САНКТ-ПЕТЕРБУРГСКОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ «САНКТ-ПЕТЕРБУРГСКИЙ ТЕХНИЧЕСКИЙ КОЛЛЕДЖ УПРАВЛЕНИЯ И КОММЕРЦИИ»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к дипломному проекту

Тема: *Разработка игры в жанре Платформер на Unity2D*

Руководитель

преподаватель профессиональных дисциплин		*А.В.Смирнова*
(должность)	*(подпись)*	*(И.О. Фамилия)*

Студент

9ПО-41		*Д.Д. Проскурня*
(группа)	*(подпись)*	*(И.О. Фамилия)*

Специальность *09.02.07 «Информационные системы и программирование»*
(шифр и наименование специальности)

Работа допущена к защите

Председатель ПЦК _____ *А.В. Смирнова*
(подпись) *(И.О. Фамилия)*

Зав. отделением _____ *О.В. Бондарук*
(подпись) *(И.О. Фамилия)*

Санкт-Петербург
2025 г.

САНКТ-ПЕТЕРБУРГСКОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ «САНКТ-ПЕТЕРБУРГСКИЙ ТЕХНИЧЕСКИЙ КОЛЛЕДЖ УПРАВЛЕНИЯ И КОММЕРЦИИ»

У Т В Е Р Ж Д А Ю

Председатель ПЦК_____/Смирнова А.В.
« 21 » апреля 2025 г.

ЗАДАНИЕ
на дипломный проект

Студент Проскурня Дмитрий Дмитриевич
Специальность 09.02.07 «Информационные системы и программирование »
Группа 9ПО-41

Тема дипломной работы *Разработка игры в жанре Платформер на Unity2D*

Тема утверждена приказом по колледжу от «29» апреля 2025 г. №21
Срок сдачи дипломной *работы* « 02 » июня 2025 г.

Содержание *работы (перечень вопросов, подлежащих рассмотрению)**:***

Введение:

- назначение программного обеспечения, актуальность темы и практическое значение, цель, поставленные задачи

Основная часть

Теоретическая часть:

- выбор среды и средств разработки;
- анализ выбранной среды разработки;
- изучение среды разработки игр Unity;

Практическая часть:

- постановка задачи и описание игрового процесса;
- проектирование и создание интерфейса пользователя;
- реализация игрового функционала;

Заключение:

- выводы и предложения как теоретического, так и практического характера, полученные в результате дипломного проектирования.

Перечень иллюстрированного материала *(кол-во листов и их содержание)*

Таблицы, рисунки, листинг программного кода

Руководитель Анастасия Владимировна Смирнова
(Имя, Отчество, Фамилия)

График выполнения проекта *(работы)*

Раздел проекта (работы)	Календарный срок выполнения	Отметка о выполнении
Подбор материала, его анализ и обобщение	30.04	
Обзор и анализ предметной области	07.05	
Программная реализация системы	17.05	
Представление раздела «Теоретическая часть»	21.05	
Представление раздела «Практическая часть»	24.05	
Проверка ВКР, составление отзыва	02.06	

Дата выдачи задания _____ *А.В. Смирнова*

(подпись руководителя, дата) (И.О.Ф. руководителя)

С заданием ознакомлен(а) _____ *Д.Д. Проскурня*

(подпись студента, дата) (И.О.Ф. студента)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....5
ГЛАВА 1. ТЕОРЕТИЧЕСКИЕ АСПЕКТЫ РАЗРАБОТКИ ИГРЫ.....7
1.1 Классификация компьютерных игр.....7
1.2 Анализ средств разработки 2D игр — игровые движки.....11
1.3 Особенности игрового движка Unity.....16
1.4 История и развитие платформеров.....20
ГЛАВА 2. РАЗРАБОТКА ИГРОВОГО ПРИЛОЖЕНИЯ.....22
2.1 Постановка задачи.....22
2.2 Реализация создания игрового проекта.....24
2.2.1 Описание структуры разрабатываемой игры.....24
2.2.2 Создание сцен для отдельных элементов игры.....25
2.2.3 Создание игровых механик игры.....29
2.2.4 Создание собираемых предметов.....30
2.2.5 Создание ловушек и врагов.....34
2.2.6 Создание системы смерти.....36
ЗАКЛЮЧЕНИЕ.....38
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ.....40
ПРИЛОЖЕНИЯ.....41

ВВЕДЕНИЕ

Современный этап развития информационных технологий характеризуется активным развитием индустрии компьютерных игр, которая становится не только одним из ключевых сегментов развлекательной индустрии, но и важным инструментом в образовании, науке и культуре. Компьютерные игры представляют собой сложные программные продукты, сочетающие в себе элементы графики, звука, физики и искусственного интеллекта. Создание таких продуктов требует не только творческого подхода, но и глубоких знаний в области программирования, проектирования и использования современных игровых движков.

Одним из наиболее популярных и доступных инструментов для разработки игр является движок Unity, который предоставляет широкие возможности для создания как 2D-, так и 3D-игр. Unity обладает интуитивно понятным интерфейсом, мощной системой визуализации и поддержкой множества платформ, что делает его идеальным выбором для разработчиков разного уровня подготовки. В рамках данного дипломного проекта была поставлена задача создания 2D-платформера на движке Unity, что позволяет изучить ключевые аспекты разработки игр, включая проектирование уровней, реализацию игровой физики, анимации и взаимодействия объектов.

Актуальность темы дипломного проекта обусловлена растущим интересом к инди-играм и 2D-платформерам, которые, благодаря своей простоте и увлекательности, продолжают оставаться популярными среди игроков. Кроме того, разработка 2D-игр является отличной возможностью для изучения основ геймдизайна и программирования, что делает данный проект полезным как с практической, так и с образовательной точки зрения.

Целью дипломного проекта является разработка компьютерной игры в жанре 2D-платформер на движке Unity, включая проектирование уровня, создание игровых механик и реализацию визуальных и звуковых эффектов.

Для достижения этой цели были поставлены следующие задачи:

1. Изучить особенности работы с движком Unity и его инструментами для создания 2D-игр.
2. Разработать концепцию игры, включая дизайн уровня и персонажей.
3. Реализовать основные игровые механики, такие как перемещение персонажа, прыжки, взаимодействие с объектами и врагами.
4. Создать анимации и визуальные эффекты.
5. Протестировать игру на наличие ошибок и оптимизировать её производительность.

В процессе работы над проектом использовались такие технологии, как C# для программирования игровой логики, встроенные инструменты Unity для работы с графикой и анимацией, а также сторонние библиотеки для оптимизации и расширения функциональности. Результатом работы стал готовый к использованию 2D-платформер, который демонстрирует возможности Unity и может служить основой для дальнейшего развития проекта.

Данный дипломный проект представляет собой пример практического применения современных технологий в разработке игр и может быть полезен для студентов, изучающих программирование и геймдизайн, а также для всех, кто интересуется созданием компьютерных игр.

ГЛАВА 1. ТЕОРЕТИЧЕСКИЕ АСПЕКТЫ РАЗРАБОТКИ ИГРЫ

1.1 Классификация компьютерных игр

Компьютерные игры представляют собой сложные программные продукты, которые можно классифицировать по различным критериям. Классификация помогает лучше понять особенности игр, их целевую аудиторию и технические аспекты. Ниже приведены основные способы классификации компьютерных игр.

