Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ
Институт информационных технологий

Факультет компьютерных технологий

Кафедра информационных систем и технологий

*К защите допустить*:

Заведующий кафедрой ИСиТ

_____А.И. Парамонов

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к дипломному проекту
на тему

# ИГРОВОЕ ПРОГРАММНОЕ СРЕДСТВО С ПРОЦЕДУРНОЙ ГЕНЕРАЦИЕЙ УРОВНЕЙ НА ПЛАТФОРМЕ UNITY

БГУИР ДП 1-40 01 01 01 010 ПЗ

| | |
|---|---|
| Студент | Д.И. Бабич |
| Руководитель | А.Л. Сицко |
| Консультанты: | |
| *от кафедры* | А.Л. Сицко |
| *по экономической части* | В.Г. Горовой |
| Нормоконтролер | А.С. Шелягович |
| Рецензент | |

Минск 2025

# Р Е Ф Е Р А Т

ИГРОВОЕ ПРОГРАММНОЕ СРЕДСТВО С ПРОЦЕДУРНОЙ ГЕНЕРАЦИЕЙ УРОВНЕЙ НА ПЛАТФОРМЕ UNITY: дипломный проект / Д.И. Бабич. – Минск: БГУИР, 2025, – п.з. – 81 с., чертежей (плакатов) – 6 л. формата А1.

Объектом проектирования является игровое программное средство с процедурной генерацией уровней.

Целью данного дипломного проекта является разработка программного средства, которое позволит создать бесконечную игровую среду с увлекательным геймплеем, динамичным взаимодействием с объектами и адаптивным игровым процессом, способным удерживать интерес пользователя.

Для решения данной задачи был разработан комплексный подход, который включает в себя: алгоритм процедурной генерации уровней, систему управления игровыми объектами и их взаимодействием, поведение врагов на основе состояний, а также игровую механику с элементами исследования.

В процессе работы над проектом были изучены современные подходы к разработке игровых программных средств, проведен анализ популярных игровых движков и инструментов, таких как Unity, а также рассмотрены успешные примеры платформеров с похожей механикой. Была разработана архитектура программного средства, которая позволяет эффективно управлять сценами, игровыми объектами и переходами между уровнями.

В процессе тестирования были проведены функциональные и производительные испытания игры, которые подтвердили корректность работы алгоритмов, стабильность генерации уровней и отсутствие критических ошибок при длительном игровом процессе.

В разделе технико-экономического обоснования были рассчитаны затраты на разработку игры и обоснована ее рентабельность. Проведенные расчеты показали экономическую целесообразность создания данного программного средства, особенно для рынка инди-игр, где востребованы проекты с уникальным игровым процессом.

В целом разработанное игровое программное средство позволяет пользователю получить увлекательный и неповторяющийся игровой опыт, благодаря процедурной генерации уровней и интерактивному окружению. Игровое решение сочетает в себе оптимизированную архитектуру, удобный интерфейс и динамичный геймплей, что способствует повышению интереса игроков и удержанию их внимания на протяжении длительного времени.

Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ
Институт информационных технологий

Факультет КТ Кафедра ИСиТ
Специальность 1-40 01 01 Специализация 01

УТВЕРЖДАЮ
А.И.Парамонов
« 24 » сентября 2024 г.

ЗАДАНИЕ
по дипломному проекту студента

**Бабич Дмитрий Иванович**
(фамилия, имя, отчество)

1. Тема проекта: **Игровое программное средство с процедурной генерацией уровней на платформе Unity**

утверждена приказом по университету от « 24 » сентября 2024 г. № 112-и

2. Срок сдачи студентом законченной работы 23 декабря 2024 года

3. Исходные данные к проекту Тип операционной системы – ОС Windows 7 и выше; Язык программирования – C#; компонентно-ориентированная архитектура. Функции: процедурная генерация уровней, управление персонажем, механика стрельбы, система анимаций, звуковая система, сохранение рекордного счёта, пользовательские настройки. Назначение разработки: развлечение игроков через интерактивные игровые процессы и интересные механики.

4. Содержание пояснительной записки (перечень подлежащих разработке вопросов):

Введение
1 Анализ предметной области
2 Моделирование предметной области
3 Проектирование и разработка программного средства
4 Тестирование программного средства
5 Руководство по эксплуатации программного средства
6 Технико-экономическое обоснование разработки игрового программного средства с процедурной генерацией уровней на платформе Unity
Заключение
Список использованных источников
Приложение А. Листинг программного средства

5. Перечень графического материала (с точным указанием наименования) и обозначения вида и типа материала):

Диаграмма вариантов использования. Плакат – формат А1, лист 1.

Диаграмма состояний. Плакат – формат А1, лист 1.

Диаграмма классов. Плакат – формат А1, лист 1.

Процедурная генерация уровня. Схема алгоритма – формат А1, лист 1.

Поведение врага. Схема алгоритма – формат А1, лист 1.

Взаимодействие персонажа с объектами игры. Схема алгоритма – формат А1, лист 1.

6. Содержание задания по технико-экономическому обоснованию

Экономическое обоснование разработки и реализации программного продукта на массовом рынке

Консультанты по дипломному проекту (с указанием разделов, по которому они консультируют):

Сицко А.Л. – разделы 1-5;

Горовой В.Г. – раздел 6 (технико-экономическое обоснование);

Шелягович А.С. – нормоконтроль.

ПРИМЕРНЫЙ КАЛЕНДАРНЫЙ ГРАФИК ВЫПОЛНЕНИЯ ДИПЛОМНОГО ПРОЕКТА

| Наименование этапов дипломного проекта | Объём готовности проекта в % | Срок выполнения этапа | Примечание |
|---|---|---|---|
| Первая опроцентовка (введение, разделы 1-3) | 30% | 16.09.24 – 25.10.24 | Консультант от кафедры |
| Вторая опроцентовка (разделы 3-5) | 60% | 26.10.24 – 30.11.24 | Руководитель проекта |
| Третья опроцентовка (разделы 5-6, заключение, список использованных источников) | 90% | 01.12.24 – 18.12.24 | Руководитель проекта |
| Консультации по оформлению графического материала и пояснительной записки, нормоконтроль | – | С 01.10.24 (согласно графику) | Нормо-контролёр |
| Итоговая проверка готовности дипломного проекта на заседании рабочей комиссии кафедры ИСиТ и допуск к защите в ГЭК | 100% | С 23.12.24 (согласно графику) | Председатель рабочей комиссии |
| Рецензирование дипломного проекта | 100% | С 26.12.24 (согласно распоряжению) | Рецензент |
| Защита дипломного проекта | 100% | С 18.01.25 (согласно приказу) | ГЭК |

Дата выдачи задания 16 сентября 2024 г. Руководитель _____ / А.Л. Сицко /

Задание принял к исполнению _____ / Д.И. Бабич /

**СОДЕРЖАНИЕ**

# ВВЕДЕНИЕ

Проект дипломной работы посвящен разработке игрового программного средства с процедурной генерацией уровней. Основная идея проекта заключается в создании динамичного игрового процесса, где уровни генерируются случайным образом при каждом запуске, предоставляя игрокам уникальные испытания. Особое внимание уделяется реализации механик взаимодействия с объектами окружения и поведения врагов, что добавляет тактический элемент в игру.

Актуальность темы заключается в возрастающей популярности игр с процедурной генерацией уровней, которые обеспечивают высокий уровень реиграбельности и разнообразия игрового опыта. Такие игры привлекают игроков за счет возможности создавать новые вызовы и интересные ситуации при каждом прохождении.

В процессе работы над проектом проведено изучение современных подходов к процедурной генерации, а также анализ методов разработки интерактивного окружения в играх. На этапе преддипломной практики был создан прототип игры, который включал основные геймплейные элементы, такие как случайная генерация уровней, взаимодействие персонажа с объектами и врагами, а также базовые механики стрельбы и разрушения сундуков. Это позволило протестировать выбранные технологии и подтвердить их применимость для реализации проекта.

Целью данной работы является разработка игрового программного средства, включающего основные элементы игрового процесса, такие как процедурная генерация уровней, механики взаимодействия объектов, поведение врагов и система подсчета очков.

Для достижения цели были поставлены следующие задачи:

– проведение анализа предметной области и существующих решений;
– определение требований к программному средству;
– проектирование архитектуры игры и её компонентов;
– разработка алгоритмов процедурной генерации уровней и поведения игровых объектов;
– тестирование реализованных механик;
– создание руководства пользователя.

Результатом работы является полнофункциональное игровое программное средство с процедурной генерацией, которая предлагает пользователю уникальные уровни, динамичное взаимодействие с окружением и высокую реиграбельность.

