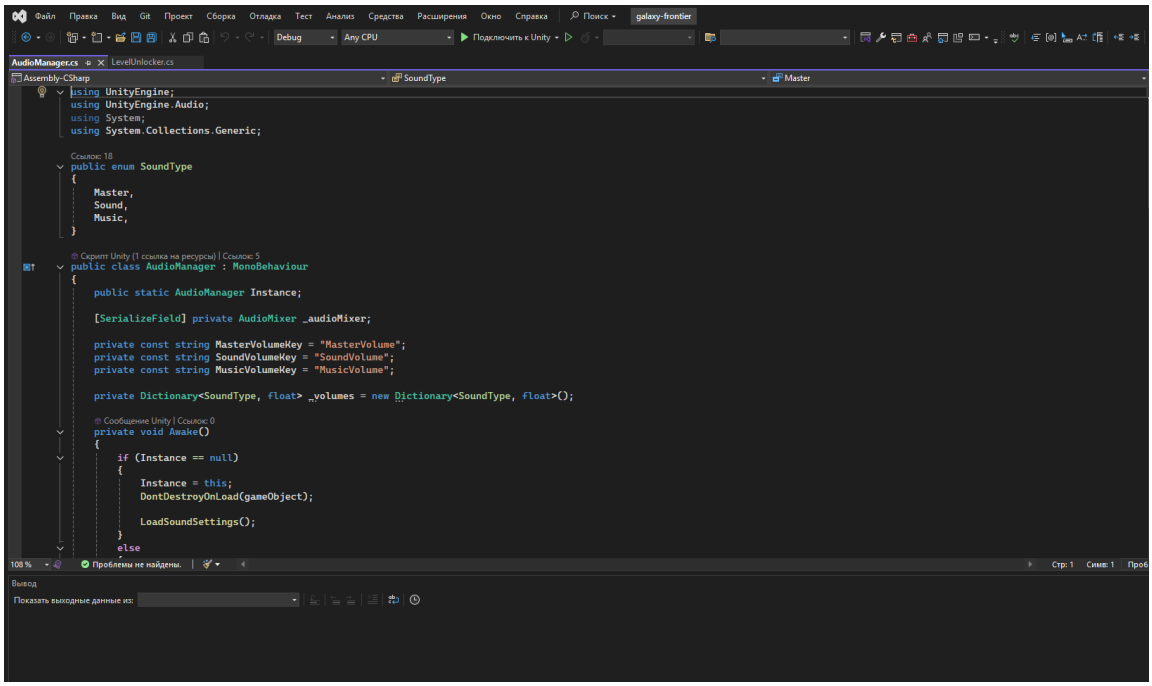


Рисунок 8 – IDE языка C#

Visual Studio предоставляет широкий спектр инструментов для написания и отладки кода, включая:

- отладчик – мощный отладчик позволяет пошагово выполнять код, просматривать значения переменных и отслеживать ход выполнения программы, что значительно упрощает поиск и исправление ошибок;

- профилировщик – инструменты профилирования позволяют выявлять узкие места в производиция Visual Studio с Unity. Visual Studio автоматически распознает проекты Unity и предоставляет специальные инструменты для работы с ними, такие как:

1) поддержка отладки непосредственно в Unity - можно отлаживать свой код C# прямо в редакторе Unity, устанавливая точки остановки, просматривая значения переменных и отслеживая ход выполнения программы;

2) автоматическая компиляция и импорт скриптов - Visual Studio автоматически компилирует скрипты C# и импортирует их в Unity, что ускоряет процесс разработки.

Можно рассмотреть и альтернативные варианты IDE, такие как Visual Studio Code или Rider. Хотя Visual Studio Code и является легким и быстрым редактором кода с множеством расширений, он не обладает такой глубокой интеграцией с Unity и таким мощным отладчиком, как Visual Studio. В свою очередь, Rider, разработанный JetBrains, является отличной альтернативой, но за его использование требуется платная лицензия, в то время как Visual Studio доступна бесплатна что для индивидуальных разработчиков идеальный вариант.

1.2.3 Графический редактор Paint 3D

Выбор Paint 3D для создания и редактирования графических элементов в игре был обусловлен несколькими ключевыми факторами, которые делают его подходящим инструментом для решения определенных задач, особенно на начальных этапах разработки и для создания простых ассетов. Важно

понимать, что Paint 3D не является профессиональным графическим редактором, таким как Adobe Photoshop или GIMP, и обладает ограниченным функционалом, но его простота и доступность делают его привлекательным для определенных целей.

Рабочее пространство Paint 3D представленно на рисунке 9.

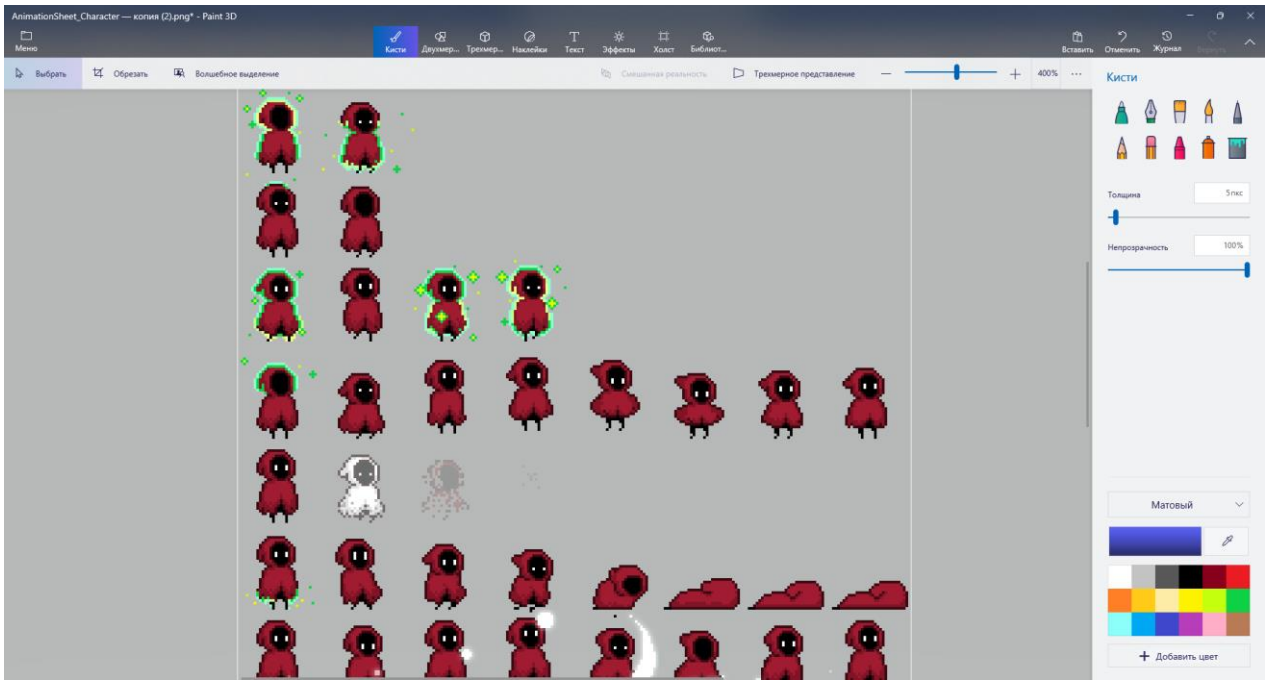


Рисунок 9 – Рабочее пространство в Paint 3D

Paint 3D, уже установленный на большинстве компьютеров с Windows 10 и 11, не требует загрузки дополнительных программ. Его интерфейс интуитивно понятен и прост в освоении, даже для тех, кто не имеет опыта работы с графическими редакторами. Это особенно важно на ранних этапах разработки, когда необходимо быстро создавать базовые графические элементы.

Данный редактор идеально подходит для создания простых спрайтов для персонажей, объектов и элементов интерфейса, а также для создания базовых текстур для использования в игровых сценах. Инструменты рисования, заливки и добавления текста позволяют быстро создавать необходимые визуальные элементы без необходимости изучения сложного функционала профессиональных графических редакторов.

Несмотря на то, что Paint 3D в первую очередь предназначен для работы с 3D-графикой, он может быть использован для создания и редактирования простых 3D-моделей, которые затем можно рендерить в 2D и использовать в качестве спрайтов. Этот подход может быть полезен для создания 2D-графики на основе 3D-форм.

Paint 3D позволяет быстро создавать и модифицировать графические элементы. Возможность быстро вносить изменения в спрайты и текстуры, без необходимости перезапускать сложный процесс редактирования в профессиональном графическом редакторе, значительно ускоряет процесс разработки и позволяет быстро тестировать различные визуальные решения.

Приложение тесно интегрирован с другими инструментами Windows, что позволяет легко обмениваться графическими элементами между различными приложениями. Например, редакторы. Это особенно важно при работе на старых компьютерах или при разработке игр для мобильных устройств.

Важно понимать, что Paint 3D имеет ряд ограничений, таких как отсутствие поддержки слоев, ограниченные возможности для работы с цветом и отсутствие продвинутых инструментов для ретуширования изображений. Поэтому для создания сложных и детализированных графических элементов потребуется использовать профессиональные графические редакторы, такие как Adobe Photoshop, GIMP или Krita.

Рассматривая альтернативы, такие как Adobe Photoshop, GIMP, Krita или даже простые онлайн-редакторы, становится очевидно, что они обладают гораздо большим функционалом и возможностями, но требуют больше времени на освоение и они сложнее в использовании для простых задач. Photoshop, безусловно, является стандартом, но он платный и требует немалых знаний. GIMP и Krita - это бесплатные и мощные альтернативы, но они также сложнее в освоении, чем Paint 3D. Бесплатные онлайн-редакторы

часто ограничены в функциональности и могут требовать стабильного интернет-соединения.

Выбор Paint 3D был обусловлен его простотой, доступностью и удобством использования для 2D игры. По мере развития проекта и увеличения сложности графики, вероятно, надо будет применять более мощные графические редакторы, но на начальных этапах Paint 3D является отличным выбором для быстрого и эффективного создания базовых ассетов.

1.2.4 Генератор фото, видео, музыки YesChat AI

Использование генератора фото, видео, музыки YesChat AI в игре на Unity обусловлено стремлением оптимизировать процесс разработки, экспериментировать с контентом и создавать уникальный игровой опыт, даже с ограниченными ресурсами. YesChat AI представляет собой ценный инструмент для достижения целей. Главная страница на рисунке 10.

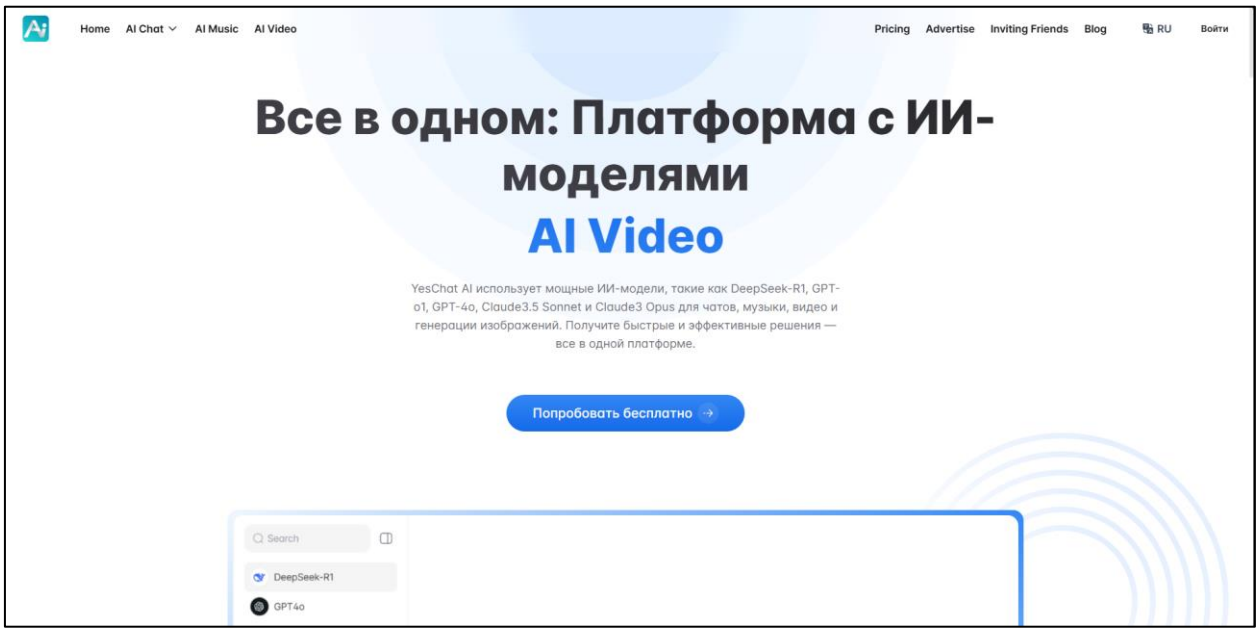


Рисунок 10 – ИИ помощник YesChat AI

Благодаря ИИ, создание графики, анимации и музыки традиционными методами требует значительного времени и ресурсов. YesChat AI позволяет генерировать необходимые ассеты в разы быстрее, освобождая время для работы над другими аспектами игры, такими как механика, геймплей и

оптимизация. Это особенно важно для инди-разработчиков, у которых каждый час на счету.

YesChat AI позволяет быстро и легко экспериментировать с различными визуальными и звуковыми стилями, не тратя время и деньги на создание полноценных ассетов. Это позволяет проверить, какие стили лучше всего соответствуют концепции игры и будут привлекать целевую аудиторию. Можно сгенерировать несколько вариантов спрайтов, текстур или музыкальных треков, сравнить их и выбрать наиболее подходящий.

С помощью YesChat AI можно сгенерировать уникальный контент, который невозможно создать обычными методами. Например, можно создать процедурно генерируемые текстуры с необычными узорами или музыку с уникальным звучанием. Это позволяет придать игре неповторимый стиль и выделиться на фоне конкурентов.

На ранних этапах разработки часто необходимо быстро заполнить игру контентом, чтобы протестировать основные механики и геймплей. Искусственный интеллект позволяет генерировать временные ассеты, не тратя время на создание полноценных спрайтов, анимаций или музыкальных треков. Впоследствии эти временные ассеты можно заменить на более качественные.

YesChat может использоваться для создания адаптивного контента, который меняется в зависимости от действий игрока или других параметров игры. Например, можно сгенерировать разные варианты текстур для окружения, в зависимости от времени суток или погоды. Или создать музыкальные треки, которые меняются в случайном порядке.

Использование ИИ позволяет значительно снизить затраты на разработку контента. Нет необходимости в художниках, аниматорах и композиторах для создания каждого спрайта, текстуры или музыкального трека. Это особенно актуально для разработки проекта с ограниченным бюджетом.

Можно генерировать оптимизированные ассеты, которые не нагружают систему и не снижают производительность игры. Например, возможно сгенерировать текстуры с небольшим разрешением или музыку с низким битрейтом. Это особенно важно для мобильных игр или игр, работающих на слабых компьютерах.

Простая интеграция с Unity позволяет напрямую импортировать сгенерированные ассеты в проект, не тратя время на конвертацию файлов и ручную настройку. Это значительно ускоряет процесс разработки и упрощает рабочий процесс.

Существуют и другие инструменты для генерации контента на основе ИИ, такие как Midjourney, DALL-E 2, Stable Diffusion для изображений, а также Jukebox и Amper Music для музыки. Midjourney и DALL-E 2 известны своей способностью генерировать высококачественные изображения с высокой степенью детализации, но являются дорогими вариантами. Stable Diffusion предоставляет большую гибкость в настройке, но требует мощного оборудования и технических знаний. Jukebox и Amper Music позволяют генерировать музыкальные треки различных жанров, но требуют ручную доработку для достижения желаемого стиля игры.

В отличие от этих специализированных инструментов, YesChat AI предлагает более универсальный подход, объединяя в себе возможности генерации как изображений, так и музыки в рамках одной платформы. Это упрощает процесс разработки, позволяя генерировать различные типы контента, не переключаясь между разными инструментами и платформами. Кроме того, YesChat AI может быть более доступным по цене в отличие от остальных. Использование его в игре на Unity позволяет ускорить разработку, экспериментировать с разными стилями, создавать уникальный контент, снизить затраты и оптимизировать производительность.

1.2.5 Видео редактор Adobe Premiere Pro

Использование Adobe Premiere Pro для создания захватывающего фона

главного меню было стратегическим решением, направленным на то, чтобы с первых секунд привлечь внимание игрока и задать позитивный тон всей игре. В отличие от более простых видеоредакторов, Premiere Pro предлагает несравненную гибкость и контроль над каждым аспектом видеомонтажа и цветокоррекции, позволяя создать визуально привлекательный и динамичный фон, который сразу же вызывает интерес и желание погрузиться в игровой мир.

На рисунке 11 представленно рабочее пространство Adobe Premiere Pro.

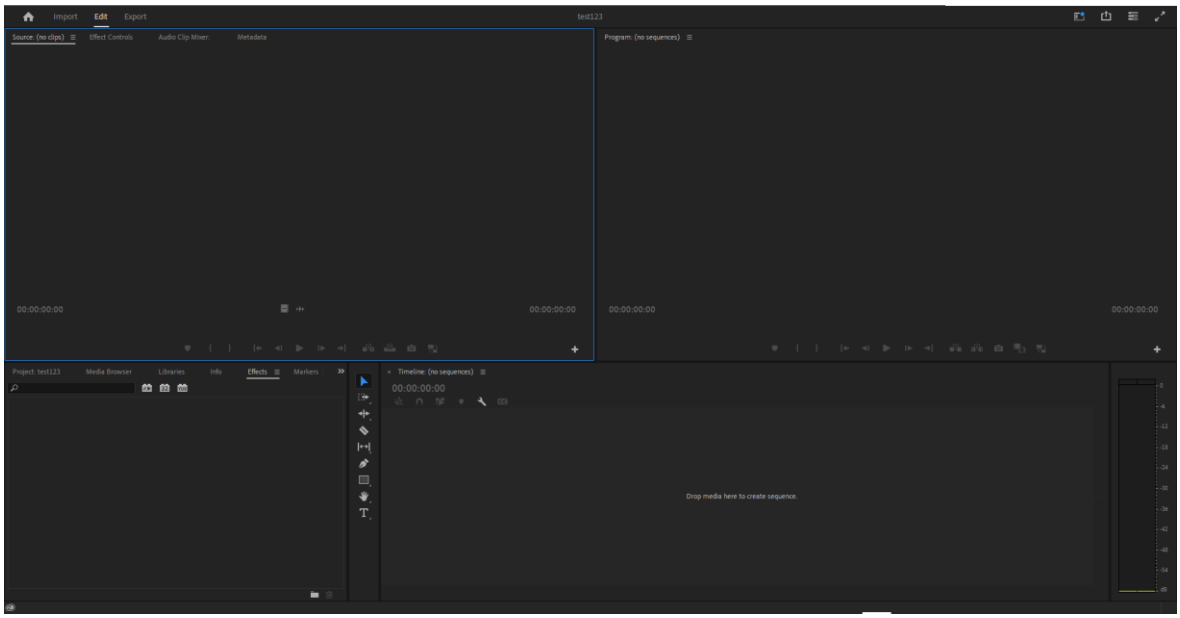


Рисунок 11 – Рабочее пространство в Adobe Premiere Pro

Premiere Pro имеет продвинутые инструменты цветокоррекции, позволяющие точно настроить цвета, контрастность и яркость видео. Это позволяет создать яркий, красочный и насыщенный фон, который сразу же привлекает внимание. Premiere Pro поддерживает широкий спектр визуальных эффектов, которые можно использовать для добавления искр, блеска, свечения и других элементов, делающих фон еще более привлекательным и запоминающимся. Используя эти инструменты, можно добиться именно того настроения - ощущение яркой, красочной и

увлекательной игры.

Premiere Pro тесно интегрирован с другими приложениями Adobe Creative Cloud, такими как After Effects и Photoshop. Это помогает легко импортировать графические элементы, созданные в Photoshop, и добавлять сложные анимации и визуальные эффекты, созданные в After Effects. Эта интеграция значительно расширяет возможности и позволяет создавать фон главного меню, который выглядит профессионально и качественно. Яркий и красочный фон, созданный в Premiere Pro, поможет игре выделиться среди других и привлечь больше игроков, желающих окунуться в увлекательный мир компьютера.

Главное преимущество Premiere Pro заключается в его широком спектре инструментов и гибкости. Конечно, существуют и другие видеоредакторы, например, Final Cut Pro, DaVinci Resolve или даже более простые программы, такие как iMovie и Movavi Video Editor. Final Cut Pro, будучи отличным инструментом, ориентирован на платформу macOS, что ограничивает его доступность. DaVinci Resolve, хоть и является мощным инструментом для цветокоррекции и постобработки, может быть избыточным для задач создания фона главного меню, требуя больше времени на освоение и ресурсы системы. iMovie и Movavi Video Editor предоставляют упрощенный интерфейс и базовый функционал, но значительно уступают Premiere Pro в возможностях тонкой настройки, цветокоррекции и интеграции с другими профессиональными инструментами. В то время как эти альтернативные видеоредакторы могут быть подходящими для простых задач, выбор Premiere Pro позволяет обеспечить наилучшее качество визуальной части игры.

1.3 Архитектура приложения

Unity изначально спроектирован с учетом компонентно-ориентированного подхода. Unity Engine основан на принципе «GameObject

+ Components». Любой объект в сцене Unity – это GameObject, к которому можно добавлять различные компоненты, определяющие его свойства и поведение.

В основе архитектуры игры лежит компонентно-ориентированный подход, где каждый игровой объект представляет собой совокупность взаимосвязанных компонентов, отвечающих за определенные аспекты его поведения. Такая структура обеспечивает гибкость, модульность и упрощает процесс разработки и отладки.

Ключевые игровые объекты наделены специализированными скриптами, определяющими их поведение.

Игра строится вокруг главного персонажа, управляемого скриптом CharacterMovement. Это скрипт, отвечающий за движение, прыжки, атаки, взаимодействие с миром и, в конечном итоге, за всё, что он делает в игре. Он является «центром управления» для персонажа. Также тесно связан с другими ключевыми скриптами. Анимации, запускаемые действиями персонажа (атака, ходьба и т.д.), управляются скриптом CharacterAnimation, получающим команды от CharacterMovement. Для отражения предметов, которые держит персонаж, используется скрипт FollowCharacter, который подчиняется движениям героя. Здоровье персонажа, его получение урона, смерть и перезапуск уровня отслеживаются в скрипте PlayerHealth. CharacterMovement отвечает за получение урона от атак врагов через взаимодействия с скриптами врагов, например, MonstersAI, Boss. Скрипт HealthBarUI берёт информацию о здоровье из PlayerHealth и отображает её на экране. Звуки действий персонажа воспроизводятся через PlayerSoundController, вызываемый опять же, скриптом CharacterMovement.

Взаимодействие с миром организовано через отдельные уровни, каждый со своей уникальной механикой.

Уровень 1 (GPU).

Фокусируется на загадке, управляемой скриптом Flower. Этот скрипт взаимодействует с ColorBox для создания предметов, необходимых для решения загадки, и с PlayerController для взаимодействия игрока с этими предметами. Решение загадки может вызвать изменение через BackgroundTintController и сохранение прогресса.

Уровень 2 (RAM).

Представляет собой мини-игру на сортировку предметов по категориям. DataGenerator создает предметы, которые сортируются DataCollector. PigeonController позволяет управлять голубем, который следует за предметами благодаря Follower. ScoreManager отслеживает процесс по сбору предметов и завершению уровня, а также сохраняет прогресс. BoundaryTrigger отвечает за условия поражения и инициирует поражение, вызывая GameOverManager.

Уровень 3 (Processor).

Задействует головоломку с кнопками, ядрами и их состояниями. Button отвечает за состояние кнопок, которые влияют на состояние Egg. ButtonStateChecker проверяет состояние этих элементов и завершает уровень, если комбинация верна.

Уровень 4 (HHD/SSD).

Различные скрипты врагов, которые управляют поведением и их атаками, наносящими урон игроку через скрипт PlayerHealth. Ghost отвечает за диалог прс и выдачу способности. Boss - скрипт, контролирующий само поведение главного антагониста игры, его атаки (Fireball, Knockback) и появление монстров.

Кроме того, в этом уровне присутствуют различные виды опасного окружения. Thorns - наносят урон игроку при соприкосновении, используя PlayerHealth. Существуют два вида: статичные и те, что падают, если игрок находится в зоне их действия. MovingPlatform - двигающиеся платформы,

которые затрудняют передвижение. BreakingPlatform - разрушающиеся платформы через время при соприкосновении игрока с ними.

Кроме того, на сцене присутствуют и другие скрипты.

В любой момент игры игрок может нажать кнопку паузы, активируя PauseController, который замораживает игровую активность, предоставляя игроку возможность обдумать свои действия, отдохнуть, настроить звук или же выйти в главное меню.

Сразу со старта игры открывается нулевая сцена главного меню, за функции которого отвечает скрипт MainMenuController. Здесь игрок может начать игру, получить доступ к настройкам или выйти из игры через скрипт ExitGame. Из стартовой локации можно выйти к разным уровням в зависимости от прогресса игрока, сохраняемого с помощью LevelUnlocker. Этот скрипт открывает доступ в локации в зависимости от пройденных уровней. После завершения игры, если игроку захочется начать всё с начала, он может воспользоваться функцией ResetPlayerPrefs, которая обнулит все сохраненные данные. Для перехода на какую-либо сцену используется SceneLoader.

Диалоги в игре управляются скриптом DialogueManager, который отвечает за отображение текста и логику диалогов. NPC (Ghost, другие NPC, и боссы) вызывают DialogueManager, чтобы активировать и отображать свои реплики. Для создания и организации диалогов используются Ink-файлы – это текстовые файлы, написанные на языке Ink, разработанном Inkle Studios. Ink позволяет легко писать диалоги, добавлять логику и переменные, что делает разработку диалогов более удобной. Подробнее о языке Ink можно узнать в Ink Documentation [5].

Вся звуковая составляющая игры контролируется AudioManager, который разделяет звуки на категории (звуки, музыка, общая громкость). Звуки персонажа и мира активируются через другие скрипты, но контролируются именно этим скриптом. VolumeController позволяет игроку

управлять общей громкостью, а ButtonController - отдельными параметрами музыки и звуков. Музыкальное сопровождение управляется MusicManager, который использует ScenePlaylist для выбора музыки для каждой сцены.

Компонентно-ориентированный подход способствует модульности кода. Каждый компонент отвечает за определенную функцию, что делает код более структурированным и организованным. Легче понимать, что делает каждый компонент, и находить нужную логику для изменения или исправления ошибок.

На рисунке 12-15 редставленны диаграмма архитектуры приложения.

