

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Пензенский государственный университет»

Политехнический институт
(наименование института полностью)

Кафедра «Системы автоматизированного проектирования»
(наименование)

09.03.01 «Информатика и вычислительная техника»
(код и наименование направления подготовки, специальности)

«Системы автоматизированного проектирования»
(направленность (профиль) / специализация)

КУРСОВАЯ РАБОТА
по курсу «Технология разработки программного
обеспечения»

на тему «Разработка 2D компьютерной игры»

Обучающийся

Д. А. Нагорная
(Инициалы Фамилия)



(личная подпись)

Руководитель

к.т.н, доцент каф. САПР В.В. Эпп

(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Пенза 2024

Оглавление

Введение.....	3
1 Анализ предметной области	4
1.1 Жанр игры. Аналоги игр в этом жанре.....	4
1.2 Описание среды разработки.....	9
1.3 Техническое задание на разработку.....	10
2 Проектирования и разработка приложения.....	14
2.1 Проектирование уровней и логики игры.....	14
2.2 Описание основных классов и скриптов	17
2.3 Ресурсы проекта	26
2.4 Основные анимации игры	29
3 Тестирование приложения	33
3.1 Описание тестовых сценариев.....	33
3.2 Сборка приложения	35
Заключение	35
Список использованных источников	38
Приложение	39

Введение

В последние годы индустрия компьютерных игр стремительно развивается, притягивая внимание многих людей. Разработка компьютерных игр является одной из самых ведущих сфер в компьютерной индустрии. Среди них особое место занимают 2D-игры, которые, несмотря на развитие технологий и появление трехмерной графики, продолжают привлекать интерес игроков благодаря своей простоте, стилю и уникальному игровому процессу. Одна из самых популярных платформ для разработки игр является кроссплатформенная среда Unity, предоставляющая разработчика мощные инструменты и возможности создания как 2D, так и 3D проектов.

Целью данной курсовой работы является исследование особенностей разработки 2D-игры с использованием платформы Unity и создание интуитивно понятной игры. В процессе разработки игры будет уделено внимание основным принципам проектирования 2D-игр, таким как создание графики, анимации, механики игры и взаимодействия с пользователем. В дальнейшем в работе будут представлены подробности о процессе разработки.

В данной курсовой работе были разработан проект, включающий такие игры как «Платформер», «Кетчер», «Шутер» и «Викторина», являющиеся одними из наиболее популярных жанров игр, которые привлекают своей динамичностью, захватывающим и интеллектуально-развлекательным процессом.

1 Анализ предметной области

Предметная область данной курсовой работы включает в себя разработку целого проекта с компьютерными 2D играми на Unity, включающих в себя различные жанры, такие как Платформер, Кетчер Шутер и Викторина.

1.1 Жанр игры. Аналоги игр в этом жанре

Каждый из жанров игр предлагает уникальный игровой опыт, основанный на различных механиках и целях. Ниже приведены особенности жанров платформера и кетчера, а также приведу примеры популярных аналогов, которые иллюстрируют их суть и привлекательность для игроков.

Суть разработки игры Платформер заключается в создании захватывающего игрового процесса, основанного на перемещении персонажа между различными платформами. Игроку предстоит преодолевать препятствия, избегать ловушек и врагов, используя прыжки. Цель игры – уничтожение всех противников.

«Super Mario Bros» – классическая игра, в которой игрок управляет Mario, преодолевая препятствия и сражаясь с врагами, чтобы спасти принцессу. Сцена игра приведена на рисунке 1.



Рисунок 1 – Сцена игры «Super Mario Bros»

«Celeste» – интенсивный платформер, где игрок управляет девушкой по имени Мадлен, стремящейся взобраться на гору, сталкиваясь с различными препятствиями и внутренними демонами. Сцена игра приведена на рисунке 2.



Рисунок 2 – Сцена игры «Celeste»

«Hollow Knight» – атмосферный платформер, в котором игрок исследует мрачный мир, сражаясь с врагами и открывая новые возможности. Сцена игра приведена на рисунке 3.



Рисунок 3 – Сцена игры «Hollow Knight»

В Кетчере акцент сделан на ловле падающих объектов. Игрок собирает правильные предметы, избегая нежелательных. Успешная ловля приносит очки, а неудача может привести к потере жизней.

«Catch the Candy» – игрок решает головоломки, чтобы помочь персонажу поймать сладости, используя различные механики взаимодействия. Сцена игра приведена на рисунке 4.



Рисунок 4 – Сцена игры «Catch the Candy»

«Fruit Ninja» – аркадная игра, в которой игрок разрезает фрукты, избегая бомб, чтобы набрать максимальное количество очков. Сцена игра приведена на рисунке 5.

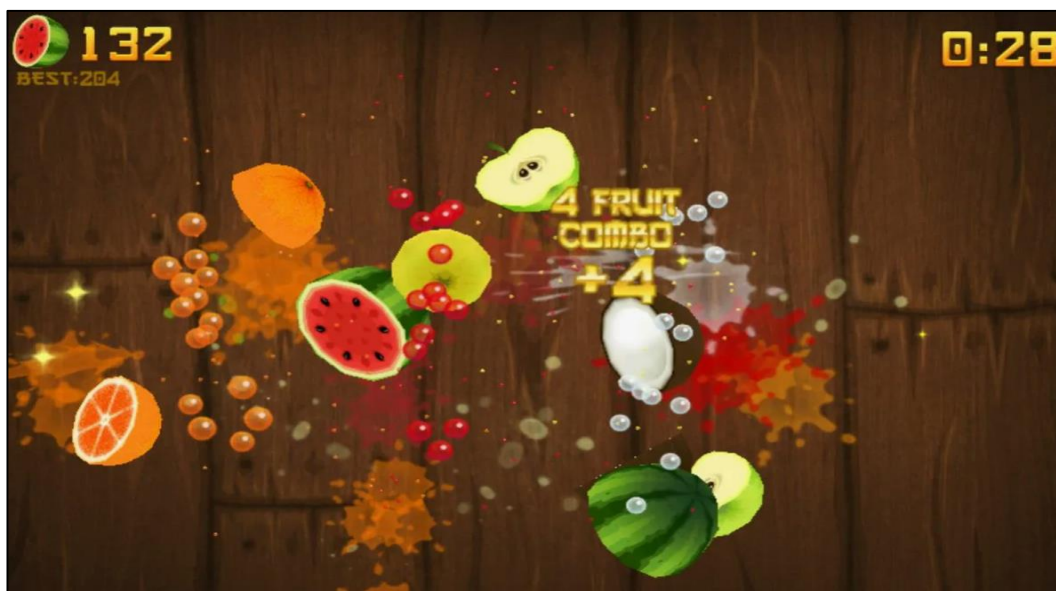


Рисунок 5 – Сцена игры «Fruit Ninja»

«Doodle Jump» – игрок управляет прыгающим персонажем, который должен подниматься вверх по платформам, избегая врагов и собирая бонусы. Сцена игра приведена на рисунке 6.

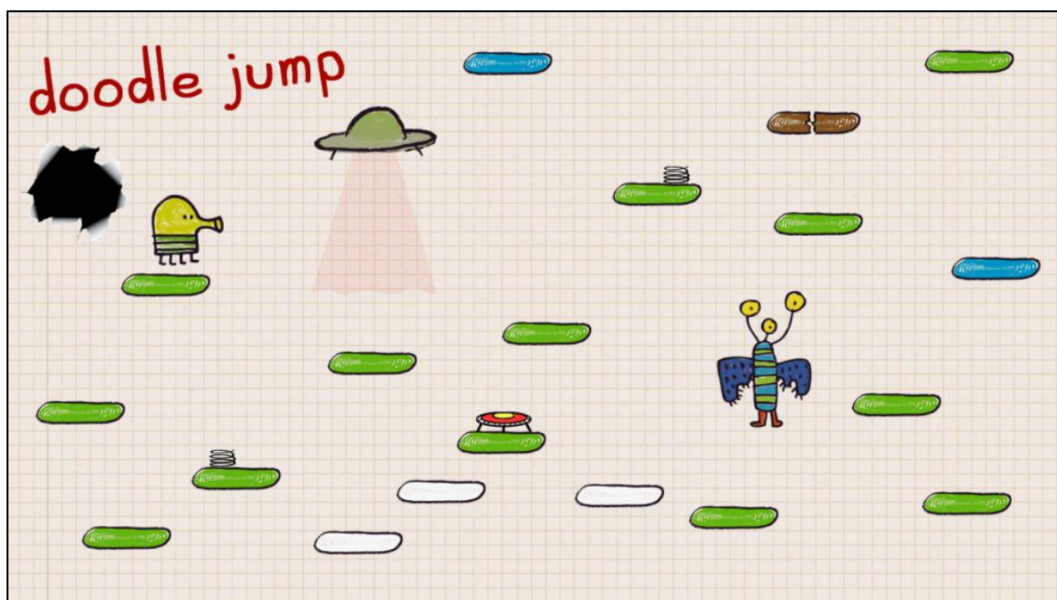


Рисунок 6 – Сцена игры «Doodle Jump»

Игра Шутер опирается на создание казуального игрового процесса, связанного со стрельбой лазером по врагам. Игрок перемещается по плоскости, уничтожая как добрых персонажей, которые не могут атаковать, так и злых, активно стреляющих в него. Это добавляет элемент напряжения и динамики, а также влияет на начисление или потерю очков.

«Contra» – классический кооперативный шутер, где игроки сражаются с инопланетными врагами, используя разнообразное оружие и тактики. Сцена игра приведена на рисунке 7.



Рисунок 7 – Сцена игры «Contra»

«Gunstar Heroes» – яркий и динамичный шутер, в котором игроки выбирают персонажей и сражаются с врагами, используя мощные комбо-атаки и уникальные способности. Сцена игра приведена на рисунке 8.



Рисунок 8 – Сцена игры «Gunstar Heroes»

«Enter the Gungeon» – рогалик-шутер, в котором игроки исследуют подземелья, полные врагов и уникального оружия, стремясь найти легендарное оружие, способное уничтожить прошлое. Сцена игра приведена на рисунке 9.

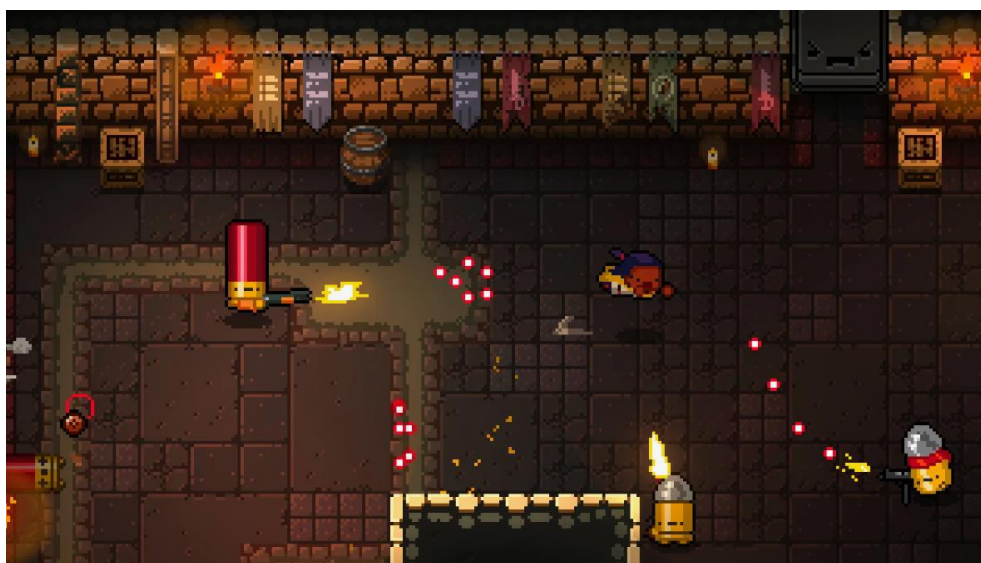


Рисунок 9 – Сцена игры «Enter the Gungeon»

Викторина – интеллектуальное состязание, основанное на проверке знаний в различных областях. Игроку предлагается ряд вопросов, на которые

необходимо дать правильный ответ, выбрав один из предложенных вариантов. Успешные ответы приносят очки, что создает соревновательную атмосферу, требующую от игрока не только знаний, но и умения быстро мыслить и анализировать. Похожими играми такого жанра являются:

«Интеллектуальный Марафон» – игра, где игрок участвует в серии тематических викторин, перемещаясь по карте знаний, где каждая область представляет собой отдельный уровень с возрастающей сложностью, а успешное прохождение открывает доступ к новым областям и позволяет получить уникальные значки за достижения.

«Вершина Знаний» – игра с уникальным форматом, где игроку предлагаются ответы-утверждения, а задача – сформулировать вопрос, который к ним подходит, где он перемещается по вертикальной башне знаний, каждый этаж которой представляет собой отдельную тематику, а чем выше поднимается игрок, тем сложнее становятся утверждения и тем выше награда, при этом неправильный вопрос может привести к падению на предыдущие уровни, добавляя элемент риска.