Дипломный проект выполнен самостоятельно, проверен в системе «Антиплагиат». Процент оригинальности составляет 85,78% и соответствует норме, установленной кафедрой. Цитирования обозначены ссылками на публикации, указанные в «Списке использованных источников». [1]

# 1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

## 1.1 Описание предметной области

Разработка игрового программного средства с процедурной генерацией уровней представляет собой создание гибкого, динамичного и увлекательного игрового мира в жанре платформер, который меняется при каждом новом запуске, предоставляя игрокам уникальные игровые ситуации и возможности для исследования.

Платформеры являются одним из самых популярных жанров в игровой индустрии благодаря сочетанию доступного управления, интуитивного игрового процесса и разнообразия механик. Этот жанр включает игры, в которых игрок управляет персонажем, перемещающимся по уровням, полным платформ, препятствий, врагов и интерактивных объектов. Главная цель таких игр заключается в преодолении сложностей, исследовании игрового пространства и достижении определённых целей, таких как сбор ресурсов, победа над врагами или завершение сюжетной линии.

Платформеры отличаются высокой гибкостью в дизайне уровней, что делает их привлекательными как для игроков, так и для разработчиков. Возможность интеграции процедурной генерации уровней — одна из современных тенденций в этом жанре. Процедурная генерация позволяет создавать уникальные карты при каждом запуске игры. Вместо фиксированных схем уровни генерируются алгоритмически, что гарантирует уникальность и разнообразие игрового процесса. Это особенно важно для игроков, стремящихся к новым испытаниям и неизведанным сценариям, а также для разработчиков, которые могут сосредоточиться на проектировании алгоритмов, избегая трудоёмкой ручной работы по созданию множества уровней.

Процедурная генерация в платформерах включает в себя такие элементы, как размещение платформ, врагов, бонусов и препятствий. Однако важно соблюдать баланс, чтобы создаваемые уровни были не только разнообразными, но и проходимыми, логичными и увлекательными. Это требует использования алгоритмов, учитывающих сложность, структуру уровней и возможности персонажа. В рассматриваемом проекте процедурная генерация осуществляется путём комбинирования заранее созданных сегментов уровней, что обеспечивает сочетание случайности и продуманной логики.

Интерактивное окружение — ещё одна важная составляющая платформеров. Оно делает игровой процесс более глубоким и увлекательным, позволяя игрокам активно взаимодействовать с объектами мира. Здесь эта механика представлена через возможность взаимодействия с сундуками. Сундуки выступают в роли интерактивных объектов, которые игрок может находить и открывать для получения ценных ресурсов. Такая система

стимулирует исследование уровней, добавляя элемент тактики в игровой процесс. Хотя взаимодействие с окружением ограничено, это позволяет игрокам ощутить, что их действия имеют значение, а также придаёт игре динамичность.

Важным аспектом платформеров является обеспечение доступного, но в то же время насыщенного игрового процесса. Интуитивное управление персонажем, плавные анимации и логичная структура уровней играют ключевую роль в удержании интереса аудитории. Для создания качественного опыта в проекте реализованы базовые механики управления: движение, прыжки, атаки и взаимодействие с объектами. Эти элементы интегрированы таким образом, чтобы геймплей оставался простым в освоении, но достаточным для создания увлекательного игрового процесса.

Целевая аудитория платформеров охватывает широкий круг игроков, от новичков до опытных геймеров. Благодаря своей универсальности, игры этого жанра подходят для разных возрастных групп и игровых предпочтений. Простота освоения механик и богатство возможностей для экспериментов делают платформеры привлекательными как для случайных игроков, так и для тех, кто стремится к более глубокому погружению в игровой процесс.

Таким образом, предметная область разрабатываемого программного средства охватывает создание платформера, который сочетает в себе элементы процедурной генерации уровней, интуитивное управление и интерактивные элементы окружения. Такой подход обеспечивает уникальность каждого игрового сеанса, поддерживает интерес аудитории за счёт разнообразия и предоставляет игрокам возможность тактического подхода к прохождению уровней. Это делает платформер не только увлекательным, но и способным удовлетворить ожидания широкой аудитории.

## 1.2 Обоснование актуальности разработки ПС

Актуальность разработки игрового программного средства с процедурной генерацией уровней обусловлена целым рядом важных факторов, которые делают такой проект перспективным и востребованным на современном игровом рынке. В наше время игровая индустрия активно развивается, стремясь предложить пользователям свежие и уникальные впечатления, которые будут захватывать их внимание и удерживать интерес на длительный срок. Одной из ключевых технологий, позволяющих достичь таких целей, является процедурная генерация контента. Она позволяет создавать новые уровни при каждом запуске игры, что обеспечивает значительное повышение реиграбельности и интереса к проекту, ведь игроки получают новые и неожиданные сценарии без необходимости ждать обновлений или дополнений. Это особенно важно для современных пользователей, которые ценят разнообразие и динамику игрового процесса.

Процедурная генерация становится востребованной и популярной технологией, поскольку она позволяет создавать уникальные игровые миры, наполненные разнообразными ландшафтами, препятствиями и элементами окружения. Платформеры с подобным подходом предлагают игрокам непредсказуемые условия, делая каждый игровой сеанс уникальным вызовом. Это подогревает интерес к игре, заставляя возвращаться к ней снова и снова в ожидании новых испытаний и возможностей. Особенно ярко актуальность процедурной генерации проявляется в играх жанра роглайк, которые выделяются высокой степенью случайности и непредсказуемости. Подобные проекты позволяют пользователям получать опыт, основанный на случайных сочетаниях объектов и препятствий, что способствует улучшению игровых навыков и развитию стратегического мышления. Игроки вынуждены адаптироваться к постоянно меняющимся условиям, разрабатывать новые тактики и стратегии, что делает игровой процесс глубже и интереснее.

Процедурная генерация уровней не только привлекает внимание геймеров, но и имеет значительное значение для оптимизации разработки. В традиционных играх создание каждого уровня требует ручной работы, что увеличивает трудозатраты и временные расходы. Процедурные методы позволяют генерировать контент автоматически на основе заданных параметров и алгоритмов, что значительно упрощает процесс разработки и снижает потребность в создании большого числа уровней вручную. Это даёт разработчикам возможность сосредоточиться на других аспектах игры, таких как улучшение механик, добавление новых игровых элементов и проработка визуальных эффектов. Такой подход позволяет даже небольшим инди-студиям конкурировать с более крупными проектами и предлагать пользователям уникальные игровые решения, сохраняя при этом высокий уровень качества.

Актуальность разработки платформеров с процедурной генерацией также подтверждается растущим интересом к инди-играм, которые часто являются источником инновационных механик и оригинальных идей. Современные игроки всё чаще ищут проекты, которые предлагают нестандартный подход и уникальные игровые механики. Такие игры часто отличаются свежестью и креативностью, что помогает им выделяться на фоне массовых, коммерчески ориентированных проектов. В этом контексте дипломный проект с процедурной генерацией уровней идеально соответствует актуальным трендам. Он предлагает игрокам уникальный набор возможностей, позволяя экспериментировать, исследовать и влиять на игровой процесс, что делает каждую игровую сессию по-настоящему особенной и запоминающейся.

Создание подобных игр способствует популяризации жанра и демонстрирует новые возможности современных технологий. Например, внедрение разрушаемых объектов, физики взаимодействия объектов и процедурной генерации делает игру более сложной и интересной с точки

зрения программирования, но при этом обеспечивает увлекательный и многослойный игровой опыт. В таких играх игроки могут наблюдать реальные последствия своих действий, что делает геймплей более глубоким и интерактивным.

Кроме того, процедурная генерация и механики разрушения объектов позволяют создавать более динамичные и адаптивные игровые миры. В таких играх игроки могут чувствовать, что мир реагирует на их действия, предоставляя новые возможности для исследования и стратегического подхода. Процедурные методы позволяют реализовать неожиданные сценарии, когда каждое действие игрока может привести к новым последствиям и открытиям. Это также помогает увеличить жизненный цикл игры, так как пользователи постоянно получают новый опыт, исследуя созданные процедурными алгоритмами миры и открывая новые возможности. Таким образом, такие игры могут оставаться актуальными и интересными для игроков гораздо дольше, чем проекты с фиксированными уровнями и заранее предсказуемым контентом.

Таким образом, разработка игрового программного средства с процедурной генерацией уровней представляет собой перспективное направление, отвечающее современным запросам игровой аудитории. Такой проект позволяет использовать передовые технологии для создания уникальных и адаптивных игровых миров, которые способны увлечь пользователей на длительный срок. Он отвечает требованиям современного рынка, где ключевую роль играет реиграбельность, креативность и возможность самовыражения через взаимодействие с игрой. Благодаря своей уникальности и высокой реиграбельности проект имеет потенциал стать востребованным продуктом, который сможет привлечь внимание широкой аудитории и занять достойное место на игровом рынке.