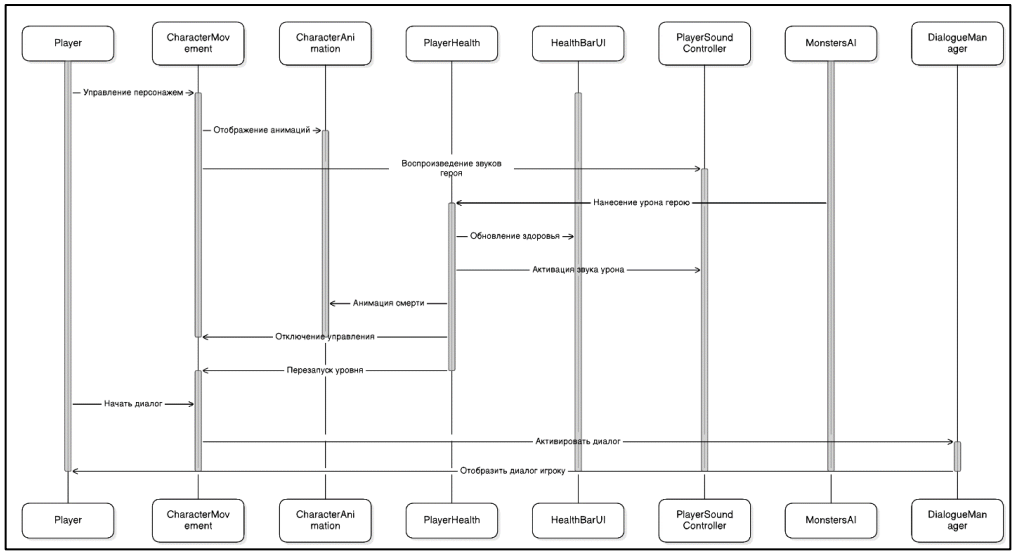


Рисунок 12 – UML – диаграмма последовательности взаимодействий героя

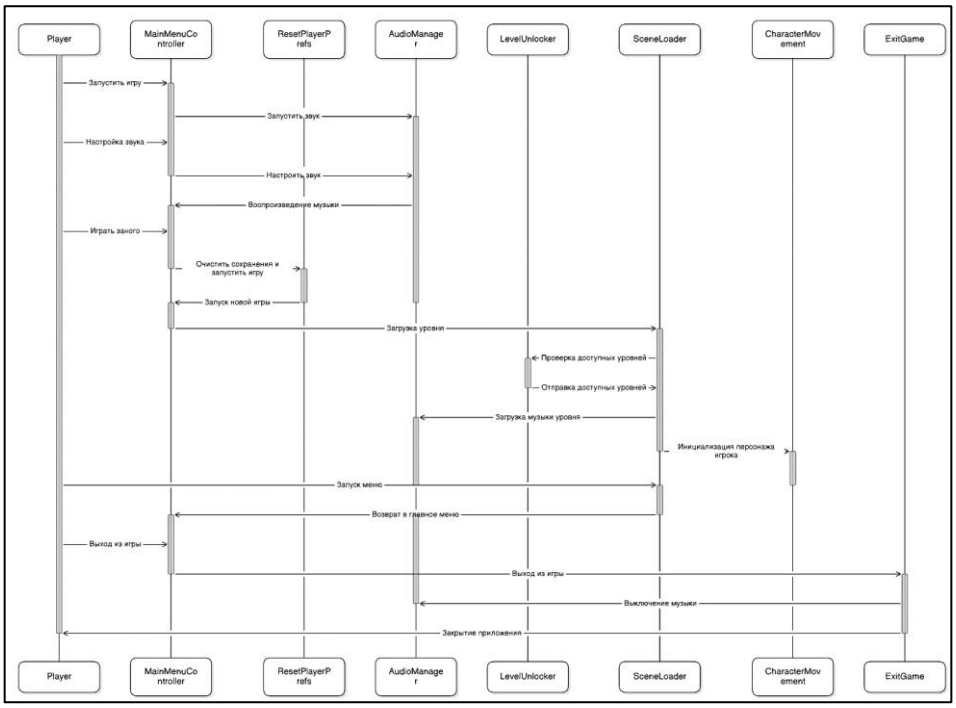


Рисунок 13 – UML – диаграмма последовательности интерфеса и загрузки

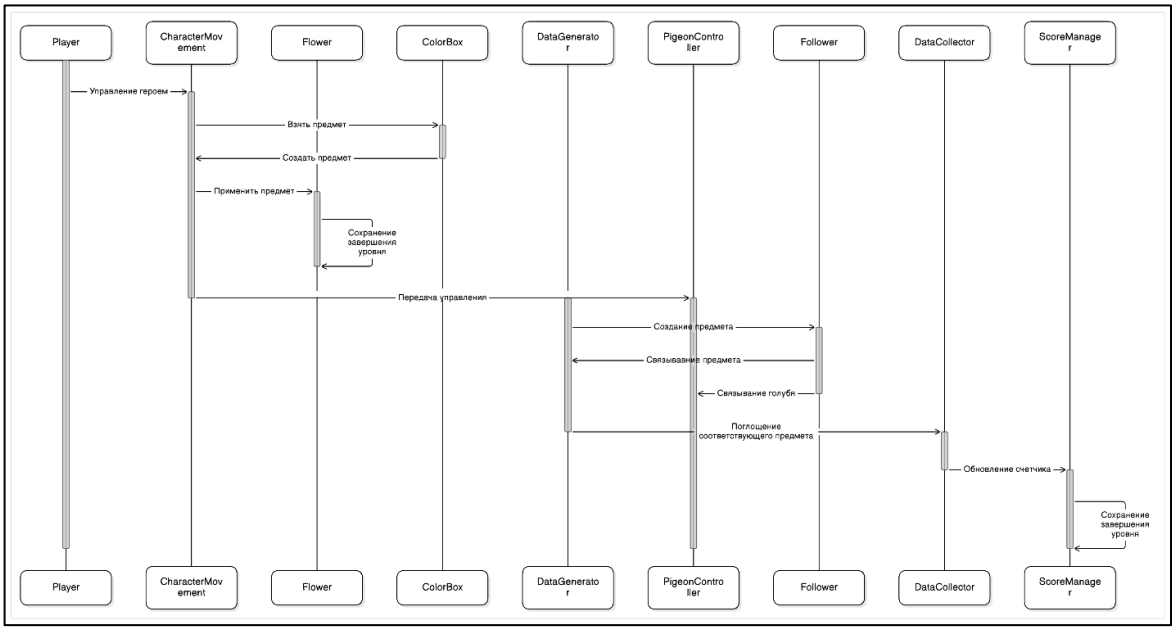


Рисунок 14 – UML – диаграмма последовательности прохождения первого и второго уровня

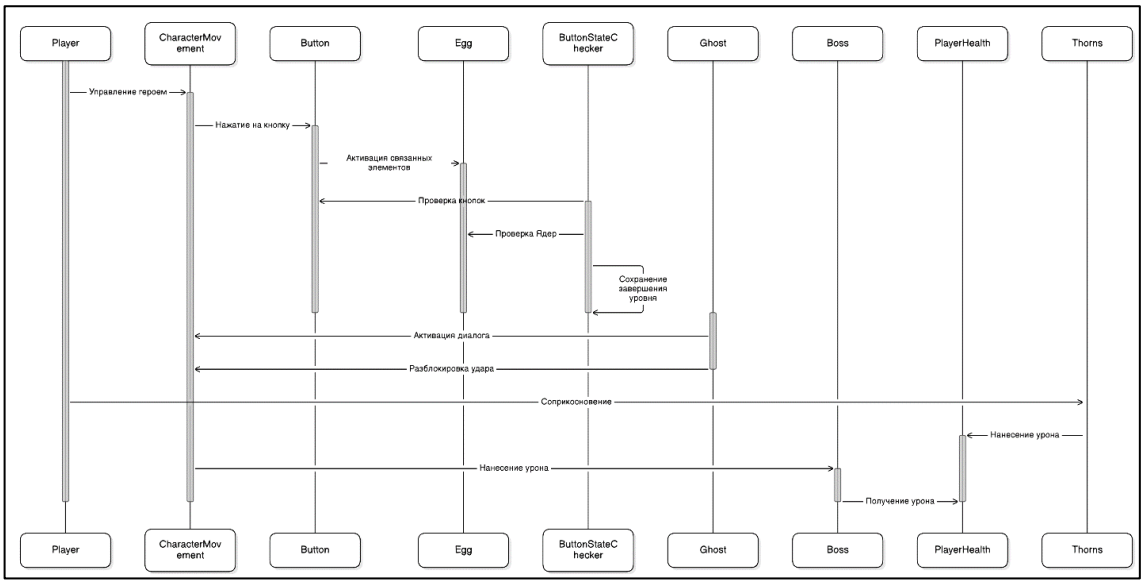


Рисунок 15 – UML – диаграмма последовательности прохождения третьего и четвертого уровня

1.4 Реализация

1.4.1 Интерфейс Unity.

Интерфейс Unity представляет собой интегрированную среду разработки (IDE), предназначенную для создания интерактивных 3D и 2D приложений, включая видеоигры, архитектурные визуализации, интерактивные симуляции и многое другое. Он состоит из нескольких основных окон, каждое из которых выполняет свою функцию в процессе разработки.

Сцена в Unity – это пространство, где размещаются игровые объекты и настраивается среда игры. В окне «Scene» можно добавлять и удалять игровые объекты, вращать, располагать их относительно друг друга или окружения, а также создавать и настраивать освещение и другие эффекты.

Сцены в Unity используются для организации игрового пространства проекта. Когда создаётся новый проект в Unity, система автоматически создает стандартную сцену. Можно создавать новые сцены и переключаться между ними, чтобы работать над различными частями проекта. Они могут

быть использованы для создания уровней игры. Можно создать несколько сцен для каждого уровня и переключаться между ними при прохождении игры. Сцена представлена на рисунке 16.

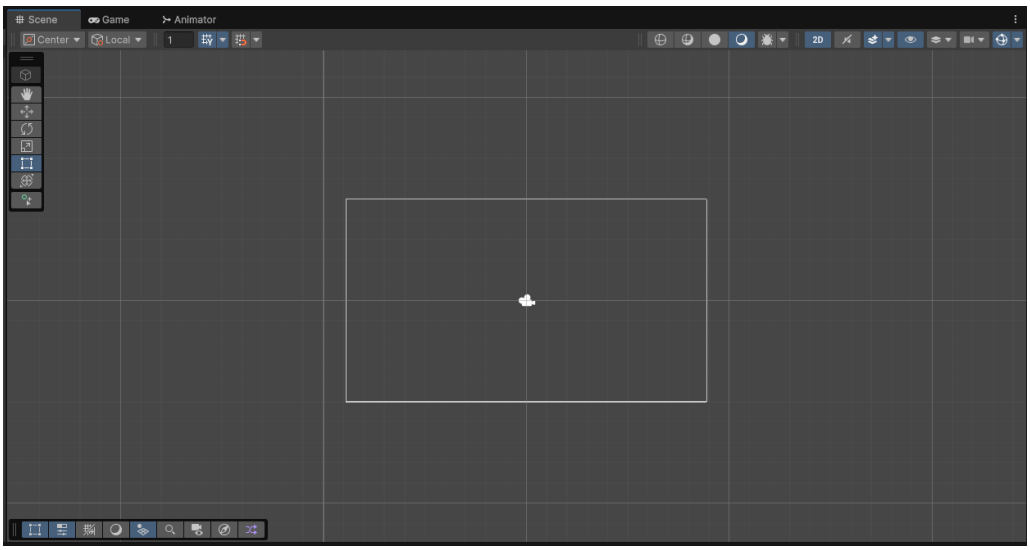


Рисунок 16 – Сцена

Окно «Game» в Unity показывает, как игра будет выглядеть для пользователя. Здесь можно сразу протестировать визуальное представление, не покидая редактор Unity. Всё это помогает быстро понять ошибку, наглядно увидеть в чём она заключается и быстро исправить ее. Окно Game представлено на рисунке 17.

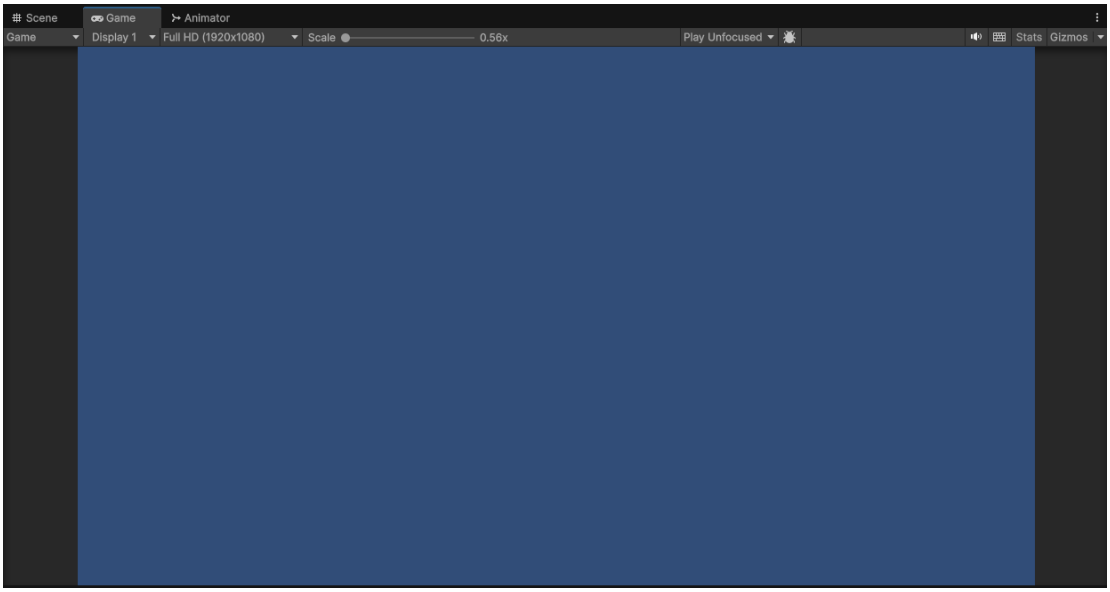


Рисунок 17 – Окно «Game»

Hierarchy» – это окно, в котором отображается иерархическая структура всех объектов в текущей сцене. Когда создаётся новый объект, он по умолчанию помещается в корень иерархии. Можно перетаскивать объекты внутри иерархии, чтобы создавать группы и связи между ними, создавая связь между родительским и дочерним объектами. Окно Иерархии представлено на рисунке 18.

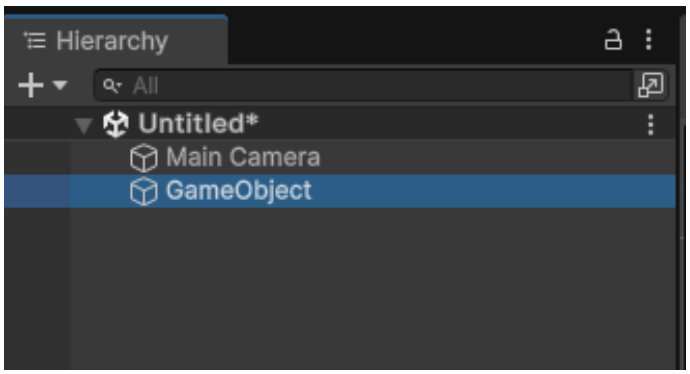


Рисунок 18 – Иерархия

«Inspector» – это окно для управления выбранным объектом. Оно отображает свойства выбранного объекта в проекте. Здесь можно менять значения компонентов, настраивая, как объект выглядит и изменять его параметры. Так же можно изменить цвет объекта, его размер, добавить к нему физику, чтобы он падал под действием гравитации, или прикрепить скрипты, чтобы он реагировал на действия игрока. Это основное окно для редактирования свойств игровых объектов. Каждый компонент имеет свои параметры, которые можно настроить в окне Inspector.

Окно инспектора представлено на рисунке 19.

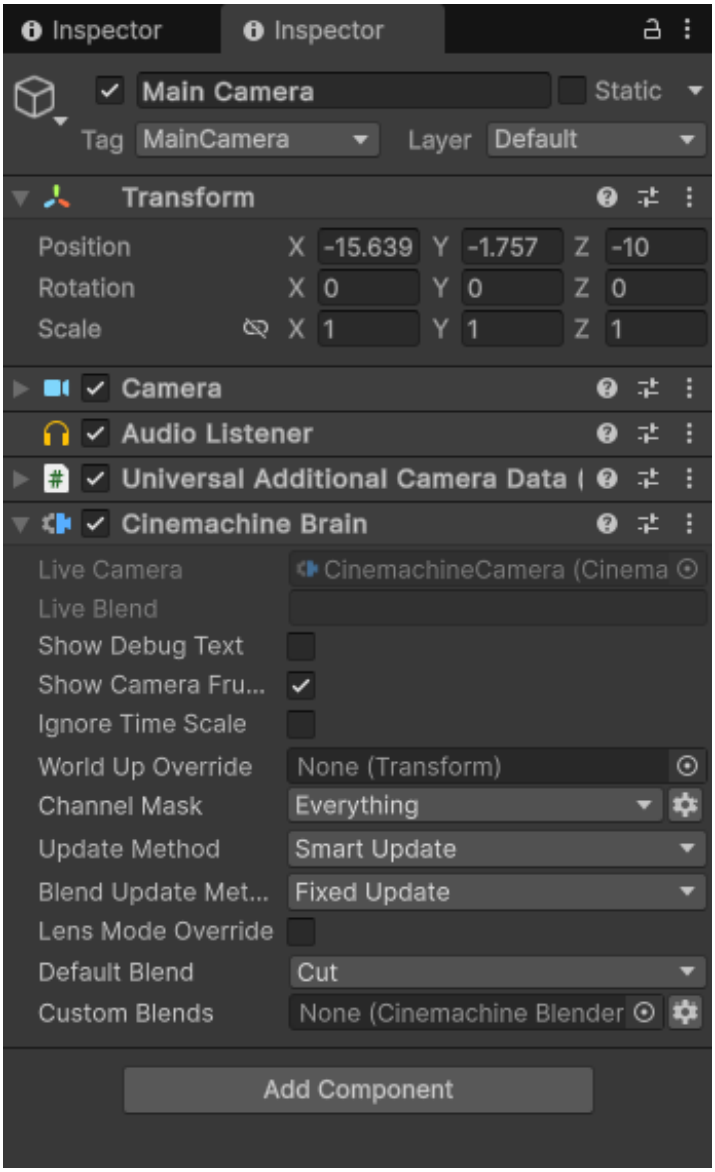


Рисунок 19 – Окно «Inspector»

Окно «Project» отображает структуру файлов и папок проекта Unity. Здесь лежат скрипты, сцены, изображения, 3D-модели, звуки и другие ресурсы. В окне Project можно создавать новые файлы и папки, а также всячески взаимодействовать: перемещать, копировать и удалять файлы. Это окно позволяет удобно организовывать проект, быстро находить нужные файлы и управлять структурой всего контента, необходимого для игры.

Окно project изображено на рисунке 20.

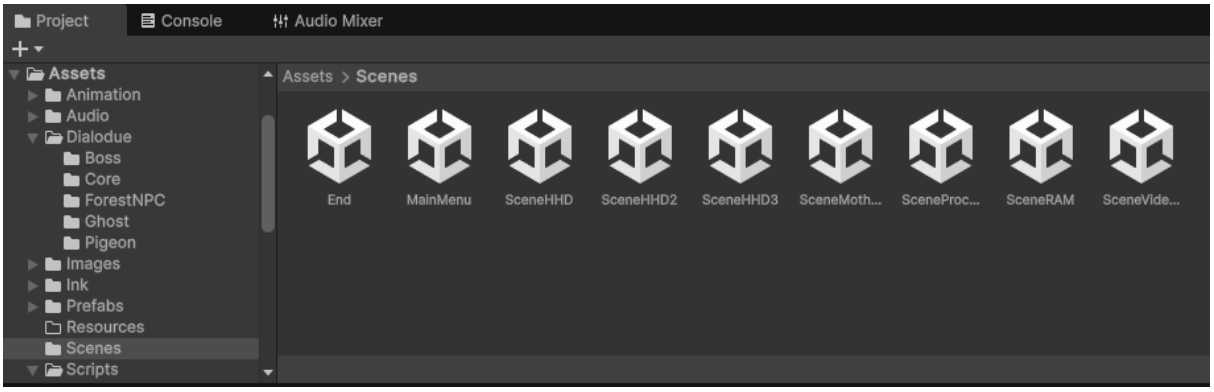


Рисунок 20 – Окно «Project»

Окно «Console» помогает получить информацию о происходящем в проекте. Основное предназначение консоли - отображать сообщения, генерируемые Unity и скриптами. Эти сообщения бывают трех основных типов: ошибки, предупреждения и отладочная информация.

Сообщения об ошибках, отмеченные красным цветом, являются самым важным сигналом, требующим немедленного внимания. Они указывают на серьезные проблемы в коде, которые приводят к сбоям, некорректному поведению или даже остановке работы игры. Первоочередная задача - устранение подобных ошибок.

Предупреждения, выделенные желтым цветом, сообщают о потенциальных проблемах в коде. Они не обязательно приводят к сбоям, но могут указывать на неэффективные элементы или устаревшие методы. Игнорирование предупреждений может привести к ухудшению производительности или возникновению ошибок в будущем.

Отладочная информация, выводимая с помощью команды Debug.Log, в скриптах, позволяет следить за ходом выполнения программы. Это незаменимый инструмент для поиска ошибок.

Окно консоли представлено на рисунке 21.

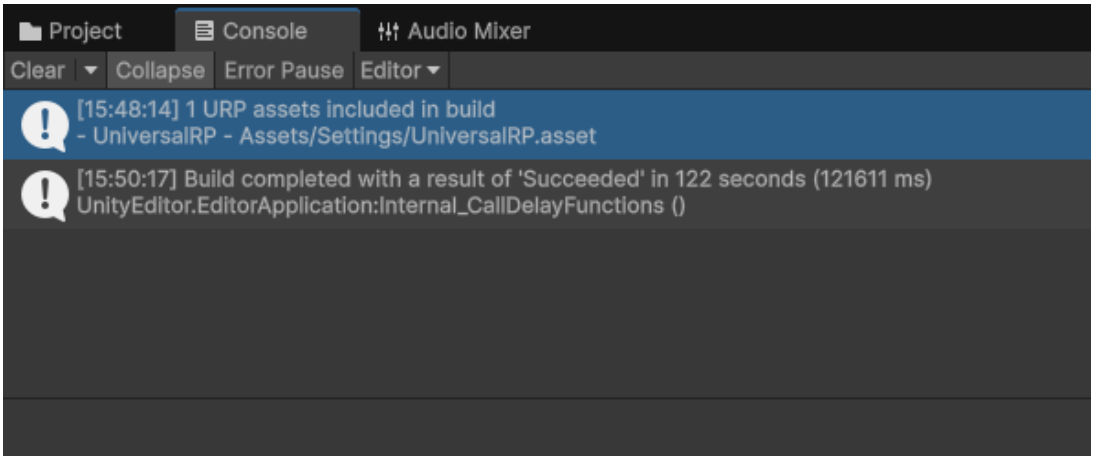


Рисунок 21 – Окно «Console»

1.4.2 Создание объекта главного героя в Unity

В Unity существуют мощные инструменты для оживления игровых персонажей и объектов: Animator и Animation. Они работают вместе, позволяя смешивать анимации, регулировать скорость и направление анимации и применять другие эффекты, такие как инверсная кинематика, чтобы сделать анимацию более естественной и реалистичной. Оба инструмента позволяют легко интегрировать анимации, взятые из различных источников, включая Unity Asset Store. В данном проекте были использованы модели и анимации, загруженные из Asset Store [6], что является распространенным и эффективным способом быстро получить готовую базу для работы.

У персонажа разработанны 7 состояний «Animation» - анимация покоя, «Animation2» - анимация ходьбы, «JumpAnimation» - анимация прыжка, «AnimationHit» - анимация удара, «Healing» - анимация восстановления здоровья, «Hit» - анимация получения урона и «AnimationDeath» - анимация смерти. В параметрах окна Animator создано 5 переменных типа trigger и одна переменная типа bool для активации анимаций.

Окна инструментов «Animation» и «Animator» представлены на рисунках 22, 23.

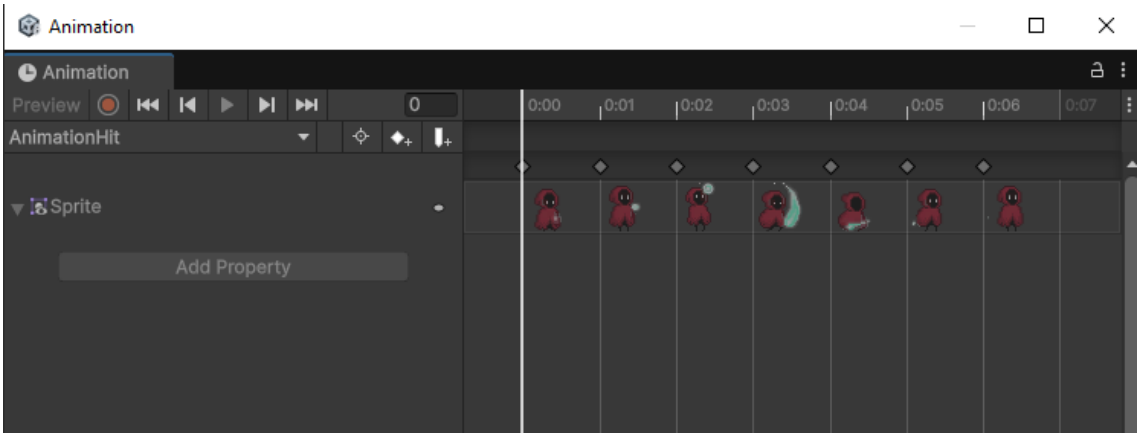


Рисунок 22 – Окно «Animation»

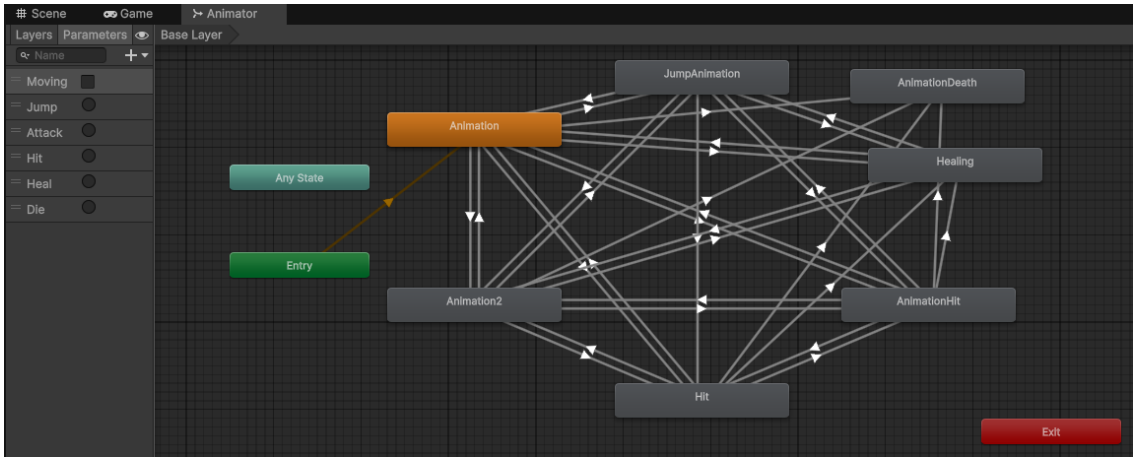


Рисунок 23 – Окно «Animator»

Для взаимодействия с объектов в Unity существует компонент Collider2D используется персонажем на всех встречающихся объектах. Этот компонент в Unity определяет границы и обнаруживает столкновения объектов в 2D-среде, определяя форму и размеры объекта, используемые для взаимодействия с другими коллайдерами в игровом мире. В результате другие объекты могут воспринимать или пересекать область, созданную компонентом Collider2D. Он обеспечивает обнаружение столкновений между объектами, представляя различные формы, включая прямоугольники, круги, многоугольники или составные формы. Кроме того, компонент Collider2D может работать вместе с компонентом Rigidbody2D для создания реалистичного поведения при столкновении. Unity отправляет сообщения и вызывает методы обработки столкновений, когда коллайдеры

объектов пересекаются или сталкиваются. Их можно использовать для реагирования на столкновения и взаимодействия с другими объектами.

Главному герои были добавлены и настроены эти компоненты, которые представлены на рисунке 24.

Рисунок 24 – Компоненты главного героя

1.4.3 Создание логики управления главным героем

«CharacterMovement».

«CharacterMovement» скрипт управляет всеми движениями, атаками, взаимодействия и даже массой персонажа.

В Start() скрипт получает ссылки на необходимые компоненты: Rigidbody2D, CharacterAnimations, Animator, Collider2D, загружает состояние canAttack из PlayerPrefs и сохраняет исходную массу тела в initialMass.