«Колесо Знаний» – игра, где игрок крутит виртуальное колесо, чтобы выбрать случайную категорию вопросов, в которой игрокам предстоит соревноваться в режиме онлайн в пошаговых дуэлях, отвечая на вопросы быстрее и точнее соперника, а победитель получает очки и может украсть знания (в виде виртуальных фишек) у соперника, при этом также есть возможность обменивать фишки на бустеры, которые могут облегчить игру.

1.2 Описание среды разработки

Для создания разработки трех жанр игр, приведенных выше, будет использоваться среда Unity, которая является мощной и гибкой игровой платформой, предоставляющей широкие возможности для разработки игр различных жанров и платформ. Она поддерживает C# в качестве основного языка программирования и предлагает широкий набор готовых инструментов и библиотек, что позволяет ускорить процесс разработки и облегчить создание игровых механик.

C# относится к языкам с C-подобным синтаксисом, из которых его синтаксис скорее близок к C++ и Java. Унаследовав многое от своих предшественников – языков C++, Java и др. – C#, опирается на практическое их использования и исключает определившие себя как проблематичные при разработке программного продукта. Например, C# в отличие от C++ не поддерживает множественное наследование классов.

Переняв многое от своих предшественников – языков C++, Java, Delphi, Модула и Smalltalk – C#, опираясь на практику их использования, исключает некоторые модели, зарекомендовавшие себя как проблематичные при разработке программных систем, например, C# в отличие от C++ не поддерживает множественное наследование классов (между тем допускается множественное наследование интерфейсов).

Данный язык был выбран в качестве основного, так как обладает нужными качествами для реализации, имеет встроенную обобщений, делегатов и событий, что облегчит реализацию.

1.3 Техническое задание на разработку

В данной курсовой работе необходимо разработать четыре игры: платформер, кетчер, шутер и викторина. Каждая игра должна иметь базовую игровую механику, включая управление персонажем и взаимодействие с объектами, соответствующую жанру. Управление должно быть интуитивно понятным и отзывчивым. Графическое оформление является простым и должно соответствовать выбранному стилю. Использование анимации персонажей в каждой из трех игр обязательно. Внедрение системы очков для увеличения вовлеченности игроков. Необходимо реализовать звуковое сопровождение: фоновая музыка и звуковые эффекты.

Ниже представлены состояния для каждой мини-игры. Диаграммы состояний показывают, как система реагирует на различные события и переходы между состояниями. Эти диаграммы позволяют понять, как игроки взаимодействуют с игрой, какие состояния они могут принимать и какие действия приводят к изменениям в игровом процессе.

Требованиями к Платформеру являются:

- создание платформ;
- передвижение игрока влево, вправо и прыжок с помощью клавиатуры;
- создание дружелюбных противников (червяков), на которые можно наносить удар;
- создание враждебных противников (бревна и монстры), которые наносят удар игроку, а один из бревен должен при соприкосновении перезагружать игру;
- создание анимации всех персонажей;
- создание здоровья у игрока и противников.

К требованиям Кетчера относятся:

- создание плоской платформы;
- передвижение игрока вправо и влево с помощью клавиатуры;
- генерация хороших и плохих предметов и их падение;
- ловля хороших предметов, которые прибавляют очки;
- ловля хороших предметов, которые прибавляют здоровье;
- ловля плохих предметов, которые убавляют здоровье;
- создание анимации игрока;
- создание фоновой музыки и звуковых эффекты;
- создание панели настроек.

Шутер имеет следующие требования:

- создание плоской платформы;
- передвижение персонажа вправо и влево с помощью курсора мыши;
- создание индикатора здоровья у всех персонажей;
- стрельба лучом в противников;
- генерация дружелюбных противников, при уничтожении лучом которых прибавляется 5 очков;
- генерация враждебных противников при уничтожении лучом которых прибавляется 10 очков;

- выигрыш при определенном наборе очков;
- проигрыш при отсутствии всех жизней игрока.

Обязательными условиями Викторины являются:

- содержание списка вопросов, каждый из которых включает текст вопроса, изображение, список ответов и индекс правильного ответа;
- вопросы должны быть случайным образом перемешаны перед началом викторины;
- текстовый элемент для отображения текущего вопроса;
- изображение, связанное с вопросом (если имеется);
- кнопки для выбора ответов, количество которых должно соответствовать количеству ответов на вопрос;
- текст для отображения текущего счета игрока;
- текст для отображения результата викторины после её завершения;
- кнопка в конце для выхода из игры;
- отслеживание количества правильных ответов игрока;
- после выбора ответа игроком, должен отображаться следующий вопрос;
- в конце викторины должна отображаться информация о количестве правильных ответов и возможность начать игру заново.

Ниже представлена диаграмма прецедентов для całego проекта. Диаграммы прецедентов представляют собой высокоуровневый обзор функциональности системы с точки зрения пользователей (актеров). Они помогают определить, какие действия могут выполнять пользователи и какие цели они могут достигать в рамках каждой игры. В контексте платформера, кетчера, шутера и викторины, диаграммы прецедентов помогут выделить ключевые функции.

Диаграмма прецедентов представлена на рисунке 10. Каждый прецедент отражает конкретную задачу или взаимодействие, которое пользователь может осуществить в игре.

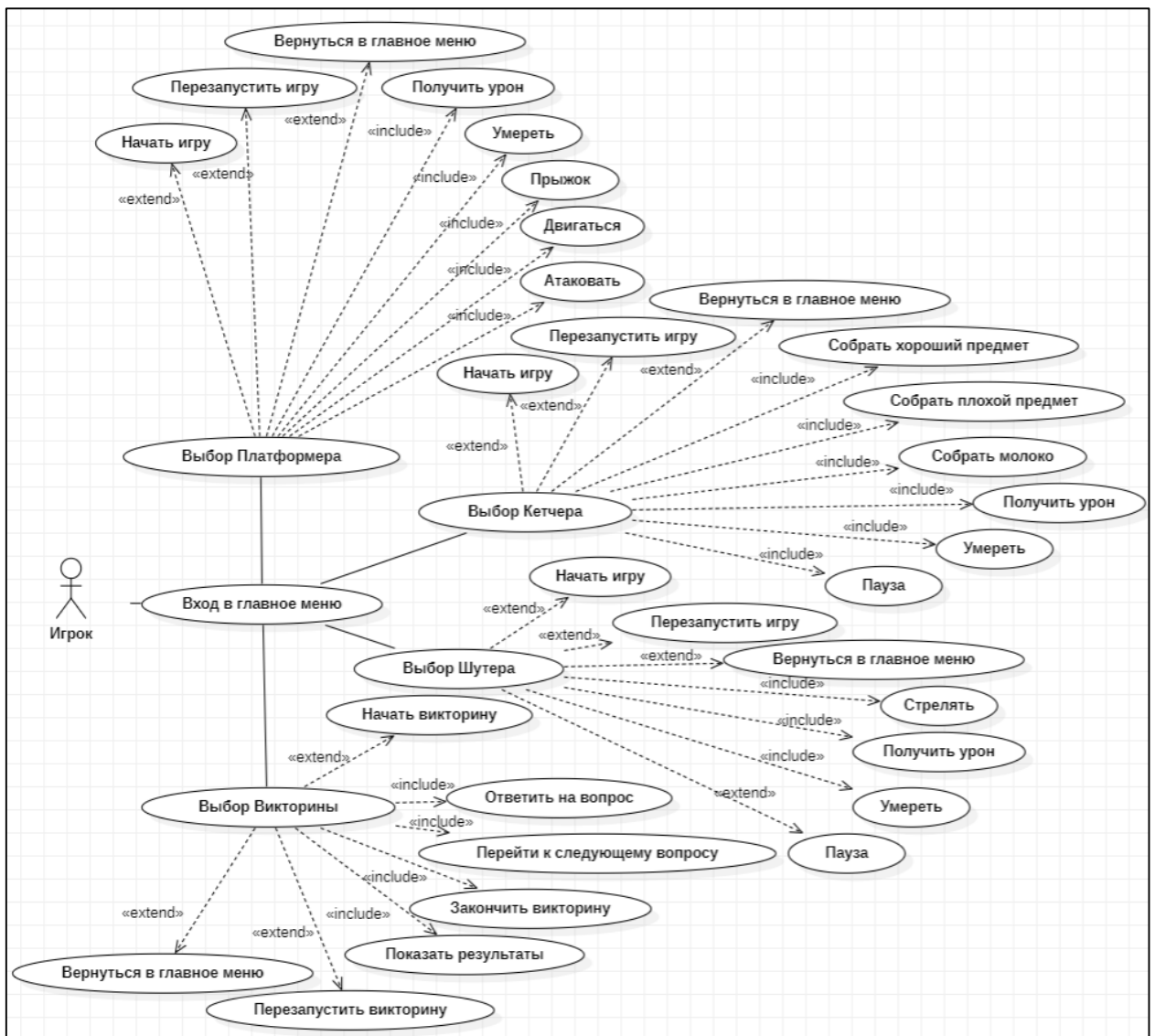


Рисунок 10 – Диаграмма прецедентов для Платформера

Диаграммы прецедентов также помогают разработчикам понять, какие механики должны быть реализованы для обеспечения положительного игрового опыта. Они служат основой для создания более детализированных технических спецификаций. Кроме того, такие диаграммы облегчают коммуникацию между членами команды, позволяя всем участникам проекта иметь общее представление о функциональности системы. В конечном итоге, четкое понимание прецедентов способствует более эффективному планированию и разработке игр.

2 Проектирования и разработка приложения

Процесс разработки включает в себя проектирование уровней и игровой логики, создание скриптов для управления игровыми механиками и персонажами, а также использование различных ресурсов и анимации для обеспечения привлекательного и динамичного игрового процесса.

2.1 Проектирование уровней и логики игры

В рамках курсовой работы был разработан игровой сборник, включающий четыре мини-игры в жанрах Платформер, Кетчер, Шутер и Викторина, реализованных на игровом движке Unity. Приложение представляет собой 2D-игру, где пользователь может выбрать одну из предложенных мини-игр через главное меню.

Каждая мини-игра реализована в виде отдельной сцены в Unity. Уровни в каждой из мини-игр представляют собой замкнутые игровые пространства без перехода на другие локации, то есть в рамках одной сцены. Для игры были разработаны уникальные наборы препятствий/объектов и механики. Главная сцена основного меню содержит три кнопки, каждая из которых ведет к отдельной мини-игре.

Уровень (сцена) Платформера представляет собой линейную сцену с препятствиями (платформы, пропасти, враги). Логика игры состоит в том, что игрок перемещается по уровню, избегая препятствий и врагов. Предусмотрено управление игроком с помощью кнопок.

У Кетчера сцена состоит из области для ловли падающих объектов. Логика игры включает в себя игрока, который ловит падающие сверху объекты, перемещаясь по плоской платформе. Предусмотрено управление игроком с помощью кнопок.

Уровень Шутера включает область, где размещен персонаж, который перемещается по плоской платформе и стреляет в противников. Логика игры складывается в том, что игрок стреляет по противникам, набирая очки. Управление стрельбой осуществляется с помощью правой кнопки мыши. Предусмотрено управление игроком с помощью курсора мыши.

У Викторины сцена имеет интерфейс, в котором расположены текст с вопросом, текст с результатом правильных ответов, кнопки с текстом вопроса. Логика игры состоит в том, что игрок проходит тест, отвечая на вопросы, а в конце выводится результат тестирования. Управление интерфейсом осуществляется с помощью мыши.

Ниже представлена диаграмма состояний для całego проекта. Диаграмма состояний показывают, как система реагирует на различные события и переходы между состояниями. Эти диаграммы позволяют понять, как игроки взаимодействуют с игрой, какие состояния они могут принимать и какие действия приводят к изменениям в игровом процессе.

В заключение, разработка данного игрового сборника позволила не только углубить знания в области программирования на Unity, но и развить навыки проектирования игровых механик. Каждая мини-игра была тщательно протестирована на удобство управления и интересность игрового процесса. Обратная связь от пользователей выявила сильные и слабые стороны каждой игры, что поможет в дальнейшем улучшении проекта. В перспективе планируется добавить новые мини-игры и расширить функционал существующих, включая мультиплеерный режим и дополнительные уровни сложности. Таким образом, проект имеет потенциал для дальнейшего развития и может стать основой для более масштабного игрового приложения.

Кроме того, интеграция системы достижений может значительно повысить интерес игроков и мотивацию к повторным играм. Мы также рассматриваем возможность внедрения обучающих уровней, которые помогут новым игрокам быстрее освоить механики каждой мини-игры.

Диаграмма состояний представлена на рисунке 11.