Жанр игры определяется её основными механиками, целями и стилем gameplay. Основные жанры включают:

- аркады (Arcade) – простые и динамичные игры с упором на реакцию и координацию (например, Pac-Man, Tetris),
- платформеры (Platformers) – игры, в которых игрок управляет персонажем, перемещающимся по платформам и преодолевающим препятствия (например, Super Mario, Celeste),
- ролевые игры (RPG — Role-Playing Games) — игры с глубоким сюжетом, развитием персонажа и элементами прокачки (например, The Witcher, Skyrim),
- стратегии (Strategy) – игры, требующие планирования и управления ресурсами (например, StarCraft, Civilization). Поджанры: пошаговые (TBS — Turn-Based Strategy) и реального времени (RTS — Real-Time Strategy),
- шутеры (Shooters) – игры с акцентом на стрельбу и боевые действия (например, Call of Duty, Counter-Strike). Поджанры: от первого лица (FPS — First-Person Shooter) и от третьего лица (TPS — Third-Person Shooter),
- приключения (Adventure) – игры с упором на исследование мира и решение головоломок (например, The Legend of Zelda, Tomb Raider),
- симуляторы (Simulators) – игры, имитирующие реальные процессы или деятельность (например, The Sims, Microsoft Flight Simulator),
- гонки (Racing) – игры, посвященные соревнованиям на транспортных средствах (например, Need for Speed, Forza Horizon),

- файтинги (Fighting) – игры, в которых игроки управляют бойцами и сражаются друг с другом (например, Mortal Kombat, Street Fighter),
- пазлы (Puzzle) – игры, требующие решения логических задач (например, Portal, Monument Valley),
- спортивные (Sports) – игры, имитирующие реальные виды спорта (например, FIFA, NBA 2K),
- хорроры (Horror) – игры, создающие атмосферу страха и напряжения (например, Resident Evil, Silent Hill),
- MMO (Massively Multiplayer Online) – игры, поддерживающие одновременное участие большого числа игроков в онлайн-мире (например, World of Warcraft, EVE Online).

По количеству игроков:

− одиночные (Single-player) – игры, рассчитанные на одного игрока (например, The Elder Scrolls V: Skyrim),

− многопользовательские (Multiplayer) – игры, поддерживающие участие нескольких игроков (например, Fortnite, Among Us). Поджанры: кооперативные (Co-op) и соревновательные (PvP — Player vs Player),

− MMO (Massively Multiplayer Online) – игры с огромным количеством игроков в одном мире (например, World of Warcraft).

По платформам:

− ПК (PC) – игры для персональных компьютеров (например, Cyberpunk 2077),

− консольные (Console) – игры для игровых приставок (например, PlayStation, Xbox, Nintendo Switch),

− мобильные (Mobile) – игры для смартфонов и планшетов (например, Candy Crush, PUBG Mobile),

− виртуальная реальность (VR) – игры, использующие технологии виртуальной реальности (например, Half-Life: Alyx).

По графике:

- 2D – игры с двухмерной графикой (например, Hollow Knight),
- 3D – игры с трехмерной графикой (например, The Legend of Zelda: Breath of the Wild),
- воксельная графика – игры, использующие воксели для создания объемных объектов (например, Minecraft),
- стилизованная графика – игры с уникальным художественным стилем (например, Cuphead, Ori and the Blind Forest).

По целям и содержанию:

- коммерческие – игры, созданные для получения прибыли (например, Call of Duty, FIFA),
- инди (Indie) – игры, разработанные небольшими студиями или независимыми разработчиками (например, Stardew Valley, Hades),
- обучающие (Educational) – игры, предназначенные для обучения и развития навыков (например, Minecraft: Education Edition),
- серьезные игры (Serious Games) – игры, созданные для решения неигровых задач (например, тренировка врачей, обучение сотрудников).

По возрастному рейтингу:

- для всех возрастов (E — Everyone) – игры, подходящие для любой аудитории (например, Mario Kart),
- для подростков (T — Teen) – игры, рекомендованные для игроков старше 13 лет (например, Fortnite),
- для взрослых (M — Mature): - игры, предназначенные для игроков старше 17 лет (например, Grand Theft Auto V).

По типу монетизации:

- платные (Premium) — игры, которые приобретаются за фиксированную стоимость (например, The Witcher 3),
- Free-to-Play (F2P) – бесплатные игры с возможностью покупки внутриигровых предметов (например, League of Legends),

− подписка (Subscription) — игры, доступные по подписке (например, Xbox Game Pass),

− реклама (Ad-supported) – бесплатные игры с показом рекламы (например, многие мобильные игры).

Классификация компьютерных игр помогает структурировать огромное разнообразие игровых продуктов и понять их особенности. Выбор жанра, платформы или типа монетизации зависит от предпочтений игроков, целей разработчиков и технических возможностей. Современные игры часто сочетают в себе элементы нескольких жанров и платформ, что делает их ещё более увлекательными и универсальными.

1.2 Анализ средств разработки 2D игр — игровые движки

Разработка 2D-игр требует выбора подходящего инструмента, который обеспечит удобство работы, поддержку необходимых функций и кроссплатформенность. Современные игровые движки предлагают широкие возможности для создания 2D-игр, и выбор зависит от задач проекта, опыта разработчика и целевой платформы. Ниже приведён анализ популярных игровых движков для разработки 2D-игр.

Особенности Unity:

- универсальность: Unity поддерживает как 2D, так и 3D-разработку, что делает его одним из самых популярных движков,
- кроссплатформенность: Игры можно выпускать на PC, мобильные устройства, консоли и веб-платформы,
- инструменты для 2D: Встроенные инструменты для работы со спрайтами, анимациями, физикой и тайлмапами,
- сообщество и документация: Огромное количество tutorials, форумов и ассетов в Asset Store.

Преимущества Unity:

- подходит для новичков и профессионалов,
- большое количество готовых решений и плагинов,
- активное сообщество и регулярные обновления.

Недостатки Unity:

- требует знания C#,
- для сложных 2D-игр может потребоваться оптимизация.

Примеры игр: Hollow Knight, Cuphead, Ori and the Blind Forest.

Особенности Godot:

- открытый исходный код: Godot полностью бесплатен и не требует лицензионных отчислений,
- 2D-ориентированность: Движок изначально заточен под 2D-разработку, что делает его легче и быстрее в использовании,

− GDScript: Простой в изучении язык программирования, похожий на Python,

− кроссплатформенность: Поддержка PC, мобильных устройств и веб-платформ.

Преимущества Godot:

− лёгкий и быстрый движок,

− подходит для инди-разработчиков,

− хорошая документация и растущее сообщество.

Недостатки Godot:

− меньше готовых ассетов и плагинов по сравнению с Unity,

− ограниченная поддержка 3D (хотя это не критично для 2D-игр).

Примеры игр: Dungeon Demons, Kingdoms of the Dump.

Особенности GameMaker Studio:

− простота использования: Визуальное программирование (Drag-and-Drop) и язык GML (GameMaker Language),

− 2D-ориентированность: Движок создан специально для 2D-игр,

− быстрая разработка: Подходит для прототипирования и создания небольших игр.

Преимущества GameMaker Studio:

− идеален для новичков,

− быстрый старт разработки,

− поддержка множества платформ.

Недостатки GameMaker Studio:

− ограниченные возможности для сложных проектов,

− платная лицензия для экспорта на некоторые платформы.

Примеры игр: Undertale, Hyper Light Drifter.

Особенности Construct:

− визуальное программирование: не требует знания языков программирования,

− быстрая разработка: подходит для создания простых игр и прототипов,

− 2D-ориентированность: упрощённые инструменты для работы с графикой и анимацией.

Преимущества Construct:

− идеален для новичков и образовательных проектов,

− быстрое создание игр без написания кода.

Недостатки Construct:

− ограниченные возможности для сложных проектов,

− платная лицензия для экспорта на некоторые платформы.

Примеры игр: Baba Is You (прототип был создан в Construct).

Особенности RPG Maker:

− специализация: движок заточен под создание 2D-ролевых игр (RPG),

− простота использования: встроенные инструменты для создания карт, диалогов и событий,

− готовые ассеты: большое количество предустановленных спрайтов, звуков и музыки.

Преимущества RPG Maker:

− идеален для создания классических RPG,

− не требует глубоких знаний программирования.

Недостатки RPG Maker:

− ограниченная гибкость для игр других жанров,

− платная лицензия.

Примеры игр: To the Moon, Ara Fell.

Особенности Phaser:

− фреймворк для веб-игр: Phaser — это JavaScript-библиотека для создания 2D-игр, работающих в браузере,

− простота интеграции: легко встраивается в веб-страницы,

− бесплатность: полностью открытый и бесплатный инструмент.