HandleInput() обрабатывает ввод игрока с клавиатуры и мыши. Здесь определяются горизонтальное направление движения и проверяется нажатие кнопки прыжка. HandleMovement() на основе ввода игрока управляет передвижением персонажа по горизонтали.

```
void HandleInput()
    {
        input.x = Input.GetAxis("Horizontal");

        if (Input.GetButtonDown("Jump"))
        {
            Jump();
            soundController.PlayJumpSound();
        }
    }
    void HandleMovement()
    {
        input.x = Input.GetAxis("Horizontal");
    }
```

Функция Move() перемещает персонажа, проверяет касание земли через CheckGround(), и синхронизирует анимации через UpdateAnimations(). UpdateAnimations() отвечает за обновление анимаций персонажа в соответствии с его действиями.

```
void Move()

    {

        float horizontalMovement = input.x * moveSpeed;

        rb.linearVelocity = new Vector2(horizontalMovement,
rb.linearVelocity.y);
```

```
        if (input.x > 0)
        {
            characterSprite.flipX = false;
        }
        else if (input.x < 0)
        {
            characterSprite.flipX = true;
        }
        moving = Mathf.Abs(input.x) > 0.01f;
    }
```

Метод Jump() обрабатывает прыжок при нажатии на кнопку. Если персонаж находится на земле, вызывается PerformJump() и воспроизводится звук прыжка. Функция PerformJump() обнуляет вертикальное движение, чтобы высота прыжка была постоянной, а затем толкает персонажа вверх с заданной силой, заставляя его подпрыгнуть. После этого запускается анимация прыжка.

```
    void Jump()
    {
        if (isGrounded)
        {
            PerformJump();
        }
    }
    void PerformJump()
    {
```

```
            rb.linearVelocity                =                new
Vector2(rb.linearVelocity.x, 0f);
            rb.AddForce(Vector2.up          *          jumpForce,
ForceMode2D.Impulse);
            animations.Jump();
        }
```

Метод Attack() запускает анимацию атаки и наносит урон врагам, если персонаж может атаковать. Он проверяет все цели, попавшие в зону поражения, и, если цель является монстром, наносит урон.

```
    void Attack()
        {
            animations.Attack();
            Debug.Log("Attack point: " + attackPoint.position);
            Collider2D[]              hitTargets              =
Physics2D.OverlapCircleAll(attackPoint.position,   attackRange,
enemyLayers);

            foreach (Collider2D target in hitTargets)
            {
                Debug.Log("We hit " + target.name);

                if (hitTargets.Length == 0)
                {
                    Debug.Log("Found 0 targets");
                    return;
                }

                foreach (Collider2D enemy in hitTargets)
```

```
                {
                    Debug.Log("Found target with layer: " +
LayerMask.LayerToName(enemy.gameObject.layer));

                }
                MonsterAI monsterAI =
target.GetComponent<MonsterAI>();
                if (monsterAI != null)
                {
                    monsterAI.TakeDamage(1);
                    continue;
                }
                MonstersAI monstersAI =
target.GetComponent<MonstersAI>();
                if (monstersAI != null)
                {
                    monstersAI.TakeDamage(1);
                    continue;
                }
                Boss boss = target.GetComponent<Boss>();
                if (boss != null)
                {
                    boss.TakeDamage(1);
                }
                FlyingEye flyingEye =
target.GetComponent<FlyingEye>();
                if (flyingEye != null)
                {
```

```
                flyingEye.TakeDamage(1);
            }
        }
    }
```

Метод SetMass устанавливает массу Rigidbody2D на указанное значение, позволяя изменять вес персонажа на третьем уровне. Метод RestoreMass() возвращает массу к исходному значению.

```
public void SetMass(float newMass)
    {
        if (rb != null)
        {
            rb.mass = newMass;
        }
    }
    public void RestoreMass()
    {
        if (rb != null)
        {
            rb.mass = initialMass;
        }
    }
```

Методы OnTriggerEnter2D() и OnTriggerExit2D() управляют взаимодействием с NPC. Когда игрок входит в зону NPC и диалог не активен, появляется значок взаимодействия. При выходе из зоны триггера значок скрывается. Переменная canInteract контролирует возможность взаимодействия с NPC. Метод SetIconDialogueActive() управляет видимостью значка диалога. Когда диалог активен, все действия персонажа

прекращаются. Анимация движения останавливается. Если в этот момент на экране отображается иконка диалога, она отключается. Также отключатся взаимодействие с другими объектами, ведь персонаж занят диалогом.

```
    private void OnTriggerEnter2D(Collider2D other)
        {
            if (other.CompareTag("NPC") && !dialogueManager.dialogueIsPlaying)
            {
                canInteract = true;
                SetIconDialogueActive(true);
            }
        }

        private void OnTriggerExit2D(Collider2D other)
        {
            if (other.CompareTag("NPC"))
            {
                canInteract = false;
                SetIconDialogueActive(false);
            }
      }
```

Обработка смерти персонажа происходит в функции Die(). Она отключает скрипт и коллайдер, чтобы прекратить ввод и взаимодействия. Движение персонажа полностью останавливается путем обнуления его скорости Rigidbody2D. Если есть аниматор, запускается анимация смерти, и после задержки вызывается RestartLevel(), перезагружающий сцену и уровень. Если аниматора нет, уровень перезапускается сразу, с выводом

ошибки в консоль. Функция RestartLevel() просто перезагружает сцену, возвращая игрока к началу уровня.

```
    public void Die()
        {
            enabled = false;
            if (characterCollider != null) { characterCollider.enabled = false; }
            if (rb != null) { rb.linearVelocity = Vector2.zero; }

            if (animator != null)
            {
                animator.SetTrigger("Death");
                Invoke("RestartLevel", deathAnimationDelay);
            }
            else
            {
                Debug.LogError("Animator component not found!");
                RestartLevel();
            }
      }
    void RestartLevel()
        {
SceneManager.LoadScene(SceneManager.GetActiveScene().name);
        }
```

OnDisable() сохраняет состояние переменной canAttack в PlayerPrefs при отключении скрипта. DisableAttack() предназначен для отключения возможности атаковать у персонажа.

```
void OnDisable()
    {
        PlayerPrefs.SetInt("CanAttack", canAttack ? 1 : 0);
        PlayerPrefs.Save();
        Debug.Log("CanAttack saved: " + canAttack);
    }
public void DisableAttack()
    {
        canAttack = false;
        PlayerPrefs.SetInt("CanAttack", 0);
        PlayerPrefs.Save();
        Debug.Log("Attack disabled via DisableAttack()");
    }
```

Метод EnableAttack() используется, чтобы включить возможность атаковать для персонажа после завершения диалога. При его вызове переменная canAttack, определяющая возможность атаки, устанавливается в значение true. Для сохранения этого состояния между сессиями игры, значение canAttack сохраняется в PlayerPrefs.

```
  public void EnableAttack()
    {
        canAttack = true;
        PlayerPrefs.SetInt("CanAttack", 1);
        PlayerPrefs.Save();
        Debug.Log("Attack enabled via EnableAttack()");
```

```
    }
```

«CharacterAnimations».

Скрипт «CharacterAnimations» управляет анимациями персонажа в Unity. Он получает компонент Animator, устанавливает булево значение Moving, чтобы указать, движется персонаж или нет, и запускает триггеры для анимаций прыжка (Jump), атаки (Attack), получения урона (Hit), лечения (Heal) и смерти (Die). Функция Attack() включает перезарядку, чтобы предотвратить спам атаками. Функция Die() отключает компоненты движения, коллайдер и физику, чтобы остановить персонажа после смерти. Все триггеры хранятся в виде хэшей для оптимизации.

«PlayerHealth».

PlayerHealth управляет здоровьем персонажа, определяя его максимум при старте и поддерживая текущее значение. Он получает урон в TakeDamage (воспроизводя звук и анимацию), лечится в Heal, и обновляет UI через UpdateHeart, отображая изображение здоровья игроку.

```
    public void TakeDamage(int damage)
        {
            Debug.Log("TakeDamage called with damage = " + damage);
            if (damage <= 0)
            {
                Debug.LogWarning("Damage value is not positive!");
                return;
            }
            currentHealth -= damage;
            currentHealth = Mathf.Max(currentHealth, 0);
```

```
            Debug.Log("Player took " + damage + " damage! Current
health: " + currentHealth);
            soundController.PlayDamageSound();

            if (healthBar != null)
            {
                healthBar.value = currentHealth;
            }
            if (characterAnimations != null)
            {
                characterAnimations.Hit();
            }
            else
            {
                Debug.LogError("CharacterAnimations  is  null!
Cannot play Hit animation.");
            }
            if (currentHealth <= 0)
            {
                Die();
            }
            UpdateHeart();
        }
    public void Heal(int amount)
        {
            currentHealth += amount;
```

```
            currentHealth   =   Mathf.Clamp(currentHealth,   0,
maxHealth);
            Debug.Log("Player healed for " + amount + " health!
Current health: " + currentHealth);
            if (healthBar != null)
            {
                healthBar.value = currentHealth;
            }
            soundController.PlayHealSound();
            UpdateHeart();
        }
```

При падении здоровья до нуля, вызывается Die, который отключает управление, воспроизводит анимацию смерти, и перезагружает сцену после задержки через Respawn.

«PlayerSoundCntroller».

«PlayerSoundCntroller» - скрипт для активации и управлению звуками героя, благодаря компоненту Audio Source, который воспроизводит звуки. Скрипт тесно связан с Character Movement, чтобы отслеживать действия игрока, такие как прыжки и атаки.

Когда игрок прыгает, вызывается функция PlayJumpSound, которая воспроизводит звук прыжка. Аналогично, функция PlayAttackSound отвечает за звук атаки, PlayDamageSound - за звук получения урона, а PlayHealSound - за звук лечения. Каждая функция использует audioSource.PlayOneShot(), чтобы воспроизвести соответствующий звуковой эффект один раз.

1.4.4 Создание новой силы героя

Скрипт Ghost – это ключевой элемент в развитии героя. После диалога призрак исчезает, оставив герою новую силу для противостояния

надвигающейся тьме.

В методе Start, скрипт загружает состояние диалога из PlayerPrefs, получает ссылку на компонент Animator визуального объекта призрака и делает призрака изначально невидимым. Он также подписывается на события начала и окончания диалога в DialogueManager, чтобы реагировать на их наступление.

```
    void Start()
        {
            _dialogueStarted                                    =
PlayerPrefs.GetInt(_dialogueStartedKey, 0) == 1;

            if (dialogueManager == null)
            {
                Debug.LogError("DialogueManager is not assigned
in Ghost script!");
                enabled = false;
                return;
            }

            if (inkFile == null)
            {
                Debug.LogError("No ink file assigned to the
Ghost!");
                enabled = false;
                return;
            }

            if (ghostVisuals != null)
            {
                _animator                                       =
ghostVisuals.GetComponent<Animator>();
            }
            SetGhostVisibility(false);
```

```
            dialogueManager.OnDialogueStarted                    +=
OnDialogueStarted;
            dialogueManager.OnDialogueEnded += OnDialogueEnded;
        }
```

В методе OnDestroy, скрипт отписывается от событий DialogueManager, чтобы избежать утечек памяти.

```
    void OnDestroy()
        {
            if (dialogueManager != null)
            {
                dialogueManager.OnDialogueStarted              -=
OnDialogueStarted;
                dialogueManager.OnDialogueEnded                -=
OnDialogueEnded;
            }
        }
```

Метод TryStartDialogue пытается запустить диалог с призраком. Если диалог еще не был начат, метод StartDialogue запускает его, устанавливает флаг _dialogueStarted и сохраняет это состояние в PlayerPrefs. Он также включает возможность атаки у персонажа.

```
        public void TryStartDialogue()
        {
            if (!_dialogueStarted)
            {
                StartDialogue();
                _dialogueStarted = true;
                PlayerPrefs.SetInt(_dialogueStartedKey, 1);
                PlayerPrefs.Save();

                if (characterMovement != null)
                {
                    characterMovement.EnableAttack();
                }
```

```
                else
                {
                    Debug.LogError("CharacterMovement component
not found!");
                }
            }
        }
```

Внутри StartDialogue, если DialogueManager и inkFile назначены, запускается диалог, используя метод EnterDialogueMode из DialogueManager.

Когда диалог начинается, метод OnDialogueStarted делает призрака видимым. Он сохраняет текущее время Time.timeScale, а затем изменяет его, чтобы замедлить время во время диалога.

```
        private void OnDialogueStarted()
        {
            _originalTimeScale = Time.timeScale;
            Time.timeScale = timeScaleDuringDialogue;
            Time.fixedDeltaTime = _originalTimeScale * 0.02f;
            SetGhostVisibility(true);
        }
```

После окончания диалога вызывается метод OnDialogueEnded. Он восстанавливает исходный Time.timeScale и запускает анимацию смерти призрака, если Animator установлен. Затем призрак уничтожается.

```
    private void OnDialogueEnded()
          {
               Time.timeScale = _originalTimeScale;
               Time.fixedDeltaTime = _originalTimeScale * 0.02f;

               if (_animator != null)
               {
                    _animator.Play(deathAnimationName);
                    Debug.Log("Анимация смерти запущена для
дочернего объекта");
```

```
            }
            else
            {
                  Debug.LogError("Animator is null");
            }
            Destroy(gameObject, destroyDelay);
      }
```

1.4.5 Создание босса игры

Босс игры управляется скриптом «Boss». Он координирует действия и реакции злодея в зависимости от ситуации и состояния игрока.

Метод Start() подготавливает босса к бою. Он инициализирует здоровье, настраивает аниматор, получает ссылки на коллайдер. Также ищет игрока в сцене и подключается к скрипту диалога босса (bossDialogue). Затем настраивает полоску здоровья. Сохраняет оригинальное значение Time.timeScale для возможности изменения времени.

Каждый кадр Update() следит за игроком и решает, когда атаковать, проверяя, жив ли босс, находится ли игрок в поле зрения и истекло ли время перезарядки атаки. Если босс во второй фазе, запускает функцию SpawnMonsterEye() с небольшими интервалами.

```
    void Update()
          {
            if (isDead) return;
            _attackTimer += Time.deltaTime;
            if (CanSeePlayer() && _attackTimer >=
attackCooldown)
            {
                Attack();
                _attackTimer = 0f;
            }

            if (_isPhase2)
            {
```

```
                if (Time.time >= _nextMonsterEyeSpawnTime)
                {
                    SpawnMonsterEye();
                    _nextMonsterEyeSpawnTime = Time.time +
monsterEyeSpawnInterval;
                }
            }
        }
```

CanSeePlayer() проверяет, находится ли игрок в пределах досягаемости босса. Если игрок находится в зоне видимости босса (visionOrigin), вычисляется расстояние до игрока и возвращается true, иначе функция выводит предупреждение и возвращает false.

```
    bool CanSeePlayer()
          {
            if (_player == null || visionOrigin == null)
            {
                Debug.LogWarning("Player or VisionOrigin is
null!");
                return false;
            }

            float distanceToPlayer =
Vector2.Distance(visionOrigin.position,
_player.transform.position);

            bool seesPlayer = distanceToPlayer <= visionRange;
            return seesPlayer;
        }
```

Функция Attack() отвечает за выбор и выполнение атаки босса. Прежде чем атаковать, она проверяет можно ли сейчас атаковать. Если все условия соблюдены, то с вероятностью 50% босс использует атаку «Fireball», проигрывая анимацию соответствующей атаки и вызывая функцию CreateFireball(). В противном случае босс атакует «Knockback», проигрывая

анимацию и запуская корутину KnockbackAttackRoutine() для выполнения серии атак. После завершения анимации атаки, вызывается функция EndAttack(), чтобы сбросить состояние атаки босса.

```
    void Attack()
          {
            if (isAttacking) return;

            if (bossDialogue == null ||
!bossDialogue.IsDialoguePlaying)
            {
                float randomValue =
UnityEngine.Random.Range(0f, 1f);

                if (randomValue < 0.5f) // 50% Fireball
                {
                    Debug.Log("Fireball Attack - Triggering
Attack animation");
                    animator.SetTrigger("Attack");
                    Invoke("CreateFireball",
attackAnimationDelay);
                }
                else // 50% Knockback
                {
                    Debug.Log("Knockback Attack - Triggering
Knockback animation");
                    animator.SetTrigger("Knockback");
                    StartCoroutine(KnockbackAttackRoutine());
                }

                Invoke("EndAttack", attackAnimationDelay);
            }
            else
            {
```

```
                Debug.Log("Boss is not attacking (dialogue is
playing).");
                // Атака не выполняется, так как идет диалог
            }
        }
```

TakeDamage описывает реакцию босса на получение урона. Прежде чем получить урон, функция проверяет, жив ли босс и не находится ли в состоянии неуязвимости. Если оба условия ложны, функция уменьшает запас здоровья босса на величину полученного урона. Обновляется индикатор здоровья, и запускается анимация получения урона. Если запас здоровья опускается ниже значения для второй фазы и босс еще не перешел в неё, запускается корутина Phase2Transition(), для того чтобы босс стал сильнее, а битва интереснее. Если же запас здоровья опускается до нуля или ниже, вызывается функция Die(), оповещающая о смерти босса, и отключается его отображение здоровья.

```
    public void TakeDamage(int damage)
          {
            if (isDead) return;
            if (isInvincible) return;

            currentHealth -= damage;
            Debug.Log(gameObject.name + " took " + damage + "
damage! Current health: " + currentHealth);

            healthBar.value = currentHealth;

            animator.SetTrigger("IsHit");

            if (currentHealth <= phase2HealthThreshold &&
!_isPhase2)
            {
                // Переход во вторую фазу
                StartCoroutine(Phase2Transition());
```

```
        }

        if (currentHealth <= 0)
        {
            Die();
            if (healthBarObject != null)
            {
                healthBarObject.SetActive(false);
            }
        }
    }
```

Сопроцесс Phase2Transition() отвечает за плавный и кинематографичный переход босса во вторую, более сложную фазу боя. Сначала устанавливаются флаги, указывающие на переход во вторую фазу, а также временную неуязвимость и невозможность атаковать. Затем запускается анимация «KnockbackBig», изображающая сильный отброс героя назад для того, чтобы игрок попробовал сразится в усложненном режиме. После окончания анимации и небольшой задержки создается объект KnockbackBig большая отбрасывающая волна энергии. После чего босс возвращается в нормальное состояние, с возможностью атаковать и быть уязвимым, а также активируется механизм спавна дополнительных врагов.

```
    private IEnumerator Phase2Transition()
    {
        _isPhase2 = true; // флаг, что босс во второй фазе
        isInvincible = true; // Босс становится неуязвимым
        isAttacking = true; // Босс не может атаковать во
время анимации KnockbackBig

        animator.SetTrigger("KnockbackBig");

        float animationLength = 0f;
```

```
            AnimationClip[] clips =
animator.runtimeAnimatorController.animationClips;
            foreach (AnimationClip clip in clips)
            {
                if (clip.name == "KnockbackBig")
                {
                    animationLength = clip.length;
                    break;
                }
            }

            yield return new WaitForSeconds(animationLength);

            float delayBeforeKnockbackBig = 1.2f;
            yield return new
WaitForSeconds(delayBeforeKnockbackBig);

            // KnockbackBig только один раз
            CreateKnockbackBig();

            isAttacking = false; // Босс снова может атаковать
            isInvincible = false; // Босс снова становится
уязвимым

            _nextMonsterEyeSpawnTime = Time.time;
        }
```

SpawnMonsterEye() эта функция отвечает за вызов подручных монстров «MonsterEye». Если у босса определен префаб monsterEyePrefab и точки спавна monsterEyeSpawnPoints, то выбирается случайная точка спавна, и создаётся экземпляр monsterEyePrefab в этой точке.

```
        private void SpawnMonsterEye()
        {
            if (monsterEyePrefab != null &&
monsterEyeSpawnPoints.Length > 0)
```

```
            {
                Transform spawnPoint =
monsterEyeSpawnPoints[UnityEngine.Random.Range(0,
monsterEyeSpawnPoints.Length)];
                Instantiate(monsterEyePrefab,
spawnPoint.position, spawnPoint.rotation);
            }
            else
            {
                Debug.LogWarning("MonsterEyePrefab is null or
no spawn points defined!");
            }
        }
```

Метод Die() срабатывает, когда здоровье босса достигает нуля. Он устанавливает флаг смерти, отключает коллайдер босса, чтобы предотвратить дальнейшее получение урона, и запускает анимацию смерти. У босса есть предсмертная фраза, которая отображается над ним во время смерти. После этого запускается сопроцесс SlowMotionDeath(), чтобы замедлить красиво время перед окончательным уничтожением объекта.

```
    void Die()
        {
            isDead = true;
            bossCollider.enabled = false;
            animator.SetTrigger("IsDead");

            // Показываем фразу смерти перед замедлением
времени
            if (bossDialogue != null &&
!string.IsNullOrEmpty(bossDialogue.deathTaunt))
            {

bossDialogue.ShowFloatingText(bossDialogue.deathTaunt);
            }
```

```
            StartCoroutine(SlowMotionDeath()); // Замедляем
время
        }
```

SlowMotionDeath() это сопроцесс, который создает эффект замедленной смерти босса. Он сохраняет текущую скорость анимации, затем замедляет время, фиксированный шаг времени и скорость анимации. После определенного времени, проведенного в замедленном режиме, восстанавливается нормальная скорость времени и анимации и объект босса уничтожается. Наконец, вызывается событие OnBossDeath, чтобы уведомить другие части игры о смерти босса чтобы предоставить герою проход дальше.

```
        private System.Collections.IEnumerator
SlowMotionDeath()
        {
            float originalAnimationSpeed = animator.speed;

            // замедление времени и анимации
            Time.timeScale = slowMotionTimeScale;
            Time.fixedDeltaTime = 0.02F * Time.timeScale;
            animator.speed = slowMotionTimeScale;
            yield return new
WaitForSecondsRealtime(slowMotionDuration);

            // восстанавление нормального времени
            Time.timeScale = originalTimeScale;
            Time.fixedDeltaTime = 0.02F;
            animator.speed = originalAnimationSpeed;

            Debug.Log("Boss is dying! Destroying GameObject.");
            Destroy(gameObject);

            if (OnBossDeath != null)
```

```
            {
                OnBossDeath();
            }
        }
```

1.4.6 Система уровней

В разработанной игре система доступа к уровням строится на основе последовательного прохождения испытаний и выполнения определенных условий на каждом этапе. Каждый уровень открывает новые возможности и механики, делая игровой процесс разнообразным и увлекательным. Для проработки этой системы, а также дизайна уровней, использовались принципы, описанные Джесси Шеллом в «Искусство геймдизайна» [7].

За проверку доступов к уровням отвечает скрипт «LevelUnlocker». Он основывается на данных, хранящихся в классе для сохранений PlayerPrefs. Сохранение прогресса с использованием PlayerPrefs обеспечивает непрерывность игрового процесса и позволяет игроку возвращаться к игре в любое удобное время. Для каждого уровня существует свой ключ в PlayerPrefs, который указывает, выполнены ли условия для его открытия. Для открытия двери второго уровня проверяется, установлено ли значение PlayerPrefs с ключом FlowerBloom равным 1. Если да, то это означает, что игрок выполнил определенное действие в игре, и уровень становится доступен. Если условия выполнены, вызывается функция UnlockLevel2(), которая делает дверь открытой, позволяя игроку перейти на этот уровень. Аналогичные проверки и разблокировки происходят для уровней 3 и 4.

```
    private void CheckLevelUnlocks()
          {
            //Разблокировка Level 2
            if (PlayerPrefs.GetInt("FlowerBloom") == 1)
            {
                UnlockLevel2();
            }
            else
```

```
        {
            Debug.Log("Flower is not yet blooming.");
        }

        // Разблокировка Level 3
        if (PlayerPrefs.GetInt("Completelevel2") == 1)
        {
            UnlockLevel3();
        }
        else
        {
            Debug.Log("Level 2 is not yet complete.");
        }

        //Разблокировка Level 4
        if (PlayerPrefs.GetInt("Completelevel3") == 1)
        {
            UnlockLevel4();
        }
        else
        {
            Debug.Log("Level 3 is not yet complete.");
        }

        PlayerPrefs.Save();
    }
```

Первый уровень знакомит игрока с базовыми механиками взаимодействия с миром. Ключевым элементом является цветок, логика которого прописана в скрипте «Flower». Цветок необходимо оживить, применив к нему три капли разного цвета. Успешное применение всех капель активирует метод Bloom(), который запускает анимацию цветения и воспроизводит звук. Условие прохождения первого уровня - цветение цветка. Это состояние сохраняется в PlayerPrefs с ключом «FlowerBloom»,

что обеспечивает сохранение прогресса между игровыми сессиями. Разблокировка второго уровня происходит тогда, когда ключ «FlowerBloom» в PlayerPrefs = 1.

```
    private void Bloom()
        {
            Debug.Log("Flower is blooming!");
            _animator.SetTrigger("Bloom");
            IsBlooming = true;
            SaveState();

            if (_bloomSound != null)
            {
                _bloomSound.Play();
            }
        }

        private void SaveState()
        {
            PlayerPrefs.SetInt("FlowerBloom", IsBlooming ? 1 :
0);
            PlayerPrefs.Save();
        }
```

Второй уровень, в отличие от первого, нацелен на проверку внимательности. Здесь игрок должен собрать определенное количество данных, распределяя их по катигориям. Ключевым показателем прохождения уровня является переменная levelComplete в скрипте «ScoreManager», которая становится истинной при выполнении необходимых условий. После успешного завершения уровня отключается генерация новых объектов. При завершении уровня активируется панель, отображающая победу на данном уровне. Статус прохождения сохраняется с помощью ключа «Completelevel2» в классе PlayerPrefs. Третий уровень становится доступным если ключ «Completelevel2» = 1.

```
        public void IncreaseScore()
        {
            if (!levelComplete)
            {
                currentScore++;
                Debug.Log("Score: " + currentScore);

                if (currentScore >= targetScore)
                {
                    LevelComplete();
                }
            }
        }

        private void LevelComplete()
        {
            levelComplete = true;
            Debug.Log("Level 2 Complete!");
            SaveState();
            if (dataGenerator != null)
            {
                dataGenerator.StopSpawning();
            }
            if (completedPanel != null)
            {
                completedPanel.SetActive(true);
            }
            else
            {
                Debug.LogWarning("Completed Panel is not
assigned!");
            }
        }
        public bool IsLevelComplete()
        {
```

```
            return levelComplete;
        }
        private void SaveState()
        {
            PlayerPrefs.SetInt("Completelevel2", 1);
            PlayerPrefs.Save();
            Debug.Log("Level 2 state saved!");
        }
```

Третий уровень предлагает игроку решить головоломку, связанную с ядрами процессора. Основная задача - привести все интерактивные кнопки к одному белому цвету. Скрипт «ButtonStateChecker» отслеживает состояние каждой кнопки и, при достижении желаемого результата, вызывает метод LevelComplete(). Прогресс игрока на этом уровне также сохраняется в PlayerPrefs вот с таким ключом: levelName + "Complete". Условие для активации четвертого уровня: "Completelevel3" = 1.

```
    private void CheckButtonStates ()
          {
            foreach (GameObject buttonObject in buttonsToCheck)
            LevelComplete();
        }
        private void LevelComplete()
        {
            levelComplete = true;
            Debug.Log("Level 3 Complete! All buttons are
white.");
            SaveLevelState();
        }
        private void SaveLevelState()
        {
            PlayerPrefs.SetInt(levelName + "Complete", 1);
            PlayerPrefs.Save();
            Debug.Log("Level " + levelName + " state saved!");
        }
```

Четвертый уровень, завершающий собой игру, представляет финальным испытанием - битву с боссом. Ключевым условием для завершения уровня является уничтожение босса, о чем сигнализирует событие OnBossDeath() в скрипте «Boss».

1.5 Тестирование

Тестирование - это процесс оценки качества игры, направленный на выявление и устранение потенциальных проблем. Оно включает в себя проверку различных аспектов игры, таких как функциональность, стабильность и удобство использования. На рисунке 25 показан пример тестирования главного меню.