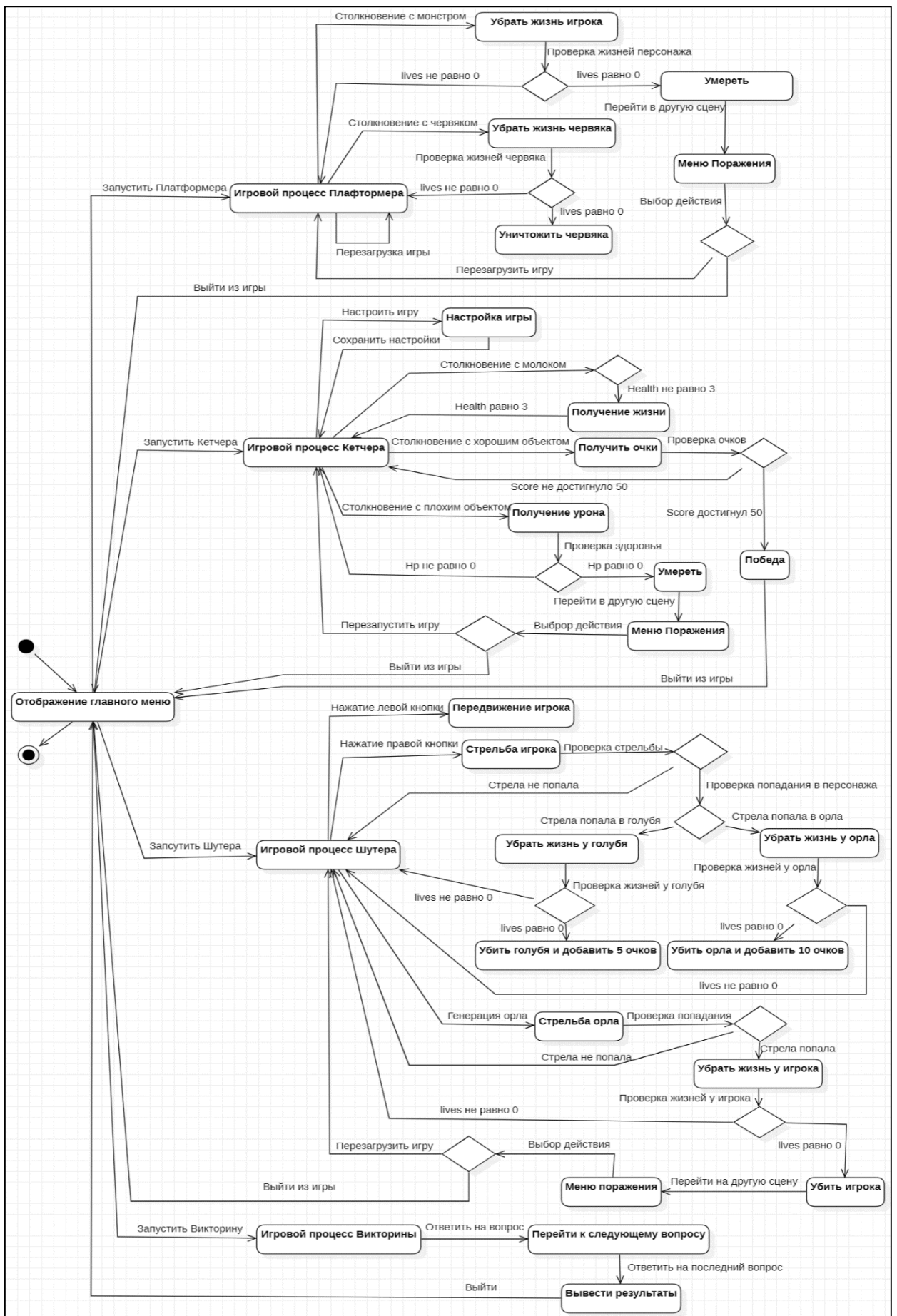


Рисунок 11 – Диаграмма состояний

2.2 Описание основных классов и скриптов

Классы и скрипты являются ключевыми компонентами для реализации игровой логики и механики. Проект состоит из четырех отдельных мини-игр: Платформера, Кетчера, Шутера и Викторины. Каждая из них имеет свой набор скриптов, отвечающих за управление персонажем, взаимодействие с объектами и логику игрового процесса. Все скрипты приведены в Приложении.

Ниже будут представлены диаграммы классов для каждой мини игры проекта. Диаграммы классов предоставляют подробный обзор структуры системы, включая классы, их атрибуты и методы, а также отношения между ними. Эти диаграммы важны для понимания архитектуры игры и могут быть использованы для проектирования системы, обеспечивая ясность в отношении того, как различные компоненты взаимодействуют друг с другом. В контексте платформера, кетчера, шутера и викторины, диаграммы прецедентов помогут выделить ключевые функции.

Скриптом, с которого нужно начать, является `ButtonScript.cs`, содержащий 5 публичных функций, которые можно привязать к кнопкам в интерфейсе Unity. Каждая функция выполняет загрузку определенной сцены.

Ниже рассматриваются ключевые скрипты для каждой мини-игры, объясняя их функциональность, принципы работы и взаимодействие с другими компонентами.

Ниже приведено описание скриптов и классов Платформера.

`CanvasButton.cs` управляет кнопками на игровом поле, позволяя перезапустить игру, выйти в главное меню или открыть указанную URL-адрес. Скрипт содержит такие методы, как:

- `RestartGame()` – перезагружает сцену;
- `Exit()` – возвращает на главную сцену;
- `URL()` – открывает URL в браузере.

`Entity.cs` определяет базовый класс `Entity`, который включает виртуальные методы для получения урона и смерти объекта, позволяя

наследующим классам переопределять их функциональность. Скрипт содержит такие методы, как:

- `GetDamage()` – виртуальный метод, который можно переопределить в наследуемых классах для обработки получения урона.

- `Die()` – уничтожает объект.

`Hero2.cs` управляет поведением игрового персонажа, включая движение, прыжки, состояние и получение урона, а также переключает анимации в зависимости от действий игрока и состояния персонажа. Скрипт содержит такие методы, как:

- `Awake()` – инициализация компонентов (`Rigidbody`, `Animator`, `SpriteRenderer`).

- `Run()` – обрабатывает движение героя влево и вправо.

- `Jump()` – обрабатывает прыжок героя.

- `CheckGround()` – проверяет, находится ли герой на земле.

- `GetDamage()` – уменьшает количество жизней и проверяет, не закончились ли они.

`RandomThree.cs` случайным образом создает один из объектов из заданного массива при старте игры, размещая его в позиции текущего игрового объекта. Скрипт содержит метод, как `Start()`, который генерирует случайный индекс и создает объект из массива `objects`.

`WalkingMonster.cs` и `WalkingMonster2.cs` управляет поведением монстра, который движется к цели (игроку) и наносит урон, если сталкивается с ним, при этом отслеживая свои жизни и уничтожая себя, когда жизни становятся меньше 1. Скрипт содержит такие методы, как:

- `Move()` – перемещает монстра в сторону героя.

- `OnCollisionEnter2D(Collision2D collision)` – всего лишь обрабатывает столкновения с героем и уменьшает его жизни.

`Wood2.cs` загружает сцену с индексом 2 при столкновении с другим объектом. Скрипт содержит метод `OnCollisionEnter2D(Collision2D collision)`, который загружает сцену, когда герой сталкивается с объектом.

Wood2.cs и Worm.cs управляет червем, который теряет жизни при столкновении с игроком и уничтожает себя, когда количество жизней достигает нуля, при этом выводя информацию о текущем количестве жизней в консоль. Скрипт содержит метод OnCollisionEnter2D(Collision2D collision), который обрабатывает столкновения с героем и уменьшает его жизни.

Диаграмма классов для Платформера представлена на рисунке 12.

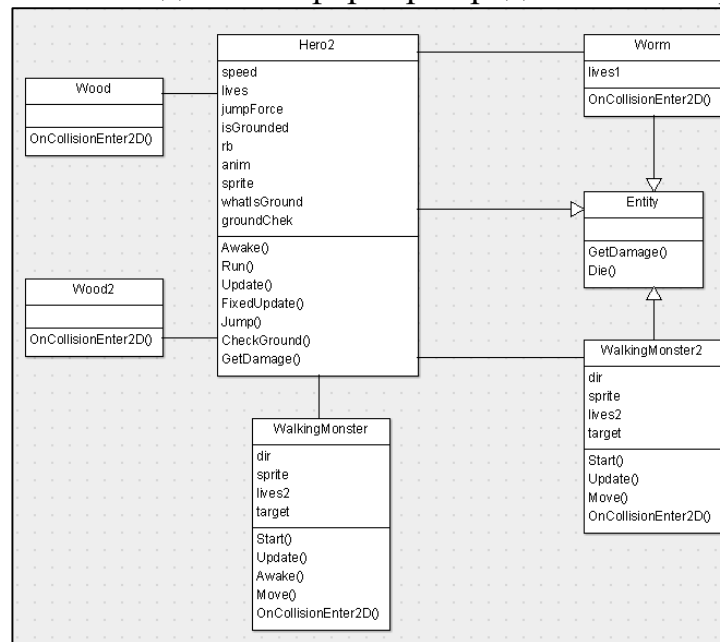


Рисунок 12 – Диаграмма классов для Платформера

Ниже приведено описание скриптов и классов Кетчера.

CanvasButton1.cs управляет кнопками на канвасе, позволяя перезапустить игру, загружая сцену с индексом 3, или выйти в главное меню, загружая сцену с индексом 0. Скрипт содержит такие методы, как:

- RestartGame() – перезагружает текущую сцену с кетчером.
- Exit() – загружает главную сцену (меню).

itemChecker.cs управляет сбором игровых объектов, увеличивая или уменьшая счет игрока и здоровье в зависимости от типа собранного предмета (хороший, плохой или молоко), а также отвечает за переход между сценами при достижении определенных условий. Скрипт содержит такие методы, как:

- Start() – инициализирует здоровье и отображает изображения здоровья.

- OnTriggerEnter2D(Collider2D other) – обрабатывает столкновения с предметами (хорошими, плохими и молоком) и обновляет счет и здоровье.

itemSpawn.cs отвечает за периодическое создание игровых объектов в заданной области, используя метод InvokeRepeating для вызова функции Spawn, которая генерирует случайные позиции и выбирает случайный предмет из массива items. Скрипт содержит такие методы, как:

- Start() – запускает процесс спавна предметов с заданной задержкой и периодичностью.

- Spawn() – генерирует случайный предмет в заданной области.

PauseMenu.cs управляет паузой в игре, позволяя игроку приостанавливать и возобновлять игру с помощью клавиши Escape, а также предоставляет функции для загрузки главного меню и настроек. Скрипт содержит такие методы, как:

- Update() – проверяет нажатие клавиши Escape для переключения состояния паузы.

- Rename() – снимает паузу и скрывает меню.

- Pause() – приостанавливает игру и отображает меню.

- LoadMenu() – загружает главное меню.

- Settings() – загружает сцену настроек.

Player.cs управляет поведением игрока в 2D-игре, включая движение, прыжки и анимации. Скрипт содержит такие методы, как:

- Awake() – инициализация компонентов (Rigidbody, Animator, SpriteRenderer).

- Run() – обрабатывает движение игрока.

- Jump() – обрабатывает прыжок игрока.

- CheckGround() – проверяет, находится ли игрок на земле.

- Update() – обрабатывает ввод от игрока и обновляет состояние анимации.

VolumeValue.cs предназначен для автоматического уничтожения игрового объекта через заданный промежуток времени. Скрипт содержит метод Start(), который уничтожает объект через заданный интервал времени.

Диаграмма прецедентов для Кетчера представлена на рисунке 13.

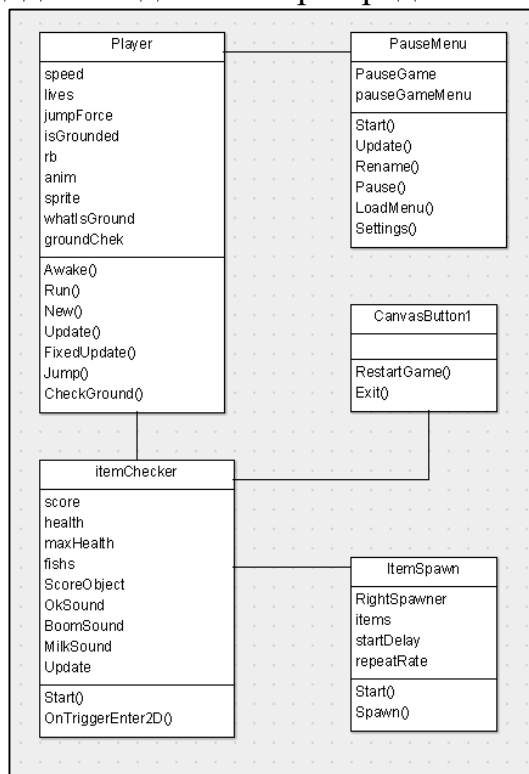


Рисунок 13 – Диаграмма классов для Кетчера

Ниже приведены описание скриптов и классов Шутера.

BirdHP.cs управляет здоровьем птицы в игре, отображая его на панели здоровья и увеличивая счет игрока при уничтожении птицы. Скрипт содержит такие методы, как:

- Start() – инициализирует здоровье и находит ScoreManager.
- MakeDamage(int damage) – уменьшает здоровье, обновляет полосу здоровья и увеличивает счет при уничтожении птицы.