Преимущества Phaser:

− подходит для веб-разработчиков,

− лёгкий и быстрый инструмент для простых игр.

Недостатки Phaser:

− ограниченные возможности для сложных проектов,

− требует знания JavaScript.

Примеры игр: Browserquest, Angry Birds (прототип).

Сравнительный анализ игровых движков представлен в таблице 1.

Выбор игрового движка для разработки 2D-игр зависит от ваших задач, опыта и предпочтений:

− Unity — универсальный выбор для сложных проектов,

− Godot — идеален для инди-разработчиков, которые хотят работать с открытым исходным кодом,

− GameMaker Studio — отличный вариант для новичков и быстрого прототипирования,

− Construct — подходит для простых игр и образовательных проектов,

− RPG Maker — специализированный инструмент для создания RPG,

− Phaser — лучший выбор для веб-игр.

Таблица 1 – Сравнительная таблица движков

Движок	Язык программирования	2D-ориентированность	Кроссплатформенность	Сложность обучения
Unity	C#	Да (и 3D)	Высокая	Средняя
Godot	GDScript, C#	Да	Высокая	Низкая
Game Maker	GML, Drag-and-Drop	Да	Высокая	Низкая
Construct	Визуальное программирование	Да	Средняя	Очень низкая
RPG Maker	Ruby (скрипты)	Да (RPG)	Ограниченная	Низкая
Phaser	JavaScript	Да	Веб	Средняя

1.3 Особенности игрового движка Unity

Unity — один из самых популярных и универсальных игровых движков, который используется для создания игр и интерактивных приложений. Он поддерживает как 2D, так и 3D-разработку, что делает его подходящим для широкого спектра проектов. Ниже приведены ключевые особенности Unity, которые делают его таким востребованным среди разработчиков.

Кроссплатформенность. Unity позволяет разрабатывать игры для более чем 25 платформ, включая:

−десктопные платформы: Windows, macOS, Linux,

−мобильные платформы: iOS, Android,

−консоли: PlayStation, Xbox, Nintendo Switch,

−виртуальная и дополненная реальность: VR (Oculus, HTC Vive), AR (ARKit, ARCore),

−веб-платформы: WebGL, WebAssembly.

Это делает Unity идеальным выбором для разработчиков, которые хотят охватить максимально широкую аудиторию.

Поддержка 2D и 3D-разработки. 2D-разработка:

−встроенные инструменты для работы со спрайтами, анимациями, тайлмапами и 2D-физикой,

−поддержка пиксель-арта и векторной графики.

Примеры игр: Hollow Knight, Cuphead.

3D-разработка:

−мощные инструменты для работы с 3D-моделями, текстурами, освещением и физикой.

−поддержка современных технологий, таких как HDRP (High Definition Render Pipeline) для высококачественной графики.

Примеры игр: Ori and the Will of the Wisps, Monument Valley.

Визуальная среда разработки. Интерфейс: Unity предоставляет интуитивно понятный интерфейс, который можно настраивать под нужды разработчика.

Редактор сцен: Позволяет визуально редактировать уровни, расставлять объекты и настраивать их свойства.

Анимация: Встроенный аниматор (Animator) и система анимаций (Timeline) для создания сложных анимаций и кат-сцен.

Язык программирования. C#: Unity использует C# как основной язык программирования, который является мощным и относительно простым в изучении.

Визуальное программирование: с помощью инструмента Bolt (или Unity Visual Scripting) можно создавать логику игры без написания кода.

Asset Store. Библиотека ассетов: Unity Asset Store содержит тысячи готовых assets (модели, текстуры, звуки, скрипты), которые можно использовать в своих проектах.

Экономия времени: Позволяет ускорить разработку, используя готовые решения.

Физика и графика. Физика: Встроенная поддержка 2D и 3D-физики (NVIDIA PhysX).

Графика: Поддержка современных технологий рендеринга, таких как:

−URP (Universal Render Pipeline): для оптимизированной графики,

−HDRP (High Definition Render Pipeline): для высококачественной графики,

−Shader Graph: визуальный редактор шейдеров.

Сообщество и документация. Активное сообщество: Unity имеет огромное сообщество разработчиков, что упрощает поиск решений и обмен опытом.

Документация: Официальная документация Unity считается одной из самых подробных и доступных.

Tutorials: Множество обучающих материалов, как официальных, так и созданных сообществом.

Поддержка VR/AR. VR: Unity поддерживает разработку для Oculus Rift, HTC Vive, PlayStation VR и других платформ.

AR: поддержка ARKit (iOS) и ARCore (Android) для создания приложений дополненной реальности.

Мультиплеер и сетевая игра. Netcode: Unity предоставляет инструменты для создания многопользовательских игр, включая Unity Netcode и поддержку сторонних решений (Photon, Mirror).

Серверные решения: возможность интеграции с облачными сервисами (PlayFab, AWS).

Мобильная разработка. Оптимизация: Unity предоставляет инструменты для оптимизации игр под мобильные устройства.

Монетизация: поддержка рекламы (Unity Ads) и внутриигровых покупок.

Преимущества Unity:

−универсальность: подходит для игр любого жанра и сложности,

−кроссплатформенность: один код — множество платформ,

−большое сообщество: легко найти помощь и готовые решения,

−гибкость: возможность настройки под нужды проекта.

Недостатки Unity:

−производительность: для сложных 3D-игр может потребоваться оптимизация,

−лицензирование: бесплатная версия (Personal) имеет ограничения, а Pro-версия требует подписки,

−кривые обучения: для новичков освоение Unity может потребовать времени.

Примеры игр, созданных на Unity,

−2D-игры: Hollow Knight, Cuphead, Ori and the Blind Forest,

−3D-игры: Monument Valley, Beat Saber, Escape from Tarkov,

−VR-игры: Superhot VR, Beat Saber.

Unity — это мощный и универсальный инструмент, который подходит как для начинающих разработчиков, так и для профессионалов. Его кроссплат-

форменность, поддержка 2D и 3D-графики, а также активное сообщество делают его одним из лучших выборов для создания игр.

1.4 История и развитие платформеров

2D-платформеры — это жанр видеоигр, который возник в начале 1980-х годов и стал одним из самых популярных и влиятельных жанров в игровой индустрии. Краткий обзор их истории и развития:

1. Ранние годы (1980-е)

Одной из первых 2D-платформенных игр считается "Donkey Kong" (1981), разработанная Nintendo. В ней игрок управляет персонажем, который должен преодолеть препятствия, чтобы спасти принцессу.

В 1985 году вышла "Super Mario Bros." от Nintendo, которая стала знаковой для жанра. Игра представила множество новых механик, таких как прыжки, сбор монет и использование различных силовых апгрейдов.

2. Золотая эра (конец 1980-х — начало 1990-х)

Платформеры стали основным жанром на консолях, таких как NES и Sega Genesis. Игры, такие как "Sonic the Hedgehog" (1991), предложили более высокую скорость и динамику.

Появились различные поджанры, включая "метроидванию" (например, "Metroid" и "Castlevania") и "платформеры с элементами головоломки" (например, "Bubsy").

3. Переход к 3D (середина 1990-х)

С появлением 3D-графики в конце 1990-х годов, таких как "Super Mario 64", жанр 2D-платформеров стал менее популярным. Однако многие классические игры продолжали оставаться актуальными.

4. Возрождение (2000-е)

В начале 2000-х годов инди-разработчики начали создавать 2D-платформеры, используя доступные инструменты и движки. Игры, такие как "Braid" (2008) и "Super Meat Boy" (2010), вернули интерес к жанру.

Многие инди-игры использовали пиксельную графику и механики, напоминающие классические платформеры, что привлекло как старых, так и новых игроков.

5. Современные 2D-платформеры (2010-е и позже)

Современные 2D-платформеры, такие как "Celeste" (2018) и "Hollow Knight" (2017), продолжают развивать жанр, добавляя новые механики, глубокие сюжеты и уникальные художественные стили.