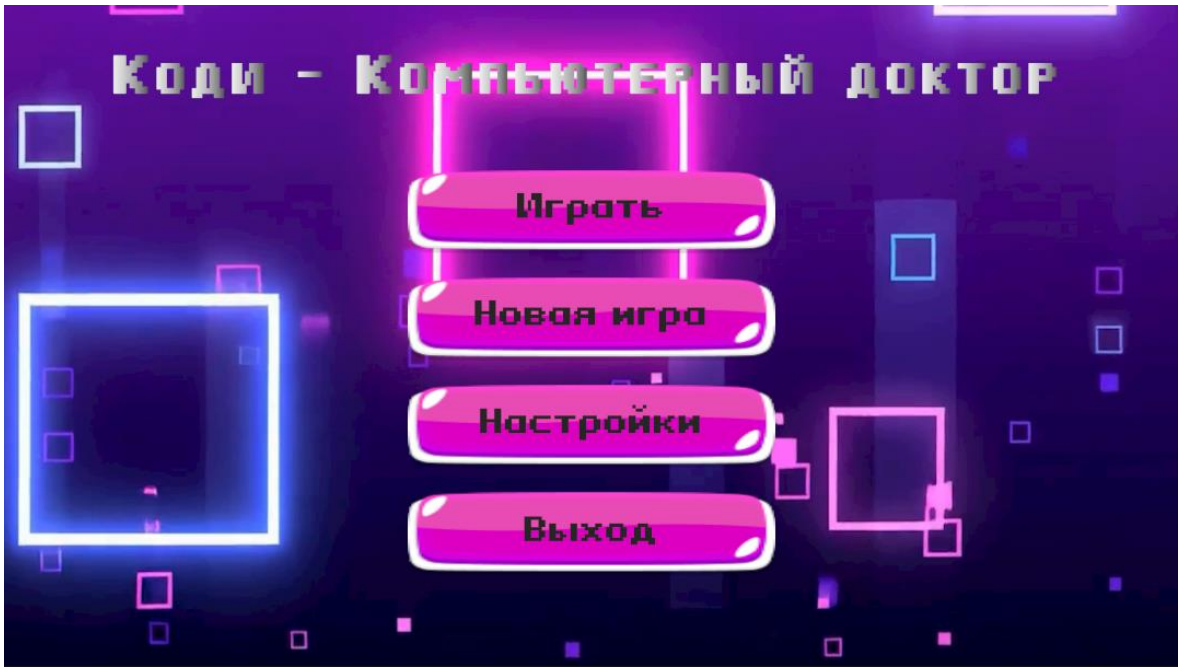


Рисунок 25 – Главное меню

На экране загрузки, показанном на рисунке 26, отображается прогресс загрузки уровней. Тестирование подтвердило стабильную работу экрана загрузки уровней. Процесс загрузки отображается корректно, без заметных задержек и зависаний.

Рисунок 26 – Экран загрузки

Первая локация игры, представленная на рисунке 27, называется «Материнская плата», является отправной точкой приключения. Взаимодействие с окружением работает как задумано, доступ ко всем уровням корректен. Критических ошибок не обнаружено.

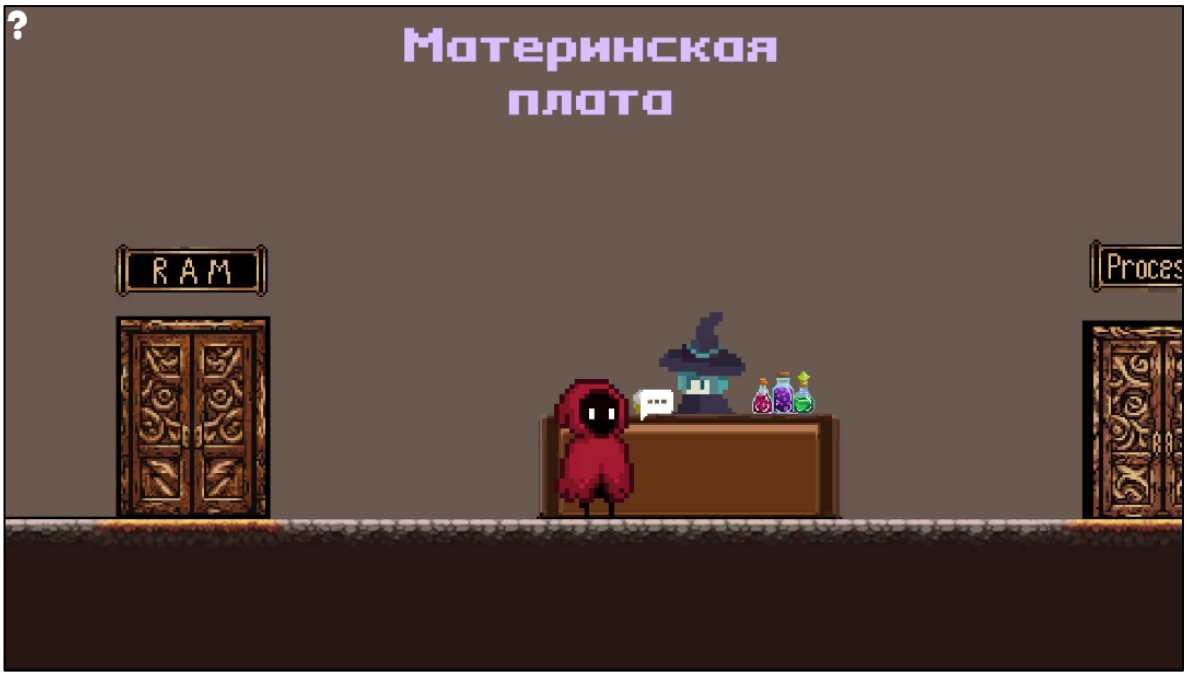


Рисунок 27 – Первая локация «Материнская плата»

Первый уровень игры «Видеокарта» представлен на рисунке 28. Он служит знакомством с основными механиками взаимодействия. Он фокусируется на обучении игрока через интуитивно понятные действия и задачи. Скрипты и триггеры функционируют исправно.

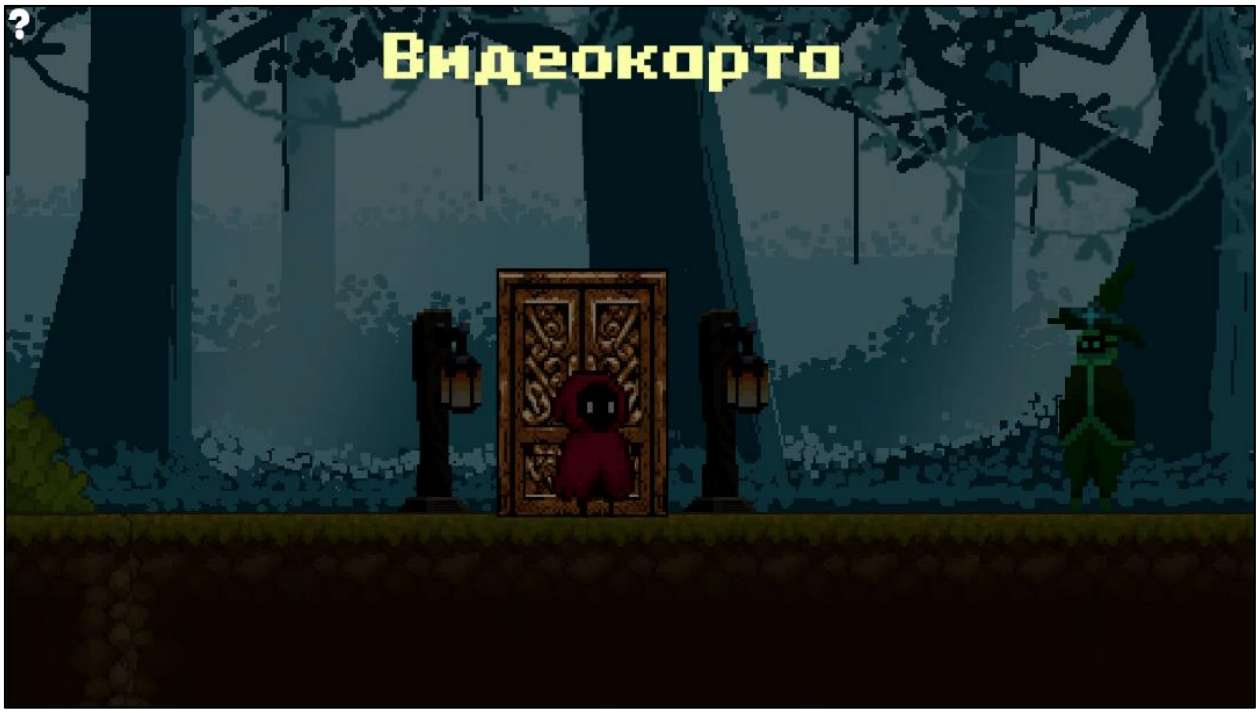


Рисунок 28 – Первый уровень «Видеокарта»

На втором уровне «Оперативная память» игроку предстоит проявить внимательность в новой мини-игре, где нужно правильно распределять поступающие данные по четырем категориям. Важно не только быстро реагировать, но и не допускать ошибок, так как неправильное распределение приводит к уничтожению данных. Механика логичная и интуитивно понятная.

Третий уровень «Процессор» (рисунок 30). Здесь, чтобы продвинуться дальше, необходимо мыслить логически и решить головоломку. Основная задача на этом уровне – правильно активировать последовательность кнопок.

Четвертый уровень, «Жесткий диск» (рисунок 31-33), - заключительный этап перед финалом игры, где герой обретает новую силу – способность очищать зараженные данные. Ему предстоит преодолеть три различные локации, каждая из которых предлагает свои испытания от уничтожения

зараженных данных, представленных в виде монстров, до избегания ловушек, поджидающих на каждом шагу. Кульминацией этого уровня и всей игры станет заключительная битва в последней локации с главным боссом, олицетворяющий вирус в компьютере. Данный уровень сбалансированной сложности, безупречно работают все игровые механики, включая новые.

Рисунок 29 – Второй уровень «Оперативная память»

Рисунок 30 – Третий уровень «Процессор»

Рисунок 31 – Четвертый уровень «Жесткий диск» первая локация

Рисунок 32 – Четвертый уровень «Жесткий диск» вторая локация

Титры игры изображены на рисунке 34. Они содержат краткий пересказ всего приключения, которое пережил главный герой. В конце представленно имя разработчика. Титры долго отображаются для прочтения, но не слишком затянуто, чтобы не наскучить игроку.

Рисунок 33 – Четвертый уровень «Жесткий диск» третья локация, комната босса

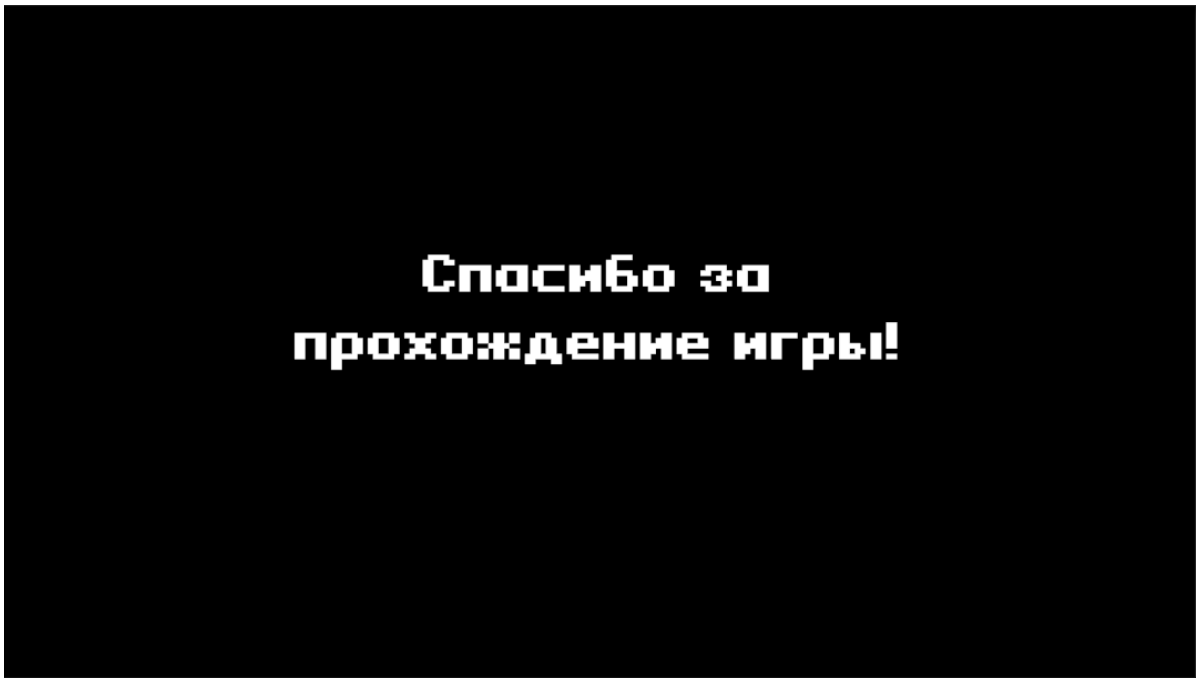


Рисунок 34 – Титры игры

2 Бзопасность жизнедеятельности

Уютное и правильно организованное рабочее место – это залог здоровья и высокой продуктивности. Обилие естественного света, проникающего в комнату, дарит энергию и тепло, создавая атмосферу, располагающую к творчеству.

Несмотря на современные достижения, такие как жидкокристаллические (LCD) и светодиодные (LED) мониторы, излучение электромагнитной энергии все еще присутствует. Хотя оно и значительно меньше, чем у устаревших электронно-лучевых трубок, рекомендуется соблюдать разумное расстояние от монитора и при возможности использовать специальные фильтры. Это поможет снизить воздействие электромагнитных волн на организм.

Необходимо избегать бликов на экране компьютера, так как они могут вызывать напряжение глаз. Важно расположить компьютер так, чтобы он не был обращен к окну или другому источнику света, который может отражаться на экране.

Не менее важна и эргономика. Монитор должен находиться на расстоянии вытянутой руки, на уровне глаз или чуть ниже. Такое положение позволит избежать напряжения в шее и глазах, предотвращая возникновение головных болей и чувства усталости. Высоту стола и стула необходимо отрегулировать так, чтобы обеспечить правильную осанку, снижая нагрузку на позвоночник. Стул с поддержкой поясницы и регулируемыми подлокотниками поможет поддерживать комфортное положение тела в течение длительного времени.

Для поддержания здорового микроклимата в помещениях с компьютерами необходимо проводить ежедневную влажную уборку и проветривать помещение каждый час. Рекомендуемая температура воздуха должна составлять +19°C - +21°C.

В современном мире, где электроника повсюду, безопасность становится критически важной. Необходимо тщательно проконтролировать подключение и заземление всех устройств. Правильно выполненное заземление предотвращает поражение электрическим током.

Важно помнить о балансе между игрой и отдыхом. Чрезмерное увлечение играми, особенно без перерывов, может привести к переутомлению, стрессу и даже проблемам со здоровьем. Важно следить за временем делать регулярные перерывы, чтобы избежать усталости и сохранить концентрацию. Правильная организация времени поможет избежать чувства перегруженности и получать удовольствие от игры без вреда для здоровья.

Заключение

В ходе выполнения данной работы была разработана 2D игра на платформе Unity с использованием языка программирования C#. Этот проект позволил больше узнать в области разработки игр, включая применение принципов объектно-ориентированного программирования, работу с движком Unity, создание игрового процесса, реализацию механик управления и взаимодействия. В ходе разработки был получен опыт для дальнейших разработок в области интерактивных развлечений.

В будущем игровое приложение планируется активно дорабатывать и расширять новыми функциями.

Список используемых источников

1. Как создать игру на Unity [Электронный ресурс]. URL: https://www.sravni.ru/kursy/info/kak-sozdat-igru-na-unity/ (дата обращения: 10.03.2025).

2. Форум Unity [Электронный ресурс]. URL: https://forum.unity.com/ (дата обращения: 13.04.2025).

3. Официальная документация Unity [Электронный ресурс]. URL: https://docs.unity3d.com/ (дата обращения: 29.02.2025).

4. Эндрю Стиллман, Дженнифер Грин «Head First C#», 2021. - 1000с.

5. Ink Documentation [Электронный ресурс]. URL: https://www.inklestudios.com/ink/ (дата обращения: 25.04.2025)

6. Asset Store Unity [Электронный ресурс]. URL: https://assetstore.unity.com/. (дата обращения: 19.03.2025).

7. Джесси Шелл «Искусство геймдизайна», 2019. - 528с.

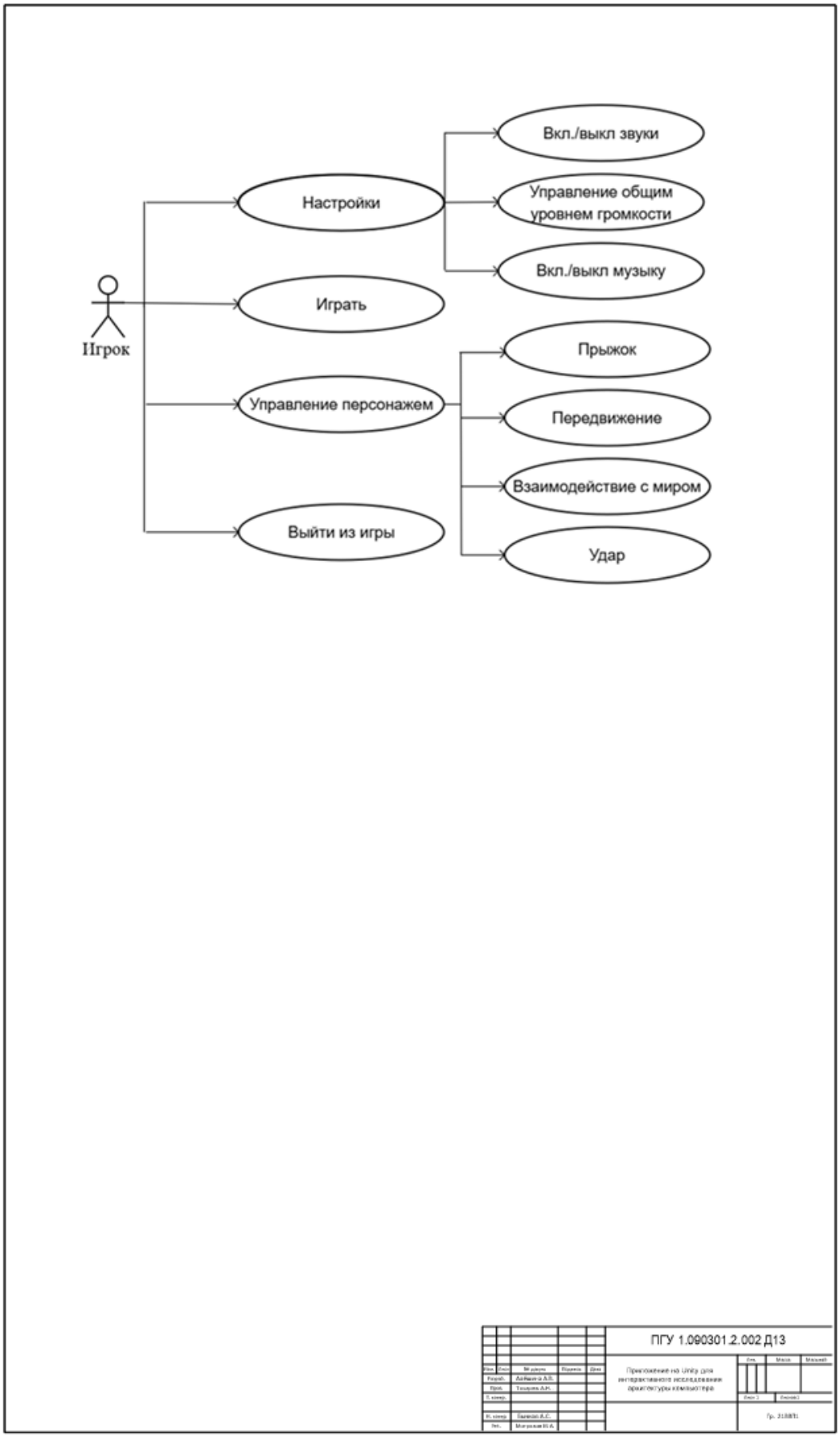
Приложение А

(Обязательное)

Приложение на Unity для интерактивного исследования архитектуры компьютера

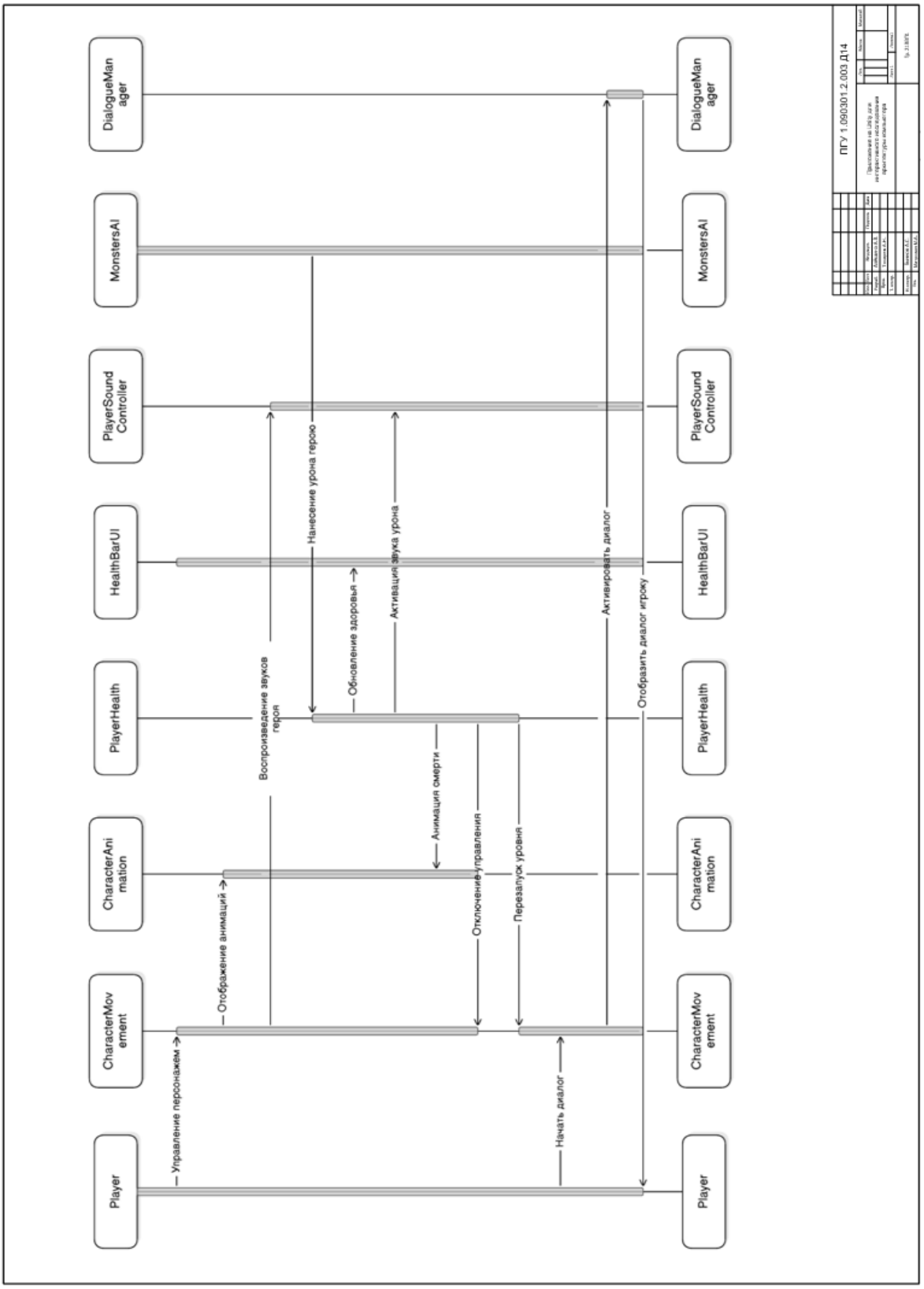
UML-диаграммы

Приложение А.1 – UML – диаграмма прецедентов пользователя

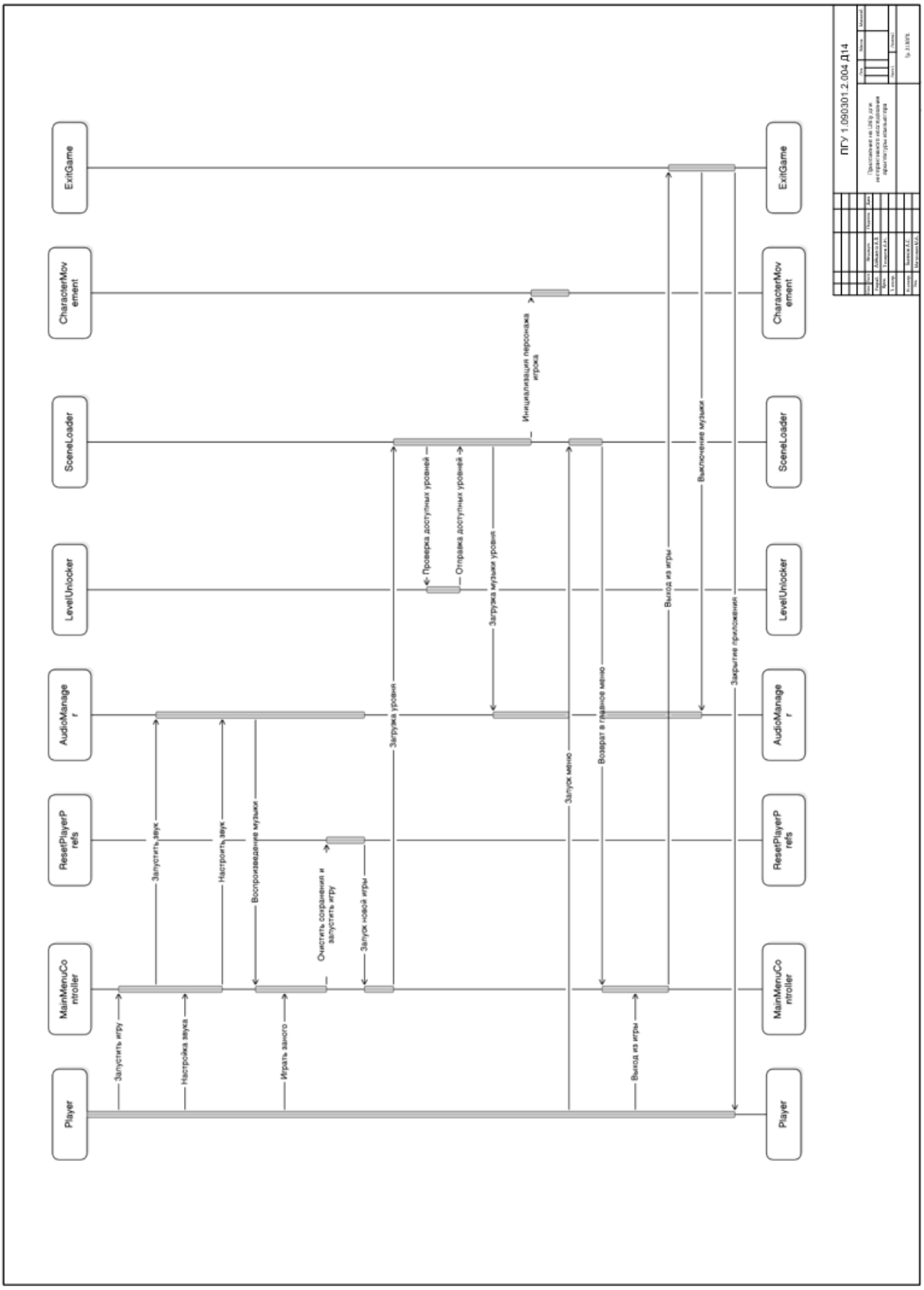


ПГУ 1.090301.2.002 Д13

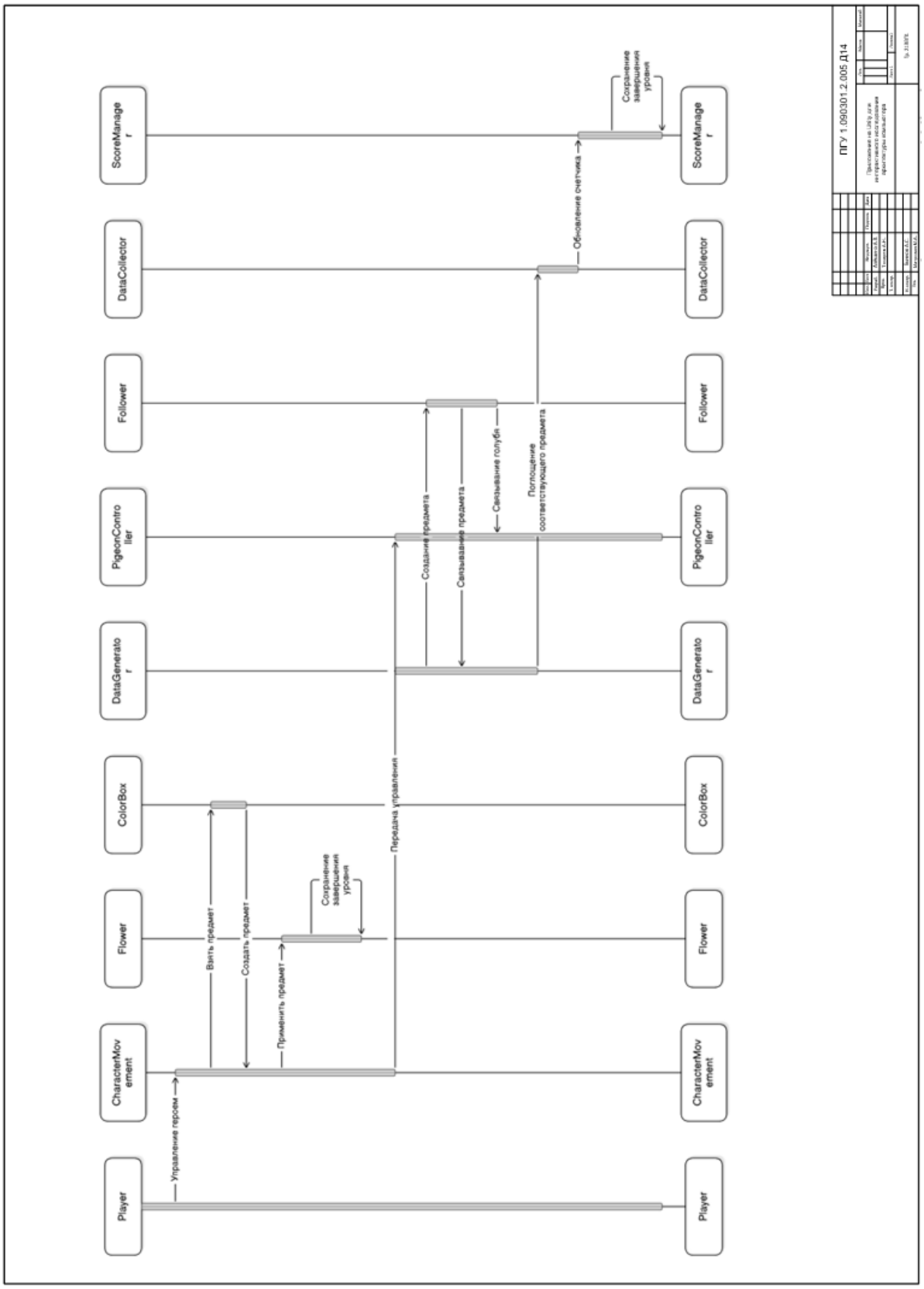
Приложение А.2 – UML – диаграмма последовательности взаимодействий героя