### 1.3 Анализ методов, способов, подходов, методик и технологий

Процедурная генерация в игровой индустрии — это метод создания контента с использованием алгоритмов и правил, что позволяет разработчикам генерировать уровни, объекты или другие элементы игрового мира без необходимости ручного проектирования. В платформерах процедурная генерация применяется для формирования уникальных уровней, которые изменяются при каждом запуске, обеспечивая разнообразие и реиграбельность.

Основные подходы к процедурной генерации:

- генерация на основе клеточных автоматов;
- генерация на основе графов;
- модульный подход;
- алгоритмы на основе шума (Perlin Noise, Simplex Noise);
- алгоритмы с использованием эволюционного подхода.

Первый метод основывается на использовании двумерных решёток, где каждая клетка может находиться в одном из нескольких состояний, например, "проходимая" или "непроходимая". Правила перехода клеток из одного состояния в другое задаются алгоритмом, что позволяет формировать уровни с логичной структурой. Клеточные автоматы хорошо подходят для создания пещер, подземелий или замкнутых пространств. Однако их применение ограничено необходимостью постобработки, чтобы обеспечить наличие связных путей для игрока.

Во втором подходе уровни представляются в виде набора узлов и соединений между ними. Каждый узел соответствует отдельной комнате, а соединения — путям между комнатами. Генерация производится путём создания графа, который удовлетворяет заданным правилам, например, наличию начальной и конечной точек, определённого числа врагов и бонусов. Этот метод популярен для игр, требующих высокой структурированности уровней, таких как роглайки или платформеры. Графовый подход позволяет легко контролировать сложность уровней и их логическую связанность.

Модульный метод предполагает использование заранее созданных кусочков уровней (сегментов), которые комбинируются случайным образом. Каждый модуль представляет собой фрагмент уровня с определённой функциональностью, например, набор платформ, врагов или препятствий. Алгоритм процедурной генерации отвечает за последовательное соединение этих модулей с учётом правил совместимости. Преимуществом данного метода является контроль над качеством каждого сегмента и относительно низкая сложность реализации. Однако такая система может ограничивать разнообразие, если число модулей невелико.

Генерация шума применяется для создания уровней с более плавными переходами между элементами. Например, шум используется для определения высоты платформ или распределения объектов. Такие алгоритмы хорошо подходят для игр с элементами случайного ландшафта, но требуют дополнительных настроек, чтобы результат был логичным и играбельным.

Эволюционные алгоритмы создают уровни путём "естественного отбора". Несколько вариантов генерируются случайным образом, затем оцениваются по заданным критериям (например, сложность, играбельность), и лучшие из них используются для создания новых уровней. Этот метод позволяет добиться высокого качества генерации, но требует значительных вычислительных ресурсов и времени на настройку.

Преимущества и недостатки методов:

– клеточные автоматы и алгоритмы шума просты в реализации, но ограничены в создании сложных структур;

– графовый и модульный подходы обеспечивают высокую логичность и структурированность уровней, однако могут быть менее вариативными;

– эволюционные алгоритмы способны создавать высококачественные уровни, но их использование может быть слишком затратным для небольших

проектов.

Для разработки игрового программного средства, описанного в данном проекте, был выбран модульный подход, сочетающий элементы графовой структуры. Уровни состоят из заранее спроектированных сегментов, которые соединяются случайным образом. Такой метод позволяет:

– обеспечить логическую связность уровней;

– сохранить контроль над качеством сегментов;

– ускорить процесс разработки за счёт использования готовых компонентов.

Этот выбор обоснован потребностью в создании уровней, которые сочетают в себе разнообразие и удобство прохождения, а также возможностью легко модифицировать и расширять набор сегментов для увеличения реиграбельности.

Для реализации процедурной генерации использованы следующие технологии:

– язык программирования C# выбран для интеграции с игровым движком Unity, предоставляющим инструменты для работы с модульным подходом;

– игровой движок Unity поддерживает возможность динамической загрузки сегментов уровней, что упрощает реализацию генерации;

– редактор Unity Tilemap использован для проектирования сегментов, позволяя легко визуализировать и корректировать их.

Процедурная генерация уровней является ключевым элементом разрабатываемого программного средства. Модульный подход обеспечивает баланс между качеством и вариативностью, позволяя создавать уникальные уровни при каждом запуске игры. Такой выбор подхода и технологий соответствует задачам проекта, обеспечивая высокую реиграбельность и разнообразие игрового процесса. [2]

### 1.4 Анализ существующих аналогов и прототипов

На рынке видеоигр существует несколько проектов, которые можно сопоставить с разрабатываемым игровым программным средством с процедурной генерацией уровней. Каждый из этих аналогов имеет свои сильные стороны и недостатки, которые важно учитывать при проектировании, чтобы предложить уникальный игровой опыт.

Одним из самых известных примеров 2D платформеров с процедурной генерацией уровней является Spelunky. Игра предлагает игрокам исследовать пещеры, каждый раз создаваемые случайным образом, что делает каждую попытку уникальной. Основное достоинство игры — это сочетание процедурной генерации и сложного, но увлекательного геймплея, где игроки сталкиваются с разнообразными ловушками, врагами и тайниками. Spelunky

обладает высокой реиграбельностью за счёт случайного размещения объектов и элементов окружения.

Рисунок 1.1 – Игра Spelunky

Тем не менее, одной из слабых сторон игры является ограниченная возможность взаимодействия с окружением. Хотя игроки могут разрушать стены и подрывать объекты, физика разрушений довольно упрощена, что не позволяет создавать сложные тактические ситуации. Также в игре отсутствует адаптивность сложности, что может отпугнуть менее опытных игроков.

Terraria — популярный 2D экшен-песочница, где игроки исследуют процедурно сгенерированный мир, который можно разрушать и модифицировать. В игре присутствует глубокая система крафта, позволяющая создавать множество предметов и строений.

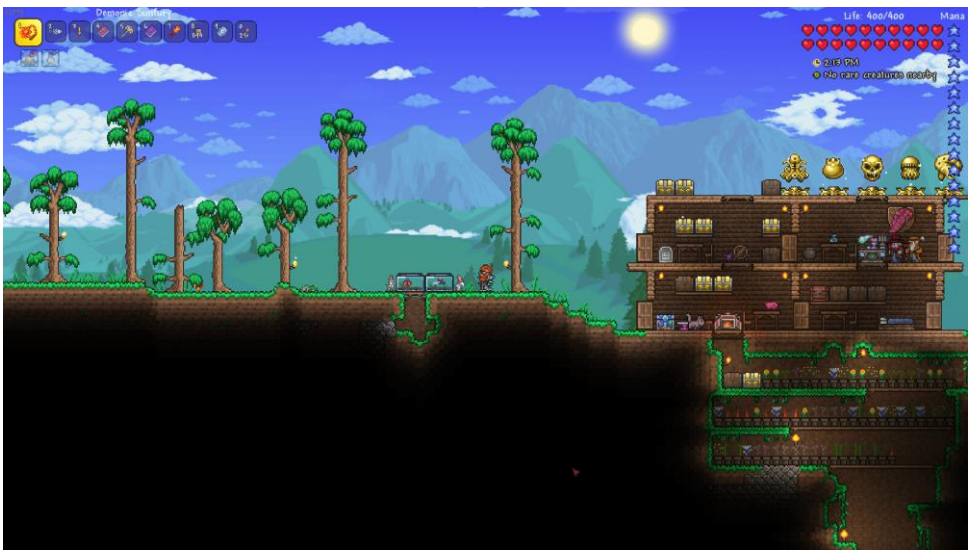

Рисунок 1.2 – Игра Terraria

Основные преимущества Terraria заключаются в свободе действий и широких возможностях для творчества. Игроки могут разрушать и модифицировать практически любые элементы игрового мира, что создаёт ощущение живого и интерактивного окружения.

Однако, несмотря на разнообразие контента, Terraria всё же больше акцентирует внимание на крафте и строительстве, чем на исследовании процедурно сгенерированных уровней в традиционном платформерном смысле. Процедурная генерация работает на уровне общего ландшафта, но не применима к детализированным уровням.

Dead Cells предлагает игрокам случайно сгенерированные уровни, где каждый забег отличается от предыдущего. Основное достоинство игры — это динамичный и плавный геймплей, который совмещает элементы роглайк с метроидванией. Процедурная генерация здесь используется для создания новых сочетаний комнат, врагов и испытаний, что поддерживает высокую реиграбельность и интерес у игроков.

Несмотря на успех, ограниченность разрушаемого окружения делает игру более линейной, так как игроки не могут активно изменять ландшафт, чтобы найти альтернативные пути или сократить дорогу к цели. Кроме того, взаимодействие с окружением в Dead Cells более ограничено по сравнению с классическими песочницами, что снижает вариативность игрового процесса.