BirdMove.cs управляет движением птицы вниз по экрану с заданной скоростью, уничтожает ее при столкновении с объектами определенных тегов и перезагружает сцену при столкновении с игроком. Скрипт содержит такие методы, как:

- Start() – инициализирует Rigidbody2D и задает начальное движение ВНИЗ.

- `OnBecameInvisible()` – уничтожает объект, если он выходит за границы видимости.

- `OnCollisionEnter2D(Collision2D something)` – обрабатывает столкновения с игроком и другими объектами.

`CanvasButtonShuter.cs` управляет действиями кнопок на канвасе, позволяя перезагрузить игру, выйти на главный экран или выполнить другую функцию, связанную с загрузкой сцены. Скрипт содержит такие методы, как:

- `RestartGame()` – перезагружает сцену шутера.
- `Exit()` – загружает главную сцену (меню).
- `Perfect2()` – загружает главную сцену (меню).

`EnemyBullet.cs` управляет поведением пули, выпущенной врагом в направлении игрока. Скрипт содержит такие методы, как:

- `Start()` – находит игрока и задает направление полета пули.
- `OnTriggerEnter2D(Collider2D other)` – обрабатывает столкновения с игроком, нанося ему урон.
- `OnBecameInvisible()` – уничтожает пулю, если она выходит за границы видимости.

`EnemyHP.cs` управляет здоровьем врага и отображает его состояние через индикатор здоровья. Скрипт содержит такие методы, как:

- `Start()` – инициализирует здоровье и находит `ScoreManager`.
- `MakeDamage(int damage)` – уменьшает здоровье, обновляет полосу здоровья и увеличивает счет при уничтожении врага.

`EnemyShoot.cs` управляет стрельбой врага, позволяя ему периодически стрелять с заданной задержкой в сторону игрока, находя его по тегу "PlayerCat". Скрипт содержит такие методы, как:

- `Start()` – находит игрока.
- `Update()` – проверяет возможность стрельбы и создает пулю.
- `firePause()` – корутин, который управляет задержкой между выстрелами.

HealthBar.cs обеспечивает постоянное выравнивание индикатора здоровья по горизонту, устанавливая его вращение в ноль в каждом кадре. Скрипт содержит метод Update(), который обнуляет вращение полосы здоровья.

HpController.cs управляет здоровьем объекта, позволяя ему получать урон и уничтожаться, когда его здоровье достигает нуля. Скрипт содержит методы MakeDamage(int damage), который уменьшает здоровье и уничтожает объект, если здоровье равно нулю.

LaserShot.cs управляет поведением лазерного снаряда в игре. Скрипт содержит такие методы, как:

- Start() – задает направление и скорость лазера.
- OnBecameInvisible() – уничтожает лазер, если он выходит за границы видимости.
- OnTriggerEnter2D(Collider2D other) – наносит урон врагу при столкновении.

ObjectSpawnerBird.cs отвечает за периодическое создание объектов в игре. Скрипт содержит такие методы, как:

- Start() – запускает процесс спавна с заданным интервалом.
- Spawn() – генерирует случайную птицу в заданной области.

PlayerCatHP.cs управляет здоровьем игрока, отображая его текущее состояние на панели здоровья и загружая новую сцену, если здоровье падает до нуля. Скрипт содержит такие методы, как:

- Start() – инициализирует здоровье и полосу здоровья.
- MakeDamage(int damage) – уменьшает здоровье и перезагружает сцену при смерти.

PlayerCatSpawner.cs отвечает за создание игрового объекта (предмета) в случайной позиции между текущей позицией спавнера и правой границей, определенной PlayerRightPositionSpawner, при инициализации сцены. Скрипт содержит методы, как Start(), который спавнит игрока в случайной позиции.

PlayerMoveCat.cs позволяет игроку управлять движением персонажа с помощью клика мыши, изменяя его позицию и наклон в зависимости от направления движения. Скрипт содержит такие методы, как:

- Start() – инициализирует Rigidbody2D и задает начальную позицию.
- Update() – обрабатывает ввод от мыши и наклон персонажа.
- FixedUpdate() – обновляет скорость движения игрока.

PlayerShoot.cs позволяет игроку стрелять лазерами при нажатии правой кнопки мыши, ограничивая частоту выстрелов с помощью задержки, заданной в delayTime. Скрипт содержит такие методы, как:

- Update() – проверяет возможность стрельбы и создает лазер.
- NoFire() – корутин, который управляет задержкой между выстрелами.

ScoreManager.cs управляет игровым счетом, обновляет текст на экране и загружает следующую сцену, когда счет достигает 50. Скрипт содержит такие методы, как:

- Update() – обновляет текст счета и проверяет, достиг ли игрок нужного количества очков для перехода на следующую сцену.
- IncreaseScore(int amount) – увеличивает счет на заданное количество.
- LoadNextScene() – переходит на следующую сцену.

Диаграмма прецедентов для Шутера представлена на рисунке 14.

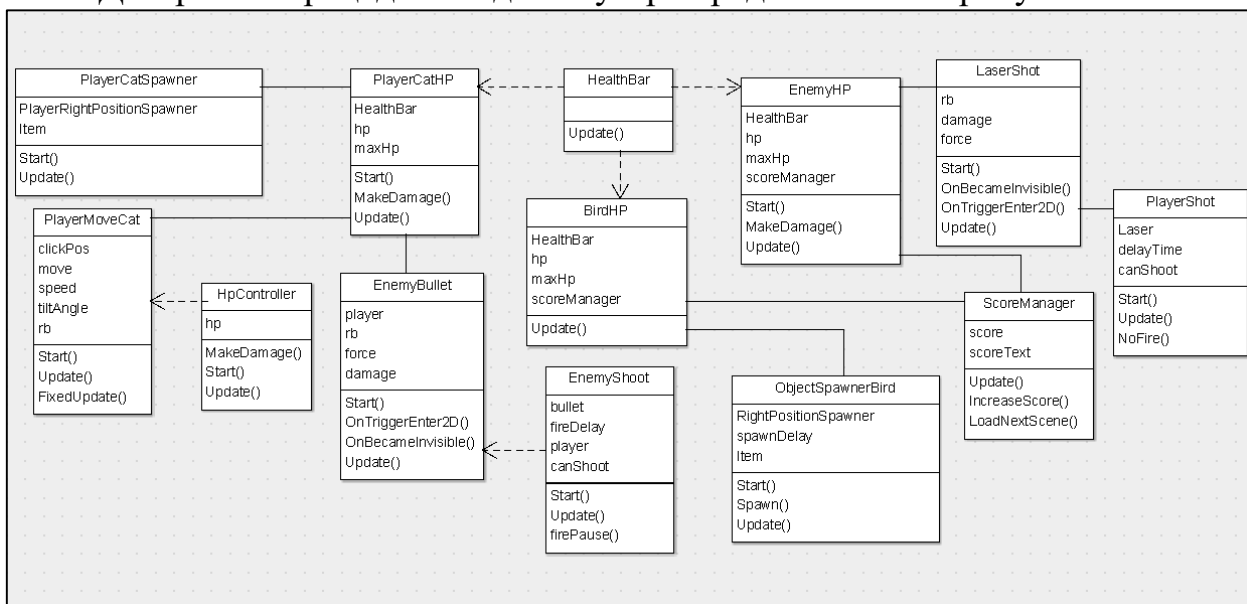


Рисунок 14 – Диаграмма классов для Шутера

Ниже приведено описание скрипта и классов Викторины.

Quiz.cs управляет викториной, отображая вопросы и варианты ответов, отслеживая правильные ответы и выводя результаты в конце, с возможностью перезапуска игры. Скрипт содержит такие методы, как:

- void Start() – инициализирует викторину, перемешивает вопросы, отображает первый вопрос и обновляет счет. Скрывает кнопку повторной попытки в начале.

- void ShuffleQuestions() – перемешивает список вопросов для случайного порядка их отображения.

- void DisplayQuestion() – отображает текущий вопрос и его возможные ответы.

- void AnswerQuestion(int index) – обрабатывает ответ пользователя. Если ответ правильный, увеличивает счетчик правильных ответов.

- void UpdateScore() – обновляет текст UI элемента, отображающего текущий счет.

- void EndQuiz() – завершает викторину, скрывает элементы UI для вопросов и отображает результат.

- void LoadNextScene() – загружает следующую сцену. В данном случае сцена с индексом 0 (можно заменить на нужное имя сцены).

Диаграмма прецедентов для Викторины представлена на рисунке 15.

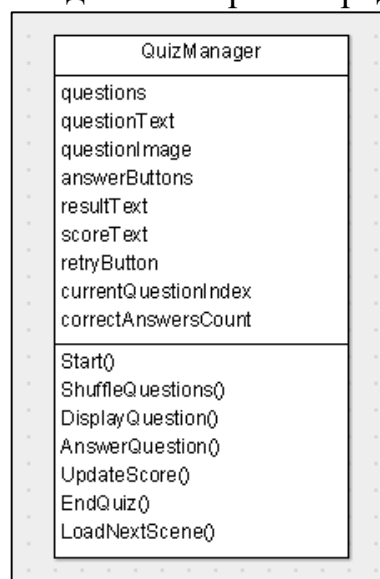


Рисунок 15 – Диаграмма классов для Викторины

2.3 Ресурсы проекта

В данном проекте используются различные ресурсы, которые обеспечивают функциональность и визуальную привлекательность мини-игр. Ниже перечислены основные категории ресурсов:

Графические ресурсы, среди которых выделяют спрайты (изображения персонажей, объектов, фонов и интерфейса), анимации (наборы кадров для анимации персонажей и объектов, которые создают плавные движения и взаимодействия в игре) и текстуры, которые используются для создания визуальных эффектов и детализации объектов, таких как стены, полы и другие элементы окружения.

Пример спрайтов приведен на рисунке 16.

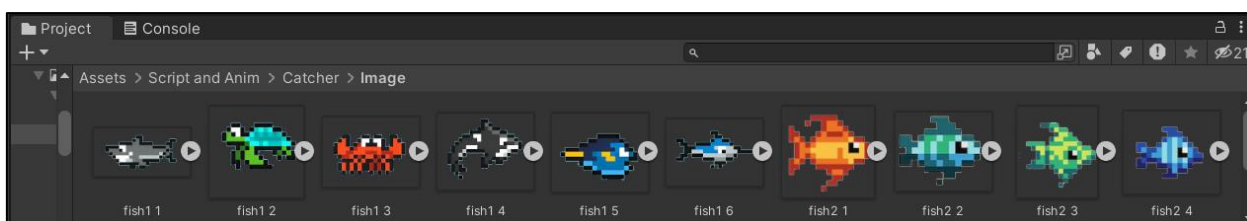


Рисунок 16 – Пример спрайтов

Пример анимации приведен на рисунке 17.

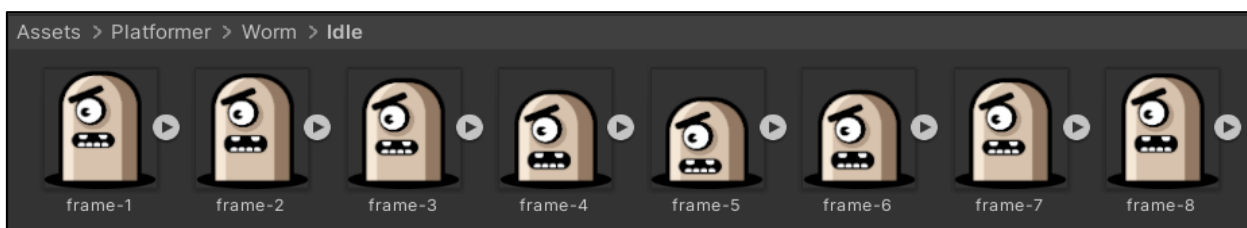


Рисунок 17 – Пример анимации

Звуковые ресурсы, в том числе музыка и звуковые эффекты. Под музыкой понимается фоновая музыка, создающая атмосферу для каждой мини-игры. Музыкальные треки могут варьироваться от энергичных до расслабляющих в зависимости от игрового процесса. Звуковые эффекты – звуки, связанные с действиями в игре, такими как прыжки, стрельба, сбор предметов и взаимодействие с объектами. Эти эффекты добавляют реалистичности и вовлеченности в игровой процесс.

Пример музыки и звуковых эффектов приведен на рисунке 18.



Рисунок 18 – Пример музыки и звуковых эффектов

Сценарные ресурсы, к числу которых относят скрипты и конфигурационные файлы. Проект состоит из основных скриптов и классов, отвечающие за логику игры, управление персонажами, взаимодействие с окружением и обработку событий. Конфигурационные файлы – файлы, содержащие настройки и параметры игры, такие как уровни сложности, количество жизней, время на выполнение задач и другие игровые параметры.

Пример скриптов приведен на рисунке 19.