Многие современные платформеры доступны на различных платформах, включая ПК, консоли и мобильные устройства, что делает их более доступными для широкой аудитории.

2D-платформеры прошли долгий путь с момента своего появления, и, несмотря на развитие технологий и появление новых жанров, они продолжают оставаться важной частью игровой культуры. Жанр продолжает эволюционировать, привнося новые идеи и механики, что делает его актуальным и интересным для игроков всех возрастов.

ГЛАВА 2. РАЗРАБОТКА ИГРОВОГО ПРИЛОЖЕНИЯ

2.1 Постановка задачи

Требуется разработать игровое приложение в 2D-пространстве на Unity с языком программирования C#.

Функционал проекта:

Проект представляет собой компьютерную платформерную (жанр Platformer) 2D-игру.

Прежде всего, проект должен отвечать следующим требованиям:

Игра должна обладать простым и понятным геймплеем.

Основная механика игры заключается в перемещении персонажа по уровню из точки А в точку В, взаимодействие с предметами, сражение с врагами.

Враги обитают в определенных местах и бродят, пока игрок не окажется у них в зоне видимости.

Ловушки находятся в определенных местах и ранят игрока, если тот соприкоснется с ними.

Должны присутствовать собираемые объекты.

У игрока определённое количество жизни. При потере всех жизней игрока, должно выводиться экран окончания игры, предлагающее либо начать сначала, либо вернуться в главное меню, либо выйти.

При достижении конца уровня, должно выводиться экран успешного прохождения уровня, предлагающее либо начать сначала, либо вернуться в главное меню, либо выйти.

При достижении чекпойнта (checkpoint – контрольная точка) и потере жизни игрока, игрок перемещается на месте последней контрольной точки.

При разработке игры, должны быть выполнены определенные задачи, а именно, разработаны:

- Меню;
- Графический интерфейс пользователя;

- Переходы между сценами;
- Боевая система.

2.2 Реализация создания игрового проекта

2.2.1 Описание структуры разрабатываемой игры

Как было описано ранее, структура игры на данном игровом движке представляет собой дерево узлов, соединяющие в себе различные сцены. Некоторые из них имеют скрипты, реализующие функционал игры. Для представления взаимодействия между сценами и механики ниже приведена блок-схема (Рисунок 1):

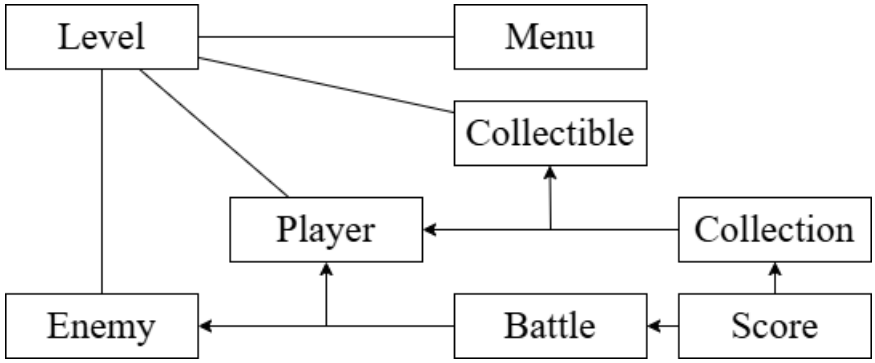

Рисунок 1 – Структура игры

После запуска игры, игровой персонаж находится в начале уровня. Есть доступ к меню по нажатию кнопки «Esc», в котором можно выйти из игры.

У персонажа должны быть следующие возможности:

- Перемещение (влево, вправо), управляемое клавишами «A» и «D»;
- Бег, реализованное клавишей «W»;
- Прыжок, реализованное клавишей «Правая кнопка мыши»;
- Комбинированная атака на земле и в воздухе, реализованные клавишей «Левая кнопка мыши».

2.2.2 Создание сцен для отдельных элементов игры

При создании нового проекта пользователь находится в первой сцене GameplayScene. К сцене добавляются игровые объекты, которые могут быть рисунками (спрайтами), слоями, персонажами, объектами уровня, музыкой и другими компонентами. В проекте может содержаться несколько сцен, будто это главное меню или сами уровни. Переход между ними осуществляется через написанные разработчиком скрипты.

Создание сцены главного меню. Эта сцена является самой начальной при запуске игры и включает в себя несколько опций.

Имеет титульное название игры сверху, сплошной бирюзовый задний фон и две опции: START – начать игру; QUIT – выйти из игры (Рисунок 2). Содержит скрипт ‘UIManager.cs’, отвечающий за функции опции после их нажатия. Данный скрипт представлен в приложении А.

Рисунок 2 – Главное меню

Создание сцены уровня. Эта сцена является самой ключевой и включает в себя несколько объектов и является самой масштабной.

Имеет «плитку» (тайлы), размещаемые на сцене; к некоторым из них добавлено столкновение. Содержит сцены игрока, собираемых объектов, врагов, ловушек (Рисунок 3).

Рисунок 3 – Сцена уровня

Создание интерфейсов. По всему экрану расположена HUD (Head-Up Display — индикация, предназначенная для просмотра без наклона головы), которая состоит из нескольких частей: количество очков, монет, жизней, бонусов, полоса с счётчиком здоровья игрока. При потере или восстановления здоровья игрока по экрану взлетит количество здоровья, которых от нас отняли или подобрали при аптечки (Рисунок 4). Содержит скрипт ‘HealthText.cs’, отвечающий за появление количества здоровья. Данный скрипт представлен в приложении Б.

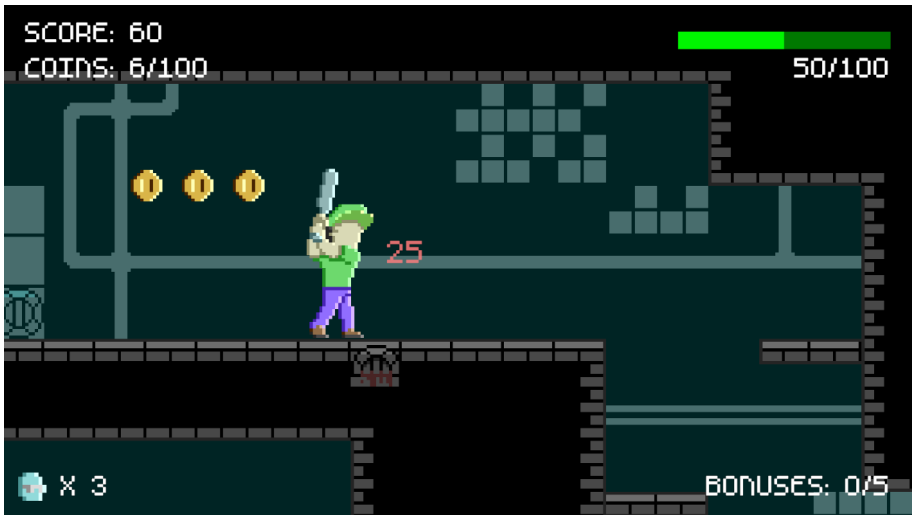

Рисунок 4 – HUD по всему экрану

Создание сцены паузы. При нажатии клавищи «Esc» сцена уровня замораживается и поверх него появляется сцена паузы.

Имеет название паузы сверху, сплошной жёлтый задний фон и три опции: RESUME – продолжить игру; MENU – главное меню; QUIT – выйти из игры (Рисунок 5).