Рисунок 1.3 – Игра Dead Cells

Broforce — это 2D экшен-платформер, в котором игроки могут разрушать все элементы ландшафта. Игра предлагает юмористический и насыщенный экшен-геймплей с разнообразными миссиями и персонажами. Основная фишка игры — полная разрушаемость окружения, что позволяет игрокам использовать ландшафт в своих целях. Например, можно создавать

обвалы или прокладывать себе путь сквозь стены, что добавляет тактического разнообразия.

Рисунок 1.4 – Игра Broforce

Тем не менее, процедурная генерация в Broforce отсутствует, и все уровни создаются вручную, что ограничивает разнообразие игрового опыта при повторном прохождении. Также игра фокусируется на экшене, что оставляет меньше места для стратегии и творчества.

Анализ существующих аналогов показывает, что хотя все вышеперечисленные игры предлагают уникальные механики и интересный геймплей, они имеют и свои ограничения. Spelunky и Dead Cells акцентируют внимание на исследовании и боевых механиках, но недостаточно развивают взаимодействие с окружающей средой. Terraria предлагает богатые возможности для крафта и модификации мира, но не делает акцент на традиционном платформерном геймплее и процедурной генерации. Broforce предоставляет игрокам разрушаемое окружение, но отсутствие процедурной генерации делает игру менее динамичной и вариативной.

Разрабатываемый проект стремится устранить недостатки существующих аналогов и предложить более глубокий и увлекательный игровой опыт за счёт следующих особенностей:

– модульная процедурная генерация;
– интерактивное окружение;
– динамическая адаптация сложности.

Уровни создаются из заранее разработанных сегментов, что обеспечивает баланс между случайностью и играбельностью. Это позволяет избежать хаотичности, свойственной многим рогаликам, сохраняя уникальность каждого уровня.

Игроки смогут взаимодействовать с элементами окружения, например, открывать сундуки, разрушать препятствия или искать скрытые бонусы. Такой подход добавляет тактическое разнообразие в геймплей.

Игра будет подстраиваться под навыки игрока, постепенно увеличивая сложность, чтобы поддерживать интерес и избегать чрезмерных сложностей для новичков.

Для реализации всех вышеописанных механик будет использован движок Unity, который позволяет работать с детализированной физикой и обеспечивает поддержку сложных графических эффектов. Это позволит создать привлекательный визуальный стиль и реалистичные анимации, которые будут способствовать погружению игроков в игровой процесс.

Анализ существующих аналогов показывает, что рынок насыщен проектами с интересными механиками, однако ни одна из рассмотренных игр не объединяет в себе процедурную генерацию уровней, разрушение объектов и адаптивную сложность в одном проекте. Разрабатываемый платформер стремится восполнить этот пробел, предлагая игрокам уникальный игровой опыт, в котором сочетаются вариативность, интерактивность и удобство для игроков разного уровня подготовки.

## 1.5 Выбор и обоснование инструментов разработки ПС

Для создания игрового программного средства с процедурной генерацией уровней были выбраны инструменты и технологии, которые обеспечивают высокую производительность, гибкость разработки и лёгкость масштабирования проекта. При выборе подходящих технологий учитывались такие факторы, как простота интеграции, возможность быстрой адаптации к новым условиям, наличие документации и активного сообщества, что помогает ускорить процесс разработки и улучшить конечный продукт. Рассмотрим подробнее выбранные языки программирования, игровые движки и вспомогательные библиотеки, а также обоснование их использования.

Основным языком разработки был выбран C#. Это один из стандартных языков программирования для работы с игровым движком Unity, который будет использоваться для создания проекта. C# предлагает удобный и интуитивно понятный синтаксис, что делает его удобным для разработчиков с разным уровнем подготовки. Высокая скорость выполнения кода и обширные возможности для реализации сложной игровой логики делают C# отличным выбором для разработки динамичных игр, требующих высокой производительности. Этот язык поддерживает широкий спектр библиотек и инструментов, что существенно ускоряет процесс разработки, а также упрощает тестирование и последующее сопровождение кода. C# обеспечивает хорошие возможности для использования объектно-ориентированного программирования, что позволяет структурировать игровой проект так, чтобы

его было легко масштабировать и адаптировать под новые функции или механики.

C# — выбран как основной язык программирования благодаря тесной интеграции с Unity и удобству работы с игровой логикой. Это один из самых популярных языков для разработки игр на платформе Unity, что гарантирует наличие большого количества документации и примеров кода, облегчающих процесс обучения и разработки. C# также поддерживает объектно-ориентированное программирование, что упрощает структуру кода, делает его более гибким и легко поддающимся изменениям. Это особенно важно для разработки сложных игровых систем, таких как генерация уровней и физические взаимодействия.

Выбор игрового движка пал на Unity. Это универсальный и популярный инструмент, который предоставляет мощные средства для создания 2D игр. Unity известен своими широкими возможностями для работы с физикой и анимацией, что особенно важно для платформеров, где плавность движений и взаимодействие с окружением играют ключевую роль. Движок поддерживает разнообразные графические эффекты, что позволяет создавать стильный и уникальный визуальный опыт. Одной из главных особенностей Unity является наличие готовых решений для процедурной генерации уровней, что упрощает реализацию алгоритмов создания новых ландшафтов и игровых карт при каждом запуске игры. Встроенные инструменты для работы с разрушаемым окружением позволяют создавать интерактивные объекты, с которыми игроки могут активно взаимодействовать, что добавляет глубину и динамичность игровому процессу. Кроме того, благодаря поддержке C#, разработка и реализация игровой логики становится удобной и быстрой. [3]

Unity также обладает большим и активным сообществом разработчиков, а также обширной поддержкой в виде форумов, документации и руководств. Это позволяет решать возникающие вопросы быстро и эффективно, используя опыт других разработчиков. Магазин ассетов Unity предлагает широкий выбор готовых модулей и плагинов, которые можно интегрировать в проект для ускорения разработки. Это снижает затраты времени и ресурсов на создание отдельных элементов, позволяя сосредоточиться на основных аспектах геймплея и уникальных особенностях игры. Unity также поддерживает кроссплатформенную разработку, что позволяет создавать одну игру и легко адаптировать её для запуска на различных устройствах и операционных системах, таких как Windows, macOS, Android, iOS и игровые консоли. Это существенно расширяет потенциальную аудиторию игры и делает проект более универсальным и коммерчески выгодным.

Unity — это мощный и гибкий инструмент для создания 2D игр, особенно когда речь идёт о проектах с элементами процедурной генерации и интерактивным окружением. Движок обеспечивает интуитивный интерфейс и обширную поддержку ассетов, что делает его идеальным выбором для разработки платформеров. Разработчики могут легко создавать уникальные

уровни, использовать готовые модули и расширять возможности игры за счёт дополнительных плагинов. Unity также поддерживает различные платформы, что позволяет запускать одну и ту же игру на разных устройствах, расширяя её потенциальную аудиторию. Кроссплатформенность и возможность интеграции с различными системами делает Unity универсальным инструментом для создания игр любого уровня сложности. [4]

Для реализации процедурной генерации уровней в разрабатываемом проекте был выбран Unity Tilemap — инструмент, предоставляемый Unity для работы с двумерными тайловыми картами. Этот редактор предлагает ряд преимуществ, которые делают его идеальным для создания и модификации игровых уровней с использованием процедурной генерации.

Использование редактора Unity Tilemap оправдано, так как он предоставляет оптимальные инструменты для разработки и редактирования уровней в 2D платформере. Возможности для работы с тайловыми картами, поддержка процедурной генерации и высокая производительность делают его отличным выбором для реализации динамичных и интерактивных уровней.

Выбор Unity в качестве движка и использование C# обусловлены необходимостью обеспечения гибкости разработки, высокой производительности и удобства в работе с различными платформами. Все эти инструменты совместно работают для создания полноценного и интерактивного игрового опыта, что делает выбранный набор технологий оптимальным для разработки 2D платформера с процедурной генерацией и интерактивным окружением, способным заинтересовать широкую аудиторию игроков. [5]

### 1.6 Спецификация требований

На основе проведенного анализа предметной области, а также с учетом требований, указанных в задании на дипломное проектирование, сформулированы нижеприведенные требования к проектируемому игровому программному средству.

**1.6.1** Назначение разработки**.** Целью разработки является создание игрового программного средства с процедурной генерацией уровней, которая будет предоставлять игрокам уникальный игровой опыт при каждом новом запуске. В основе игры лежит алгоритм, генерирующий уровни случайным образом, что обеспечивает разнообразие игровых ситуаций, повышая реиграбельность. Игра ориентирована на создание увлекательного, динамичного геймплея, где каждый уровень предоставляет игрокам новые испытания и возможности для исследования.