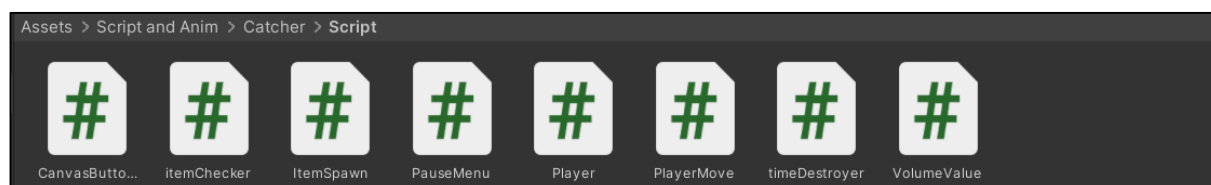


Рисунок 19 – Пример скриптов

Уровни и окружение, к числу которых относят сцены и элементы окружения. Сцены – переходы между сценариями. Сцены создаются с учетом игрового процесса и механик. Элементы окружения – объекты, с которыми игрок может взаимодействовать, такие как платформы, препятствия, враги и предметы для сбора.

Пример сцены приведен на рисунке 20.



Рисунок 20 – Пример сцены

Пример элементов окружения приведен на рисунке 21.

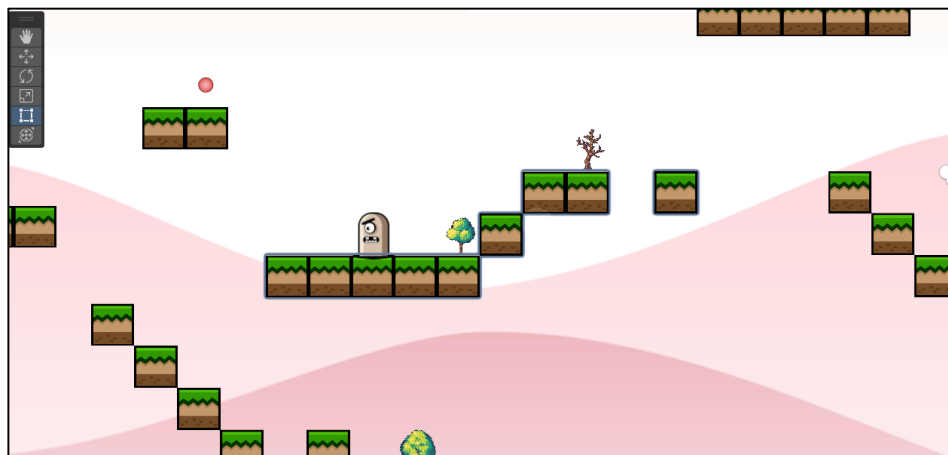


Рисунок 21 – Пример элементов окружения

Интерфейс пользователя включает UI элементы и меню. UI элементы – кнопки, текстовые поля, индикаторы здоровья и другие элементы интерфейса, которые помогают игрокам взаимодействовать с игрой и отслеживать свой прогресс. Меню – страницы меню для начала игры, выбора уровней, просмотра результатов и настроек.

Пример UI-элементов приведен на рисунке 22.



Рисунок 23 – Пример UI-элементов

Пример меню приведен на рисунке 24.

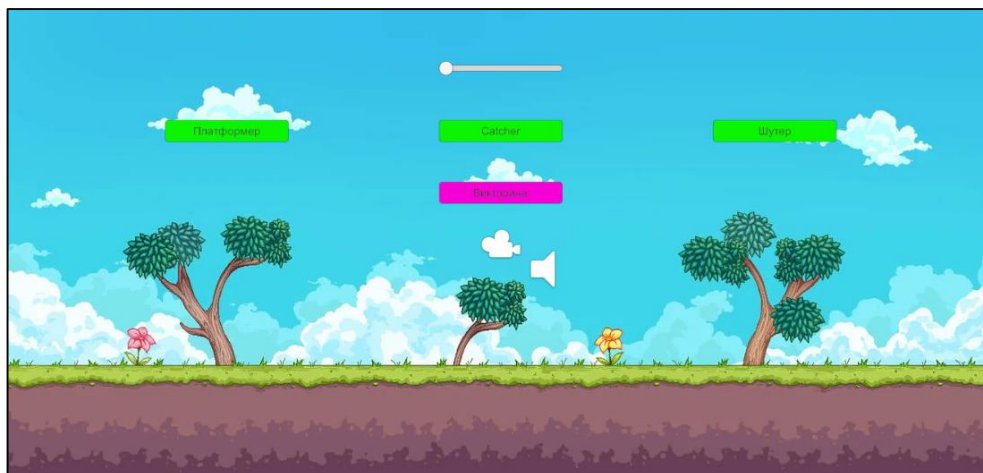


Рисунок 24 – Пример меню

2.4 Основные анимации игры

В каждой игре, кроме Викторины, заложена анимация персонажей, которая происходит по кнопке.

В Платформере анимацию имеют все персонажи: игрок, дружелюбные и враждебные противники. Игрок имеет 4 анимации: покоя, бега, прыжка и дополнительную анимацию по кнопке. Дружелюбные противники (червяки) имеют анимацию покоя, враждебные игроки имеют анимацию ходьбы вправо и влево.

На рисунке 25 приведены спрайты для анимации персонажа для трех мини-игр.



Рисунок 25 – Спрайты для анимации персонажа

На рисунке 26 настроены переходы между состояниями анимации персонажа.

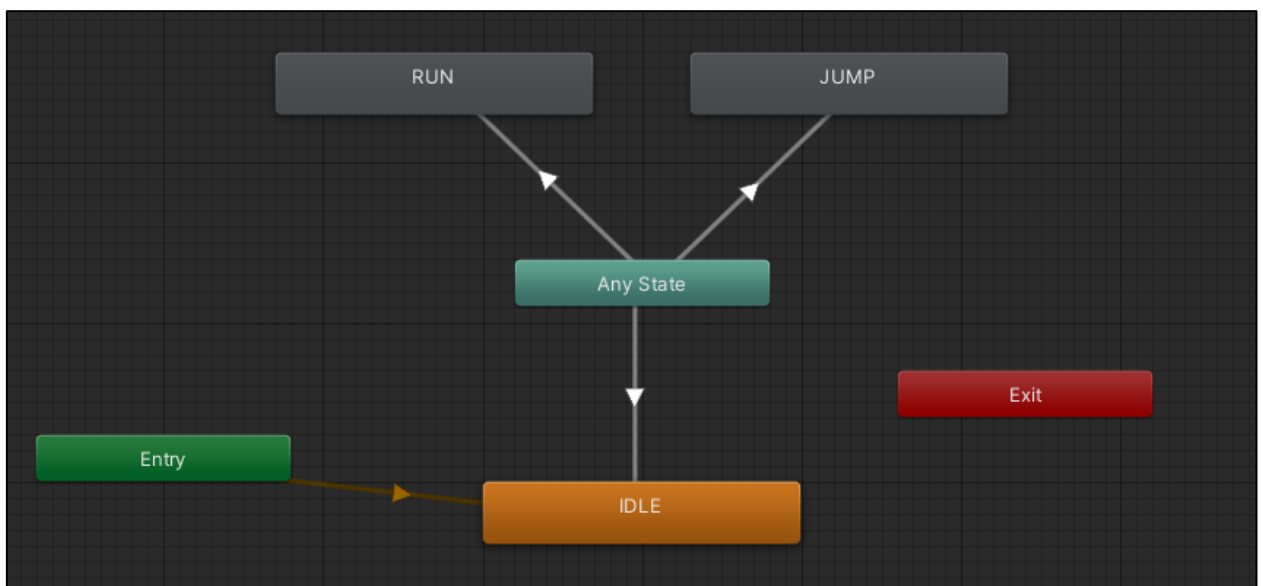


Рисунок 26 – Аниматор персонажа

Скрипт содержит логику для переключения состояний. Анимации правильно связаны с состояниями. В методе Update() есть логика для установки состояния.

```
void Update()
{
    if (isGrounded) State = States.IDLE;
    if (Input.GetButton("Horizontal"))
    {
        Run();
    }
    if (isGrounded && Input.GetKeyDown(KeyCode.Q))
    {
        New();
    }
    if (isGrounded && Input.GetButtonDown("Jump"))
    {
        Jump();
    }
}
```

На рисунке 27 приведены спрайты для анимации дружелюбного персонажа (червяка).

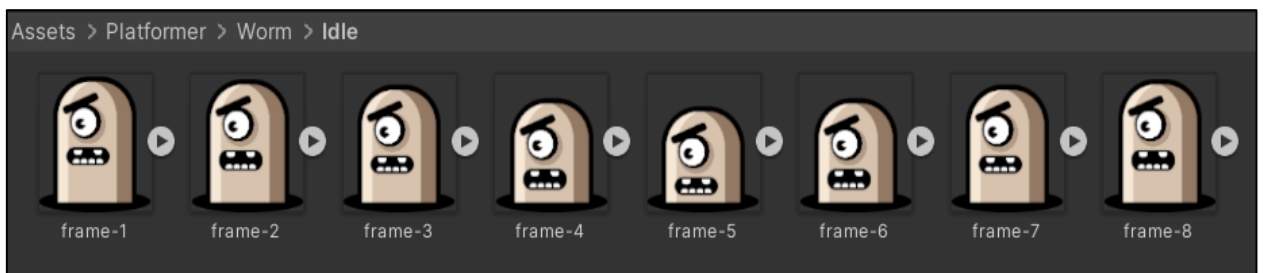


Рисунок 27 – Спрайты для анимации дружелюбного персонажа

На рисунке 28 настроены переходы между состояниями анимации дружелюбного персонажа.

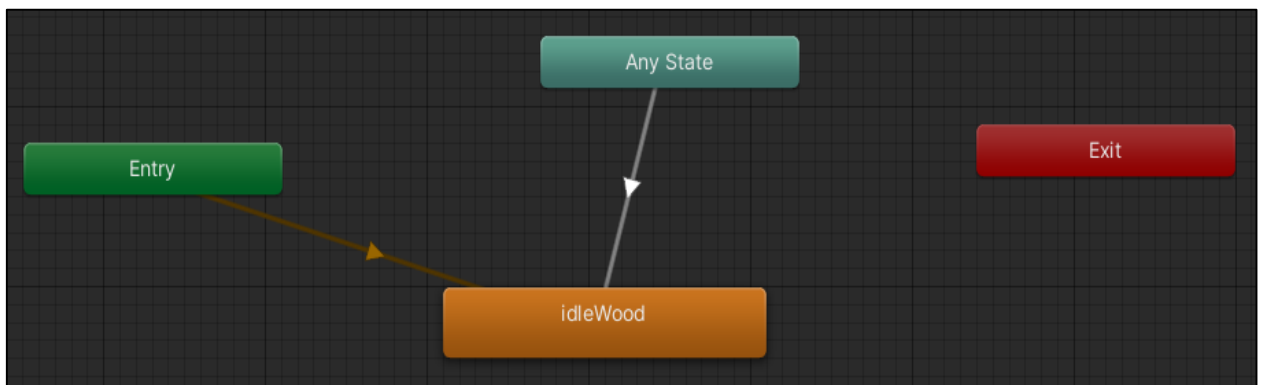


Рисунок 28 – Аниматор дружелюбного персонажа

На рисунке 29 приведены спрайты для анимации враждебного персонажа (монстра).

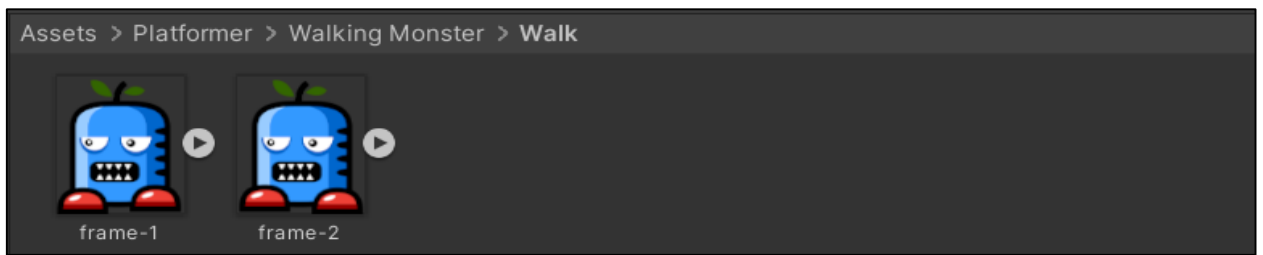


Рисунок 29 – Спрайты для анимации врага

На рисунке 30 настроены переходы между состояниями анимации враждебного персонажа.

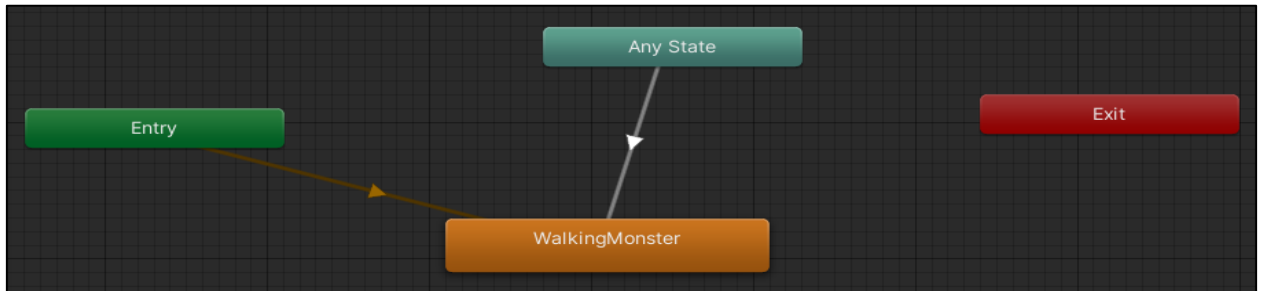


Рисунок 31 – Аниматор врага

В Кетчере анимацию имеет только игрок. Это анимация покоя, бега и прыжка. Скрипт похож на тот, который у персонажа в Платформере, за исключением дополнительной анимации.