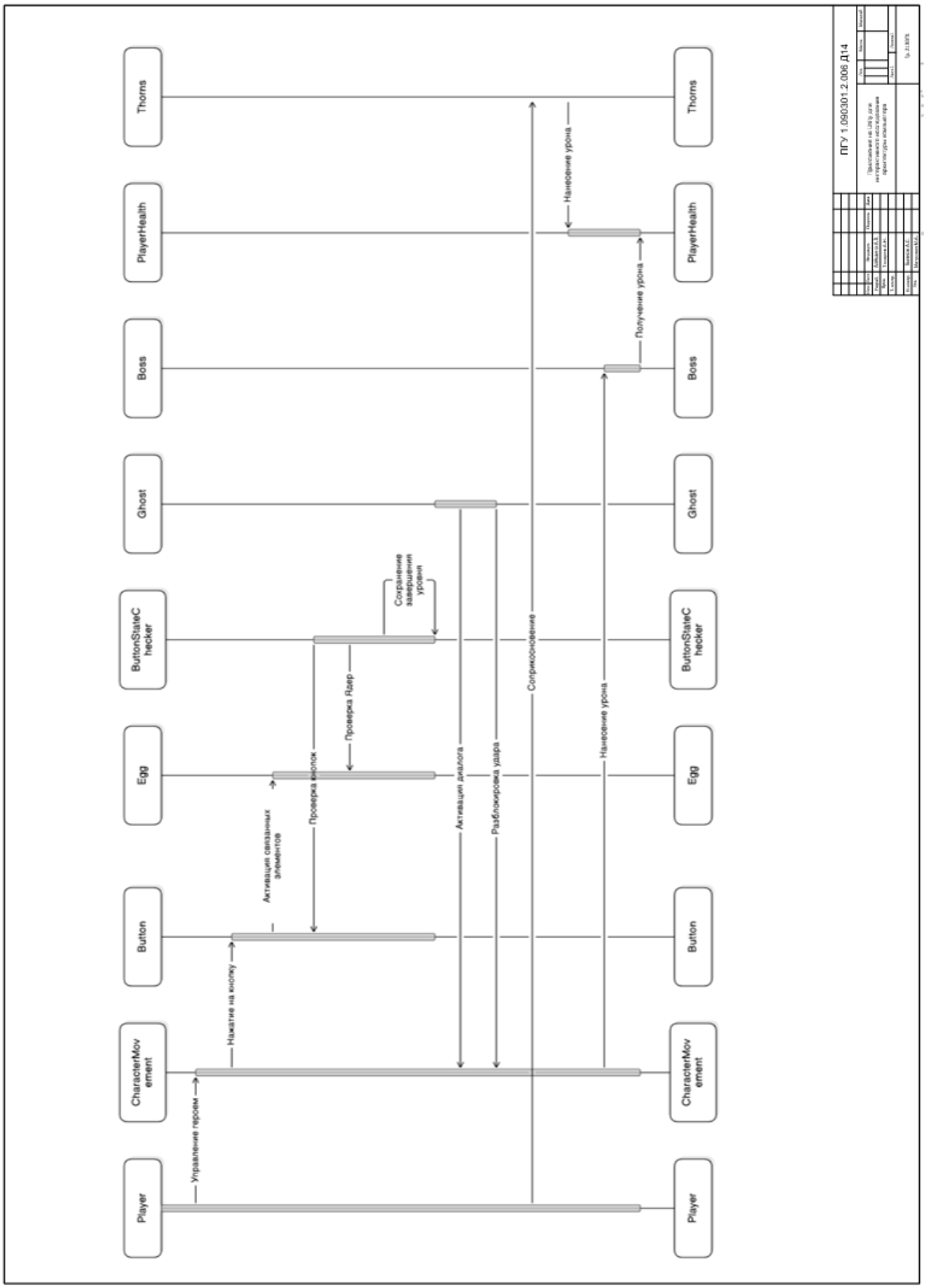
Приложение А.3 – UML – диаграмма последовательности интерфеса и загрузки



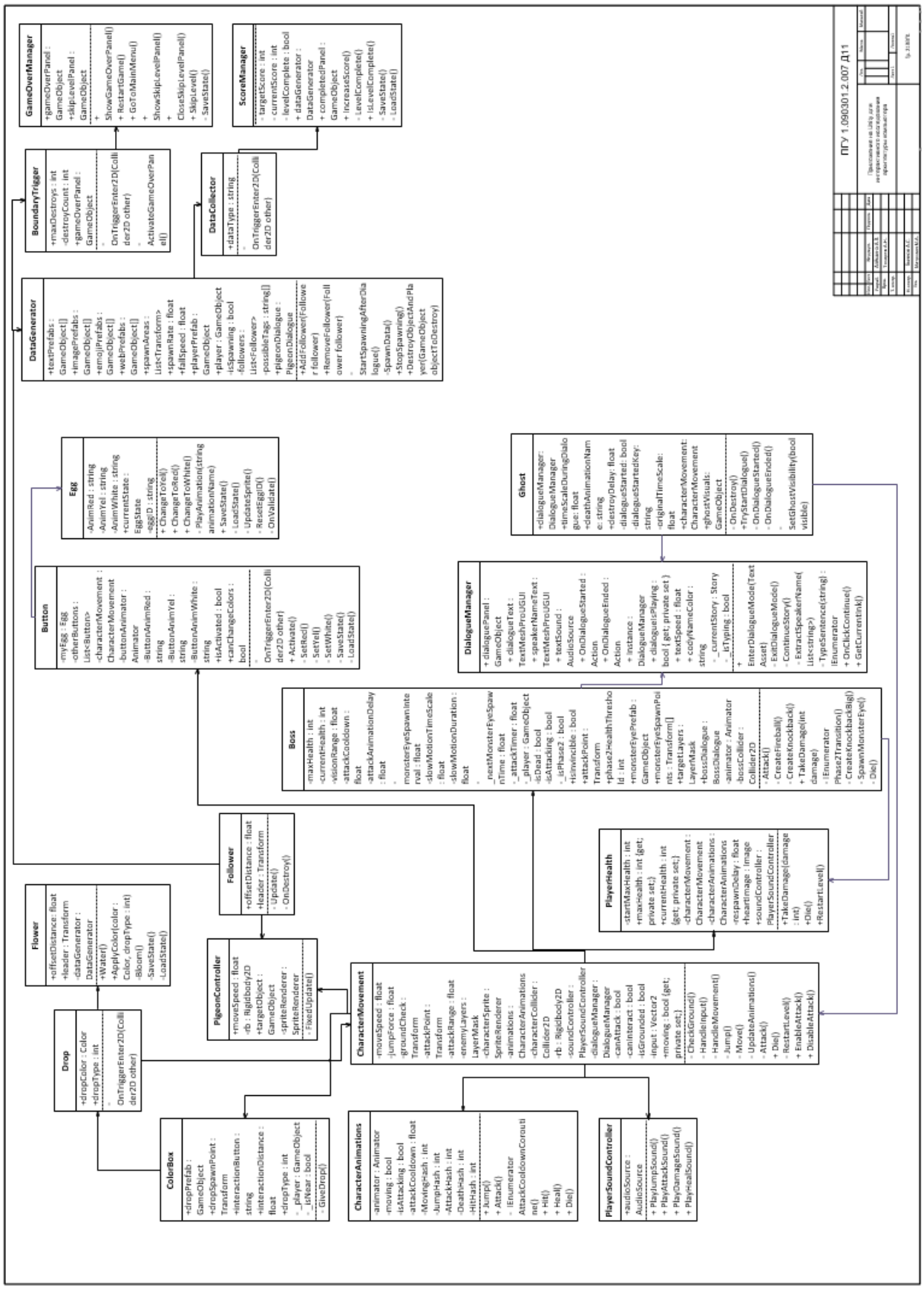
Приложение А.4 – UML – диаграмма последовательности прохождения первого и второго уровня



Приложение А.5 – UML – диаграмма последовательности прохождения первого и второго уровня



Приложение А.6 – UML – Диаграмма классов



Приложение Б

(Обязательное)

Приложение на Unity для интерактивного исследования архитектуры компьютера

Листинг программы

Скрипты героя:

Файл CharacterMovement.cs:

```
using UnityEngine;
using UnityEngine.SceneManagement;
using System;
using System.Collections;

public class CharacterMovement : MonoBehaviour
{
    [Header("Movement")]
    [SerializeField] private float moveSpeed = 5f;

    [Header("Jump")]
    [SerializeField] private float jumpForce = 10f;
    [SerializeField] private Transform groundCheck;
    [SerializeField] private float groundCheckRadius = 0.2f;
    [SerializeField] private LayerMask whatIsGround;

    private float initialMass;

    [Header("Attack")]
    [SerializeField] private Transform attackPoint; //Точка
атаки
    [SerializeField] private float attackRange = 0.5f;
//Радиус атаки
    [SerializeField] private LayerMask enemyLayers; //Слои
врагов, чтобы атаковать только их
    public bool canAttack = false;

    [Header("Knockback")]
    [SerializeField] private float knockbackForce = 5f;


    [Header("Death")]
    [SerializeField] private float deathAnimationDelay = 2f;
// Задержка перед перезапуском уровня

    [Header("Components")]
    private CharacterAnimations animations;
    [SerializeField] private SpriteRenderer characterSprite;
    private Animator animator;
    private Collider2D characterCollider;
    private Rigidbody2D rb;

    [Header("Interaction")]
    public GameObject iconDialogue; // Ссылка на иконку
диалога
    private bool canInteract = false; // Флаг, показывающий,
может ли игрок начать диалог
```

```
    [Header("Sound")]
    public PlayerSoundController soundController;


    public DialogueManager dialogueManager;

    private bool isGrounded;
    private Vector2 input;
    public bool moving { get; private set; }

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        animations =
GetComponentInChildren<CharacterAnimations>();
        animator = GetComponent<Animator>();
        characterCollider = GetComponent<Collider2D>();

        canAttack = PlayerPrefs.GetInt("CanAttack", 0) == 1;

        if (rb != null)
        {
            initialMass = rb.mass;
        }
        else
        {
            Debug.LogError("Rigidbody2D component not found on
this GameObject.");
        }

        if (rb == null)
        {
            Debug.LogError("Rigidbody2D component not
found!");
        }

        if (animations == null)
        {
            Debug.LogError("CharacterAnimations component not
found!");
        }

        if (characterSprite == null)
        {
            Debug.LogError("SpriteRenderer component not
found! Please assign it in the Inspector.");
        }

        if (attackPoint == null)
        {
```

```
            Debug.LogError("Attack Point component not found!
Please assign it in the Inspector.");
        }
        if (characterCollider == null)
        {
            Debug.LogError("Collider2D component not found on
" + gameObject.name);
        }
        if (soundController == null)
        {
            Debug.LogError("PlayerSoundController is not
assigned! Please assign it in the Inspector.");
            enabled = false;
            return;
        }
        if (iconDialogue != null)
        {
            iconDialogue.SetActive(false);
        }
    }

    void Update()
    {
        if (!dialogueManager.dialogueIsPlaying)
        {
            HandleInput();
            HandleMovement();
            UpdateAnimations();

            if (canAttack && Input.GetKeyDown(KeyCode.E))
            {
                Attack();
                soundController.PlayAttackSound();
            }
        }
        else
        {
            // Если диалог активен, останавливаем движение
            rb.linearVelocity = Vector2.zero;
            animations.Moving = false; // Останавливаем
анимацию движения и отключаем иконку диалога
            if (iconDialogue != null)
            {
                iconDialogue.SetActive(false);
            }
            canInteract = false;
        }
    }

    void FixedUpdate()
    {
        if (!dialogueManager.dialogueIsPlaying)
```

```
        {
            Move();
            CheckGround();
            UpdateAnimations();
        }
    }

    public void SetMass(float newMass)
    {
        if (rb != null)
        {
            rb.mass = newMass;
        }
    }

    // Метод для восстановления исходной массы
    public void RestoreMass()
    {
        if (rb != null)
        {
            rb.mass = initialMass;
        }
    }

    void CheckGround()
    {
        bool wasGrounded = isGrounded;
        isGrounded =
Physics2D.OverlapCircle(groundCheck.position,
groundCheckRadius, whatIsGround);
    }

    void HandleInput()
    {
        input.x = Input.GetAxis("Horizontal");

        if (Input.GetButtonDown("Jump"))
        {
            Jump();
            soundController.PlayJumpSound();
        }
    }


    void HandleMovement()
    {
        input.x = Input.GetAxis("Horizontal");
    }

    void Jump()
    {
        if (isGrounded)
```

```
        {
            PerformJump();
        }
    }

    void PerformJump()
    {
        rb.linearVelocity = new Vector2(rb.linearVelocity.x,
0f); // Сброс вертикальной скорости
        rb.AddForce(Vector2.up * jumpForce,
ForceMode2D.Impulse);
        animations.Jump();
    }

    void Move()
    {
        float horizontalMovement = input.x * moveSpeed;
        rb.linearVelocity = new Vector2(horizontalMovement,
rb.linearVelocity.y);

        if (input.x > 0)
        {
            characterSprite.flipX = false;
        }
        else if (input.x < 0)
        {
            characterSprite.flipX = true;
        }

        moving = Mathf.Abs(input.x) > 0.01f;

    }

    void UpdateAnimations()
    {
        animations.Moving = Mathf.Abs(rb.linearVelocity.x) >
0.1f;
    }

    void Attack()
    {
        animations.Attack();
        Debug.Log("Attack point: " + attackPoint.position);
        Collider2D[] hitTargets =
Physics2D.OverlapCircleAll(attackPoint.position, attackRange,
enemyLayers);

        foreach (Collider2D target in hitTargets)
        {
            Debug.Log("We hit " + target.name);

            if (hitTargets.Length == 0)
```

```
            {
                Debug.Log("Found 0 targets");
                return;
            }

            foreach (Collider2D enemy in hitTargets)
            {
                Debug.Log("Found target with layer: " +
LayerMask.LayerToName(enemy.gameObject.layer));

            }
            MonsterAI monsterAI =
target.GetComponent<MonsterAI>();
            if (monsterAI != null)
            {
                monsterAI.TakeDamage(1);
                continue;
            }

            MonstersAI monstersAI =
target.GetComponent<MonstersAI>();
            if (monstersAI != null)
            {
                monstersAI.TakeDamage(1);
                continue;
            }

            Boss boss = target.GetComponent<Boss>();
            if (boss != null)
            {
                boss.TakeDamage(1);
            }

            FlyingEye flyingEye =
target.GetComponent<FlyingEye>();
            if (flyingEye != null)
            {
                flyingEye.TakeDamage(1);
            }
        }
    }

    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Monster"))
        {
            Vector2 direction = ((Vector2)transform.position -
(Vector2)collision.transform.position).normalized;
            ApplyKnockback(direction);

            MonsterAI monsterAI =
collision.gameObject.GetComponent<MonsterAI>();
```

```
            if (monsterAI != null)
            {
                monsterAI.ApplyKnockback(-direction);
            }
        }
    }

    // Метод для вкл/выкл IconDialogue
    private void SetIconDialogueActive(bool active)
    {
        if (iconDialogue != null)
        {
            iconDialogue.SetActive(active);
        }
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        // Проверяем, что вошли в триггер NPC и диалог не идет
        if (other.CompareTag("NPC") &&
!dialogueManager.dialogueIsPlaying)
        {
            canInteract = true; // Разрешаем взаимодействие
            SetIconDialogueActive(true); // Включаем иконку
        }
    }

    private void OnTriggerExit2D(Collider2D other)
    {
        // Вышли из триггера NPC
        if (other.CompareTag("NPC"))
        {
            canInteract = false; // Запрещаем взаимодействие
            SetIconDialogueActive(false); // Выключаем иконку
        }
    }

    private void Knockback(Vector2 direction)
    {
        rb.AddForce(direction * knockbackForce,
ForceMode2D.Impulse);
    }

    public void ApplyKnockback(Vector2 direction)
    {
        Knockback(direction);
    }

    public void Die()
    {
        // Отключаем управление и коллизии
        enabled = false;
```

```
        if (characterCollider != null) {
characterCollider.enabled = false; }
        if (rb != null) { rb.linearVelocity = Vector2.zero; }


        if (animator != null)
        {
            animator.SetTrigger("Death");

            // Перезапускаем уровень после задержки
            Invoke("RestartLevel", deathAnimationDelay);
        }
        else
        {
            Debug.LogError("Animator component not found!
Cannot play death animation.");
            RestartLevel();
        }
    }

    void RestartLevel()
    {

SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    void OnDisable()
    {
        PlayerPrefs.SetInt("CanAttack", canAttack ? 1 : 0);
        PlayerPrefs.Save();
    }

    public void EnableAttack()
    {
        canAttack = true;
        PlayerPrefs.SetInt("CanAttack", 1);
        PlayerPrefs.Save();
        Debug.Log("Attack enabled via EnableAttack()");
    }

    public void DisableAttack()
    {
        canAttack = false;
        PlayerPrefs.SetInt("CanAttack", 0);
        PlayerPrefs.Save();
        Debug.Log("Attack disabled via DisableAttack()");
    }

    void OnDrawGizmosSelected()
    {
        if (attackPoint == null) return;
```

```
        Gizmos.DrawWireSphere(attackPoint.position,
attackRange);
    }
}
```

Файл CharacterAnimation.cs:

```
using UnityEngine;
using System.Collections;

public class CharacterAnimations : MonoBehaviour
{
    private Animator animator;
    private bool moving;
    private bool isAttacking = false;
    [SerializeField] private float attackCooldown = 0.3f;

    // Хэши триггеров анимаций
    private static readonly int MovingHash =
Animator.StringToHash("Moving");
    private static readonly int JumpHash =
Animator.StringToHash("Jump");
    private static readonly int AttackHash =
Animator.StringToHash("Attack");
    private static readonly int DeathHash =
Animator.StringToHash("Die");
    private static readonly int HitHash =
Animator.StringToHash("Hit");

    void Start()
    {
        animator = GetComponent<Animator>();
        if (animator == null)
        {
            Debug.LogError("Animator component not found!");
            enabled = false;
            return;
        }
    }

    public bool Moving
    {
        get { return animator.GetBool(MovingHash); }
        set { animator.SetBool(MovingHash, value); }
    }

    void Awake()
    {
        animator = GetComponent<Animator>();
        if (animator == null)
        {
```

```
            Debug.LogError("Animator component not found on
this GameObject!");
            enabled = false;
        }
    }

    public void Jump()
    {
        animator.SetTrigger(JumpHash);
    }

    public void Attack()
    {
        if (isAttacking) return;

        animator.SetTrigger(AttackHash);
        isAttacking = true;
        StartCoroutine(AttackCooldownCoroutine()); //
Запускаем корутину перезарядки
    }

    private IEnumerator AttackCooldownCoroutine()
    {
        yield return new WaitForSeconds(attackCooldown);
        isAttacking = false;
    }

    public void Hit()
    {
        animator.SetTrigger(HitHash);
    }

    public void Heal()
    {
        animator.SetTrigger("Heal");
    }

    public void Die()
    {
        animator.SetTrigger(DeathHash);
        CharacterMovement characterMovement =
GetComponent<CharacterMovement>();
        if (characterMovement != null)
        {
            characterMovement.enabled = false;
        }
        Collider2D collider2D = GetComponent<Collider2D>();
        if (collider2D != null)
        {
            collider2D.enabled = false;
        }
        Rigidbody2D rb = GetComponent<Rigidbody2D>();
```

```
        if (rb != null)
        {
            rb.linearVelocity = Vector2.zero;
            rb.bodyType = RigidbodyType2D.Static;
        }
    }
}
```

Файл PlayerHealth.cs:

```
public void TakeDamage(int damage)

    {

        Debug.Log("TakeDamage called with damage = " + damage);

        if (damage <= 0)

        {

            Debug.LogWarning("Damage value is not positive!");

            return;

        }

        currentHealth -= damage;

        currentHealth = Mathf.Max(currentHealth, 0);

        Debug.Log("Player took " + damage + " damage! Current
health: " + currentHealth);

        soundController.PlayDamageSound();



        if (healthBar != null)

        {

            healthBar.value = currentHealth;

        }
```

```
        if (characterAnimations != null)
        {
            characterAnimations.Hit();
        }
        else
        {
            Debug.LogError("CharacterAnimations is null! Cannot
play Hit animation.");
        }

        if (currentHealth <= 0)
        {
            Die();
        }
        UpdateHeart();
    }
public void Heal(int amount)
    {
        currentHealth += amount;
        currentHealth = Mathf.Clamp(currentHealth, 0,
maxHealth);
        Debug.Log("Player healed for " + amount + " health!
Current health: " + currentHealth);
```

```
        if (healthBar != null)
        {
            healthBar.value = currentHealth;
        }
        soundController.PlayHealSound();
        UpdateHeart();
    }
```

Файл PlayerSoundController.cs:

```
using UnityEngine;

public class PlayerSoundController : MonoBehaviour
{
    public AudioSource audioSource;
    private CharacterMovement characterMovement;

    [Header("Прыжки")]
    public AudioClip jumpSound;
    [Header("Атаки")]
    public AudioClip attackSound;

    [Header("Урон")]
```

```
    public AudioClip damageSound; // Звук получения урона


    [Header("Лечение")]
    public AudioClip healSound; // Звук лечения


    void Start()
    {
        if (audioSource == null)
        {
            Debug.LogError("Audio Source is not assigned! Please
assign it in the Inspector.");
            enabled = false;
            return;
        }


        // Получаем ссылку на CharacterMovement
        characterMovement = GetComponent<CharacterMovement>();
        if (characterMovement == null)
        {
            Debug.LogError("CharacterMovement component not
found! Please attach this script to the same object as
CharacterMovement.");
            enabled = false;
```

```
            return;
        }
    }

    public void PlayJumpSound()
    {
        audioSource.PlayOneShot(jumpSound);
    }

    public void PlayAttackSound()
    {
        audioSource.PlayOneShot(attackSound);
    }

    public void PlayDamageSound()
    {
        audioSource.PlayOneShot(damageSound);
    }

    public void PlayHealSound()
    {
        audioSource.PlayOneShot(healSound);
```

```
    }
}
```

Скрипты первого уровня:

Файл Flower.cs:

```
using UnityEngine;


public class Flower : MonoBehaviour
{
    private bool _hasDrop1 = false;
    private bool _hasDrop2 = false;
    private bool _hasDrop3 = false;
    public bool IsBlooming { get; private set; } // Публичное
чтение, приватная запись


    private Animator _animator;
    private AudioSource _bloomSound;


    void Start()
    {
        _animator = GetComponent<Animator>();
        if (_animator == null)
        {
```

```
            Debug.LogError("Animator component not found on
Flower GameObject.");
            enabled = false;
            return;
        }

        _bloomSound = GetComponent<AudioSource>();
        if (_bloomSound == null)
        {
            Debug.LogWarning("AudioSource component not found on
Flower GameObject.  Bloom sound will not play.");
        }
        LoadState();

        if (IsBlooming)
        {
            Bloom();
        }
    }

    public void Water()
    {
        Debug.Log("Flower was watered!");
```

```
    }

    public void ApplyColor(Color color, int dropType)
    {
        Debug.Log("Applying color: " + color + ", dropType: " +
dropType);
        Renderer renderer = GetComponent<Renderer>();
        if (renderer != null)
        {
            renderer.material.color = color;

            switch (dropType)
            {
                case 1:
                    _hasDrop1 = true;
                    break;
                case 2:
                    _hasDrop2 = true;
                    break;
                case 3:
                    _hasDrop3 = true;
                    break;
```

```
            }


            if (_hasDrop1 && _hasDrop2 && _hasDrop3)
            {
                Bloom();
            }
        }
        else
        {
            Debug.LogError("Renderer component not found on
Flower GameObject.");
        }
    }


    private void Bloom()
    {
        Debug.Log("Flower is blooming!");
        _animator.SetTrigger("Bloom");
        IsBlooming = true;
        SaveState();
        if (_bloomSound != null)
        {
```

```
            _bloomSound.Play();
        }
    }
    private void SaveState()
    {
        PlayerPrefs.SetInt("FlowerBloom", IsBlooming ? 1 : 0);
        PlayerPrefs.Save();
    }
    private void LoadState()
    {
        IsBlooming = PlayerPrefs.GetInt("FlowerBloom", 0) == 1;
    }
}
```

Файл Drop.cs:

```
using UnityEngine;


public class Drop : MonoBehaviour
{
    public Color dropColor = Color.white;
    public int dropType = 1; // Тип капли


    private void OnTriggerEnter2D(Collider2D other)
```

```
    {
        if (other.CompareTag("Flower"))
        {
            Flower flower = other.GetComponent<Flower>();
            if (flower != null)
            {
                flower.ApplyColor(dropColor, dropType); //
Передаем тип капли
                Destroy(gameObject);
            }
        }
    }
}
```

Файл ColorBox.cs:

```
using UnityEngine;


public class ColorBox : MonoBehaviour
{
    public GameObject dropPrefab;
    public Transform dropSpawnPoint;
    public string interactionButton = "F";
    public float interactionDistance = 2f;
```

```
    public int dropType = 1; // Тип капли (1, 2, 3)


    private GameObject _player;
    private bool _isNear = false;


    void Start()
    {
        _player = GameObject.FindGameObjectWithTag("Player");
        if (_player == null)
        {
            Debug.LogError("Player not found. Make sure the
Player has the 'Player' tag.");
            enabled = false;
            return;
        }


        if (dropSpawnPoint == null)
        {
            Debug.LogError("Drop Spawn Point is not assigned!");
            enabled = false;
            return;
        }
```

```
    }


    void Update()
    {
        float distanceToPlayer =
Vector2.Distance(transform.position,
_player.transform.position);
        _isNear = distanceToPlayer <= interactionDistance;


        if (_isNear && Input.GetButtonDown(interactionButton))
        {
            GiveDrop();
        }
    }


    void GiveDrop()
    {
        if (_player.GetComponent<PlayerController>().HasDrop())
        {
            Debug.Log("Player already has a drop!");
            return;
        }
```

```
        //Создание капли и передача цвета
        GameObject drop = Instantiate(dropPrefab,
dropSpawnPoint.position, Quaternion.identity);
        Drop dropScript = drop.GetComponent<Drop>();
        if (dropScript != null)
        {
            dropScript.dropColor =
GetComponent<SpriteRenderer>().color;
            dropScript.dropType = dropType; // Передаем тип
капли
        }
        drop.transform.SetParent(dropSpawnPoint);

        _player.GetComponent<PlayerController>().SetDrop(drop);
        Debug.Log("Gave drop to player!");
    }

    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(transform.position,
interactionDistance);
    }
```

```
}
```