Разработка направлена на реализацию механики взаимодействия с окружающим миром, включая разрушение объектов (например, сундуков) и использование этих взаимодействий для создания новых тактических

возможностей. Проект будет включать элементы платформера, в которых игроки должны преодолевать препятствия, сражаться с врагами, искать скрытые объекты.

Основными задачами разработки являются:

– создание системы процедурной генерации уровней;
– реализация механики разрушения объектов;
– обеспечение адаптивной сложности;
– реализация всех механик классического платформера.

**1.6.2** Перечень основных выполняемых функций. Нижеприведенные функции направлены на создание уникального и увлекательного игрового опыта, который будет сочетать элементы случайности, интерактивности и стратегии, делая игру динамичной и интересной для пользователей:

1 Процедурная генерация уровней. Алгоритм случайной генерации уровней создает уникальные карты при каждом запуске игры. Включает случайное размещение заготовленных фрагментов уровней, обеспечивая разнообразие и динамичность игрового процесса.

2 Механика разрушения объектов. Игроки могут разрушать элементы окружения, такие как сундуки с монетами, что позволяет получать награды.

3 Динамичное управление персонажем. Игрок управляет персонажем с возможностью бега, прыжков и стрельбы. Реализована плавная анимация движений, что способствует комфортному и интуитивно понятному управлению.

4 Взаимодействие с окружением. Игрок может взаимодействовать с различными объектами на уровне (например, открывать сундуки). Взаимодействие с миром увеличивает вариативность и стратегические возможности.

5 Адаптивная сложность. Игра автоматически подстраивается под уровень мастерства игрока, увеличивая или уменьшая сложность в зависимости от его успехов в процессе прохождения.

6 Визуальные и звуковые эффекты. Реализация анимаций разрушений, взаимодействий и изменений на уровне. Включает в себя создание визуальных эффектов разрушений, а также соответствующих звуковых эффектов для усиления атмосферы.

**1.6.3** Входные данные. Нижеприведенные входные данные позволяют правильно настроить игровую среду и взаимодействие игрока с ней, обеспечивая плавный и динамичный игровой процесс:

1 Параметры процедурной генерации уровней. Входными данными для алгоритма процедурной генерации являются стартовый фрагмент уровня и массив фрагметов для дальнейшей генерации.

2 Данные о персонаже. Входные данные для управления персонажем включают:

– позиция на уровне;
– скорость и управление;
– состояние здоровья и другие параметры.

3 Информация об объектах окружения. Для взаимодействия с окружением необходимо учитывать следующие данные:
– типы объектов;
– состояние объектов.

4 Аудио и визуальные данные. Для воспроизведения различных эффектов и анимаций важны:
– аудиофайлы;
– графика и анимации.

5 Уровни и статистика. Данные о предыдущих уровнях или текущем прогрессе:
– текущий счет;
– рекордный счет.

**1.6.4** Выходные данные. Нижеприведенные выходные данные необходимы для дальнейшей обработки в рамках игрового процесса, поддержания взаимодействий с окружением, отображения прогресса игрока и обеспечения плавной работы всех игровых механик:

1 Сгенерированные уровни. Основным выходом системы является генерация уровня, которая включает заранее выбранные фрагменты уровней для генерации.

2 Состояние персонажа. Выходными данными являются характеристики и состояние персонажа, такие как:
– текущая позиция;
– здоровье и другие параметры;
– взаимодействия с объектами.

3 Враги. Для каждого врага в игре генерируется:
– позиции врагов;
– состояние врагов;
– алгоритм поведения врагов.

4 Интеракция с окружением. Все взаимодействия игрока с окружением формируют состояния объектов.

5 Результаты игрового процесса. В процессе игры генерируются данные, отображающие финальный и рекордный счет.

6 Аудио и визуальные эффекты. В процессе игры воспроизводятся анимации и звуковые эффекты.

**1.6.5** Среда эксплуатации. Среда эксплуатации разрабатываемого программного средства — это комплекс аппаратных и программных условий, в которых будет использоваться приложение. Включает в себя требования как к вычислительным мощностям, так и к операционным системам и

программному обеспечению. Учитывая специфику проекта (игровое программное средство с процедурной генерацией уровней), эксплуатационная среда будет следующей:

1 Операционные системы:

– Windows;
– macOS;
– Linux.

2 Видеокарты:

– NVIDIA GeForce GTX 1050 или эквивалент;
– AMD Radeon RX 560 или эквивалент.

Видеокарты с более низким уровнем также поддерживают работу игры, но могут возникнуть проблемы с производительностью при генерации сложных уровней или высоком уровне детализации.

3 Оперативная память. Минимальные требования к оперативной памяти:

– 4 GB RAM.

4 Процессор. Разрабатываемая игра имеет средние требования к процессору, и для комфортной работы потребуется:

– Intel Core i3 или AMD Ryzen 3.

5 Жесткий диск. Игра будет требовать наличия свободного места на жестком диске для установки и хранения данных:

– минимум 300 MB свободного места для установки игры и дополнительных данных.

6 Программное обеспечение:

– Unity;
– C#;
– Visual Studio;
– Adobe Photoshop.

7 Внешние устройства:

– мышь и клавиатура (стандартные устройства ввода, которые обеспечивают управление персонажем и взаимодействие с игрой);
– экран (подходит для работы на экранах с разрешением от 1280x720 пикселей, рекомендуется 1920x1080 для обеспечения лучшего визуального восприятия и детализации).

Таким образом, среда эксплуатации разрабатываемого программного средства предполагает использование стандартного оборудования и программного обеспечения, доступного большинству пользователей, что обеспечивает широкую доступность игры.

# 2 МОДЕЛИРОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

## 2.1 Разработка диаграммы вариантов использования ПС

Диаграмма вариантов использования (Use Case Diagram) является инструментом для визуализации функциональности программного средства с точки зрения пользователя. Она описывает взаимодействие акторов (пользователей или внешних систем) с системой, а также демонстрирует, какие задачи пользователь может выполнять в программном средстве. [6]

В рамках разработки игрового программного средства была построена диаграмма вариантов использования, чтобы отразить основные функции, доступные пользователю.

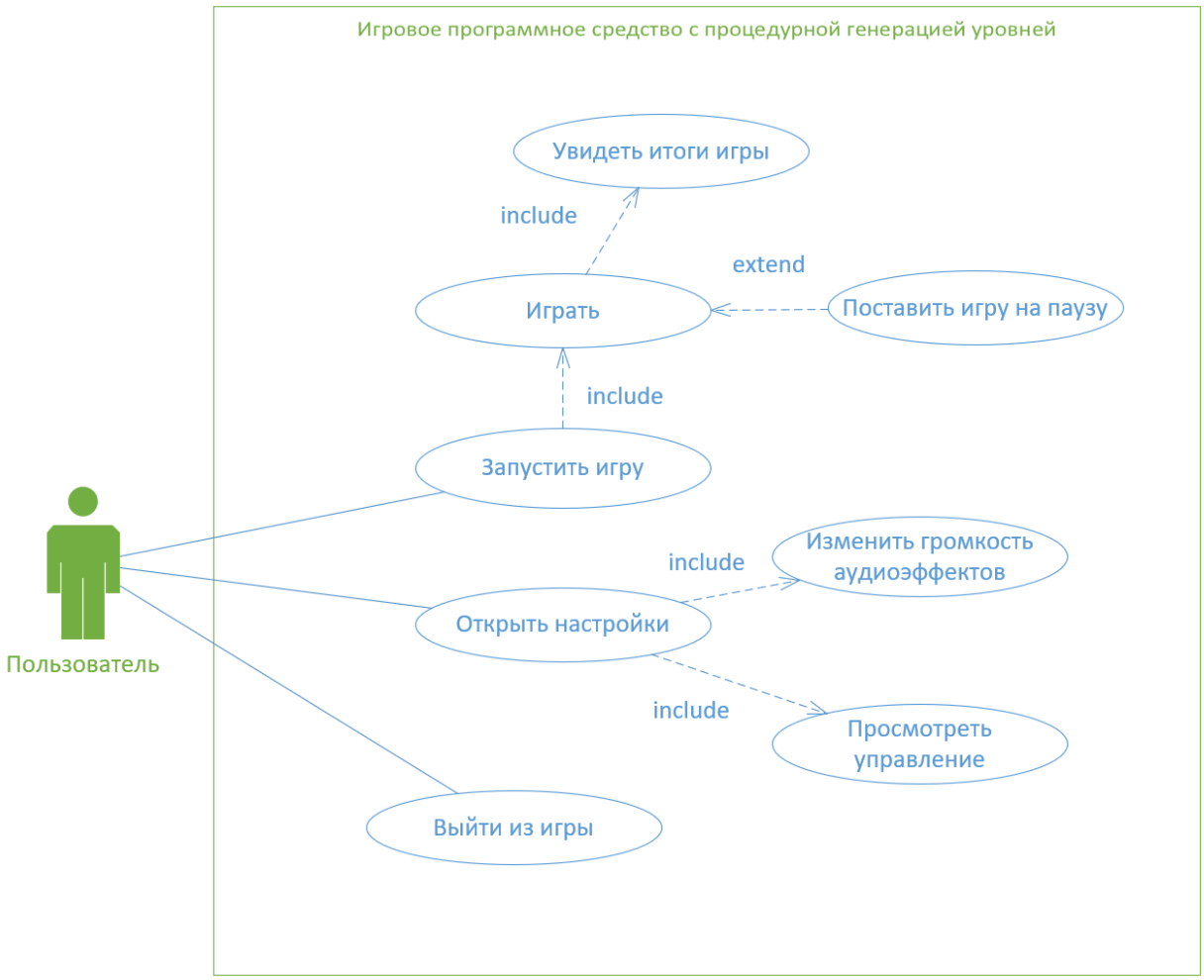


Рисунок 2.1 – Диаграмма вариантов использования

Пользователь — актор, который взаимодействует с программным средством и управляет персонажем в игре.