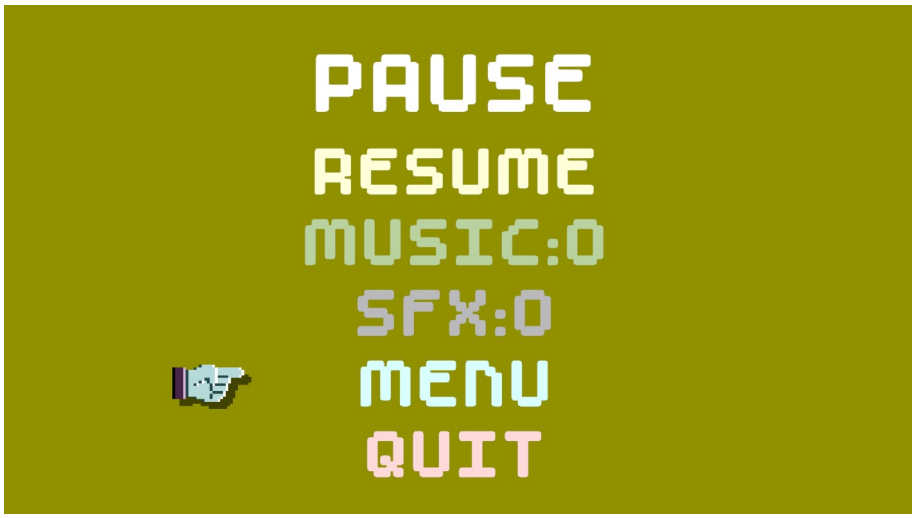

Рисунок 5 – Сцена паузы

Создание сцены окончания игры. После смерти игрока сцена уровня замораживается и поверх него появляется сцена окончания игры.

Имеет название окончания игры сверху, сплошной красный задний фон и три опции: RESTART – начать игру заново; MENU – главное меню; QUIT – выйти из игры (Рисунок 6).

Рисунок 6 – Сцена окончания игры

Создание сцены успешного прохождения игры. После достижения окончания уровня сцена уровня замораживается и поверх него появляется сцена успешного прохождения игры (Рисунок 7).

Имеет название “LEVEL CLEARED!” сверху, сплошной задний фон и три опции: RESTART – начать игру заново; MENU – главное меню; QUIT – выйти из игры (Рисунок 8).

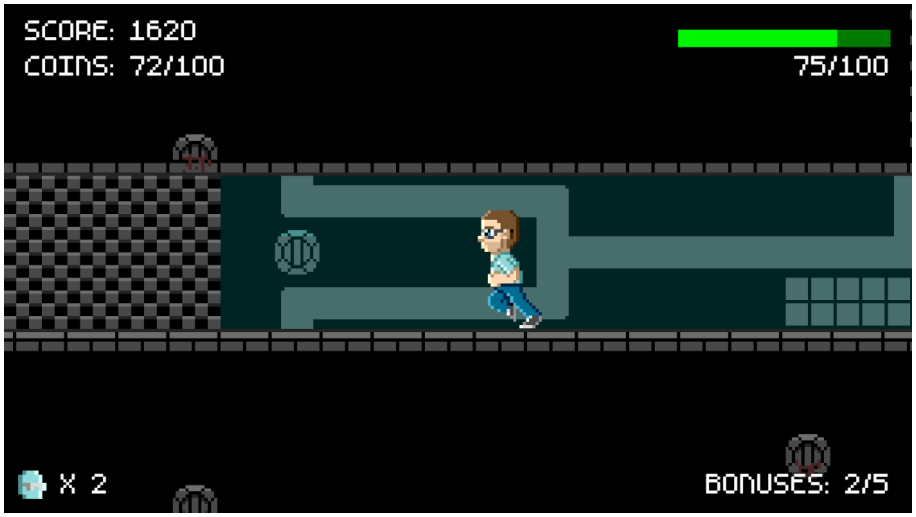

Рисунок 7 – Финишная комната в конце уровня

Рисунок 8 – Сцена успешного прохождения игры

2.2.3 Создание игровых механик игры

Создание управления. С помощью Rigidbody2D и Animator в Unity, наш игрок имеет динамическую физику, взаимодействие и соприкосновение с уровнем и объектами уровня. Игрок может ходить в обе стороны, бегать, прыгать, атаковать на суше и в воздухе и совершать комбинированные атаки. (Рисунок 9). Содержит скрипт ‘Attack.cs’, отвечающий за атаку. Данный скрипт представлен в приложении В.

Рисунок 9 – Прыжок игрока через пропасть

Создание камеры. Камера — отдельный объект в иерархии уровня, предназначенный отображать сам игровой процесс и следить за игроком в целом. Камера создана вместе с проектом. С помощью ассета Cinemachine камера плавно движется и останавливается в зависимости от действий игрока.

2.2.4 Создание собираемых предметов

Создание монет. Монеты — обычные собираемые объекты, предназначенные для увеличения очков и открытия дверей для сбора особых бонусов. Так всего в игре сто монет (Рисунок 10). Содержит скрипт ‘CoinManager.cs’, отвечающий за сбор монет и открытии дверей. Данный скрипт представлен в приложении Г.

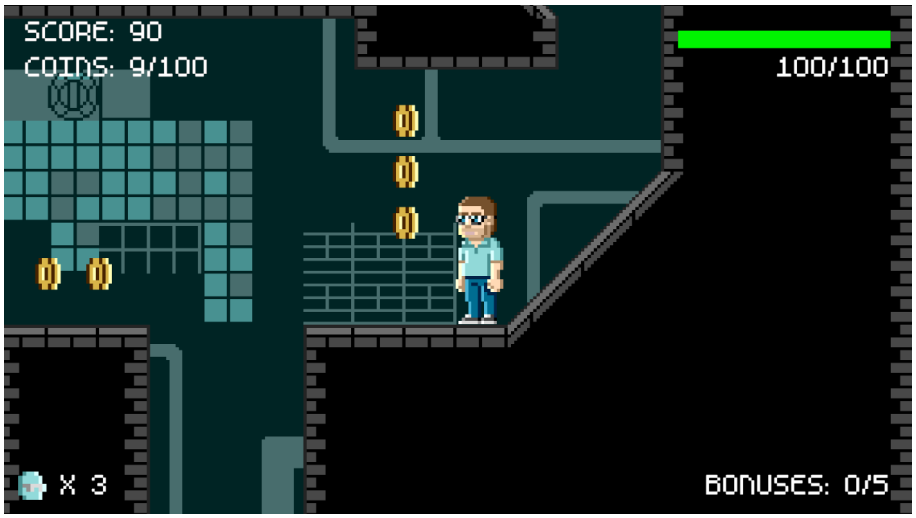

Рисунок 10 – Три монеты слева от игрока

Создание бонусов. При наличии определенного количества монет, дверь, блокирующая доступ к бонусу, исчезает и даёт игроку доступ к бонусу (Рисунок 11). Так всего в игре пять бонусов (Рисунок 12).

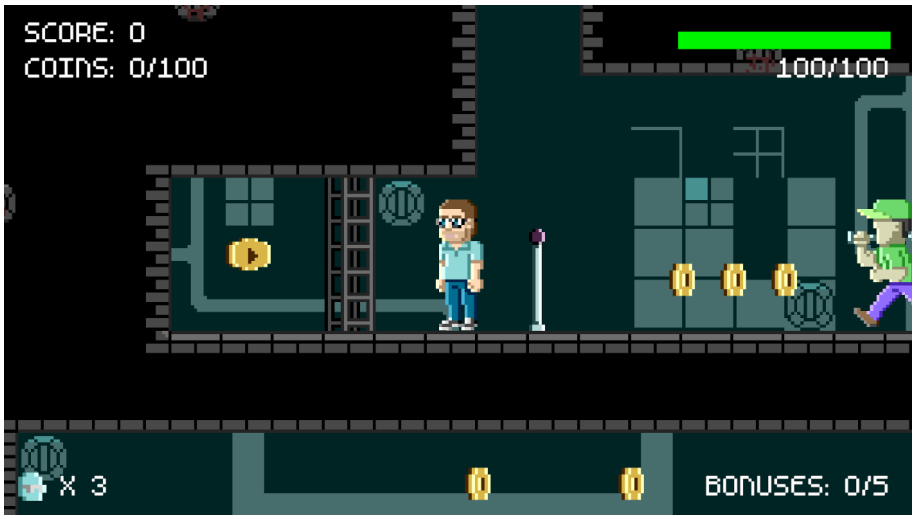

Рисунок 11 – Бонус, закрытый для игрока

Рисунок 12 – Бонус, открытый для игрока

Создание аптечек. Когда игрок потерял часть здоровья, ему надо восстанавливаться. Аптечки предназначены для лечения игрока. Если игрок полностью здоров, аптечки не подбираются (Рисунок 13).