Файл ColorBox.cs:

```
package MyProgramm;


}
```

Файл PlayerController.cs:

```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    private GameObject _currentDrop = null;
    public string interactionButton = "F";
    public float interactionDistance = 2f;
    private GameObject _flower;
    public Transform dropSpawnPoint;

    private SpriteRenderer _spriteRenderer;

    void Start()
    {
        _flower = GameObject.FindGameObjectWithTag("Flower");
        if (_flower == null)
        {
            Debug.LogError("Flower not found. Make sure the
Flower has the 'Flower' tag.");
            enabled = false;
            return;
        }

        if (dropSpawnPoint == null)
        {
            Debug.LogError("Drop Spawn Point is not
assigned!");
            enabled = false;
            return;
        }
    }

    void Update()
    {
        UpdateDropSpawnPointPosition();

        if (_currentDrop != null && IsNearFlower() &&
Input.GetButtonDown(interactionButton))
        {
```

```
            WaterFlower();
        }
    }

    private bool IsNearFlower()
    {
        float distanceToFlower =
Vector2.Distance(transform.position,
_flower.transform.position);
        return distanceToFlower <= interactionDistance;
    }

    void WaterFlower()
    {
        Debug.Log("Watering the flower with " +
_currentDrop.name);

        Destroy(_currentDrop);
        _currentDrop = null;
    }

    public void SetDrop(GameObject drop)
    {
        _currentDrop = drop;
    }

    public bool HasDrop()
    {
        return _currentDrop != null;
    }

    void UpdateDropSpawnPointPosition()
    {
        float facingDirection = _spriteRenderer.flipX ? -1f :
1f;

        dropSpawnPoint.localScale = new
Vector3(facingDirection, 1f, 1f);
    }
}
```

Файл BackgroundTintController.cs:

```
using UnityEngine;
using UnityEngine.UI;

public class BackgroundTintController : MonoBehaviour
{
    public Flower flower;
    public Color unbloomedTint = new Color(0.53f, 0.53f,
0.53f); // Цвет #878787
```

```
    public Color bloomedColor = new Color(0.92f, 1.0f, 0.55f);
// Цвет #EBFF8D

    private SpriteRenderer spriteRenderer;
    private Image image;

    void Start()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
        if (spriteRenderer == null)
        {
            image = GetComponent<Image>();
            if (image == null)
            {
                Debug.LogError("BackgroundTintController
requires a SpriteRenderer or Image component on the same
GameObject!");
                enabled = false;
            }
        }

        if (flower == null)
        {
            enabled = false;
        }
        SetBackgroundColor(unbloomedTint);
    }

    void Update()
    {
        if (flower.IsBlooming)
        {
            SetBackgroundColor(bloomedColor);
        }
        else
        {
            SetBackgroundColor(unbloomedTint);
        }
    }

    void SetBackgroundColor(Color color)
    {
        if (spriteRenderer != null)
        {
            spriteRenderer.color = color;
        }
        else if (image != null)
        {
            image.color = color;
        }
    }
}
```

Скрипты второго уровня:

Файл DataGenerator.cs:

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using System.Collections.Generic;

public class DataGenerator : MonoBehaviour
{
    public GameObject[] textPrefabs;
    public GameObject[] imagePrefabs;
    public GameObject[] emojiPrefabs;
    public GameObject[] webPrefabs;
    public List<Transform> spawnAreas;

    public float spawnRate = 2f;
    public float fallSpeed = 1f;

    public GameObject playerPrefab; // Префаб голубя
    public GameObject player;        // Ссылка на экземпляр
голубя
    public GameObject currentObject; // Текущий объект на
сцене
    private bool isSpawning = true; // Флаг для управления
генерацией

    private List<Follower> followers = new List<Follower>();

    private string[] possibleTags = { "Image", "Emoji", "Web",
"Text" };

    [Header("Dialogue")]
    public PigeonDialogue pigeonDialogue;

    void Start()
    {
        if (pigeonDialogue == null)
        {
            Debug.LogError("PigeonDialogue is not assigned to
DataGenerator!");
            enabled = false;
            return;
        }

        //  Запускаем корутину если диалог не активен
        if (!pigeonDialogue.IsDialoguePlaying)
        {
            StartCoroutine(SpawnData());
        }
```

```
        else
        {
            pigeonDialogue.OnDialogueEnded +=
StartSpawningAfterDialogue;
            Debug.Log("DataGenerator: Waiting for dialogue to
end before spawning.");
        }
    }

    // Метод для добавления последователя в список и установки
ему лидера
    public void AddFollower(Follower follower)
    {
        followers.Add(follower);
        if (player != null)
        {
            follower.leader = player.transform;  // Передаем
transform лидера
        }
    }

    // Метод для удаления последователя из списка
    public void RemoveFollower(Follower follower)
    {
        followers.Remove(follower);
    }

    // Метод, вызываемый после завершения диалога
    void StartSpawningAfterDialogue()
    {
        Debug.Log("DataGenerator: Dialogue ended, starting
spawning.");
        pigeonDialogue.OnDialogueEnded -=
StartSpawningAfterDialogue; // Отписываемся от события
        isSpawning = true;
        StartCoroutine(SpawnData()); // Запускаем генерацию
    }

    IEnumerator SpawnData()
    {
        while (isSpawning)
        {

            if (pigeonDialogue.IsDialoguePlaying)
            {
                yield return null;
                continue;
            }

            if (currentObject == null ||
!pigeonDialogue.IsDialoguePlaying)
            {
```

```
                // Выбираем случайный тег для голубя
                string tagToAssign = GetRandomTag();

                // Создаем голубя с выбранным тегом
                Transform selectedSpawnArea =
GetRandomSpawnArea();
                Vector2 spawnPosition =
GetSpawnPositionNearPlayer(selectedSpawnArea, player);

                if (player == null)
                {
                    // Создаем лидера
                    player = Instantiate(playerPrefab,
spawnPosition, Quaternion.identity, transform);
                    player.tag = tagToAssign;

                    // Устанавливаем ему последователя
                    foreach (Follower follower in followers)
                    {
                        follower.leader = player.transform;
                    }
                }

                // Создаем объект с тегом голубя
                GameObject prefabToSpawn =
GetPrefabWithTag(tagToAssign);

                if (prefabToSpawn != null)
                {
                    // Создаем объект
                    currentObject = Instantiate(prefabToSpawn,
spawnPosition, Quaternion.identity, transform);

                    PigeonController pigeonController =
player.GetComponent<PigeonController>();

                    if (pigeonController != null)
                    {
                        pigeonController.targetObject =
currentObject;
                    }
                    else
                    {
                        Debug.LogError("PigeonController not
found on the Player after instantiating target object");
                    }
                }
                else
                {
                    Debug.LogError("No prefab found with tag:
" + tagToAssign);
                }
```

```
            }
            yield return new WaitUntil(() => currentObject ==
null);
            yield return new WaitForSeconds(1f / spawnRate);
        }
    }

    // Метод для остановки спавна
    public void StopSpawning()
    {
        isSpawning = false;
    }

    string GetRandomTag()
    {
        int randomIndex = Random.Range(0,
possibleTags.Length);
        return possibleTags[randomIndex];
    }

    GameObject GetPrefabWithTag(string tag)
    {
        switch (tag)
        {
            case "Image":
                if (imagePrefabs.Length > 0)
                {
                    Debug.Log("Image prefabs available.
Returning a random one.");
                    return imagePrefabs[Random.Range(0,
imagePrefabs.Length)];
                }
                break;
            case "Emoji":
                if (emojiPrefabs.Length > 0)
                {
                    Debug.Log("Emoji prefabs available.
Returning a random one.");
                    return emojiPrefabs[Random.Range(0,
emojiPrefabs.Length)];
                }
                break;
            case "Web":
                if (webPrefabs.Length > 0)
                {
                    Debug.Log("Web prefabs available.
Returning a random one.");
                    return webPrefabs[Random.Range(0,
webPrefabs.Length)];
                }
                break;
            case "Text":
```

```
                if (textPrefabs.Length > 0)
                {
                    Debug.Log("Text prefabs available.
Returning a random one.");
                    return textPrefabs[Random.Range(0,
textPrefabs.Length)];
                }
                break;
            default:
                Debug.LogError("Invalid tag: " + tag);
                return null;
        }

        Debug.LogWarning("No prefabs found for tag: " + tag);
        return null;
    }

    Vector2 GetSpawnPositionNearPlayer(Transform spawnArea,
GameObject player)
    {
        if (player == null)
        {
            return GetRandomSpawnPosition(spawnArea);
        }

        Rect spawnRect =
spawnArea.GetComponent<RectTransform>().rect;

        Vector3 playerPosition = player.transform.position;

        float minX = playerPosition.x + spawnRect.xMin;
        float maxX = playerPosition.x + spawnRect.xMax;

        float x = Random.Range(minX, maxX);
        float y = spawnArea.position.y;
        Vector3 worldPosition = spawnArea.TransformPoint(new
Vector3(x, 0, 0));
        return new Vector2(worldPosition.x, worldPosition.y);
    }

    Vector2 GetRandomSpawnPosition(Transform spawnArea)
    {
        Rect spawnRect =
spawnArea.GetComponent<RectTransform>().rect;
        float x = Random.Range(spawnRect.xMin,
spawnRect.xMax);
        float y = spawnRect.yMax;
        Vector3 worldPosition = spawnArea.TransformPoint(new
Vector3(x, 0, 0));
        return new Vector2(worldPosition.x, worldPosition.y);
    }
```

```
    Transform GetRandomSpawnArea()
    {
        int randomIndex = Random.Range(0, spawnAreas.Count);
        return spawnAreas[randomIndex];
    }

    GameObject GetRandomPrefab()
    {
        int randomIndex = Random.Range(0, textPrefabs.Length +
imagePrefabs.Length + emojiPrefabs.Length +
webPrefabs.Length);

        if (randomIndex < textPrefabs.Length)
        {
            return textPrefabs[randomIndex];
        }
        else if (randomIndex < textPrefabs.Length +
imagePrefabs.Length)
        {
            return imagePrefabs[randomIndex -
textPrefabs.Length];
        }
        else if (randomIndex < textPrefabs.Length +
imagePrefabs.Length + emojiPrefabs.Length)
        {
            return emojiPrefabs[randomIndex -
textPrefabs.Length - imagePrefabs.Length];
        }
        else
        {
            return webPrefabs[randomIndex - textPrefabs.Length
- imagePrefabs.Length - emojiPrefabs.Length];
        }
    }

    // Метод для уничтожения объекта и голубя
    public void DestroyObjectAndPlayer(GameObject
objectToDestroy)
    {
        if (objectToDestroy == currentObject)
        {
            currentObject = null;
            if (player != null)
            {
                Destroy(player); // Уничтожаем голубя
                player = null; // Обнуляем ссылку на голубя
            }
            Destroy(objectToDestroy);
        }
    }
}
```

Файл DataCollector.cs:

```
using UnityEngine;

using UnityEngine.UI;


public class DataCollector : MonoBehaviour

{

    public string dataType; // Тип данных для данной
коробки

Text, Image, Emoji, Web


    void OnTriggerEnter2D(Collider2D other)

    {

        // Проверяем, соответствует ли тэг объекта типу голубя

        if (other.gameObject.CompareTag(dataType))

        {

            Destroy(other.gameObject);

            Debug.Log(dataType + " collected!");


            // Увеличиваем счетчик собранных данных

            ScoreManager.Instance.IncreaseScore();

        }

        else
```

```
        {

            // Объект проходит сквозь коробку

            Debug.Log(other.gameObject.name + " passed through "
+ dataType + " collector.");

        }

    }

}
```

Файл BoundaryTrigger.cs:

```
using UnityEngine;

public class BoundaryTrigger : MonoBehaviour
{
    public int maxDestroys = 3; // Максимальное количество
уничтожений перед вызовом панели поражения
    private int destroyCount = 0; // Начальный счетчик
    public GameObject gameOverPanel; // Ссылка на панель
поражения

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.gameObject != gameObject)
        {
            // Уничтожаем любые объекты
            Destroy(other.gameObject);
            destroyCount++; // Увеличиваем счетчик уничтожений

            // Проверяем, нужно ли вызвать панель поражения
            if (destroyCount >= maxDestroys)
            {
                ActivateGameOverPanel();
            }
        }
    }

    void ActivateGameOverPanel()
    {
        if (gameOverPanel != null)
        {
            GameOverManager gameOverManager =
gameOverPanel.GetComponent<GameOverManager>();
```

```
            if (gameOverManager != null)
            {
                gameOverManager.ShowGameOverPanel();
            }
            else
            {
                Debug.LogError("GameOverManager не найден!");
            }
        }
        else
        {
            Debug.LogError("GameOverPanel не назначен в
BoundaryTrigger!");
        }
    }
}
```

Файл ScoreManager.cs:

```
using UnityEngine;
using UnityEngine.UI;

public class ScoreManager : MonoBehaviour
{
    public int targetScore = 10;
    private int currentScore = 0;
    private bool levelComplete = false;

    public DataGenerator dataGenerator;
    public GameObject completedPanel; // Ссылка на панель
победы

    public static ScoreManager Instance;

    void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
        LoadState();
    }

    public void IncreaseScore()
    {
        if (!levelComplete)
```

```
        {
            currentScore++;
            Debug.Log("Score: " + currentScore);

            if (currentScore >= targetScore)
            {
                LevelComplete();
            }
        }
    }

    private void LevelComplete()
    {
        levelComplete = true;
        Debug.Log("Level 2 Complete!");

        SaveState();

        // Отключаем генерацию новых объектов и голубей
        if (dataGenerator != null)
        {
            dataGenerator.StopSpawning();
        }

        if (completedPanel != null)
        {
            completedPanel.SetActive(true);
        }
        else
        {
            Debug.LogWarning("Completed Panel is not
assigned!");
        }
    }

    public bool IsLevelComplete()
    {
        return levelComplete;
    }

    private void SaveState()
    {
        PlayerPrefs.SetInt("Completelevel2", 1);
        PlayerPrefs.Save();
        Debug.Log("Level 2 state saved!");
    }

    private void LoadState()
    {
        levelComplete = PlayerPrefs.GetInt("Completelevel2",
0) == 1;
```

```
        if (levelComplete)
        {
            Debug.Log("Level 2 already complete!");
            if (dataGenerator != null)
            {
                dataGenerator.StopSpawning();
            }
            if (completedPanel != null)
            {
                completedPanel.SetActive(true);
            }
        }
        else
        {
            Debug.Log("Level 2 not yet complete.");
        }
    }
}
```

Файл GameOverManager.cs:

```
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;

public class GameOverManager : MonoBehaviour
{
    public GameObject gameOverPanel; // Ссылка на панель
поражения
    public GameObject skipLevelPanel;  // Ссылка на панель
пропуска уровня

    void Start()
    {
        if (gameOverPanel == null)
        {
            Debug.LogError("GameOverPanel не назначен в
GameOverManager!");
            enabled = false;
            return;
        }
        gameOverPanel.SetActive(false);
        if (skipLevelPanel != null)
        {
            skipLevelPanel.SetActive(false);
        }
    }

    void Update()
    {
```

```
        if (skipLevelPanel != null &&
skipLevelPanel.activeSelf)
        {
            if (Input.GetKeyDown(KeyCode.Escape))
            {
                CloseSkipLevelPanel();
            }

          EventSystem.current.SetSelectedGameObject(skipLevelPa
          nel); // Фокус на панель
        }
        else
        {
            if (EventSystem.current.currentSelectedGameObject
== skipLevelPanel)
            {

EventSystem.current.SetSelectedGameObject(null);
            }
        }
    }

    public void ShowGameOverPanel()
    {
        gameOverPanel.SetActive(true);
        Time.timeScale = 0f; //1 - норм скорость
    }

    public void RestartGame()
    {
        Time.timeScale = 1f;

UnityEngine.SceneManagement.SceneManager.LoadScene(UnityEngine
.SceneManagement.SceneManager.GetActiveScene().name);
    }

    public void GoToMainMenu()
    {
        Time.timeScale = 1f;

UnityEngine.SceneManagement.SceneManager.LoadScene("MainMenu")
;
    }

    public void ShowSkipLevelPanel()
    {
        skipLevelPanel.SetActive(true);
    }

    public void CloseSkipLevelPanel()
    {
        skipLevelPanel.SetActive(false);
```

```
    }

    public void SkipLevel()
    {
        SaveState();
        Debug.Log("Skipping level...");
        Time.timeScale = 1f; // Возобновляем время

UnityEngine.SceneManagement.SceneManager.LoadScene("SceneMothe
rboard");
    }

    private void SaveState()
    {
        PlayerPrefs.SetInt("Completelevel2", 1);
        PlayerPrefs.Save();
        Debug.Log("Level 2 state saved!");
    }
}
```

Скрипты третьего уровня:

Файл Button.cs:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine.SceneManagement;

public class Button : MonoBehaviour
{
    [SerializeField] private Egg myEgg;
    [SerializeField] private List<Button> otherButtons;
    [SerializeField] private CharacterMovement
characterMovement;

    private Animator buttonAnimator;

    private const string ButtonAnimRed = "RedButton";
    private const string ButtonAnimYel = "YelButton";
    private const string ButtonAnimWhite = "WhiteButton";

    public bool isActivated = false;

    public static bool canChangeColors = true;

    private string sceneName;

    void Start()
    {
```

```
        buttonAnimator = GetComponent<Animator>();
        if (buttonAnimator == null)
        {
            Debug.LogError("Animator component not found on
Button GameObject.");
            enabled = false;
            return;
        }
        sceneName = SceneManager.GetActiveScene().name;

        LoadState();

        if
(buttonAnimator.GetCurrentAnimatorStateInfo(0).IsName("RedButton
"))
            SetRed();
        else if
(buttonAnimator.GetCurrentAnimatorStateInfo(0).IsName("YelButton
"))
            SetYel();
        else
            SetWhite();

        characterMovement.SetMass(50f);
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player") && enabled &&
canChangeColors)
        {
            Activate();
        }
    }

    public void Activate()
    {
        if (!canChangeColors) return;
        if (isActivated)
        {
            SetRed();
            myEgg.ChangeToRed();
        }
        else
        {
            SetYel();
            myEgg.ChangeToYel();
        }
        isActivated = !isActivated;
        SaveState();

        // Активируем связанные кнопки
```

```
        foreach (Button button in otherButtons)
        {
            if (button != null)
            {
                if (button.isActivated)
                {
                    button.SetRed();
                    button.myEgg.ChangeToRed();
                    button.isActivated = false;
                    button.SaveState();
                }
                else
                {
                    button.SetYel();
                    button.myEgg.ChangeToYel();
                    button.isActivated = true;
                    button.SaveState();
                }
            }
        }

        // Проверяем, все ли яйца желтые
        Egg[] allEggs =
FindObjectsByType<Egg>(FindObjectsSortMode.None);
        bool allYel = allEggs.All(egg => egg.currentState ==
Egg.EggState.Yel);
        if (allYel)
        {
            foreach (Egg egg in allEggs)
            {
                egg.ChangeToWhite();
            }

            foreach (Button button in
FindObjectsByType<Button>(FindObjectsSortMode.None))
            {
                button.SetWhite();
                button.SaveState();
            }

            canChangeColors = false; // Отключаем возможность
менять цвета
            characterMovement.RestoreMass();
            SceneState.levelCompleted = true;
        }
    }

    private void SetRed()
    {
        buttonAnimator.Play(ButtonAnimRed);
    }
```

```
    private void SetYel()
    {
        buttonAnimator.Play(ButtonAnimYel);
    }

    private void SetWhite()
    {
        buttonAnimator.Play(ButtonAnimWhite);
    }

    private void SaveState()
    {
        // Сохраняем прогресс уровня, чтобы кнопки оставались
белыми
        PlayerPrefs.SetInt(sceneName + "_" + gameObject.name +
"_IsActivated", isActivated ? 1 : 0);
        PlayerPrefs.SetInt("Completelevel3", 1);
        PlayerPrefs.Save();
    }

    private void LoadState()
    {
        isActivated = PlayerPrefs.GetInt(sceneName + "_" +
gameObject.name + "_IsActivated", 0) == 1;

        if (SceneState.levelCompleted)
        {
            SetWhite();
            canChangeColors = false; // Отключаем возможность
менять цвета
        }
        else
        {
            // Иначе восстанавливаем состояние кнопок
            if (isActivated)
            {
                SetYel();
            }
            else
            {
                SetRed();
            }
        }
    }
}
```

Файл Egg.cs:

```
using UnityEngine;

public class Egg : MonoBehaviour
```

```
{
    private Animator animator;

    private const string AnimRed = "Red";
    private const string AnimYel = "Yel";
    private const string AnimWhite = "White";

    public EggState currentState = EggState.Red;
    public enum EggState
    {
        Red,
        Yel,
        White
    }
    // Уникальный идентификатор для яйца
    [SerializeField] private string eggID;

    void Start()
    {
        animator = GetComponent<Animator>();
        if (animator == null)
        {
            Debug.LogError("Animator component not found on this
GameObject.");
            enabled = false;
            return;
        }

        // Генерация уникального ID, если его нет
        if (string.IsNullOrEmpty(eggID))
        {
            eggID = System.Guid.NewGuid().ToString();
            Debug.LogWarning($"EggID not set, generated new ID:
{eggID}.  Consider setting it manually to avoid ID changes on
future scene loads.");
        }

        LoadState();

        PlayAnimation(AnimRed);
    }

    public void ChangeToYel()
    {
        if (!Button.canChangeColors) return;
        if (currentState == EggState.Red)
        {
            PlayAnimation(AnimYel);
            currentState = EggState.Yel;
            SaveState();
        }
    }
```

```
    public void ChangeToRed()
    {
        if (!Button.canChangeColors) return;
        if (currentState == EggState.Yel)
        {
            PlayAnimation(AnimRed);
            currentState = EggState.Red;
            SaveState();
        }
    }

    public void ChangeToWhite()
    {
        if (!Button.canChangeColors) return;
        if (currentState == EggState.Yel)
        {
            PlayAnimation(AnimWhite);
            currentState = EggState.White;
            SaveState();
        }
    }

    void PlayAnimation(string animationName)
    {
        animator.Play(animationName);
    }

    public void SaveState()
    {
        PlayerPrefs.SetInt(eggID + "_EggState",
(int)currentState);
        PlayerPrefs.Save();
    }

    private void LoadState()
    {
        currentState = (EggState)PlayerPrefs.GetInt(eggID +
"_EggState", 0); // 0 - Red по умолчанию
        UpdateSprite();
    }

    // Обновление спрайта в соответствии с состоянием
    private void UpdateSprite()
    {
        switch (currentState)
        {
            case EggState.Red:
                PlayAnimation(AnimRed);
                break;
            case EggState.Yel:
                PlayAnimation(AnimYel);
```

```
                break;
            case EggState.White:
                PlayAnimation(AnimWhite);
                break;
        }
    }

    //Добавлен метод ResetEggID для редактора, чтобы
сгенерировать новый EggID
    [ContextMenu("Reset EggID")]
    private void ResetEggID()
    {
        eggID = System.Guid.NewGuid().ToString();
    }

    //Проверка на уникальный ID
    private void OnValidate()
    {
        if (string.IsNullOrEmpty(eggID))
        {
            Debug.LogWarning($"EggID not set on
{gameObject.name}.  Consider setting it manually or allowing the
script to generate one on start. Right click component and press
'Reset EggID'");
        }
    }
}
```

Файл ButtonStateChecker.cs:

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections.Generic;

public class ButtonStateChecker : MonoBehaviour
{
    public string levelName;
    public List<GameObject> buttonsToCheck;
    private bool levelComplete = false;

    void Awake()
    {
        LoadLevelState();
    }

    void Start()
    {
        if (string.IsNullOrEmpty(levelName))
        {
            Debug.LogError("Level Name is not assigned in
ButtonStateChecker!");
```

```
            enabled = false;
            return;
        }

        if (buttonsToCheck == null || buttonsToCheck.Count ==
0)
        {
            Debug.LogError("No buttons assigned in
ButtonStateChecker!");
            enabled = false;
            return;
        }
    }

    void Update()
    {
        if (!levelComplete)
        {
            CheckButtonStates();
        }
    }

    private void CheckButtonStates()
    {
        // Проверяем, что все кнопки белые
        foreach (GameObject buttonObject in buttonsToCheck)

        LevelComplete();
    }

    private void LevelComplete()
    {
        levelComplete = true;
        Debug.Log("Level 3 Complete! All buttons are white.");

        SaveLevelState();
    }

    private void SaveLevelState()
    {
        PlayerPrefs.SetInt(levelName + "Complete", 1);
        PlayerPrefs.Save();
        Debug.Log("Level " + levelName  + " state saved!");
    }

    private void LoadLevelState()
    {
        levelComplete = PlayerPrefs.GetInt(levelName +
"Complete", 0) == 1;

        if (levelComplete)
        {
```

```
            Debug.Log("Level 3 " + levelName + " already
complete!");
            enabled = false;
        }
        else
        {
            Debug.Log("Level 3 " + levelName + " not yet
complete.");
        }
    }
}
```

Файл SceneState.cs:

```
using UnityEngine;

using UnityEngine.SceneManagement;



public class SceneState : MonoBehaviour

{

    public static bool levelCompleted = false;

    public static float initialMass = -1f; // -1f масса еще не
сохранена

}
```

Скрипты четвертого уровня:

Файл Ghost.cs:

```
using UnityEngine;

public class Ghost : MonoBehaviour
{
    public DialogueManager dialogueManager;
    public TextAsset inkFile;
    public float timeScaleDuringDialogue = 1f;
    public string deathAnimationName = "Death";
    public float destroyDelay = 1f;
    private Animator _animator;
    private bool _dialogueStarted = false;
```

```
    private string _dialogueStartedKey =
"GhostDialogueStarted";
    private float _originalTimeScale;
    public CharacterMovement characterMovement;

    // GameObject визуальное представление призрака
    public GameObject ghostVisuals;

    void Start()
    {
        _dialogueStarted =
PlayerPrefs.GetInt(_dialogueStartedKey, 0) == 1;

        if (dialogueManager == null)
        {
            Debug.LogError("DialogueManager is not assigned in
Ghost script!");
            enabled = false;
            return;
        }

        if (inkFile == null)
        {
            Debug.LogError("No ink file assigned to the
Ghost!");
            enabled = false;
            return;
        }

        if (ghostVisuals != null)
        {
            _animator = ghostVisuals.GetComponent<Animator>();
            if (_animator == null)
            {
                Debug.LogError("Animator не найден на дочернем
объекте ghostVisuals!");
            }
        }
        else
        {
            Debug.LogError("ghostVisuals не назначен!
Перетащите дочерний объект в это поле в инспекторе.");
        }

        // Изначально скрываем визуальную часть призрака
        SetGhostVisibility(false);

        dialogueManager.OnDialogueStarted +=
OnDialogueStarted;
        dialogueManager.OnDialogueEnded += OnDialogueEnded;
    }
```

```
    void OnDestroy()
    {
        if (dialogueManager != null)
        {
            dialogueManager.OnDialogueStarted -=
OnDialogueStarted;
            dialogueManager.OnDialogueEnded -=
OnDialogueEnded;
        }
    }


    public void TryStartDialogue()
    {
        if (!_dialogueStarted)
        {
            StartDialogue();
            _dialogueStarted = true;
            PlayerPrefs.SetInt(_dialogueStartedKey, 1);
            PlayerPrefs.Save();

            if (characterMovement != null)
            {
                characterMovement.EnableAttack();
            }
            else
            {
                Debug.LogError("CharacterMovement не назначен
в DialogueScript!");
            }
        }
    }

    private void StartDialogue()
    {
        if (dialogueManager != null && inkFile != null)
        {
            dialogueManager.EnterDialogueMode(inkFile);
            Debug.Log("Ghost Dialogue Started!");
        }
        else
        {
            Debug.LogError("DialogueManager is not assigned or
inkFile is not assigned!");
        }
    }

    private void OnDialogueStarted()
    {
        _originalTimeScale = Time.timeScale;
        Time.timeScale = timeScaleDuringDialogue;
        Time.fixedDeltaTime = _originalTimeScale * 0.02f;
```