Диаграмма включает следующие ключевые варианты использования:

– запустить игру (пользователь выбирает эту функцию в главном меню

для начала игрового процесса);

– играть (пользователь управляет персонажем и взаимодействует с игровыми объектами);

– поставить игру на паузу (пользователь может приостановить игровой процесс, чтобы сделать перерыв или вернуться в меню);

– открыть настройки (пользователь может просматривать параметры управления и громкости звука);

– увидеть итоги игры (после завершения игровой сессии пользователю предоставляется итоговая статистика);

– изменить громкость аудиоэффектов (через меню настроек пользователь может управлять громкостью звукового сопровождения);

– просмотреть управление (пользователь может ознакомиться с элементами управления в разделе настроек);

– выйти из игры (пользователь может вернуться в систему).

Пользователь связан со всеми вариантами использования, так как он инициирует все основные действия.

Вариант использования “Открыть настройки” включает в себя варианты использования “Изменить громкость аудиоэффектов” и “Просмотреть управление”. Вариант использования “Запустить игру” включает в себя вариант использования “Играть”, который в свою очередь включает в себя вариант использования “Увидеть итоги игры” и расширяется вариантом использования “Поставить игру на паузу”.

Диаграмма вариантов использования позволяет:

– понять ключевые сценарии взаимодействия пользователя с системой;

– определить, какие функции критически важны для реализации;

– визуализировать связи между функциями и пользователем.

Диаграмма предоставляет общий взгляд на систему и упрощает процесс её проектирования, помогая сконцентрироваться на реализации наиболее важных функций. [7]

## 2.2 Разработка диаграммы состояний программного средства

Диаграмма состояний (State Diagram) является важным элементом проектирования программного средства. Она позволяет описать возможные состояния системы или её компонентов, а также переходы между ними, вызванные определёнными событиями или действиями пользователя. Для разрабатываемого игрового программного средства с процедурной генерацией уровней диаграмма состояний отражает ключевые этапы и сценарии функционирования игрового программного средства в целом. [8]

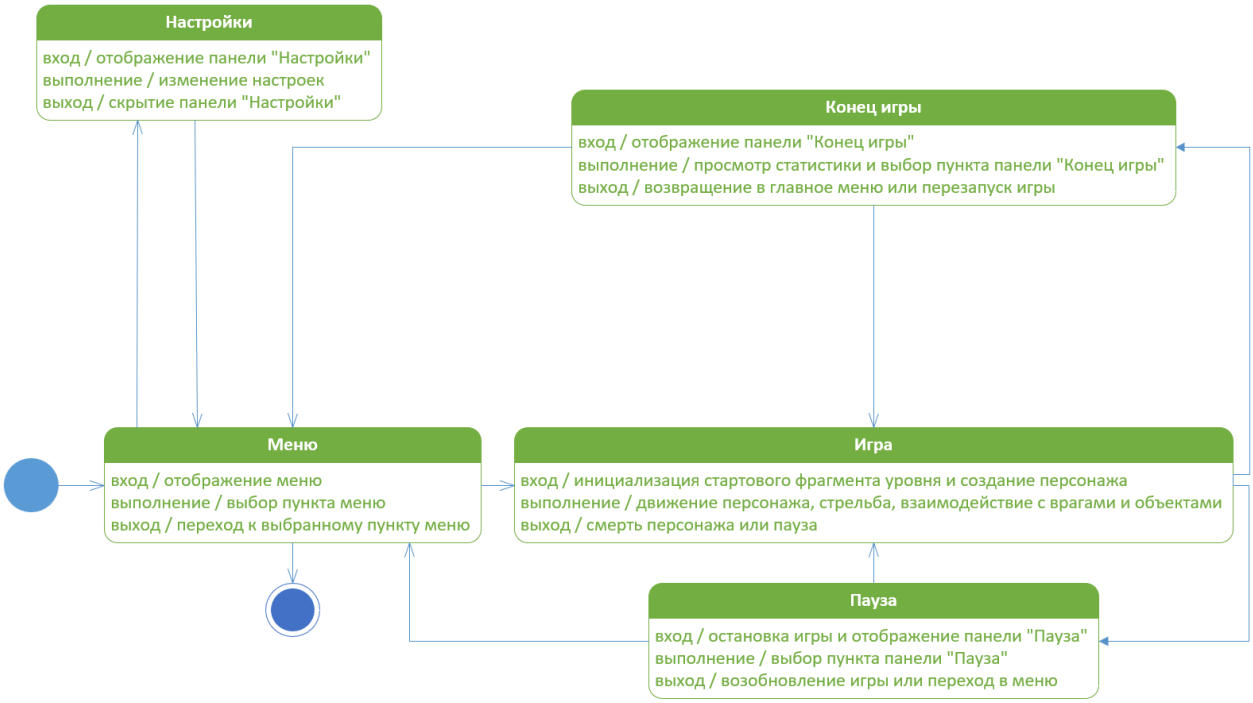


Рисунок 2.2 – Диаграмма состояний

В ходе разработки были выделены следующие состояния программного средства:

– меню (начальное состояние игры, где пользователь может выбрать дальнейшие действия: начать игру, перейти к настройкам или выйти из программы);

– игра (активное состояние, в котором происходит управление персонажем, взаимодействие с окружением и выполнение игровых задач);

– пауза (состояние, при котором игровой процесс временно приостанавливается. Пользователь может продолжить игру или выйти в главное меню);

– конец игры (финальное состояние, возникающее при смерти персонажа. Пользователь видит результаты и может начать игру заново или выйти в меню);

– настройки (состояние, в котором пользователь может регулировать параметры игры, такие как громкость звука или просматривать управление).

Были выделены следующие переходы между состояниями:

– из главного меню в игру (осуществляется выбором соответствующей опции в меню);

– из игры в паузу (происходит, если пользователь нажимает кнопку паузы);

– из паузы в игру (возвращение к игровому процессу по нажатию кнопки продолжения);

– из игры в конец игры (возникает, если персонаж погибает);

– из конца игры в игру (возникает, если пользователь начнет новую

игру);

– из конца игры в главное меню (переход, если игрок выбирает выход в меню);

– из главного меню в настройки (осуществляется выбором соответствующей опции в меню);

– из настроек в главное меню (возвращение происходит после завершения изменения параметров).

В каждом состоянии система выполняет определённые действия (entry, do, exit). В состоянии «Игра» система обрабатывает ввод пользователя, обновляет игровую физику и отображает графику. В состоянии «Пауза» все действия в игре временно приостанавливаются, но доступно меню паузы. События являются триггерами, которые вызывают переходы между состояниями.

Диаграмма помогает наглядно представить, как игра реагирует на действия пользователя и как организованы переходы между состояниями. Предоставляет чёткую структуру, что позволяет легче реализовать логику работы игры, а также помогает учесть все возможные переходы и состояния, чтобы избежать ошибок в проектировании. [9]

Диаграмма состояний для разрабатываемого игрового программного средства является инструментом, который не только упрощает процесс разработки, но и позволяет эффективно организовать управление игровыми состояниями, обеспечивая удобство взаимодействия пользователя с системой. [10]

### 2.3 Логическая модель взаимодействия объектов ПС

Логическая модель взаимодействия объектов программного средства описывает структуру и связи между ключевыми элементами игры. Для разрабатываемого игрового программного средства с процедурной генерацией уровней логическая модель позволяет проанализировать, как объекты взаимодействуют друг с другом, какие данные передаются между ними, а также определить их зависимости.

Основные объекты программного средства:

– игрок (Player);
– враг (Enemy);
– сундук (Chest);
– уровни (Area);
– управление уровнями (LevelManager);
– пуля (Bullet);
– меню (Menu);
– аудио-менеджер (AudioManager);
– параллакс (ParallaxEffect);
– монета (Coin);

– сердце (Heart).