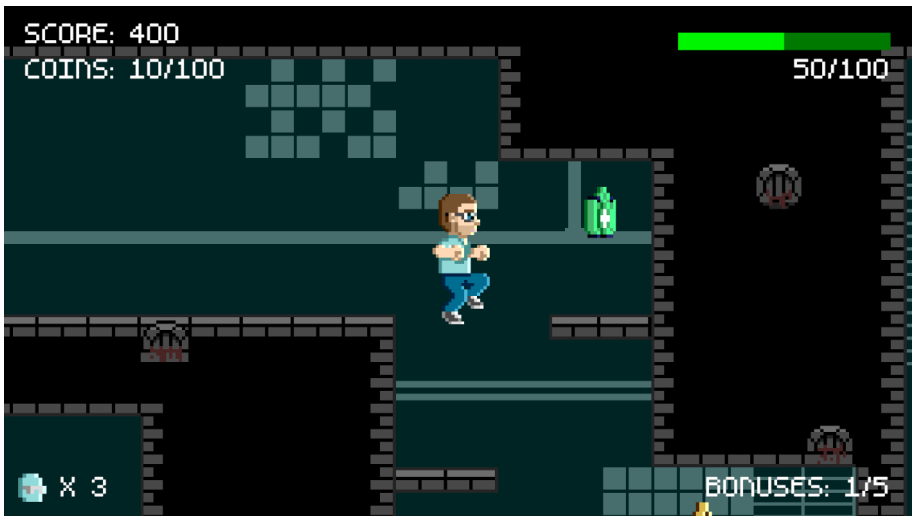

Рисунок 13 – Аптечка справа от игрока

Создание дополнительных жизней. У игрока изначально всего три жизни. При смерти одна жизнь теряется. Дополнительная жизнь представлена в виде голубого сердца и при подборе количество жизней увеличивается (Рисунок 14).

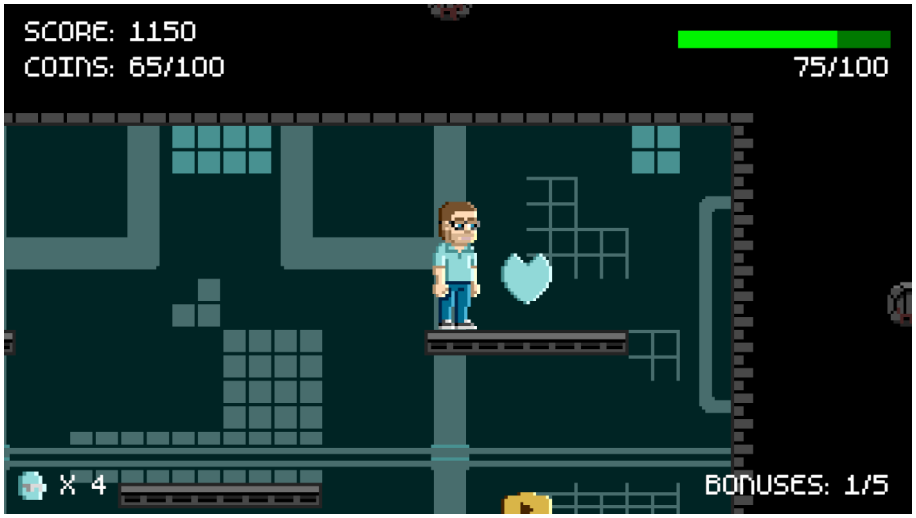

Рисунок 14 – Дополнительная жизнь справа от игрока

Создание контрольных точек. Контрольные точки предназначены для перемещения на более близкие расстояния от возможной смерти, чтобы не передвигаться с самого начала (Рисунок 15).

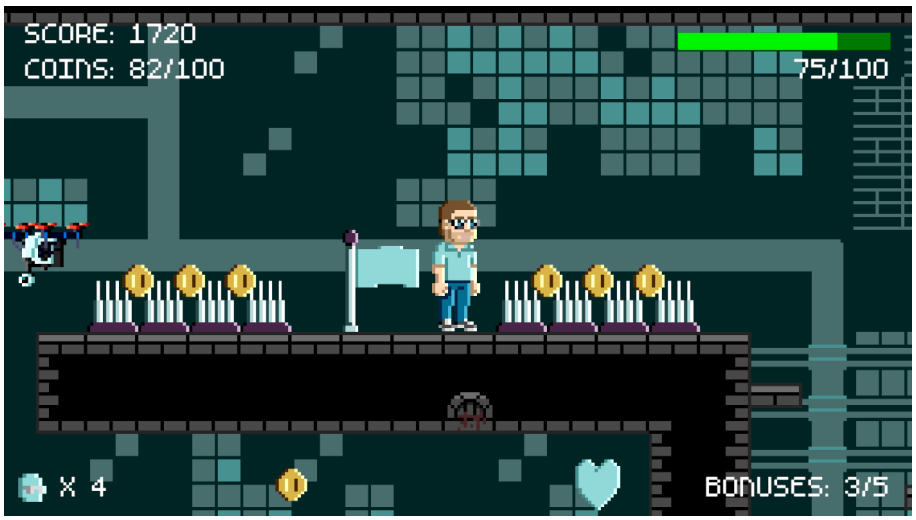

Рисунок 15 – Достигнутая контрольная точка слева от игрока

2.2.5 Создание ловушек и врагов

Создание ловушек. Всего в игре три ловушки — шипы, циркулярные пилы и пропасти (Рисунок 16). Шипы — простые ловушки, которые со временем опускают шипы и снова поднимают, нанося игроку урон. Циркулярные пилы в отличие от шипов передвигаются по назначенным координатам. Их нельзя уничтожить, как врагов. Пропасти сразу убивают игрока.

Рисунок 16 – Пять шипов и циркулярная пила

Создание врагов (Рисунок 17). Бегущий враг — мужчина с бейсбольной бите бежит по кругу. Когда игрок оказывается в зоне видимости врага, он начинает атаковать, размахивая битой. Дрон — враг летающий. Как и в случае с пилой, передвигается по заданным координатам. Он бьёт током игрока, если игрок окажется под дроном. Врагов можно уничтожить. Содержит скрипт ‘FlyingEnemy.cs’, отвечающий за логику дрона. Данный скрипт представлен в приложении Д.

Рисунок 17 – Бегущий и летающий враги

2.2.6 Создание системы смерти

Логика урона персонажам. При ударе игрок и враги теряют здоровье, попеременно исчезают и становятся неуязвимыми на секунду, чтобы выйти из зона дискомфорта. Они все умирают, если здоровье совсем закончилось (Рисунок 18). Содержит скрипт ‘Damageable.cs’, отвечающий за логику жизни и урона персонажей. Данный скрипт представлен в приложении Е.

Рисунок 18 – Игрок умирает, упав в пропасть

С помощью коллайдеров, которые отвечают за столкновения/взаимодействия объектов и стен, игрок не сможет провалиться сквозь пол уровня.

В игре также добавлены звуковые эффекты. Они были заимствованы с игры Super Mario Bros. Они воспроизводятся по заданным в скрипте методам. С помощью компонента AudioClip, который хранится в специальном скрипте SoundManager, звуки проигрывают в сцене, подобно тому, как динамик в реальном мире воспроизводит звук.

Создание скриптов необходимо для игры. AnimationStrings отвечает за хранение строк узлов, отвечающих за переход между анимациями персонажей; Attack — нанесение вреда персонажу; BonusManager — поведение бонусов после подбора игроком; CoinManager — поведение монет после подбора игроком; Damageable — здоровье персонажа и его уменьшение; DetectionZone — зона видимости врага; Enemy — логика врага; ExtraLifeManager — поведение дополнительных жизней после подбора игроком; FlyingEnemy — логика летающего врага (дрона); HealthBar — полоса здоровья; HealthPickup — поведение аптечки после подбора игроком; HealthText — текст отнятого здоровья; PlayerController — логика игрока и его управления; PlayerRespawn — возвращение игрока в последнюю контрольную точку после смерти; Saw — логика циркулярной пилы; ScoreManager — добавление очков к игре; SelectionArrow — стрелка опции в меню; SoundManager — звуковые эффекты в игре; TouchingDirections — логика прикосновения поверхности; UIManager — менеджер интерфейсов. Все скрипты в папке Scripts представлены в приложении Ж.