В Шутере анимацию так же имеет только игрок. Это изменение уровня наклона игрока в зависимости от того, в какую сторону он переместился. Для этого не понадобилось создание аниматора. Анимация в данном скрипте реализована через наклон персонажа в зависимости от направления его движения. Когда персонаж движется вправо или влево, его угол наклона изменяется, что создает визуальный эффект движения. Если персонаж останавливается, он возвращается в вертикальное положение. Это добавляет динамичности и реалистичности движению персонажа. Ниже приведен скрипт анимации персонажа. Скрипт содержит логику для переключения состояний:

```
Vector3 clickPos;
Vector3 move;
public float speed = 1;
public float tiltAngle = 15f; // Угол наклона
Rigidbody2D rb;

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

```

        clickPos = transform.position;
    }

    void Update()
    {
        if (Input.GetMouseButton(0))
        {
            clickPos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
            clickPos.z = 0;
        }

        move = clickPos - transform.position;

        // Наклон персонажа в зависимости от направления движения
        if (move.x > 0.1f) // Движение вправо
        {
            transform.rotation = Quaternion.Euler(0, 0, -tiltAngle);
        }
        else if (move.x < -0.1f) // Движение влево
        {
            transform.rotation = Quaternion.Euler(0, 0, tiltAngle);
        }
        else // Если не движется, возвращаем в исходное положение
        {
            transform.rotation = Quaternion.Euler(0, 0, 0);
        }
    }

    void FixedUpdate()
    {
        rb.velocity = new Vector2(move.x, move.y) * speed;
    }

```

На рисунке 33 представлен наклон персонажа вправо.



Рисунок 33 – Анимация персонажа

На рисунке 34 представлен наклон персонажа влево.



Рисунок 34 – Анимация персонажа

3 Тестирование приложения

3.1 Описание тестовых сценариев

В процессе тестирования приложения были разработаны следующие сценарии для каждой из мини-игр, приведенный в таблице 1.

Таблица 1 – Тестирование проекта

Шаг	Ожидаемый результат
Платформер	
Проверка управления персонажем: движение влево и вправо, прыжок и дополнительная анимация.	Успешно
Проверка столкновений с врагами (WalkingMonster, Worm) и взаимодействия с игровыми объектами (Wood, Wood2).	Успешно
Проверка уменьшения жизней при получении урона и перехода на главный экран при их исчерпании.	Успешно
Кетчер	
Проверка правильности начисления очков при захвате объектов с тегом "Good".	Успешно
Проверка уменьшения очков при столкновении с объектами с тегом "Bad".	Успешно
Проверка увеличения здоровья при захвате объектов с тегом "Milk".	Успешно
Проверка отображения текущего состояния здоровья через интерфейс (изображения рыбок).	Успешно
Проверка уменьшения здоровья при столкновении с объектами с тегом "Bad" и перехода на экран "Game Over" при исчерпании здоровья.	Успешно
Проверка перехода на следующий уровень при достижении 150 очков.	Успешно
Проверка корректности работы кнопок "Restart" и "Exit" в меню паузы.	Успешно
Проверка воспроизведения звуков при захвате объектов (OkSound, BoomSound, MilkSound).	Успешно
Проверка, что звуки воспроизводятся в правильных ситуациях.	Успешно

Продолжение таблицы 1

Шаг	Ожидаемый результат
Проверка активации и деактивации меню паузы по нажатию клавиши Escape.	Успешно
Проверка перехода на главное меню и настройки из меню паузы.	Успешно
Шутер	
Проверка корректного отображения здоровья игрока и врагов через интерфейс (HealthBar).	Успешно
Проверка уменьшения здоровья при столкновении с врагами и их пулями.	Успешно
Проверка перехода на экран "Game Over" при исчерпании здоровья игрока.	Успешно
Проверка стрельбы игрока с помощью правой кнопки мыши.	Успешно
Проверка правильности создания и движения лазеров (Laser) в сторону врагов.	Успешно
Проверка нанесения урона врагам при столкновении с лазерами.	Успешно
Проверка корректного движения врагов (BirdMove) вниз по экрану.	Успешно
Проверка стрельбы врагов (EnemyShoot) по игроку с заданным интервалом.	Успешно
Проверка корректного уничтожения врагов при получении урона.	Успешно
Проверка увеличения счета (score) при уничтожении всех врагов.	Успешно
Проверка перехода на следующий уровень при достижении 50 очков.	Успешно
Проверка работы кнопок "Restart" и "Exit" в меню паузы подсцены.	Успешно
Проверка корректности перехода между сценами при нажатии кнопок.	Успешно

Продолжение таблицы 1

Шаг	Ожидаемый результат
Викторина	
Тест на отображение вопроса	Текст вопроса и изображение должны соответствовать заданному вопросу.
Тест на отображение ответов	Кнопки должны содержать все варианты ответов, соответствующие текущему вопросу.
Тест на правильный ответ	Счет должен увеличиться на 1.
Тест на неправильный ответ	Счет должен остаться прежним.
Тест на завершение викторины	Сообщение о результате должно соответствовать количеству правильных ответов.
Тест на кнопку повторной попытки	Викторина должна перезапуститься, и первый вопрос должен отображаться заново.

3.2 Сборка приложения

Для сборки приложения (рисунок 35) использовалась функция Build Settings в Unity. Процесс сборки включал следующие шаги:

- открытие меню File в Build Settings.
- выбор платформы Windows для сборки.
- добавление всех сцен в сборку через кнопку Add Open Scenes.
- настройка параметров сборки, таких как имя приложения и папка для сохранения.
- нажатие кнопки Build для создания исполняемого файла.

После завершения сборки было проведено тестирование на целевой платформе для проверки работоспособности всех мини-игр и их взаимодействия. Также было создано видео, в котором демонстрируется процесс прохождения всех мини-игр, что позволяет наглядно оценить функциональность и стабильность приложения.

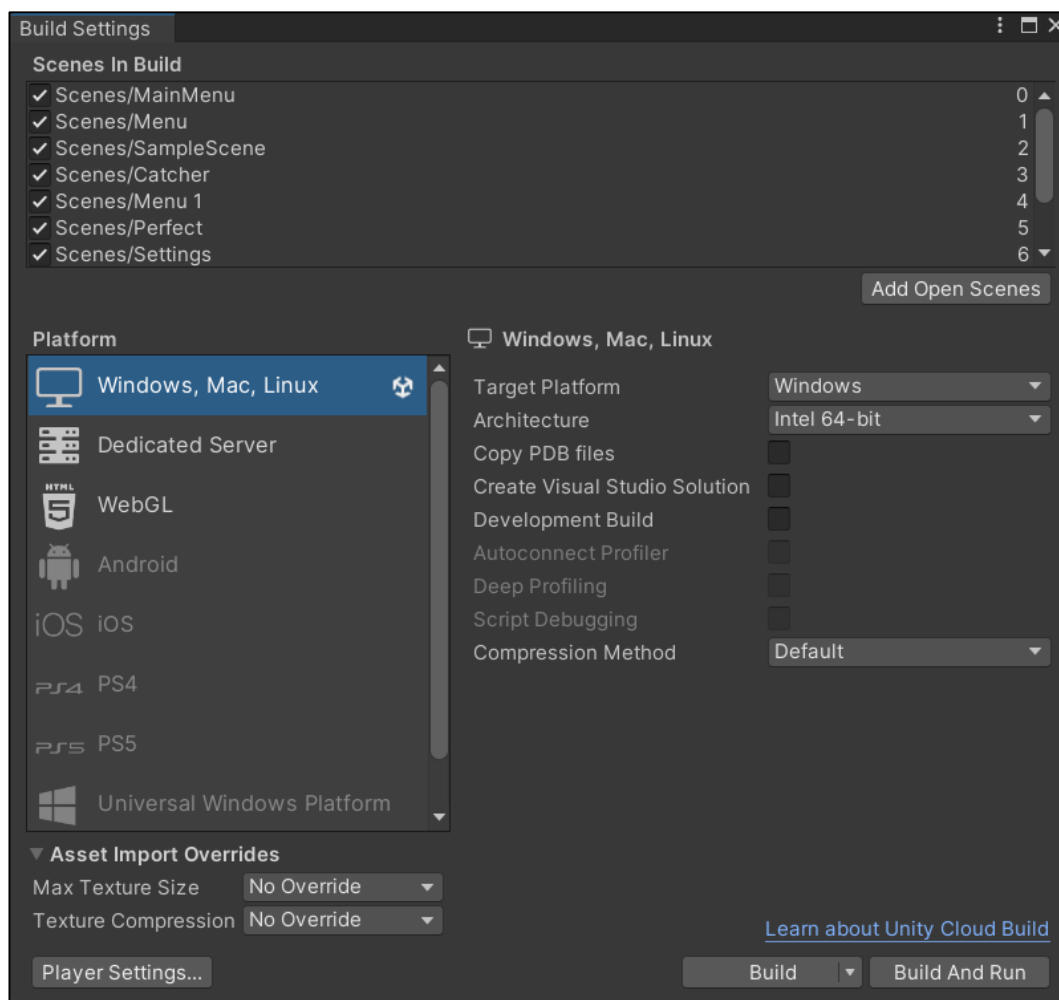


Рисунок 35 – Сборка проект

Также было создано видео, в котором демонстрируется процесс прохождения всех мини-игр, что позволяет наглядно оценить функциональность и стабильность приложения.

Заключение

В заключение, разработка 2D-игр в среде Unity представляет собой увлекательный и многогранный процесс, который сочетает в себе как технические, так и творческие аспекты. В ходе выполнения данной курсовой работы были исследованы ключевые принципы проектирования и реализации различных игровых жанров, таких как Платформеры, Кетчеры, Шутеры и Викторины. Каждый из этих жанров обладает своими уникальными характеристиками и механиками, что позволяет создавать разнообразный и интерактивный игровой опыт.

Работа над проектом не только углубила понимание особенностей разработки игр, но и продемонстрировала, как важны элементы графики, анимации и взаимодействия с пользователем для создания увлекательного игрового процесса. Таким образом, результаты данной курсовой работы могут служить основой для дальнейшего изучения и совершенствования навыков в области разработки игр, а также вдохновить на создание новых, оригинальных проектов в сфере 2D-гейминга.

Одним из основных инструментов для описания и анализа программных систем являются диаграммы, такие как диаграммы классов, диаграммы прецедентов и диаграмма состояний. Эти диаграммы позволяют разработчикам получить наглядное представление о структуре и поведении системы, что, в свою очередь, способствует более эффективному процессу разработки.

Список использованных источников

1. Могилев, А.В. Информатика. / Н.И. Пак, Е.К. Хённер – М.: Академия, 2004. – 848 с. 2. Шеллинг, Т. Стратегия конфликта. / М.: ИРИСЭН, 2007. – 376 с.
2. Диксит, А. Теория игр: искусство стратегического мышления в бизнесе и жизни/ Б. Нейлбафф, – М.: Манн, Иванов и Фербер, 2014. – 464 с.
3. Мураховский, В.И. Компьютерная графика – М.: АСТ-ПРЕСС 2002 – 638 с.
4. Корабельникова, Г.Б. Adobe Photoshop в теории и на практике. / Ю.А.Гурский – М.: Новое Знание, 2002. – 528 с.
5. Макарова, Н. В. Информатика / Л.А. Матвеев, В.Л. Бройдо – М.: Финансы и Статистика, 2002. – 768 с.
6. Пауэль, Т.А. Справочник программиста. / Т.А. Пауэль, Д. Уитворт – М.: АСТ, 2005. – 470 с.
7. Симонович, С.В. Практическая информатика. / Г.А. Евсеев – М.: АСТ-ПРЕСС КНИГА, 2005. – 480 с.
8. Морелос-Сарагоса, Р. Искусство помехоустойчивого кодирования. Методы, алгоритмы, применение – М.: ТЕХНОСФЕРА, 2006. – 320 с.
9. Блэкманн, С. Моделирование с помощью Unity – М.: АСТПРЕСС, 2011. – 147 с.
10. Голдстон, В. Предметы первой необходимости для создания игр на Unity – М.: Академия, 2009 – 201 с.
11. Игошев, Б. ЭВТ: знакомимся, делаем, играем. / М. Галагузова, Д. Комский – М.: Молодая гвардия, 1989. – 156 с.
12. Лысовский, А. Как сибирские программисты заставили весь мир играть. / А.Егошин, А.Бочкарев - М.:Русский Репортер №10, 2012. - 56- 61с.

Приложение

(Обязательное)

Разработка 2D компьютерной игры

Листинг программы

Файл ButtonScript.cs

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

public class ButtonScript : MonoBehaviour
{
    public void StartMeinMenu()
    {
        SceneManager.LoadScene(0);
    }

    public void StartPlatformer()
    {
        SceneManager.LoadScene(2);
    }

    public void StartCatcher()
    {
        SceneManager.LoadScene(3);
    }

    public void StartShuter()
    {
        SceneManager.LoadScene(7);
    }

    public void StartQuiz()
    {
        SceneManager.LoadScene(8);
    }
}
```