Игрок — это основной объект, управляемый пользователем. Он выполняет такие действия, как перемещение, прыжки, стрельба, взаимодействие с объектами.

Игрок связан с подсистемами HealthManager, PlayerManager, PlayerShoot, LevelCounter. HealthManager — для контроля уровня здоровья. PlayerManager — для управления основными игровыми механиками игрока (состояния, взаимодействие с окружением, смерть). PlayerShoot — отвечает за генерацию пуль при стрельбе. LevelCounter — для подсчёта пройденного расстояния.

Враг управляется системой искусственного интеллекта, который выполняет такие действия, как перемещение и атака игрока. Взаимодействует с игроком и пулями. С игроком, чтобы нанести ему урон, а с пулями, чтобы получить урон.

Сундук — это объект, с которым взаимодействует игрок для получения монет. При открытии сундука генерируются монеты.

Уровни — это отдельные фрагменты игрового пространства, которые процедурно генерируются. Они связаны с объектом LevelManager, который отвечает за создание и удаление частей уровня по мере продвижения игрока.

LevelManager контролирует генерацию уровней и удаление устаревших фрагментов. Использует объект Игрок для отслеживания текущей позиции и создания новых участков при приближении к границе уровня.

Пуля взаимодействует с врагами и разрушаемыми объектами. Генерируется объектом PlayerShoot.

Меню управляет навигацией по игровым состояниям (главное меню, настройки, выход из игры) и взаимодействует с AudioManager для изменения громкости звуков.

Аудио-менеджер управляет звуковыми эффектами и фоновыми мелодиями в игре. Реагирует на изменения в меню (регулировка громкости) и переключение игровых состояний.

Параллакс отвечает за визуальный эффект движения заднего фона, который реагирует на перемещение объекта Игрок.

При столкновении враг наносит урон игроку через HealthManager, а Игрок может уничтожить врага с помощью пуль. Игрок взаимодействует с сундуками для получения монет.

LevelManager следит за положением игрока и добавляет новые части уровня (Area) при необходимости, а также удаляет устаревшие части уровня для оптимизации.

Меню управляет переходами между состояниями игры (запуск, настройки, выход из игры). Связь с AudioManager позволяет регулировать громкость звука.

Логическая модель предоставляет структурированный взгляд на взаимодействие объектов, что упрощает реализацию их функциональности.

Чётко определённые связи и зависимости между объектами помогают учесть все элементы игры при внесении изменений. Модель позволяет выявить избыточные связи или неэффективные взаимодействия между объектами, что улучшает производительность.

Логическая модель взаимодействия объектов программного средства служит основой для дальнейшего проектирования и реализации программного средства, позволяя добиться её целостности и функциональности.

## 2.4 Спецификация функциональных требований

Спецификация функциональных требований определяет основные функциональные возможности разрабатываемого игрового программного средства с процедурной генерацией уровне. Служит основой для проектирования, разработки и тестирования, а также для проверки соответствия готового продукта заявленным требованиям.

Функциональные требования:

1 Игрок:

– перемещение влево/вправо;

– прыжки;

– стрельба;

– взаимодействие с объектами (например, сундуками, элементами окружения).

2 Процедурная генерация уровней:

– уровни должны генерироваться динамически на основе заранее подготовленных сегментов;

– новый сегмент уровня должен добавляться, когда игрок приближается к концу текущего сегмента;

– одновременно в памяти должно находиться не более трёх сегментов уровня (для оптимизации);

– разрушаемое окружение должно корректно обрабатываться при генерации новых сегментов.

3 Игрок должен взаимодействовать с объектами, включая:

– сундуки (открытие сундуков для получения монет);

– сердца (восстановление здоровья).

Взаимодействие должно инициировать соответствующую анимацию и изменение игровых параметров (например, получение очков или восстановление здоровья).

4 Враги и искусственный интеллект:

– враги должны патрулировать, атаковать игрока, реагировать на его действия;

– враги уничтожаются при попадании пуль;

– при столкновении с игроком враги должны наносить урон.

5 Система разрушений:

– игрок может разрушать элементы окружения с помощью пуль.

6 Главное меню:

– запустить игру;

– открыть настройки;

– выйти из игры.

7 Настройки:

– возможность регулировать громкость звуковых эффектов и музыки;

– просмотр и изменение схемы управления.

8 Игровой интерфейс:

– количество жизней игрока;

– счёт;

– уровень.

9 Экран паузы:

– возможность поставить игру на паузу, продолжить игру или выйти в главное меню.

10 Конец игры:

– финальный счет;

– рекордный счет.

11 Звуковые эффекты должны сопровождать следующие действия:

– выстрелы;

– прыжки;

– нападение врагов;

– сбор монет;

– фоновая музыка;

– динамическая смена фоновой музыки при переходе между игровыми состояниями (например, игра, пауза, конец игры).

12 Регулировка звука:

– в настройках игрок должен иметь возможность изменять громкость звуков и музыки.

13 Сохранение прогресса:

– игра должна автоматически сохранять рекордный счет.

14 Производительность:

– игра должна стабильно работать с частотой 60 кадров в секунду на целевых устройствах.

15 Оптимизация процедурной генерации:

– одновременно в памяти не должно быть более трёх сегментов уровня;

– разрушенные объекты должны удаляться, если они больше не влияют на игровой процесс.

16 Обработка выхода за границы уровня:

– если игрок или враг покидает границы активного сегмента уровня, объект удаляется.

17 Сбой генерации уровня:

– если новый сегмент уровня не был корректно сгенерирован, необходимо повторить попытку генерации.

18 Ошибка взаимодействия с объектами:

– если объект, с которым игрок пытается взаимодействовать, не найден, действие отменяется без вылета из игры.

Спецификация функциональных требований является ключевым документом, который детализирует основные аспекты работы программного обеспечения, определяя его функциональность, ожидаемое поведение и ключевые характеристики. Этот документ служит основой для проектирования, разработки и тестирования программного продукта, позволяя сформировать единое понимание между разработчиками, дизайнерами и другими участниками процесса создания программного обеспечения.

В рамках данной спецификации была подробно описана функциональность, необходимая для реализации игрового приложения, включая элементы игрового процесса, взаимодействие с пользователем, структуру интерфейса и требования к производительности. Особое внимание уделялось тому, чтобы каждая функция была не только тщательно спроектирована, но и соответствовала ожиданиям целевой аудитории, обеспечивая пользователям удобство, удовольствие от игрового процесса и стабильную работу приложения на различных устройствах.

Удовлетворение заданных требований играет решающую роль в создании качественного игрового программного средства. Это включает разработку продуманного игрового процесса, способного вовлечь пользователя, создание интуитивно понятного интерфейса, который не требует длительного освоения, и обеспечение высокой производительности приложения, позволяющей избежать задержек и сбоев даже в условиях высокой нагрузки. Спецификация станет инструментом контроля на всех этапах работы над проектом.

# 3 ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

## 3.1 Проектирование архитектуры программного средства

Проектирование архитектуры программного средства направлено на определение структуры и взаимодействия основных компонентов системы, обеспечивающих выполнение всех заявленных функциональных и нефункциональных требований. В основе архитектуры лежат принципы модульности, переиспользуемости, масштабируемости и производительности.

Для реализации игрового программного средства используется компонентно-ориентированный подход, который основан на возможностях игрового движка Unity. Каждый объект (игрок, враг, предметы и т.д.) реализуется в виде компонента, что упрощает управление объектами и их взаимодействие.

Основные компоненты архитектуры:

1 Меню и пользовательский интерфейс, основной класс MenuEvents:

– главное меню (обеспечивает запуск игры, переход в настройки и выход);

– экран паузы (управление состоянием игры);

– настройки (регулировка звуков и управление);

– экран итогов (отображение рекордов после завершения игры).

2 Игровой процесс, основные классы (LevelManager, Player, Enemy):

– управление игроком (управление перемещением, прыжками, стрельбой (Player));

– генерация уровней (динамическое создание и удаление сегментов уровня (LevelManager));

– взаимодействие с врагами (контроль поведения врагов и реакции на игрока (Enemy)).

3 Физика и взаимодействие объектов реализуется средствами физического движка Unity:

– обработка столкновений игрока с врагами, предметами, пулями;

– разрушение окружения.

4 Аудио и эффекты (AudioManager):

– управление звуками и музыкой в зависимости от состояния игры.

5 Система событий и состояний (PlayerManager)

– реализация состояний (игра, пауза, конец игры);

– управление переходами между состояниями.