У каждого персонажа есть узлы: Animator — система, которая управляет анимациями. В каждом элементе аниматора можно добавить Behaviour – класс сценария, который работает с системой Unity Animator. Он позволяет определять действия для входа, выхода и обновления различных анимационных состояний.

Также была проведена работа по разработке и анимации спрайтов персонажей, объектов и уровня.

ЗАКЛЮЧЕНИЕ

Согласно последним статическим данным, игровая индустрия в нашей стране имеет положительную динамику роста и приносит разработчикам доходы в миллиардах рублей. Следовательно, развитие в сфере разработок компьютерных игр и приложений становится очень перспективным.

Целью данного дипломного проекта было разработать десктопное игровое приложение 2D-игры с помощью Unity.

В ходе выполнения проекта были исследованы понятия: компьютерная игра, десктопные приложения, 2D-игры.

Проанализированы популярные средства разработки 2D-игр. В ходе анализа, было проведено их сравнение и выбран наиболее удобный игровой движок. Выбор приоритетных средств разработки проходил по двум критериям: доступность и функциональность.

В процессе работы над проектом использовались такие технологии, как C# для программирования игровой логики, встроенные инструменты Unity для работы с графикой и анимацией, а также сторонние библиотеки для оптимизации и расширения функциональности. Результатом работы стал готовый к использованию 2D-платформер, который демонстрирует возможности Unity и может служить основой для дальнейшего развития проекта.

В ходе проектирования были выполнены следующие задачи:

- Выбран и изучен игровой движок Unity;
- Изучен встроенный язык программирования C#;
- Выполнена реализация игрового приложения, а именно, разработаны:
    - Меню;
    - Графический интерфейс пользователя;
    - Боевая система.

Разработанная игра соответствует поставленным задачам и заданным проектом требованиям. Иными словами, цель дипломного проекта достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Хокинг Джозеф. «Unity в дейтсвии» / Д. Хокинг – СПб.: Питер, 2023. - 336 с.

2. Бонд Джереми Гибсон. «Unity и C#. Геймдев от идеи до реализации» / Д. Гибсон Бонд – СПб.: Питер, 2020. - 928 с.

3. Денисов Дмитрий. «Разработка игры на Unity. С нуля до публикации» / Д. Денисов - Автор, 2021. - 177 с.

4. Корнилов Андрей. «Unity. Полное руководство» / А. Корнилов – СПб.: Наука и техника, 2021. - 496 с.

5. Гейг Майк. «Разработка игр на Unity за 24 урока. 4-е издание» / М. Гейг - М.: БОМБОРА, 2022. - 450 с.

ПРИЛОЖЕНИЯ

Приложение А. Листинг кода скрипта ‘UIManager.cs’, отвечающий за функции интерфейса

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

public class UIManager : MonoBehaviour
{
    public GameObject damageTextPrefab;
    public GameObject healthTextPrefab;
    public GameObject scoreTextPrefab;
    public GameObject extraLifeTextPrefab;

    [Header ("Game Over")]
    [SerializeField]
    private GameObject gameOverScreen;

    [Header("Results")]
    [SerializeField]
    private GameObject resultsScreen;

    [Header("Pause")]
    [SerializeField]
    private GameObject pauseScreen;

    public Canvas gameCanvas;

    private void Awake()
    {
        gameCanvas = FindObjectOfType<Canvas>();
        gameOverScreen.SetActive(false);
        resultsScreen.SetActive(false);
        pauseScreen.SetActive(false);
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (pauseScreen.activeInHierarchy)
                PauseGame(false);
            else
                PauseGame(true);
        }
    }

    private void OnEnable()
    {
        CharacterEvents.characterDamaged += CharacterTookDamage;
        CharacterEvents.characterHealed += CharacterHealed;
        CharacterEvents.characterScored += CharacterScored;
    }
```

```
    private void OnDisable()
    {
        CharacterEvents.characterDamaged -= CharacterTookDamage;
        CharacterEvents.characterHealed -= CharacterHealed;
        CharacterEvents.characterScored -= CharacterScored;
    }

    public void CharacterTookDamage(GameObject character, int damageReceived)
    {
        Vector3 spawnPosition = Camera.main.WorldToScreenPoint(character.transform.position);

                    TMP_Text tmpText = Instantiate(damageTextPrefab, spawnPosition, Quaternion.identity, gameCanvas.transform).GetComponent<TMP_Text>();

        tmpText.text = damageReceived.ToString();
    }

    public void CharacterHealed(GameObject character, int healthRestored)
    {
        Vector3 spawnPosition = Camera.main.WorldToScreenPoint(character.transform.position);

                    TMP_Text tmpText = Instantiate(healthTextPrefab, spawnPosition, Quaternion.identity, gameCanvas.transform).GetComponent<TMP_Text>();

        tmpText.text = healthRestored.ToString();
    }

    public void CharacterScored(GameObject character, int scoreReceived)
    {
        Vector3 spawnPosition = Camera.main.WorldToScreenPoint(character.transform.position);

                    TMP_Text tmpText = Instantiate(scoreTextPrefab, spawnPosition, Quaternion.identity, gameCanvas.transform).GetComponent<TMP_Text>();

        tmpText.text = scoreReceived.ToString();
    }


    #region Game Over
    public void GameOver()
    {
        gameOverScreen.SetActive(true);
    }

    public void RestartGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }

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

    public void QuitGame()
    {
        Application.Quit();
        #if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
```

```
        #endif
    }
    #endregion

    #region Results
    public void Results()
    {
        resultsScreen.SetActive(true);
    }
    #endregion

    #region Pause
    public void PauseGame(bool status)
    {
        pauseScreen.SetActive(status);
        Time.timeScale = System.Convert.ToInt32(!status);
    }

    public void ResumeGame()
    {
        SceneManager.LoadScene(1);
        Time.timeScale = System.Convert.ToInt32(true);
    }

    public void SoundVolume()
    {
        SoundManager.instance.ChangeSoundVolume(0.2f);
    }

    public void MusicVolume()
    {
        SoundManager.instance.ChangeMusicVolume(0.2f);
    }
    #endregion
}
```

Приложение Б. Листинг кода скрипта текста здоровья после урона ил восстановления ‘HealthText.cs’

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

public class HealthText : MonoBehaviour
{
    public Vector3 moveSpeed = new Vector3(0, 100, 0);
    public float timeToFade = 1f;

    RectTransform textTransform;
    TextMeshProUGUI textMeshPro;

    private float timeElapsed = 0f;
    private Color startColor;

    private void Awake()
    {
        textTransform = GetComponent<RectTransform>();
        textMeshPro = GetComponent<TextMeshProUGUI>();
        startColor = textMeshPro.color;
    }

    private void Update()
    {
        textTransform.position += moveSpeed * Time.deltaTime;

        timeElapsed += Time.deltaTime;

        if (timeElapsed < timeToFade)
        {
            float fadeAlpha = startColor.a * (1 - (timeElapsed / timeToFade));
            textMeshPro.color = new Color(startColor.r, startColor.g, startColor.b, fadeAlpha);
        }
        else
        {
            Destroy(gameObject);
        }
    }
}
```

Приложение В. Листинг кода скрипта атаки персонажа ‘Attack.cs’