Файл CanvasButton.cs

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

public class CanvasButton : MonoBehaviour
{
    public void RestartGame()
    {
        SceneManager.LoadScene(2);
    }

    public void Exit()
    {
        SceneManager.LoadScene(0);
    }

    public void URL()
```

```

    {
        Application.OpenURL("https://pnzgu.ru/");
    }
}

```

Файл Entity.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Entity : MonoBehaviour
{
    public virtual void GetDamage()
    {

    }

    public virtual void Die()
    {
        Destroy(this.gameObject);
    }
}

```

Файл Hero2.cs

```

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

public class Hero2 : MonoBehaviour
{
    [SerializeField] private float speed = 3f; // скорость
    [SerializeField] public int lives = 5; // количество жизни
    [SerializeField] private float jumpForce = 6f; // сила прыжка

    private bool isGrounded = false;

    private Rigidbody2D rb;
    private Animator anim;
    private SpriteRenderer sprite;

    public static Hero2 Instance { get; set; }

    public LayerMask whatIsGround;
    public Transform groundChek;

    private States State
    {
        get { return (States)anim.GetInteger("state"); }
        set { anim.SetInteger("state", (int)value); }
    }
}

```

```

private void Awake()
{
    sprite = GetComponentInChildren<SpriteRenderer>(); // обращение к
спрайту
    anim = GetComponent<Animator>();
    rb = GetComponent<Rigidbody2D>();
    Instance = this;
}

private void Run()
{
    if (isGrounded) State = States.RUN;
    Vector3 dir = transform.right * Input.GetAxis("Horizontal");
    transform.position = Vector3.MoveTowards(transform.position,
transform.position + dir, speed * Time.deltaTime);
    sprite.flipX = dir.x < 0.0f; //поворот глаз
}

private void New()
{
    if (Input.GetKeyDown(KeyCode.Q)) State = States.New;
}

// Update is called once per frame
void Update()
{
    if (isGrounded) State = States.IDLE;
    if (Input.GetButton("Horizontal"))
    {
        Run();
    }
    if (isGrounded && Input.GetKeyDown(KeyCode.Q))
    {
        New();
    }
    if (isGrounded && Input.GetButtonDown("Jump"))
    {
        Jump();
    }
}

private void FixedUpdate()
{
    Debug.Log("isGrounded: " + isGrounded);
    CheckGround();
}

private void Jump()
{
    Debug.Log("Jump!");
    rb.AddForce(transform.up * jumpForce, ForceMode2D.Impulse);
}

private void CheckGround()
{

```

```

        Collider2D[] collider =
Physics2D.OverlapCircleAll(transform.position, 0.3f);
        isGrounded = collider.Length > 1;
        if (!isGrounded) State = States.JUMP;
        //isGrounded = Physics2D.OverlapPoint(groundChek.position,
whatIsGround);
    }

    public void GetDamage()
    {
        lives--;
        Debug.Log("Количество жизни у персонажа: " + lives);
        if (lives == 0)
        {
            Destroy(gameObject);
            SceneManager.LoadScene(0); // загружаем 1 сцену
        }
    }

    public enum States
    {
        IDLE,
        RUN,
        JUMP,
        New
    }
}

```

Файл RandomThree.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RandomThree : MonoBehaviour
{
    public GameObject[] objects;
    void Start()
    {
        int rand = Random.Range(0, objects.Length);
        Instantiate(objects[rand], transform.position, Quaternion.identity);
    }
}

```

Файл WalkingMonster.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class WalkingMonster : MonoBehaviour
{
    private Vector3 dir;
    private SpriteRenderer sprite;
}

```

```

private int lives2 = 6;

private void Start()
{
    dir = transform.right;
}

protected void Update()
{
    Move();
}

protected void Awake()
{
    sprite = GetComponentInChildren<SpriteRenderer>();
}

private void Move()
{
    Collider2D[] colliders =
Physics2D.OverlapCircleAll(transform.position + transform.up * 0.1f +
transform.right * dir.x * 0.7f, 0.01f);
    if(colliders.Length > 0)
    {
        dir *= -1f;
    }
    transform.position = Vector3.MoveTowards(transform.position,
transform.position + dir, Time.deltaTime);
}

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject == Hero2.Instance.gameObject)
    {
        Hero2.Instance.GetDamage();
    }
    if (collision.gameObject == Hero2.Instance.gameObject)
    {
        //Hero2.Instance.GetDamage();
        lives2--;
        Debug.Log("Количество жизни у монстра: " + lives2);
    }
    if (lives2 < 1)
    {
        Destroy(gameObject);
    }
}
}

```

Файл WalkingMonster2.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

```

```

public class WalkingMonster2 : MonoBehaviour
{
    private Vector3 dir;
    private SpriteRenderer sprite;
    private int lives2 = 6;
    private Transform target;

    private void Start()
    {
        dir = transform.right;
        target = GameObject.FindGameObjectWithTag("Hero").transform;
    }

    protected void Update()
    {
        Move();
    }

    protected void Awake()
    {
        sprite = GetComponentInChildren<SpriteRenderer>();
    }

    private void Move()
    {
        Vector3 direction = (target.position -
transform.position).normalized;
        transform.position = Vector3.MoveTowards(transform.position,
transform.position + direction, Time.deltaTime);
    }

    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject == Hero2.Instance.gameObject)
        {
            Hero2.Instance.GetDamage();
        }
        if (collision.gameObject == Hero2.Instance.gameObject)
        {
            //Hero2.Instance.GetDamage();
            lives2--;
            Debug.Log("Количество жизни у монстра: " + lives2);
        }
        if (lives2 < 1)
        {
            Destroy(gameObject);
        }
    }
}

```

Файл Wood.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

```

```

public class Wood : MonoBehaviour
{
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject == Hero2.Instance.gameObject)
        {
            Hero2.Instance.GetDamage();
        }
    }
}

```

Файл Wood2.cs

```

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

public class Wood2 : MonoBehaviour
{
    private void OnCollisionEnter2D(Collision2D collision)
    {
        SceneManager.LoadScene(2); // загружаем 1 сцену
    }
}

```

Файл Worm.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Worm : Entity
{
    [SerializeField] private int lives1 = 3;

    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject == Hero2.Instance.gameObject)
        {
            //Hero2.Instance.GetDamage();
            lives1--;
            Debug.Log("Количество жизни у червяка: " + lives1);
        }
        if (lives1 < 1)
        {
            Destroy(gameObject);
        }
    }
}

```

Файл CanvasButton1.cs

```

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

public class CanvasButton1 : MonoBehaviour
{
    public void RestartGame()
    {
        SceneManager.LoadScene(3); // загружаем 0 сцену
    }
    public void Exit()
    {
        SceneManager.LoadScene(0); // загружаем 1 сцену
    }
}

```

Файл itemChecker.cs

```

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

public class itemChecker : MonoBehaviour
{
    public int score;
    [SerializeField] private Image[] fishs;
    public GameObject ScoreObject;
    Text scoreText;

    public AudioClip OkSound, BoomSound, MilkSound;

    public int health; // Здоровье персонажа
    public int maxHealth = 3; // Максимальное здоровье
    void Start()
    {
        health = maxHealth; // Инициализируем здоровье
        scoreText = ScoreObject.GetComponent<Text>();

        // Сначала все изображения здоровья отображаем
        for (int i = 0; i < fishs.Length; i++)
        {
            fishs[i].enabled = true;
        }
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.gameObject.tag == "Good")
        {
            AudioSource.PlayClipAtPoint(OkSound, other.transform.position);
            score += 10;
            scoreText.text = score.ToString();
        }
    }
}

```

```

        Destroy(other.gameObject);
        if (score >= 150)
        {
            SceneManager.LoadScene(5); // загружаем 1 сцену
        }
    }
    else if (other.gameObject.tag == "Bad")
    {
        AudioSource.PlayClipAtPoint(BoomSound, other.transform.position);
        score -= 10;
        health--;
        if (health > 0)
        {
            fishs[health].enabled = false;
        }
        else
        {
            SceneManager.LoadScene(4); // загружаем 1 сцену
            Debug.Log("Game Over!");
            // Добавьте сюда код для завершения игры
        }
        Destroy(other.gameObject);
    }
    else if (other.gameObject.tag == "Milk")
    {
        AudioSource.PlayClipAtPoint(MilkSound, other.transform.position);
        if (health < maxHealth)
        {
            // Включаем текущее изображение здоровья
            fishs[health].enabled = true;

            // Увеличиваем здоровье
            health++;
        }
        else
        {
            score += 50; // Если здоровье максимальное, увеличиваем счет
        }

        // Обновляем текст счета
        scoreText.text = score.ToString();
        // Удаляем объект "Milk"
        Destroy(other.gameObject);
    }
}

void Update()
{
}
}

```

Файл itemSpawn.cs

```
using System.Collections;
```

```

using System.Collections.Generic;
using UnityEngine;

public class ItemSpawn : MonoBehaviour
{
    public GameObject RightSpawner;
    public GameObject[] items;
    public float startDelay, repeatRate;
    // Start is called before the first frame update
    void Start()
    {
        InvokeRepeating("Spawn", startDelay, repeatRate);
    }

    // Update is called once per frame
    void Spawn()
    {
        Vector3 pos = new
Vector3(Random.Range(gameObject.transform.position.x,
RightSpawner.transform.position.x), gameObject.transform.position.y, 0);
        Instantiate(items[Random.Range(0, items.Length)], pos,
gameObject.transform.rotation);
    }
}

```

Файл PauseMenu.cs

```

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

public class PauseMenu : MonoBehaviour
{
    // Start is called before the first frame update
    public bool PauseGame;
    public GameObject pauseGameMenu;
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.Escape))
        {
            if(PauseGame)
            {
                Rename();
            }
            else
            {
                Pause();
            }
        }
    }
}

```

```

    }
}

public void Rename()
{
    pauseGameMenu.SetActive(false);
    Time.timeScale = 1f;
    PauseGame = false;
}
public void Pause()
{
    pauseGameMenu.SetActive(true);
    Time.timeScale = 0f;
    PauseGame = true;
}
public void LoadMenu()
{
    Time.timeScale = 1f;
    SceneManager.LoadScene("MainMenu");
}

public void Settings()
{
    Time.timeScale = 1f;
    SceneManager.LoadScene("Settings");
}
}

```

Файл Player.cs

```

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

public class Player : MonoBehaviour
{
    [SerializeField] private float speed = 3f; // скорость игрока
    [SerializeField] public int lives = 5; // количество жизни игрока
    [SerializeField] private float jumpForce = 6f; // сила прыжка

    private bool isGrounded = false;

    private Rigidbody2D rb;
    private Animator anim;
    private SpriteRenderer sprite;

    public LayerMask whatIsGround;
    public Transform groundChek;

    private States state
    {
        get { return (States)anim.GetInteger("state"); }
        set { anim.SetInteger("state", (int)value); }
    }
}

```

```

private void Awake()
{
    sprite = GetComponentInChildren<SpriteRenderer>(); // обращение к
спрайту
    anim = GetComponent<Animator>();
    rb = GetComponent<Rigidbody2D>();
}

private void Run()
{
    if (isGrounded) State = States.RUN;
    Vector3 dir = transform.right * Input.GetAxis("Horizontal");
    transform.position = Vector3.MoveTowards(transform.position,
transform.position + dir, speed * Time.deltaTime);
    sprite.flipX = dir.x < 0.0f; //поворот глаз
}

private void New()
{
    if (Input.GetKeyDown(KeyCode.Q)) State = States.New;
}

void Update()
{
    if (isGrounded) State = States.IDLE;
    if (Input.GetButton("Horizontal"))
    {
        Run();
    }
    if (isGrounded && Input.GetKeyDown(KeyCode.Q))
    {
        New();
    }
    if (isGrounded && Input.GetButtonDown("Jump"))
    {
        Jump();
    }
}

private void FixedUpdate()
{
    Debug.Log("isGrounded: " + isGrounded);
    CheckGround();
}

private void Jump()
{
    Debug.Log("Jump!");
    rb.AddForce(transform.up * jumpForce, ForceMode2D.Impulse);
}

private void CheckGround()
{

```

```

        isGrounded = Physics2D.OverlapPoint(groundChek.position,
whatIsGround);
        if (!isGrounded) State = States.JUMP;
    }

    public enum States
    {
        IDLE,
        RUN,
        JUMP,
        New
    }
}

```

Файл VolumeValue.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class timeDestroyer : MonoBehaviour
{
    public float aliveTimer;
    // Start is called before the first frame update
    void Start()
    {
        Destroy(gameObject, aliveTimer);
    }

    // Update is called once per frame
    void Update()
    {
    }
}