```
        // Делаем призрака видимым, когда диалог начинается
        SetGhostVisibility(true);
    }

    private void OnDialogueEnded()
    {
        Time.timeScale = _originalTimeScale;
        Time.fixedDeltaTime = _originalTimeScale * 0.02f;

        if (_animator != null)
        {
            _animator.Play(deathAnimationName);
            Debug.Log("Анимация смерти запущена для дочернего
объекта");
        }
        else
        {
            Debug.LogError("Animator is null.  Make sure it's
properly initialized.");
        }

        Destroy(gameObject, destroyDelay);

    }

    // Метод для установки видимости визуальной части призрака
    private void SetGhostVisibility(bool visible)
    {
        if (ghostVisuals != null)
        {
            ghostVisuals.SetActive(visible);
        }
        else
        {
            Debug.LogWarning("ghostVisuals is null, visibility
cannot be set.");
        }
    }
}
```

Скрипты шипов:

Файл Thorns.cs:

```
using UnityEngine;

public class Thorns : MonoBehaviour
{
    [SerializeField] private int damage = 1;
```

```
    [SerializeField] private float damageInterval = 1f; //
Интервал в секундах
    private float lastDamageTime; // Время последнего
нанесения урона

    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            PlayerHealth playerHealth =
collision.gameObject.GetComponent<PlayerHealth>();

            if (playerHealth != null)
            {
                playerHealth.TakeDamage(damage);
                lastDamageTime = Time.time; // Запоминаем
время последнего нанесения урона
            }
        }
    }

    private void OnCollisionStay2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            if (Time.time - lastDamageTime >= damageInterval)
            {
                PlayerHealth playerHealth =
collision.gameObject.GetComponent<PlayerHealth>();

                if (playerHealth != null)
                {
                    playerHealth.TakeDamage(damage);
                    lastDamageTime = Time.time; // Запоминаем
время последнего нанесения урона
                }
            }
        }
    }
}
```

Файл Thorns2.cs:

```
using UnityEngine;
using System.Collections;

public class Thorns2 : MonoBehaviour

{
    [SerializeField] private int damage = 2;
```

```
    [SerializeField] private float disappearDelay = 2f; //
Задержка перед исчезновением

    private bool hasFallen = false;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (!hasFallen &&
collision.gameObject.CompareTag("Player"))
        {
            // Шипы падают только раз
            hasFallen = true;

            Rigidbody2D rb = GetComponent<Rigidbody2D>();
            if (rb == null)
            {
                rb = gameObject.AddComponent<Rigidbody2D>();
            }
            rb.bodyType = RigidbodyType2D.Dynamic;
            rb.gravityScale = 2f;
            StartCoroutine(DisappearAfterDelay()); // Запускаем
корутину исчезновения
        }
    }

    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (hasFallen &&
collision.gameObject.CompareTag("Player"))
        {
            PlayerHealth playerHealth =
collision.gameObject.GetComponent<PlayerHealth>();

            if (playerHealth != null)
            {
                playerHealth.TakeDamage(damage);
                Debug.Log("Player hit by falling thorns!");
            }
        }
    }

    IEnumerator DisappearAfterDelay()
    {
        yield return new WaitForSeconds(disappearDelay);

        Destroy(gameObject);
    }
}
```

Скрипты платформ:

Файл BreakingPlatform.cs:

```
using UnityEngine;
using System.Collections;

public class BreakingPlatform : MonoBehaviour
{
    [SerializeField] private float delayBeforeBreaking = 1.0f;
    [SerializeField] private float delayBeforeDestroy = 3.0f;

    private Animator _animator;
    private Collider2D _collider;
    private static readonly int IsBreaking =
Animator.StringToHash("IsBreaking");

    void Start()
    {
        _animator = GetComponent<Animator>();
        if (_animator == null)
        {
            Debug.LogError("Animator component not found on
BreakingPlatform GameObject.");
            enabled = false;
            return;
        }

        _collider = GetComponent<Collider2D>(); // Получаем
ссылку на *физический* коллайдер
        if (_collider == null)
        {
            Debug.LogError("Collider2D component not found on
BreakingPlatform GameObject.");
            enabled = false;
            return;
        }
    }

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            StartCoroutine(BreakPlatform());
        }
    }

    private IEnumerator BreakPlatform()
    {
        yield return new WaitForSeconds(delayBeforeBreaking);
        _animator.SetBool(IsBreaking, true);

        float animationLength = 0f;
        AnimationClip[] clips =
_animator.runtimeAnimatorController.animationClips;
        foreach (AnimationClip clip in clips)
```

```
        {
            if (clip.name == "Breaking")
            {
                animationLength = clip.length;
                break;
            }
        }
        yield return new WaitForSeconds(animationLength);
        yield return new WaitForSeconds(delayBeforeDestroy);

        Destroy(gameObject);
    }
}
```

Файл MovingPlatform.cs:

```
using UnityEngine;

public class MovingPlatform : MonoBehaviour
{
    [Tooltip("Точка, до которой будет двигаться платформа.")]
    public Transform targetPoint;

    [Tooltip("Скорость движения платформы.")]
    public float speed = 2f;

    [Tooltip("Время ожидания в конечной точке, прежде чем
двигаться обратно.")]
    public float waitTime = 1f;

    private Vector2 _startPosition;
    private Vector2 _targetPosition;
    private Rigidbody2D _rb;
    private bool _movingToTarget = true;
    private float _waitTimer = 0f;

    void Start()
    {
        _rb = GetComponent<Rigidbody2D>();

        if (_rb == null)
        {
            Debug.LogError("Rigidbody2D component not found on
MovingPlatform GameObject.");
            enabled = false;
            return;
        }

        if (targetPoint == null)
        {
```

```
            Debug.LogError("Target Point is not assigned to
MovingPlatform.");
            enabled = false;
            return;
        }

        _startPosition = transform.position;
        _targetPosition = targetPoint.position;

        // Отсоединяем targetPoint от платформы, чтобы он не
двигался вместе с ней
        targetPoint.SetParent(null);
    }

    void FixedUpdate()
    {
        if (_movingToTarget)
        {

_rb.MovePosition(Vector2.MoveTowards(transform.position,
_targetPosition, speed * Time.fixedDeltaTime));

            if (Vector2.Distance(transform.position,
_targetPosition) < 0.01f)
            {
                _movingToTarget = false;
                _waitTimer = 0f;
            }
        }
        else
        {
            _waitTimer += Time.fixedDeltaTime;

            if (_waitTimer >= waitTime)
            {
                _movingToTarget = true;
            }
        }

        if (!_movingToTarget)
        {

_rb.MovePosition(Vector2.MoveTowards(transform.position,
_startPosition, speed * Time.fixedDeltaTime));
            if (Vector2.Distance(transform.position,
_startPosition) < 0.01f)
            {
                _movingToTarget = true;
                _waitTimer = 0f;
            }
        }
    }
```

```
}
```

Файл PlatformAttach.cs:

```
using UnityEngine;

public class PlatformAttach : MonoBehaviour
{
    private Transform _platformTransform;

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            _platformTransform = transform;
            other.transform.SetParent(_platformTransform);
        }
    }

    void OnTriggerExit2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            other.transform.SetParent(null);
        }
    }
}
```

Скрипты монстров:

Файл MonsterAI.cs:

```
using UnityEngine;
using System.Collections;

public class MonsterAI : MonoBehaviour
{
    [SerializeField] private float moveSpeed = 2f;
    [SerializeField] private float chaseSpeed = 4f;
    [SerializeField] private float patrolDistance = 5f;
    private Vector2 startPosition;
    private bool movingRight = true;
    private bool isChasingPlayer = false;

    [SerializeField] private float detectionRange = 5f;
    [SerializeField] private LayerMask playerLayer;
    private Transform player;

    [SerializeField] private int maxHealth = 3;
    private int currentHealth;
```

```
    [SerializeField] private int damageAmount = 1;

    [SerializeField] private float knockbackForce = 3f;

    [SerializeField] private float attackDelay = 1f;
    private bool canAttack = true;

    private Animator animator;
    private Collider2D monsterCollider;
    private Rigidbody2D rb;
    private Ghost _ghost;

    private bool damageAppliedThisFrame = false;

    void Start()
    {
        startPosition = transform.position;
        currentHealth = maxHealth;
        animator = GetComponent<Animator>();
        rb = GetComponent<Rigidbody2D>();
        monsterCollider = GetComponent<Collider2D>();
        _ghost = FindFirstObjectByType<Ghost>();

        if (animator == null)
        {
            Debug.LogError("Animator component not found on " +
gameObject.name);
        }
        if (rb == null)
        {
            Debug.LogError("Rigidbody2D component not found on "
+ gameObject.name);
        }

        if (monsterCollider == null)
        {
            Debug.LogError("Collider2D component not found on "
+ gameObject.name);
        }
    }

    void Update()
    {
        if (currentHealth > 0)
        {
            DetectPlayer();

            if (isChasingPlayer)
            {
                ChasePlayer();
                if (_ghost != null)
                {
```

```
                    _ghost.TryStartDialogue();
                }
            }
            else
            {
                Patrol();
            }
            UpdateAnimations();
        }
    }

    void DetectPlayer()
    {
        Collider2D hit =
Physics2D.OverlapCircle(transform.position, detectionRange,
playerLayer);

        if (hit != null && hit.CompareTag("Player"))
        {
            isChasingPlayer = true;
            player = hit.transform;
        }
        else
        {
            isChasingPlayer = false;
            player = null;
        }
    }

    void Patrol()
    {
        Vector2 targetPosition;
        if (movingRight)
        {
            targetPosition = startPosition + Vector2.right *
patrolDistance;
            if (transform.position.x > targetPosition.x)
            {
                movingRight = false;
                Flip();
            }
        }
        else
        {
            targetPosition = startPosition + Vector2.left *
patrolDistance;
            if (transform.position.x < targetPosition.x)
            {
                movingRight = true;
                Flip();
            }
        }
```

```
        if (movingRight)
        {
            rb.MovePosition(rb.position + Vector2.right *
moveSpeed * Time.deltaTime);
        }
        else
        {
            rb.MovePosition(rb.position + Vector2.left *
moveSpeed * Time.deltaTime);
        }
    }

    void ChasePlayer()
    {
        if (player != null)
        {
            Vector2 direction = (player.position -
transform.position).normalized;
            rb.MovePosition(rb.position + direction * chaseSpeed
* Time.deltaTime);
            if (direction.x > 0 && !movingRight)
            {
                Flip();
                movingRight = true;
            }
            else if (direction.x < 0 && movingRight)
            {
                Flip();
                movingRight = false;
            }
        }
        else
        {
            isChasingPlayer = false;
        }
    }

    void Flip()
    {
        transform.localScale = new Vector3(-
transform.localScale.x, transform.localScale.y,
transform.localScale.z);
    }

    public void TakeDamage(int damage)
    {
        Debug.Log("TakeDamage called!  Incoming damage: " +
damage + ", currentHealth BEFORE damage: " + currentHealth);
        Debug.Log(gameObject.name + " took " + damage + "
damage! Current health: " + currentHealth);
        currentHealth -= damage;
```

```
        Debug.Log("currentHealth AFTER damage: " +
currentHealth);
        Debug.Log("New currentHealth: " + currentHealth);
        if (currentHealth <= 0)
        {
            Die();
        }
    }

    void Die()
    {
        Debug.Log(gameObject.name + " died!");
        animator.SetBool("IsDead", true);
        monsterCollider.enabled = false;

        StartCoroutine(DelayedDestroy());
    }

    private IEnumerator DelayedDestroy()
    {
        float animationLength = 0f;
        if (animator.runtimeAnimatorController != null)
        {
            AnimationClip[] clips =
animator.runtimeAnimatorController.animationClips;
            foreach (AnimationClip clip in clips)
            {
                if (clip.name == "Death")
                {
                    animationLength = clip.length;
                    break;
                }
            }
        }
        yield return new WaitForSeconds(animationLength);

        Destroy(gameObject);
    }

    private void OnCollisionStay2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Player") &&
canAttack && !damageAppliedThisFrame)
        {
            PlayerHealth playerHealth =
collision.gameObject.GetComponent<PlayerHealth>();
            if (playerHealth != null)
            {
                playerHealth.TakeDamage(damageAmount);

                Vector2 direction = ((Vector2)transform.position
- (Vector2)collision.transform.position).normalized;
```

```
                ApplyKnockback(direction);

                CharacterMovement characterMovement =
collision.gameObject.GetComponent<CharacterMovement>();
                if (characterMovement != null)
                {
                    characterMovement.ApplyKnockback(-
direction);
                }

                StartCoroutine(AttackDelay());
                damageAppliedThisFrame = true;
            }
        }
    }

    private void LateUpdate()
    {
        damageAppliedThisFrame = false;
    }

    private IEnumerator AttackDelay()
    {
        canAttack = false;
        yield return new WaitForSeconds(attackDelay);
        canAttack = true;
    }

    private void Knockback(Vector2 direction)
    {
        rb.AddForce(direction * knockbackForce,
ForceMode2D.Impulse);
    }

    public void ApplyKnockback(Vector2 direction)
    {
        Knockback(direction);
    }

    void UpdateAnimations()
    {
        animator.SetBool("Idle", !isChasingPlayer);
        animator.SetBool("Run", isChasingPlayer);  // Run, когда
преследует
    }

    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.green;
        Gizmos.DrawWireSphere(transform.position,
detectionRange);
    }
```

```
}
```

Файл MonstersAI.cs:

```
using UnityEngine;
using System.Collections;

public class MonstersAI : MonoBehaviour
{
    [SerializeField] private int maxHealth = 1;
    private int currentHealth;
    private Animator animator;
    private Collider2D monsterCollider;
    private bool isDead = false;

    //  Статическое поле для хранения ссылки на объект игрока
    public static GameObject Player { get; set; }

    void Start()
    {
        currentHealth = maxHealth;
        animator = GetComponent<Animator>();
        monsterCollider = GetComponent<Collider2D>();

        if (animator == null)
        {
            Debug.LogError("Animator component not found!");
            return;
        }

        animator.Play("Idle");
    }

    public void TakeDamage(int damage)
    {
        if (isDead) return;

        currentHealth -= damage;
        Debug.Log(gameObject.name + " took " + damage + "
damage! Current health: " + currentHealth);

        if (currentHealth <= 0)
        {
            Die();
        }
    }

    void Die()
    {
        isDead = true;
        Debug.Log(gameObject.name + " died!");
```

```
        animator.Play("Death");
        monsterCollider.enabled = false;
        StartCoroutine(DelayedDestroy());
    }

    private IEnumerator DelayedDestroy()
    {
        float animationLength = 0f;
        AnimationClip[] clips =
animator.runtimeAnimatorController.animationClips;
        foreach (AnimationClip clip in clips)
        {
            if (clip.name == "Death")
            {
                animationLength = clip.length;
                break;
            }
        }
        yield return new WaitForSeconds(animationLength);

        if (Player != null)
        {
            PlayerHealth playerHealth =
Player.GetComponent<PlayerHealth>();
            CharacterAnimations playerAnimator =
Player.GetComponent<CharacterAnimations>();

            if (playerHealth != null && playerHealth.enabled &&
playerAnimator != null && playerAnimator.enabled)
            {
                playerHealth.Heal(2); // Лечим игрока
                playerAnimator.Heal();
            }
            else
            {
                Debug.LogWarning("Player is not fully
initialized. Skipping heal.");
            }
        }
        else
        {
            Debug.LogWarning("Player object is null. Skipping
heal.");
        }

        Destroy(gameObject);
    }
}
```

Файл FlyingEye.cs:

```
using UnityEngine;
using System.Collections;

public class FlyingEye : MonoBehaviour
{
    public float moveSpeed = 3f;
    public int damage = 1;
    public float detectionRange = 10f;
    public float attackRange = 1f;
    public int health = 1;

    [SerializeField] private float attackDelay = 1f;
    private bool canAttack = true;

    [SerializeField] private float knockbackForce = 3f;

    private Transform player;
    private Animator animator;
    private Rigidbody2D rb;
    private bool isDead = false;

    void Start()
    {
        animator = GetComponent<Animator>();
        rb = GetComponent<Rigidbody2D>();

        if (animator == null)
        {
            Debug.LogError("Animator не найден!");
            enabled = false;
        }

        if (rb == null)
        {
            Debug.LogError("Rigidbody2D не найден!");
            enabled = false;
        }
        FindPlayer();
    }

    void FindPlayer()
    {
        if (player == null)
        {
            GameObject playerObject =
GameObject.FindGameObjectWithTag("Player");
            if (playerObject != null)
            {
                player = playerObject.transform;
            }
            else
            {
```

```
                Debug.LogWarning("Игрок с тегом 'Player' не
найден!");
            }
        }
    }

    void Update()
    {
        if (isDead) return;

        FindPlayer();

        if (player == null) return;

        float distanceToPlayer =
Vector2.Distance(transform.position, player.position);

        if (distanceToPlayer <= detectionRange)
        {
            // Перемещение к игроку
            Vector2 direction = (player.position -
transform.position).normalized;
            rb.linearVelocity = direction * moveSpeed;

            // Проверка attackRange и возможность атаки
            if (distanceToPlayer <= attackRange && canAttack)
            {
                Attack();
            }
        }
        else
        {
            rb.linearVelocity = Vector2.zero;
        }
    }

    void Attack()
    {
        // Нанести урон игроку
        PlayerHealth playerHealth =
player.GetComponent<PlayerHealth>();
        if (playerHealth != null)
        {
            playerHealth.TakeDamage(damage);
        }

        StartCoroutine(AttackDelay()); // Запускаем кулдаун
атаки
    }


    private void OnCollisionStay2D(Collision2D collision)
```

```
    {
        if (collision.gameObject.CompareTag("Player") &&
canAttack)
        {
            PlayerHealth playerHealth =
collision.gameObject.GetComponent<PlayerHealth>();
            if (playerHealth != null)
            {
                playerHealth.TakeDamage(damage);

                Vector2 direction =
((Vector2)transform.position -
(Vector2)collision.transform.position).normalized;
                ApplyKnockback(direction);

                CharacterMovement characterMovement =
collision.gameObject.GetComponent<CharacterMovement>();
                if (characterMovement != null)
                {
                    characterMovement.ApplyKnockback(-
direction);
                }

                StartCoroutine(AttackDelay());
            }
        }
    }

    private IEnumerator AttackDelay()
    {
        canAttack = false;
        yield return new WaitForSeconds(attackDelay);
        canAttack = true;
    }

    public void TakeDamage(int damage)
    {
        health -= damage;
        if (health <= 0)
        {
            Die();
        }
    }

    private void Die()
    {
        animator.SetBool("IsDead", true);
        GetComponent<Collider2D>().enabled = false;
        rb.linearVelocity = Vector2.zero; // Останавливаем
монстра
        rb.isKinematic = true; // Отключаем физику
```

```
        StartCoroutine(DelayedDestroy());
    }

    private IEnumerator DelayedDestroy()
    {
        float animationLength = 0f;
        if (animator.runtimeAnimatorController != null)
        {
            AnimationClip[] clips =
animator.runtimeAnimatorController.animationClips;
            foreach (AnimationClip clip in clips)
            {
                if (clip.name == "Death")
                {
                    animationLength = clip.length;
                    break;
                }
            }
        }
        yield return new WaitForSeconds(animationLength);

        Destroy(gameObject);
    }

    private void Knockback(Vector2 direction)
    {
        rb.AddForce(direction * knockbackForce,
ForceMode2D.Impulse);
    }

    public void ApplyKnockback(Vector2 direction)
    {
        Knockback(direction);
    }

    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(transform.position,
detectionRange);
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position,
attackRange);
    }
}
```

Файл Boss.cs:

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
```

```
using System;

public class Boss : MonoBehaviour
{
    [Header("Health")]
    [SerializeField] private int maxHealth = 10;
    private int currentHealth;
    public Slider healthBar;
    public GameObject healthBarObject;

    [Header("Vision")]
    [SerializeField] private float visionRange = 10f;
    [SerializeField] private LayerMask targetLayers;
    [SerializeField] private Transform visionOrigin;

    [Header("Attack")]
    [SerializeField] private float attackCooldown = 2f;
    [SerializeField] private float attackAnimationDelay = 0.5f;
    [SerializeField] public GameObject fireBallPrefab;
    [SerializeField] public GameObject knockbackPrefab;
    public GameObject knockbackBigPrefab;
    [SerializeField] public Transform attackPoint;

    public int phase2HealthThreshold = 4; // Порог здоровья для
перехода во вторую фазу
    public bool _isPhase2 = false; // Флаг, показывающий, что
босс во второй фазе
    public bool isInvincible = false; // Флаг неуязвимости босса

    [Header("Phase 2")]
    [SerializeField] private GameObject monsterEyePrefab;
    [SerializeField] private Transform[] monsterEyeSpawnPoints;
    [SerializeField] private float monsterEyeSpawnInterval = 5f;
// Интервал спавна MonsterEye в секундах
    private float _nextMonsterEyeSpawnTime; // Время следующего
спавна MonsterEye

    [Header("Slowmotion")]
    public float slowMotionTimeScale = 0.2f; // Степень
замедления времени
    public float slowMotionDuration = 3f;   // Длительность
замедленного времени

    [Header("Dialogue")]
    public BossDialogue bossDialogue;

    [Header("Components")]
    private Animator animator;
    private Collider2D bossCollider;
    private float originalTimeScale;
    private GameObject _player;
    private bool isDead = false;
```

```
    private float _attackTimer = 0f;
    private bool isAttacking = false;

    public static event Action OnBossDeath;

    void Start()
    {
        currentHealth = maxHealth;
        animator = GetComponent<Animator>();
        bossCollider = GetComponent<Collider2D>();

        if (animator == null)
        {
            Debug.LogError("Animator component not found!");
            enabled = false;
            return;
        }
        if (bossCollider == null)
        {
            Debug.LogError("Collider2D component not found!");
            enabled = false;
            return;
        }

        _player = GameObject.FindGameObjectWithTag("Player");
        if (_player == null)
        {
            Debug.LogError("Player not found. Make sure the
player has the 'Player' tag.");
            enabled = false;
            return;
        }

        if (bossDialogue == null)
        {
            bossDialogue = GetComponent<BossDialogue>();
            if (bossDialogue == null)
            {
                Debug.LogError("BossAttack requires a
BossDialogue component!");
            }
        }

        healthBar.maxValue = maxHealth;
        healthBar.value = currentHealth;
        if (healthBarObject == null)
        {
            healthBarObject = healthBar.gameObject;
        }
        originalTimeScale = Time.timeScale;
    }
```

```
    void Update()
    {
        if (isDead) return;

        _attackTimer += Time.deltaTime;

        if (CanSeePlayer() && _attackTimer >= attackCooldown)
        {
            Attack();
            _attackTimer = 0f;
        }

        if (_isPhase2)
        {
            // Логика спавна MonsterEye
            if (Time.time >= _nextMonsterEyeSpawnTime)
            {
                SpawnMonsterEye();
                _nextMonsterEyeSpawnTime = Time.time +
monsterEyeSpawnInterval;
            }
        }
    }

    bool CanSeePlayer()
    {
        if (_player == null || visionOrigin == null)
        {
            Debug.LogWarning("Player or VisionOrigin is null!");
            return false;
        }

        float distanceToPlayer =
Vector2.Distance(visionOrigin.position,
_player.transform.position);

        bool seesPlayer = distanceToPlayer <= visionRange;
        return seesPlayer;
    }

    void Attack()
    {
        if (isAttacking) return;

        if (bossDialogue == null ||
!bossDialogue.IsDialoguePlaying)
        {
            float randomValue = UnityEngine.Random.Range(0f,
1f);

            if (randomValue < 0.5f) // 50% Fireball
            {
```

```
                Debug.Log("Fireball Attack - Triggering Attack
animation");
                animator.SetTrigger("Attack");
                Invoke("CreateFireball", attackAnimationDelay);
            }
            else // 50% Knockback
            {
                Debug.Log("Knockback Attack - Triggering
Knockback animation");
                animator.SetTrigger("Knockback");
                StartCoroutine(KnockbackAttackRoutine()); //
Запускаем корутину для серии атак Knockback
            }

            Invoke("EndAttack", attackAnimationDelay);
        }
        else
        {
            Debug.Log("Boss is not attacking (dialogue is
playing).");
            // Атака не выполняется, так как идет диалог
        }
    }

    void CreateFireball()
    {
        GameObject fireBall = Instantiate(fireBallPrefab,
attackPoint.position, Quaternion.identity);

    }

    private IEnumerator KnockbackAttackRoutine()
    {
        animator.SetTrigger("Knockback");

        float knockbackDelay = 0.2f;

        yield return new WaitForSeconds(0.5f);

        for (int i = 0; i < 3; i++)
        {
            CreateKnockback();
            yield return new WaitForSeconds(knockbackDelay);
        }
    }


    void CreateKnockback()
    {
        GameObject knockback = Instantiate(knockbackPrefab,
attackPoint.position, Quaternion.identity);
    }
```

```
    public void EndAttack()
    {
        animator.SetBool("IsAttacking", false);
        animator.ResetTrigger("Attack");
        animator.ResetTrigger("Knockback");
        animator.ResetTrigger("KnockbackBig");
    }

    public void TakeDamage(int damage)
    {
        if (isDead) return;
        if (isInvincible) return;

        currentHealth -= damage;
        Debug.Log(gameObject.name + " took " + damage + "
damage! Current health: " + currentHealth);

        healthBar.value = currentHealth;

        animator.SetTrigger("IsHit");

        if (currentHealth <= phase2HealthThreshold &&
!_isPhase2)
        {
            // Переход во вторую фазу
            StartCoroutine(Phase2Transition());
        }

        if (currentHealth <= 0)
        {
            Die();
            if (healthBarObject != null)
            {
                healthBarObject.SetActive(false);
            }
        }
    }

    void CreateKnockbackBig()
    {
        Debug.Log("CreateKnockbackBig() called");
        // Создаем KnockbackBig
        GameObject knockbackBig =
Instantiate(knockbackBigPrefab, attackPoint.position,
Quaternion.identity);
    }

    private IEnumerator Phase2Transition()
    {
        _isPhase2 = true; // Устанавливаем флаг, что босс во
второй фазе
```

```
        isInvincible = true; // Босс становится неуязвимым
        isAttacking = true; // Босс не может атаковать во время
анимации KnockbackBig

        animator.SetTrigger("KnockbackBig");

        // Получаем длительность анимации KnockbackBig
        float animationLength = 0f;
        AnimationClip[] clips =
animator.runtimeAnimatorController.animationClips;
        foreach (AnimationClip clip in clips)
        {
            if (clip.name == "KnockbackBig")
            {
                animationLength = clip.length;
                break;
            }
        }

        yield return new WaitForSeconds(animationLength);

        float delayBeforeKnockbackBig = 1.2f;
        yield return new
WaitForSeconds(delayBeforeKnockbackBig);

        // Создаём KnockbackBig один раз
        CreateKnockbackBig();

        isAttacking = false; // Босс снова может атаковать
        isInvincible = false; // Босс снова становится уязвимым

        _nextMonsterEyeSpawnTime = Time.time;
    }

    private void SpawnMonsterEye()
    {
        if (monsterEyePrefab != null &&
monsterEyeSpawnPoints.Length > 0)
        {
            // Выбираем случайную точку спавна
            Transform spawnPoint =
monsterEyeSpawnPoints[UnityEngine.Random.Range(0,
monsterEyeSpawnPoints.Length)];
            Instantiate(monsterEyePrefab, spawnPoint.position,
spawnPoint.rotation);
        }
        else
        {
            Debug.LogWarning("MonsterEyePrefab is null or no
spawn points defined!");
        }
    }
```

```
    void Die()
    {
        isDead = true;
        bossCollider.enabled = false;
        animator.SetTrigger("IsDead");

        // Показываем фразу смерти перед SlowMotion
        if (bossDialogue != null &&
!string.IsNullOrEmpty(bossDialogue.deathTaunt))
        {

bossDialogue.ShowFloatingText(bossDialogue.deathTaunt);
        }

        StartCoroutine(SlowMotionDeath()); // Замедляем время
    }

    private System.Collections.IEnumerator SlowMotionDeath()
    {
        float originalAnimationSpeed = animator.speed;

        // Замедляем время и анимацию
        Time.timeScale = slowMotionTimeScale;
        Time.fixedDeltaTime = 0.02F * Time.timeScale; //
Обновляем fixedDeltaTime, чтобы физика работала корректно в
замедленном времени
        animator.speed = slowMotionTimeScale;

        // Ждем, пока длится замедленное время
        yield return new
WaitForSecondsRealtime(slowMotionDuration);

        // Восстанавливаем нормальное время и скорость анимации
        Time.timeScale = originalTimeScale;
        Time.fixedDeltaTime = 0.02F;
        animator.speed = originalAnimationSpeed;

        Debug.Log("Boss is dying! Destroying GameObject.");
        Destroy(gameObject);

        // Вызываем событие смерти босса
        if (OnBossDeath != null)
        {
            OnBossDeath();
        }
    }

    private void OnDrawGizmosSelected()
    {
        if (monsterEyeSpawnPoints != null)
        {
```

```
            Gizmos.color = Color.yellow;
            foreach (Transform spawnPoint in
monsterEyeSpawnPoints)
            {
                if (spawnPoint != null)
                {
                    Gizmos.DrawSphere(spawnPoint.position,
0.5f);
                }
            }
        }
    }
}
```