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

public class Attack : MonoBehaviour
{
   public int attackDamage = 25;
   public Vector2 knockback = Vector2.zero;

   public void OnTriggerEnter2D(Collider2D collision)
   {
      Damageable damageable = collision.GetComponent<Damageable>();

      if (damageable != null)
      {
            Vector2 deliveredKnockback = transform.parent.localScale.x > 0 ? knockback : new Vector2(-knockback.x,
knockback.y);

         bool gotHit = damageable.Hit(attackDamage, deliveredKnockback);
      }
   }
}
```

Приложение Г. Листинг кода скрипта монет ‘CoinManager.cs’

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

public class CoinManager : MonoBehaviour
{
   public int coinCount;
   public TMP_Text coinText;
   public GameObject DoorOne;
   bool isDoorOneDestroyed;
   public GameObject DoorTwo;
   bool isDoorTwoDestroyed;
   public GameObject DoorThree;
   bool isDoorThreeDestroyed;
   public GameObject DoorFour;
   bool isDoorFourDestroyed;
   public GameObject DoorFive;
   bool isDoorFiveDestroyed;

   [SerializeField]
   public AudioClip doorSound;

   void Update()
   {
      coinText.text = "COINS: " + coinCount.ToString() + "/100";
      OnDoor();
   }

   public void OnDoor()
```

```
    {
        if (coinCount == 5 && !isDoorOneDestroyed)
            Destroy(DoorOne);

        else if (coinCount == 20 && !isDoorTwoDestroyed)
            Destroy(DoorTwo);

        else if (coinCount == 50 && !isDoorThreeDestroyed)
            Destroy(DoorThree);

        else if (coinCount == 80 && !isDoorFourDestroyed)
            Destroy(DoorFour);

        else if (coinCount == 100 && !isDoorFiveDestroyed)
            Destroy(DoorFive);
    }
}
```

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

public class FlyingEnemy : MonoBehaviour
{
    public float flightSpeed = 2f;
    public float waypointReachedDistance = 0.1f;
    public DetectionZone shockDetectionZone;
    public Collider2D deathCollider;
    public List<Transform> waypoints;

    Animator animator;
    Rigidbody2D rb;
    Damageable damageable;

    public ScoreManager sm;

    [SerializeField]
    public AudioClip hitSound;

    [SerializeField]
    public AudioClip deathSound;

    Transform nextWaypoint;
    int waypointNum = 0;

    public bool _hasTarget = false;

    public bool HasTarget
    {
        get
        {
            return _hasTarget;
        }
        private set
        {
            _hasTarget = value;
            animator.SetBool(AnimationStrings.hasTarget, value);
        }
    }

    public bool CanMove
    {
        get
        {
            return animator.GetBool(AnimationStrings.canMove);
        }
    }

    private void Awake()
    {
        animator = GetComponent<Animator>();
        rb = GetComponent<Rigidbody2D>();
        damageable = GetComponent<Damageable>();
```

```
}

private void Start()
{
    nextWaypoint = waypoints[waypointNum];
}

private void OnEnable()
{
    damageable.damageableDeath.AddListener(OnDeath);
}

void Update()
{
    HasTarget = shockDetectionZone.detectedColliders.Count > 0;
}

private void FixedUpdate()
{
    if (damageable.IsAlive)
    {
        if (CanMove)
        {
            Flight();
        }
        else
        {
            rb.velocity = Vector3.zero;
        }
    }
}

private void Flight()
{
    Vector2 directionToWaypoint = (nextWaypoint.position - transform.position).normalized;

    float distance = Vector2.Distance(nextWaypoint.position, transform.position);

    rb.velocity = directionToWaypoint * flightSpeed;
    UpdateDirection();

    if (distance <= waypointReachedDistance)
    {
        waypointNum++;

        if (waypointNum >= waypoints.Count)
        {
            waypointNum = 0;
        }

        nextWaypoint = waypoints[waypointNum];
    }
}

private void UpdateDirection()
{
    Vector3 locScale = transform.localScale;

    if (transform.localScale.x > 0)
```

```
        {
            if (rb.velocity.x < 0)
            {
                transform.localScale = new Vector3(-1 * locScale.x, locScale.y, locScale.z);
            }
        }
        else
        {
            if (rb.velocity.x > 0)
            {
                transform.localScale = new Vector3(-1 * locScale.x, locScale.y, locScale.z);
            }
        }
    }

    public void OnHit()
    {
        if (damageable.Health > 0)
            SoundManager.instance.PlaySound(hitSound);
    }

    public void OnDeath()
    {
        rb.gravityScale = 2f;
        rb.velocity = new Vector2(0, rb.velocity.y);
        deathCollider.enabled = true;

        if (damageable.Health <= 0)
        {
            sm.scoreCount += 100;
            SoundManager.instance.PlaySound(deathSound);
        }
    }
}
```

Приложение Е. Листинг кода скрипта здоровья персонажей и их урона ‘Damageable.cs’

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

public class Damageable : MonoBehaviour
{
    public UnityEvent<int, Vector2> damageableHit;
    public UnityEvent damageableDeath;
    public UnityEvent<int, int> healthChanged;

    Animator animator;
    Transform currentCheckpoint;

    ExtraLifeManager lm;

    [SerializeField]
    private int _maxHealth = 100;

    public int MaxHealth
    {
        get
        {
            return _maxHealth;
        }
        set
        {
            _maxHealth = value;
        }
    }

    [SerializeField] private int _health = 100;

    public int Health
    {
        get
        {
            return _health;
        }
        set
        {
            _health = value;
            healthChanged?.Invoke(_health, MaxHealth);

            if (_health <= 0)
            {
                IsAlive = false;
                _health = 0;
            }
        }
    }
```

```
[SerializeField] private bool _isAlive = true;

[SerializeField] private bool isInvincible = false;

private float timeSinceHit = 0;
public float invincibilityTime = 3f;

[SerializeField] private int numberOfFlashes = 3;
private SpriteRenderer sr;

public bool IsAlive
{
   get
   {
      return _isAlive;
   }
   set
   {
      _isAlive = value;
      animator.SetBool(AnimationStrings.isAlive, value);

      if (value == false)
         damageableDeath.Invoke();
   }
}

public bool LockVelocity
{
   get
   {
      return animator.GetBool(AnimationStrings.lockVelocity);
   }
   set
   {
      animator.SetBool(AnimationStrings.lockVelocity, value);
   }
}

private void Awake()
{
   animator = GetComponent<Animator>();
   sr = GetComponent<SpriteRenderer>();
}

private void Update()
{
   if (isInvincible)
   {
      if (timeSinceHit > invincibilityTime)
      {
         isInvincible = false;
         timeSinceHit = 0;
      }

      timeSinceHit += Time.deltaTime;
```

```
    }

}

public bool Hit(int damage, Vector2 knockback)
{
    if (IsAlive && !isInvincible)
    {
        Health -= damage;
        isInvincible = true;

        animator.SetTrigger(AnimationStrings.hitTrigger);
        LockVelocity = true;
        damageableHit?.Invoke(damage, knockback);
        CharacterEvents.characterDamaged.Invoke(gameObject, damage);
        StartCoroutine(Invunerability());

        return true;
    }

    return false;
}

public bool Heal(int healthRestore)
{
    if (IsAlive && Health < MaxHealth)
    {
        int maxHeal = Mathf.Max(MaxHealth - Health, 0);
        int actualHeal = Mathf.Min(maxHeal, healthRestore);
        Health += actualHeal;
        CharacterEvents.characterHealed(gameObject, actualHeal);
        return true;
    }

    return false;
}

public void Respawn()
{
    IsAlive = true;
    Heal(MaxHealth);

    animator.Play("Base Layer.GroundStates.player_idle");
    StartCoroutine(Invunerability());
    isInvincible = true;
}

private IEnumerator Invunerability()
{
    Physics2D.IgnoreLayerCollision(10, 11, true);

    for (int i = 0; i < numberOfFlashes; i++)
    {
        sr.color = new Color(1, 1, 1, 0f);
        yield return new WaitForSeconds(invincibilityTime / (numberOfFlashes * 4));
```

```
            sr.color = Color.white;
            yield return new WaitForSeconds(invincibilityTime / (numberOfFlashes * 4));
        }

        Physics2D.IgnoreLayerCollision(10, 11, false);
    }
}
```

Приложение Ж. Папка со скриптами