```

Файл BirdHP.cs

```

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

public class BirdHP : MonoBehaviour
{
    public GameObject HealthBar;
    Image img;
    public int hp;
    float maxHp;

    // Ссылка на ScoreManager
    private ScoreManager scoreManager;
}

```

```

void Start()
{
    img = HealthBar.GetComponent<Image>();
    maxHp = hp;
    img.fillAmount = hp / maxHp;

    // Находим ScoreManager в сцене
    scoreManager = FindObjectOfType<ScoreManager>();
}

void MakeDamage(int damage)
{
    hp -= damage;
    if (hp <= 0)
    {
        // Увеличиваем счет на 5
        scoreManager.IncreaseScore(5);
        Destroy(gameObject);
    }
    img.fillAmount = hp / maxHp;
}

void Update()
{
}
}

```

Файл BirdMove.cs

```

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

public class BirdMove : MonoBehaviour
{
    public float speed;
    Rigidbody2D rb;
    // Start is called before the first frame update
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        Vector3 move = new Vector3(0, -1, 0);
        rb.velocity = move * speed;
    }

    void OnBecameInvisible()
    {
        Destroy(gameObject);
    }

    void OnCollisionEnter2D(Collision2D something)
    {
        if(something.gameObject.tag == "PlayerCat")

```

```

        {
            Destroy(gameObject);
            SceneManager.LoadScene(SceneManager.GetActiveScene().name);
        }
        if (something.gameObject.tag == "layer")
        {
            Destroy(gameObject);
        }
    }

    // Update is called once per frame
    void Update()
    {

    }
}

```

Файл CanvasButtonShuter.cs

```

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

public class CanvasButtonShuter : MonoBehaviour
{
    public void RestartGame()
    {
        SceneManager.LoadScene(7);
    }
    public void Exit()
    {
        SceneManager.LoadScene(0);
    }
    public void Perfect2()
    {
        SceneManager.LoadScene(0);
    }
}

```

Файл EnemyBullet.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EnemyBullet : MonoBehaviour
{
    GameObject player;
    Rigidbody2D rb;
    public float force;
    public int damage;
    // Start is called before the first frame update
    void Start()
    {
    }
}

```

```

{
    rb = GetComponent<Rigidbody2D>();
    player = GameObject.FindWithTag("PlayerCat");
    if (player != null)
    {
        Vector3 dir = player.transform.position - transform.position;
        float angle = Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg;
        transform.Rotate(0, 0, angle);
        rb.AddForce(dir.normalized * force, ForceMode2D.Impulse);

    }
    else
    {
        Destroy(gameObject);
    }
}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "PlayerCat")
    {
        other.gameObject.SendMessage("MakeDamage", damage,
        SendMessageOptions.DontRequireReceiver);
        Destroy(gameObject);
    }
}

private void OnBecameInvisible()
{
    Destroy(gameObject);
}

// Update is called once per frame
void Update()
{
}
}

```

Файл EnemyHP.cs

```

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

public class EnemyHP : MonoBehaviour
{
    public GameObject HealthBar;
    Image img;
    public int hp;
    float maxHp;

    // Ссылка на ScoreManager

```

```

private ScoreManager scoreManager;

void Start()
{
    img = HealthBar.GetComponent<Image>();
    maxHp = hp;
    img.fillAmount = hp / maxHp;

    // Находим ScoreManager в сцене
    scoreManager = FindObjectOfType<ScoreManager>();
}

void MakeDamage(int damage)
{
    hp = hp - damage;
    if (hp <= 0)
    {
        // Увеличиваем счет на 10
        scoreManager.IncreaseScore(10);
        Destroy(gameObject);
    }
    img.fillAmount = hp / maxHp;
}

// Update is called once per frame
void Update()
{
}
}

```

Файл EnemyShoot.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EnemyShoot : MonoBehaviour
{
    public GameObject bullet;
    public float fireDelay;
    GameObject player;
    bool canShoot = true;
    // Start is called before the first frame update
    void Start()
    {
        player = GameObject.FindWithTag("PlayerCat");
    }

    // Update is called once per frame
    void Update()
    {
        if (canShoot && player != null)
        {
            canShoot = false;

```

```

        Instantiate(bullet, transform.position, Quaternion.identity);
        StartCoroutine(firePause());
    }
}

IEnumerator firePause()
{
    yield return new WaitForSeconds(fireDelay);
    canShoot = true;
}
}

```

Файл HealthBar.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class HealthBar : MonoBehaviour
{
    void Update()
    {
        // Обнуляем вращение healthbar
        transform.rotation = Quaternion.Euler(0, 0, 0);
    }
}

```

Файл HpController.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class HpController : MonoBehaviour
{
    public int hp;

    void MakeDamage(int damage)
    {
        hp = hp - damage;
        if(hp <= 0)
        {
            Destroy(gameObject);
        }
    }
    // Start is called before the first frame update
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {
    }
}

```

```
}  
}
```

Файл LaserShot.cs

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class LaserShot : MonoBehaviour  
{  
    Rigidbody2D rb;  
    public int damage;  
    public float force;  
    // Start is called before the first frame update  
    void Start()  
    {  
        rb = GetComponent<Rigidbody2D>();  
        Vector3 direction = new Vector3(0, force, 0);  
        rb.AddForce(direction, ForceMode2D.Impulse);  
    }  
  
    void OnBecameInvisible()  
    {  
        Destroy(gameObject);  
    }  
  
    private void OnTriggerEnter2D(Collider2D other)  
    {  
        if (other.gameObject.tag == "Enemy")  
        {  
            other.gameObject.SendMessage("MakeDamage", damage,  
SendMessageOptions.DontRequireReceiver);  
            Destroy(gameObject);  
        }  
    }  
  
    // Update is called once per frame  
    void Update()  
    {  
    }  
}
```

Файл ObjectSpawnerBird.cs

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class ObjectSpawnerBird : MonoBehaviour  
{  
    public Transform RightPositionSpawner;
```

```

public float spawnDelay;
public GameObject[] Item;
// Start is called before the first frame update
void Start()
{
    InvokeRepeating("Spawn", spawnDelay, spawnDelay);
}

void Spawn()
{
    Vector3 spawnPos = new Vector3(Random.Range(transform.position.x,
RightPositionSpawner.position.x), transform.position.y, 0);
    int i = Random.Range(0, Item.Length);
    Instantiate(Item[i], spawnPos, transform.rotation);
}

// Update is called once per frame
void Update()
{
}
}

```

Файл PlayerCatHP.cs

```

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

public class PlayerCatHP : MonoBehaviour
{
    public GameObject HealthBar;
    Image img;
    public int hp;
    float maxHp;
    // Start is called before the first frame update
    void Start()
    {
        img = HealthBar.GetComponent<Image>();
        maxHp = hp;
        img.fillAmount = hp / maxHp;
    }

    void MakeDamage(int damage)
    {
        hp = hp - damage;
        if (hp <= 0)
        {
            SceneManager.LoadScene(9);
        }
        img.fillAmount = hp / maxHp;
    }
}

```

```

    void Update()
    {
    }
}

```

Файл PlayerCatSpawner.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerCatSpawner : MonoBehaviour
{
    public Transform PlayerRightPositionSpawner;
    public GameObject Item;
    void Start()
    {
        Vector3 spawnPos = new Vector3(Random.Range(transform.position.x,
PlayerRightPositionSpawner.position.x), transform.position.y, 0);
        Instantiate(Item, spawnPos, transform.rotation);
    }

    void Update()
    {
    }
}

```

Файл PlayerMoveCat.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerMoveCat : MonoBehaviour
{
    Vector3 clickPos;
    Vector3 move;
    public float speed = 1;
    public float tiltAngle = 15f; // Угол наклона
    Rigidbody2D rb;

    // Start is called before the first frame update
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        clickPos = transform.position;
    }

    void Update()
    {
        if (Input.GetMouseButton(0))
        {

```

```

        clickPos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        clickPos.z = 0; // Убедитесь, что z-координата равна 0
    }

    move = clickPos - transform.position;

    // Наклон персонажа в зависимости от направления движения
    if (move.x > 0.1f) // Движение вправо
    {
        transform.rotation = Quaternion.Euler(0, 0, -tiltAngle);
    }
    else if (move.x < -0.1f) // Движение влево
    {
        transform.rotation = Quaternion.Euler(0, 0, tiltAngle);
    }
    else // Если не движется, возвращаем в исходное положение
    {
        transform.rotation = Quaternion.Euler(0, 0, 0);
    }
}

void FixedUpdate()
{
    rb.velocity = new Vector2(move.x, move.y) * speed;
}
}

```

Файл PlayerShoot.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerShot : MonoBehaviour
{
    public GameObject Laser;
    public float delayTime;
    bool canShoot = true;
    // Start is called before the first frame update
    void Start()
    {
    }

    void Update()
    {
        if (canShoot && Input.GetMouseButton(1))
        {
            canShoot = false;
            Instantiate(Laser, transform.position, transform.rotation);
            StartCoroutine(NoFire());
        }
    }

    IEnumerator NoFire()
    {
    }
}

```

```

        yield return new WaitForSeconds(delayTime);
        canShoot = true;
    }
}

```

Файл ScoreManager.cs

```

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

public class ScoreManager : MonoBehaviour
{
    public static int score = 0;
    public Text scoreText;

    void Update()
    {
        scoreText.text = "Score: " + score;

        if (score >= 50)
        {
            LoadNextScene();
        }
    }

    public void IncreaseScore(int amount)
    {
        score += amount;
    }

    private void LoadNextScene()
    {
        SceneManager.LoadScene(10);
    }
}

```

Файл Quiz.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;
using TMPro;

public class QuizManager : MonoBehaviour
{
    [System.Serializable]
    public class Question
    {
        public string questionText;
    }
}

```

```

        public Sprite questionImage;
        public List<string> answers;
        public int correctAnswerIndex;
    }

    public List<Question> questions;
    public Text questionText;
    public Image questionImage;
    public Button[] answerButtons;
    public Text resultText;
    public Text scoreText; // Добавляем ссылку на текст для отображения счета
    public Button retryButton; // Добавляем ссылку на кнопку для перехода

    private int currentQuestionIndex = 0;
    private int correctAnswersCount = 0;

    void Start()
    {
        ShuffleQuestions();
        DisplayQuestion();
        UpdateScore(); // Обновляем счет в начале
        retryButton.gameObject.SetActive(false); // Скрываем кнопку в начале
    }

    void ShuffleQuestions()
    {
        for (int i = 0; i < questions.Count; i++)
        {
            Question temp = questions[i];
            int randomIndex = Random.Range(i, questions.Count);
            questions[i] = questions[randomIndex];
            questions[randomIndex] = temp;
        }
    }

    void DisplayQuestion()
    {
        if (currentQuestionIndex < questions.Count)
        {
            Question currentQuestion = questions[currentQuestionIndex];
            questionText.text = currentQuestion.questionText;
            questionImage.sprite = currentQuestion.questionImage;

            for (int i = 0; i < answerButtons.Length; i++)
            {
                answerButtons[i].GetComponentInChildren<Text>().text =
currentQuestion.answers[i];
                int index = i; // Лямбда замыкание
                answerButtons[i].onClick.RemoveAllListeners();
                answerButtons[i].onClick.AddListener(() =>
AnswerQuestion(index));
            }
        }
        else
        {
            EndQuiz();
        }
    }

```

```

    }
}

void AnswerQuestion(int index)
{
    if (index == questions[currentQuestionIndex].correctAnswerIndex)
    {
        correctAnswersCount++;
        Debug.Log("Правильный ответ! Текущий счет: " +
correctAnswersCount);
    }
    UpdateScore(); // Обновляем счет после ответа
    currentQuestionIndex++;
    DisplayQuestion();
}

void UpdateScore()
{
    scoreText.text = "Счет: " + correctAnswersCount + " / " +
questions.Count; // Обновляем текст счета
}

void EndQuiz()
{
    questionText.gameObject.SetActive(false);
    questionImage.gameObject.SetActive(false);
    foreach (Button button in answerButtons)
    {
        button.gameObject.SetActive(false);
    }

    // Логика для вывода результата
    if (correctAnswersCount >= 6)
    {
        resultText.text = "Вы прошли викторину! Вы ответили правильно на
" + correctAnswersCount + " из " + questions.Count + " вопросов.";
    }
    else
    {
        resultText.text = "Вы не прошли викторину. Вы ответили правильно
на " + correctAnswersCount + " из " + questions.Count + " вопросов.";
    }

    resultText.gameObject.SetActive(true);
    retryButton.gameObject.SetActive(true); // Показываем кнопку в конце
викторины
    retryButton.onClick.AddListener(LoadNextScene); // Привязываем метод
к кнопке
}

void LoadNextScene()
{
    // Замените "NextSceneName" на имя вашей следующей сцены
    SceneManager.LoadScene(0);
}
}

```