Файл Fireball.cs:

```
using UnityEngine;

public class Fireball : MonoBehaviour
{
    public float speed = 5f;
    public float lifeTime = 2f;
    public int damage = 1;
    public float trackingTime = 2f; // Время преследования цели

    private Vector2 _direction;
    private Transform _player;
    private float _trackingTimer;

    void Start()
    {
        Destroy(gameObject, lifeTime);
        _player =
GameObject.FindGameObjectWithTag("Player").transform;
        if (_player == null)
        {
            Debug.LogWarning("Player not found!");
            Destroy(gameObject);
        }
        _trackingTimer = 0f; // Инициализируем таймер
        if (_player != null)
        {
            _direction = ((Vector2)_player.position -
(Vector2)transform.position).normalized; // Начальное
направление
        }
    }

    void Update()
    {
        if (_player == null)
```

```
        {
            Destroy(gameObject);
            return;
        }

        if (_trackingTimer < trackingTime) // Если время
преследования еще не истекло
        {
            _direction = ((Vector2)_player.position -
(Vector2)transform.position).normalized; // Обновляем
направление
            _trackingTimer += Time.deltaTime; // Увеличиваем
таймер
        }

        transform.Translate(_direction * speed *
Time.deltaTime);
    }

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            // Наносим урон игроку
            PlayerHealth playerHealth =
other.GetComponent<PlayerHealth>();
            if (playerHealth != null)
            {
                playerHealth.TakeDamage(damage);
            }
            Destroy(gameObject);
        }
    }
}
```

Файл Knockback.cs:

```
using UnityEngine;

public class Knockback : MonoBehaviour
{
    public float speed = 5f;
    public float lifeTime = 2f; // Время жизни объекта
    public int pushForce = 10;
    public float trackingTime = 2f; // Время преследования
цели

    private Vector2 _direction;
    private Transform _player;
    private float _trackingTimer;
```

```
    void Start()
    {
        Destroy(gameObject, lifeTime);
        _player =
GameObject.FindGameObjectWithTag("Player").transform;
        if (_player == null)
        {
            Debug.LogWarning("Player not found!");
            Destroy(gameObject);
        }

        _trackingTimer = 0f; // Инициализируем таймер
        if (_player != null)
        {
            _direction = ((Vector2)_player.position -
(Vector2)transform.position).normalized; // Начальное
направление
        }
    }

    void Update()
    {
        if (_player == null)
        {
            Destroy(gameObject);
            return;
        }

        if (_trackingTimer < trackingTime) // Если время
преследования еще не истекло
        {
            _direction = ((Vector2)_player.position -
(Vector2)transform.position).normalized; // Обновляем
направление
            _trackingTimer += Time.deltaTime; // Увеличиваем
таймер
        }

        transform.Translate(_direction * speed *
Time.deltaTime);
    }

    void OnTriggerEnter2D(Collider2D other)
    {
        // Проверяем, столкнулся ли объект с игроком
        if (other.gameObject.CompareTag("Player"))
        {
            Rigidbody2D playerRb =
other.GetComponent<Rigidbody2D>();
            if (playerRb != null)
            {
                Vector2 pushDirection = _direction;
```

```
                playerRb.AddForce(pushDirection * pushForce,
ForceMode2D.Impulse);
                Debug.Log("Player pushed!");
            }
            else
            {
                Debug.LogWarning("Player has no
Rigidbody2D!");
            }
        }
    }
}
```

Скрипты диалогов:
Файл DialogueManager.cs:

```
using UnityEngine;
using UnityEngine.UI;
using Ink.Runtime;
using TMPro;
using System;
using System.Collections;
using System.Collections.Generic;

public class DialogueManager : MonoBehaviour
{
    public GameObject dialoguePanel;
    public TextMeshProUGUI dialogueText;
    public TextMeshProUGUI speakerNameText;
    public AudioSource textSound;

    public event Action OnDialogueStarted; // Событие начала
диалога
    public event Action OnDialogueEnded;   // Событие
окончания диалога

    public static DialogueManager Instance;

    private Story _currentStory;
    public bool dialogueIsPlaying { get; private set; }

    [Tooltip("Characters per second")]
    public float textSpeed = 50f;

    [Tooltip("Color of Cody's name (hex code)")]
    public string codyNameColor = "BC0741";

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

```
            Debug.LogWarning("More than one Dialogue Manager
in the scene!");
        }
        Instance = this;
    }

    private void Start()
    {
        dialogueIsPlaying = false;
        dialoguePanel.SetActive(false);
    }

    private void Update()
    {
        if (dialogueIsPlaying && (Input.GetMouseButtonDown(0)
|| Input.GetKeyDown(KeyCode.Space)))
        {
            OnClickContinue();
        }
    }

    public void EnterDialogueMode(TextAsset inkStory)
    {
        _currentStory = new Story(inkStory.text);
        _currentStory.ChoosePathString("main");
        dialogueIsPlaying = true;
        dialoguePanel.SetActive(true);

        ContinueStory();
        OnDialogueStarted?.Invoke();
    }

    private void ExitDialogueMode()
    {
        StopAllCoroutines();
        dialogueIsPlaying = false;
        dialoguePanel.SetActive(false);
        dialogueText.text = "";
        speakerNameText.text = "";
        OnDialogueEnded?.Invoke();
    }

    private void ContinueStory()
    {
        if (_currentStory.canContinue)
        {
            string nextLine = _currentStory.Continue();
            string speakerName =
ExtractSpeakerName(_currentStory.currentTags);

            if (speakerName == "Коди")
            {
```

```
                speakerNameText.text =
$"<color=#{codyNameColor}>{speakerName}</color>";
            }
            else
            {
                speakerNameText.text = speakerName;
            }

            StopAllCoroutines();
            StartCoroutine(TypeSentence(nextLine));
        }
        else
        {
            ExitDialogueMode();
        }
    }

    private string ExtractSpeakerName(List<string>
currentTags)
    {
        foreach (string tag in currentTags)
        {
            string[] parts = tag.Split(':');
            if (parts.Length == 2 && parts[0].Trim() ==
"speaker")
            {
                return parts[1].Trim();
            }
        }
        return "";
    }

    private IEnumerator TypeSentence(string sentence)
    {
        dialogueText.text = "";
        float timePerCharacter = 1f / textSpeed;
        for (int i = 0; i < sentence.Length; i++)
        {
            dialogueText.text += sentence[i];

            if (i % 4 == 0)
            {
                textSound.Play();
            }

            yield return new WaitForSeconds(timePerCharacter);
        }
    }

    public void OnClickContinue()
    {
        ContinueStory();
```

```
    }

    public TextAsset GetCurrentInk()
    {
        if (_currentStory != null)
        {
            //  Находим TextAsset по имени
            string storyName =
_currentStory.ToJson().Substring(2, 12);

            // Возвращаем inkFile
            return Resources.Load<TextAsset>(storyName);

        }
        else
        {
            Debug.LogError("No current story!");
            return null;
        }
    }
}
```

Файл NPCCore.cs:

```
using UnityEngine;

public class NPCCore : MonoBehaviour
{
    public DialogueManager dialogueManager;
    public TextAsset[] inkFiles;  // Массив диалогов
    private bool _isInRange;
    private bool _dialogueStarted;

    void Start()
    {
        if (inkFiles == null || inkFiles.Length == 0)
        {
            Debug.LogError("No ink files assigned to the
NPC!");
            enabled = false;
            return;
        }
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Q) && _isInRange &&
!_dialogueStarted)
        {
            StartDialogue();
        }
```

```
    }

    public void StartDialogue()
    {
        TextAsset inkToUse = null;

        // Определяем, какой файл Ink использовать
        if (PlayerPrefs.GetInt("Completelevel3") == 1 &&
inkFiles.Length > 3)
        {
            inkToUse = inkFiles[3];
            Debug.Log("Загружен Ink файл 3 (Completelevel3 ==
1)");
        }
        else if (PlayerPrefs.GetInt("Completelevel2") == 1 &&
inkFiles.Length > 2)
        {
            inkToUse = inkFiles[2];
            Debug.Log("Загружен Ink файл 2 (Completelevel2 ==
1)");
        }
        else if (PlayerPrefs.GetInt("FlowerBloom") == 1 &&
inkFiles.Length > 1)
        {
            inkToUse = inkFiles[1];
            Debug.Log("Загружен Ink файл 1 (FlowerBloom ==
1)");
        }
        else
        {
            inkToUse = inkFiles[0];
            Debug.Log("Загружен Ink файл 0 (По умолчанию)");
        }

        // Запускаем диалог
        if (dialogueManager != null && inkToUse != null)
        {
            dialogueManager.EnterDialogueMode(inkToUse);
            _dialogueStarted = true;
            PlayerPrefs.SetInt("DefaultDialogueRead", 1);
            dialogueManager.OnDialogueEnded +=
OnDialogueEnded;
        }
        else
        {
            Debug.LogError("DialogueManager is not assigned or
inkFile is null!");
        }
    }

    void OnDialogueEnded()
    {
```

```
        dialogueManager.OnDialogueEnded -= OnDialogueEnded;
        _dialogueStarted = false;
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            _isInRange = true;
            Debug.Log("Player in range");
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            _isInRange = false;
        }
    }
}
```

Скрипты переходов:

Файл LevelTransition.cs:

```
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections;

public class LevelTransition : MonoBehaviour
{
    [Tooltip("Индекс сцены, на которую нужно переключиться.")]
    public int sceneToLoad;

    [Tooltip("Объект, который нужно активировать после
срабатывания триггера (например, дверь). Оставьте пустым, если
не нужно активировать объект.")]
    public GameObject objectToActivate;

    [Tooltip("Тэг объекта, который может активировать триггер.
Например, 'Player'.")]
    public string triggerTag = "Player";

    [Tooltip("Задержка перед переходом на новую сцену (в
секундах).")]
    public float delayBeforeLoad = 0.7f;

    private bool _transitioning = false; //флаг, что переход
уже запущен
```

```
    void Start()
    {
        if (objectToActivate != null)
        {
            objectToActivate.SetActive(false); // Изначально
деактивируем объект
        }
    }

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag(triggerTag) && !_transitioning)
        {
            _transitioning = true; //блокируем повторное
срабатывание

            CompleteLevel();
            Debug.Log("Уровень пройден!");
            if (objectToActivate != null)
                objectToActivate.SetActive(true);

            // Запускаем корутину для смены сцены с задержкой
            StartCoroutine(LoadSceneWithDelay(sceneToLoad,
delayBeforeLoad));
        }
    }

    void CompleteLevel()
    {
        // Устанавливаем в PlayerPref, что SceneHHD была
пройдена
        PlayerPrefs.SetInt("SceneHHDCompleted", 1);
        PlayerPrefs.Save();
    }

    // Корутина для загрузки сцены с задержкой
    private IEnumerator LoadSceneWithDelay(int sceneIndex,
float delay)
    {
        Debug.Log("Начало ожидания перед загрузкой сцены...");
        yield return new WaitForSeconds(delay);
        Debug.Log("Время ожидания истекло. Загрузка сцены с
индексом: " + sceneIndex);
        SceneManager.LoadScene(sceneIndex);
    }
}
```

Файл LevelTransitionButton.cs:

```
using UnityEngine;
using UnityEngine.SceneManagement;
```

```
public class LevelTransitionButton : MonoBehaviour
{
    [Tooltip("Индекс сцены, на которую нужно переключиться.")]
    public int sceneToLoad;

    [Tooltip("Объект, который нужно активировать после нажатия
кнопки (например, дверь). Оставьте пустым, если не нужно
активировать объект.")]
    public GameObject objectToActivate;

    [Tooltip("Тэг объекта, который может активировать триггер.
Например, 'Player'.")]
    public string triggerTag = "Player";

    [Tooltip("Кнопка, которую нужно нажать для взаимодействия
с дверью. Например, 'w'.")]
    public KeyCode interactionKey = KeyCode.W;

    [Tooltip("Задержка перед переходом на новую сцену (в
секундах).")]
    public float delayBeforeLoad = 1.1f;

    [Tooltip("Объект игрока, который нужно отключить на время
перехода.")]
    public GameObject playerObject;

    private bool _transitioning = false; // Флаг, что переход
уже запущен
    private bool _playerInRange = false;

    void Start()
    {
        if (objectToActivate != null)
        {
            objectToActivate.SetActive(false); // Изначально
деактивируем объект
        }
    }

    void Update()
    {
        if (_playerInRange && Input.GetKeyDown(interactionKey)
&& !_transitioning)
        {
            _transitioning = true; // Блокируем повторное
срабатывание

            Debug.Log("Кнопка взаимодействия нажата. Объект с
тегом " + triggerTag + " начал переход.");

            if (objectToActivate != null)
```

```
            {
                objectToActivate.SetActive(true);
                Debug.Log("Активирован объект: " +
objectToActivate.name);
            }

            // Отключаем управление игроком
            DisablePlayerControl();

            // Запускаем корутину для смены сцены с задержкой
            StartCoroutine(LoadSceneWithDelay(sceneToLoad,
delayBeforeLoad));
        }
    }

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag(triggerTag))
        {
            Debug.Log("Игрок вошел в зону действия двери.");
            _playerInRange = true;
        }
    }

    void OnTriggerExit2D(Collider2D other)
    {
        if (other.CompareTag(triggerTag))
        {
            Debug.Log("Игрок вышел из зоны действия двери.");
            _playerInRange = false;
        }
    }

    // Корутина для загрузки сцены с задержкой
    private System.Collections.IEnumerator
LoadSceneWithDelay(int sceneIndex, float delay)
    {
        Debug.Log("Начало ожидания перед загрузкой сцены...");
        yield return new WaitForSeconds(delay);
        // Находим объект SceneState и уничтожаем его
        SceneState sceneState =
FindAnyObjectByType<SceneState>();
        if (sceneState != null)
        {
            Destroy(sceneState.gameObject);
            Debug.Log("SceneState destroyed before loading new
scene.");
        }

        Debug.Log("Время ожидания истекло. Загрузка сцены с
индексом: " + sceneIndex);
        SceneManager.LoadScene(sceneIndex); // Загружаем сцену
```

```
    }

    // Метод для отключения управления игроком
    private void DisablePlayerControl()
    {
        if (playerObject != null)
        {
            CharacterMovement characterMovement =
playerObject.GetComponent<CharacterMovement>();
            if (characterMovement != null)
            {
                characterMovement.enabled = false;
            }

            SpriteRenderer spriteRenderer =
playerObject.GetComponent<SpriteRenderer>();
            if (spriteRenderer != null)
            {
                spriteRenderer.enabled = false;
            }

            Animator animator =
playerObject.GetComponent<Animator>();
            if (animator != null)
            {
                animator.enabled = false;
            }

            Debug.Log("Управление игроком отключено.");
        }
    }
}
```

Скрипты сохранений:

Файл LevelUnlocker.cs:

```
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class LevelUnlocker : MonoBehaviour
{
    public GameObject level1Door;
    public GameObject level2Door;
    public GameObject level3Door;
    public GameObject level4Door;

    void Start()
    {
        // Скрываем двери уровней
```

```
        level1Door.SetActive(false);
        level2Door.SetActive(false);
        level3Door.SetActive(false);
        level4Door.SetActive(false);

        CheckLevelUnlocks();
    }

    void Update()
    {
        // Постоянно проверяем состояние двери Level1
        CheckLevel1Unlock();
    }

    private void CheckLevel1Unlock()
    {
        if (PlayerPrefs.GetInt("DefaultDialogueRead") == 1)
        {
            UnlockLevel1();
        }
    }

    private void CheckLevelUnlocks()
    {
        //Разблокировка Level 2
        if (PlayerPrefs.GetInt("FlowerBloom") == 1) /
        {
            UnlockLevel2();
        }
        else
        {
            Debug.Log("Flower is not yet blooming.");
        }

        // Разблокировка Level 3
        if (PlayerPrefs.GetInt("Completelevel2") == 1)
        {
            UnlockLevel3();
        }
        else
        {
            Debug.Log("Level 2 is not yet complete.");
        }

        //Разблокировка Level 4
        if (PlayerPrefs.GetInt("Completelevel3") == 1)
        {
            UnlockLevel4();
        }
        else
        {
            Debug.Log("Level 3 is not yet complete.");
```

```
        }

        PlayerPrefs.Save();
    }

    private void UnlockLevel1()
    {
        level1Door.SetActive(true);
        Debug.Log("Level 1 unlocked!");
    }

    private void UnlockLevel2()
    {
        level2Door.SetActive(true);
        Debug.Log("Level 2 unlocked!");
    }

    private void UnlockLevel3()
    {
        level3Door.SetActive(true);
        Debug.Log("Level 3 unlocked!");
    }

    private void UnlockLevel4()
    {
        level4Door.SetActive(true);
        Debug.Log("Level 4 unlocked!");
    }
}
```

Файл ResetPlayerPrefs.cs:

```
using UnityEngine;

public class ResetPlayerPrefs : MonoBehaviour
{
    public void ResetAllPlayerPrefs()
    {
        PlayerPrefs.DeleteAll();  // Удаляет все сохраненные
данные
        PlayerPrefs.Save();
        Debug.Log("PlayerPrefs reset!");
    }
}
```

Скрипт перехода между сценами:

Файл SceneLoader.cs:

```
using UnityEngine;
using UnityEngine.SceneManagement;
```

```
public class SceneLoader : MonoBehaviour
{
    public string nextSceneName;

    public void LoadNextScene()
    {
        SceneManager.LoadScene(nextSceneName);
    }
}
```

Скрипты аудио:

Файл AudioManager.cs:

```
using UnityEngine;
using UnityEngine.Audio;
using System;
using System.Collections.Generic;

public enum SoundType
{
    Master,
    Sound,
    Music,
}

public class AudioManager : MonoBehaviour
{
    public static AudioManager Instance;

    [SerializeField] private AudioMixer _audioMixer;

    private const string MasterVolumeKey = "MasterVolume";
    private const string SoundVolumeKey = "SoundVolume";
    private const string MusicVolumeKey = "MusicVolume";

    private const string MIXER_MASTER_VOLUME = "MasterVolume";
    private const string MIXER_SOUND_VOLUME = "SoundVolume";
    private const string MIXER_MUSIC_VOLUME = "MusicVolume";

    private bool _musicMuted = false;
    private bool _soundMuted = false;

    private float _soundVolume = 1f;
    private float _musicVolume = 1f;

    private Dictionary<SoundType, float> _volumes = new
Dictionary<SoundType, float>();

    private void Awake()
```

```
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);

            LoadSoundSettings();
        }
        else
        {
            Destroy(gameObject);
            return;
        }
    }

    public void SetVolume(SoundType soundType, float volume)
    {
        volume = Mathf.Clamp01(volume);
        string parameterName = soundType.ToString() + "Volume";
        float logVolume = Mathf.Log10(Mathf.Clamp(volume,
0.0001f, 1f)) * 20f;
        _audioMixer.SetFloat(parameterName, logVolume);
        _volumes[soundType] = volume;
        SaveVolume(soundType, volume);
    }
    public float GetVolume(SoundType soundType)
    {
        if (_volumes.ContainsKey(soundType))
        {
            return _volumes[soundType];
        }

        float volume = 1f;
        string key = GetKeyForSoundType(soundType);

        if (PlayerPrefs.HasKey(key))
        {
            volume = PlayerPrefs.GetFloat(key, 1f);
        }

        _volumes[soundType] = volume;
        return volume;
    }

    public void ToggleMusicMute()
    {
        _musicMuted = !_musicMuted;
        _audioMixer.SetFloat(MIXER_MUSIC_VOLUME, _musicMuted ? -
80f : Mathf.Log10(Mathf.Max(_musicVolume, 0.0001f)) * 20);
        Debug.Log("Music Muted: " + _musicMuted);
    }
```

```
    public void ToggleSoundMute()
    {
        _soundMuted = !_soundMuted;
        _audioMixer.SetFloat(MIXER_SOUND_VOLUME, _soundMuted ? -
80f : Mathf.Log10(Mathf.Max(_soundVolume, 0.0001f)) * 20);
        Debug.Log("Sound Muted: " + _soundMuted);
    }

    private void SaveVolume(SoundType soundType, float volume)
    {
        string key = GetKeyForSoundType(soundType);
        PlayerPrefs.SetFloat(key, volume);
        PlayerPrefs.Save();
    }

    private string GetKeyForSoundType(SoundType soundType)
    {
        switch (soundType)
        {
            case SoundType.Master:
                return MasterVolumeKey;
            case SoundType.Sound:
                return SoundVolumeKey;
            case SoundType.Music:
                return MusicVolumeKey;
            default:
                Debug.LogError("Invalid SoundType!");
                return "";
        }
    }

    public void LoadSoundSettings()
    {
        SetVolume(SoundType.Master,
GetVolume(SoundType.Master));
        SetVolume(SoundType.Sound, GetVolume(SoundType.Sound));
        SetVolume(SoundType.Music, GetVolume(SoundType.Music));
    }
}
```

Файл GameSettingsController.cs:

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Audio;

public class GameSettingsController : MonoBehaviour
{
    [SerializeField]
    private AudioMixer _audioMixer;
```

```
    [SerializeField]
    private VolumeController _masterVolumeController;

    private void Start()
    {
        if (_audioMixer == null)
        {
            Debug.LogError("AudioMixer не назначен в
GameSettingsController!");
            return;
        }

        if (_masterVolumeController == null)
        {
            Debug.LogError("Master VolumeController не
назначен в GameSettingsController!");
            return;
        }

        _masterVolumeController.soundType = SoundType.Master;

        Slider masterSlider =
_masterVolumeController.GetComponent<Slider>();
        if (masterSlider == null)
        {
            Debug.LogError("Slider не найден на GameObject с
VolumeController!");
            return;
        }

        masterSlider.value =
AudioManager.Instance.GetVolume(SoundType.Master);
    }
}
```

Файл MusicManager.cs:

```
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections.Generic;

public class MusicManager : MonoBehaviour
{
    public static MusicManager Instance; // Singleton

    [SerializeField] private AudioSource musicSource;

    [SerializeField] private List<ScenePlaylist> scenePlaylists
= new List<ScenePlaylist>();

    private string _currentSceneName;
```

```
    private int _currentTrackIndex = 0;

    // Словарь для быстрого поиска плейлиста по имени сцены
    private Dictionary<string, List<AudioClip>>
_scenePlaylistsDict = new Dictionary<string, List<AudioClip>>();

    private void Awake()
    {
        // Singleton pattern
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject); // Сохраняем
MusicManager между сценами
        }
        else
        {
            Destroy(gameObject);
            return;
        }

        // Заполняем словарь из списка
        foreach (var scenePlaylist in scenePlaylists)
        {
            if
(!_scenePlaylistsDict.ContainsKey(scenePlaylist.sceneName))
            {
                _scenePlaylistsDict.Add(scenePlaylist.sceneName,
scenePlaylist.playlist);
            }
        }
    }

    void Start()
    {
        // Подписываемся на событие загрузки сцены
        SceneManager.sceneLoaded += OnSceneLoaded;

        // Запускаем музыку для первой сцены
        _currentSceneName = SceneManager.GetActiveScene().name;
        PlayAppropriateTrack(); // Используем новый метод
    }

    void OnSceneLoaded(Scene scene, LoadSceneMode mode)
    {
        _currentSceneName = scene.name;
        PlayAppropriateTrack(); // Используем новый метод
    }

    void PlayAppropriateTrack()
    {
```

```
        // Если мы в SceneHHD или SceneHHD2, играем один и тот
же трек
        if (_currentSceneName == "SceneHHD" || _currentSceneName
== "SceneHHD2")
        {
            PlaySpecificTrack("SceneHHD"); // Используем
плейлист SceneHHD
        }
        else
        {
            // Для других сцен используем стандартный плейлист
            PlayNormalTrack(_currentSceneName);
        }
    }

    // Метод для воспроизведения конкретного трека для SceneHHD
и SceneHHD2
    void PlaySpecificTrack(string sceneName)
    {
        // Проверяем, есть ли плейлист для указанной сцены
        if (!_scenePlaylistsDict.ContainsKey(sceneName))
        {
            Debug.LogWarning("No playlist found for scene: " +
sceneName);
            musicSource.Stop();
            return;
        }

        List<AudioClip> playlist =
_scenePlaylistsDict[sceneName];

        // Проверяем, что плейлист не пуст
        if (playlist == null || playlist.Count == 0)
        {
            Debug.LogWarning("Playlist for scene " + sceneName
+ " is empty.");
            musicSource.Stop();
            return;
        }

        // Получаем первый трек из плейлиста
        AudioClip trackToPlay = playlist[0];

        // Проигрываем трек, если он отличается от текущего
        if (musicSource.clip != trackToPlay)
        {
            musicSource.clip = trackToPlay;
            musicSource.loop = true;
            musicSource.Play();
            Debug.Log("Playing specific track " +
trackToPlay.name + " for " + _currentSceneName);
        }
```

```
        else
        {
            Debug.Log("Specific track already playing in " +
_currentSceneName);
            musicSource.loop = true;  // Убеждаемся, что трек
зациклен
        }
    }

    // Метод для других сцен
    void PlayNormalTrack(string sceneName)
    {
        // Проверяем, есть ли плейлист для указанной сцены
        if (!_scenePlaylistsDict.ContainsKey(sceneName))
        {
            Debug.LogWarning("No playlist found for scene: " +
sceneName);
            musicSource.Stop();
            return;
        }

        List<AudioClip> playlist =
_scenePlaylistsDict[sceneName];

        // Проверяем, что плейлист не пуст
        if (playlist == null || playlist.Count == 0)
        {
            Debug.LogWarning("Playlist for scene " + sceneName
+ " is empty.");
            musicSource.Stop();
            return;
        }

        // Случайный выбор трека
        int randomIndex = Random.Range(0, playlist.Count);
        if (playlist.Count > 1 && randomIndex ==
_currentTrackIndex)
        {
            randomIndex = (randomIndex + 1) % playlist.Count;
        }
        _currentTrackIndex = randomIndex;

        // Получаем трек
        AudioClip trackToPlay = playlist[randomIndex];

        // Проигрываем трек, если он отличается от текущего
        if (musicSource.clip != trackToPlay)
        {
            musicSource.clip = trackToPlay;
            musicSource.loop = false; // Отключаем зацикливание,
чтобы корутина работала
            musicSource.Play();
```

```
            Debug.Log("Playing normal track " + trackToPlay.name
+ " in scene " + sceneName);

            StartCoroutine(TrackEndRoutine());
        }
        else
        {
            Debug.Log("Normal track already playing in scene " +
sceneName);
        }
    }

    private System.Collections.IEnumerator TrackEndRoutine()
    {
        yield return new WaitUntil(() =>
!musicSource.isPlaying);
        PlayAppropriateTrack();
    }

    private void OnDestroy()
    {
        SceneManager.sceneLoaded -= OnSceneLoaded;
    }
}
```

Приложение В

(Обязательное)

Приложение на Unity для интерактивного исследования архитектуры компьютера

Презентация