Логическая структура включает следующие подсистемы:

– подсистема меню (отвечает за работу главного меню, настроек, паузы и экрана итогов);

– подсистема игрового процесса (реализует взаимодействие игрока, врагов и объектов в игровом мире);

– подсистема управления уровнями (обеспечивает генерацию уровней и их оптимизацию);

– подсистема звука (отвечает за звуковое сопровождение и музыкальные эффекты).

Оптимизация процедурной генерации:

– генерация уровней выполняется заранее (до того, как игрок достигнет конца текущего сегмента);

– одновременная загрузка не более 3 сегментов уровня для снижения нагрузки на память.

Оптимизация звуков:

– использование аудиомикшера Unity для динамического управления звуками.

Игрок управляется через Player, который обрабатывает ввод, взаимодействие с окружением и столкновения. Взаимодействует с LevelManager для проверки текущего состояния уровня. Генерация уровней осуществляется LevelManager, который использует данные из Area (сегментов уровня). Уровни корректно обновляются при переходе персонажа на новые сегменты. Враги управляются Enemy, который контролирует их патрулирование, атаки и поведение. Меню и настройки управляются MenuEvents, который взаимодействует с AudioManager и PlayerManager.

Проектирование архитектуры обеспечивает модульную, гибкую и легко масштабируемую структуру программного средства. Взаимодействие между компонентами сведено к минимуму, что упрощает сопровождение и добавление нового функционала. Архитектура поддерживает стабильность игрового процесса, визуальные и звуковые эффекты, а также производительность игры. [11]

## 3.2 Разработка алгоритмов программного средства

Разработка алгоритмов программного средства является ключевым этапом, обеспечивающим реализацию всех заявленных функциональных требований игры. Алгоритмы детализируют логику работы системы, взаимодействие объектов и реакцию на действия пользователя.

1 Алгоритм процедурной генерации уровня:

Алгоритм отвечает за динамическое создание новых фрагментов уровня по мере продвижения игрока. [12]

Основные шаги алгоритма:

– проверить, находится ли игрок на расстоянии 15 единиц по оси X от конца последнего сегмента уровня (если верно, то продолжает алгоритм);

– случайным образом выбрать следующий сегмент уровня;

– разместить его в конце текущего уровня;

– удалить самый старый сегмент, если на экране больше 3 сегментов;

– обновить список активных сегментов уровня.

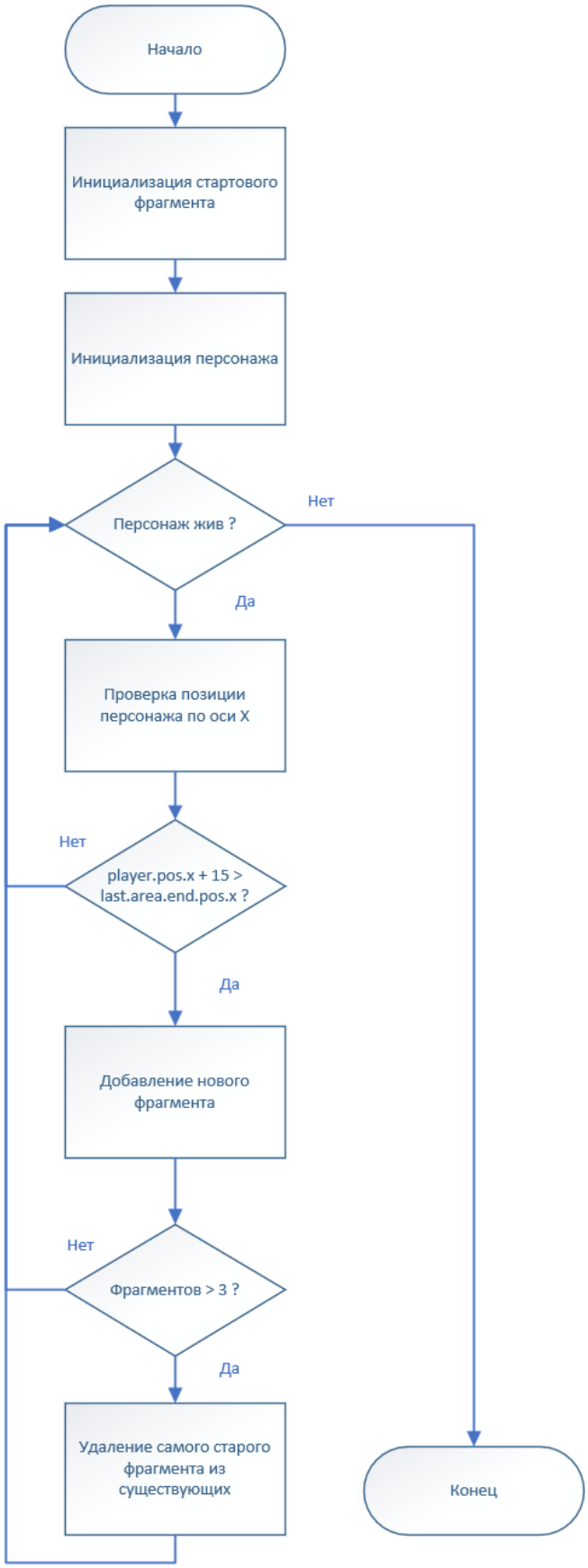


Рисунок 3.1 – Процедурная генерация уровня. Схема алгоритма

2 Алгоритм поведения врага:

Определяет действия врагов в зависимости от положения игрока. [13]

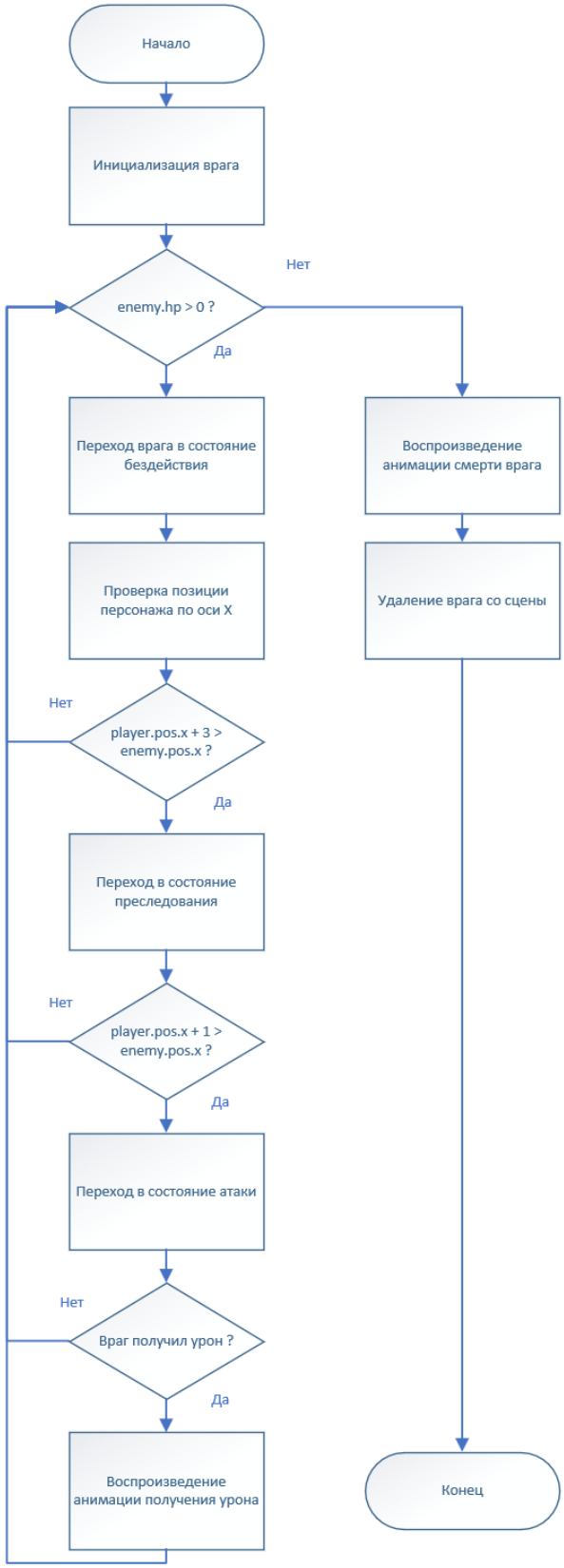


Рисунок 3.2 – Поведение врага. Схема алгоритма

Основные шаги алгоритма:

– враг определяет расстояние до игрока;
– если игрок находится в радиусе обнаружения;
– перейти в состояние преследования;
– следовать за игроком и атаковать, если расстояние позволяет;
– перейти в состояние бездействия (если игрок далеко);
– воспроизведение анимации при получении урона.

3 Алгоритм взаимодействия персонажа с объектами игры:

Обеспечивает реакцию игры на столкновения игрока с объектами, такими как сундуки, сердца, монеты или ловушки. [14]

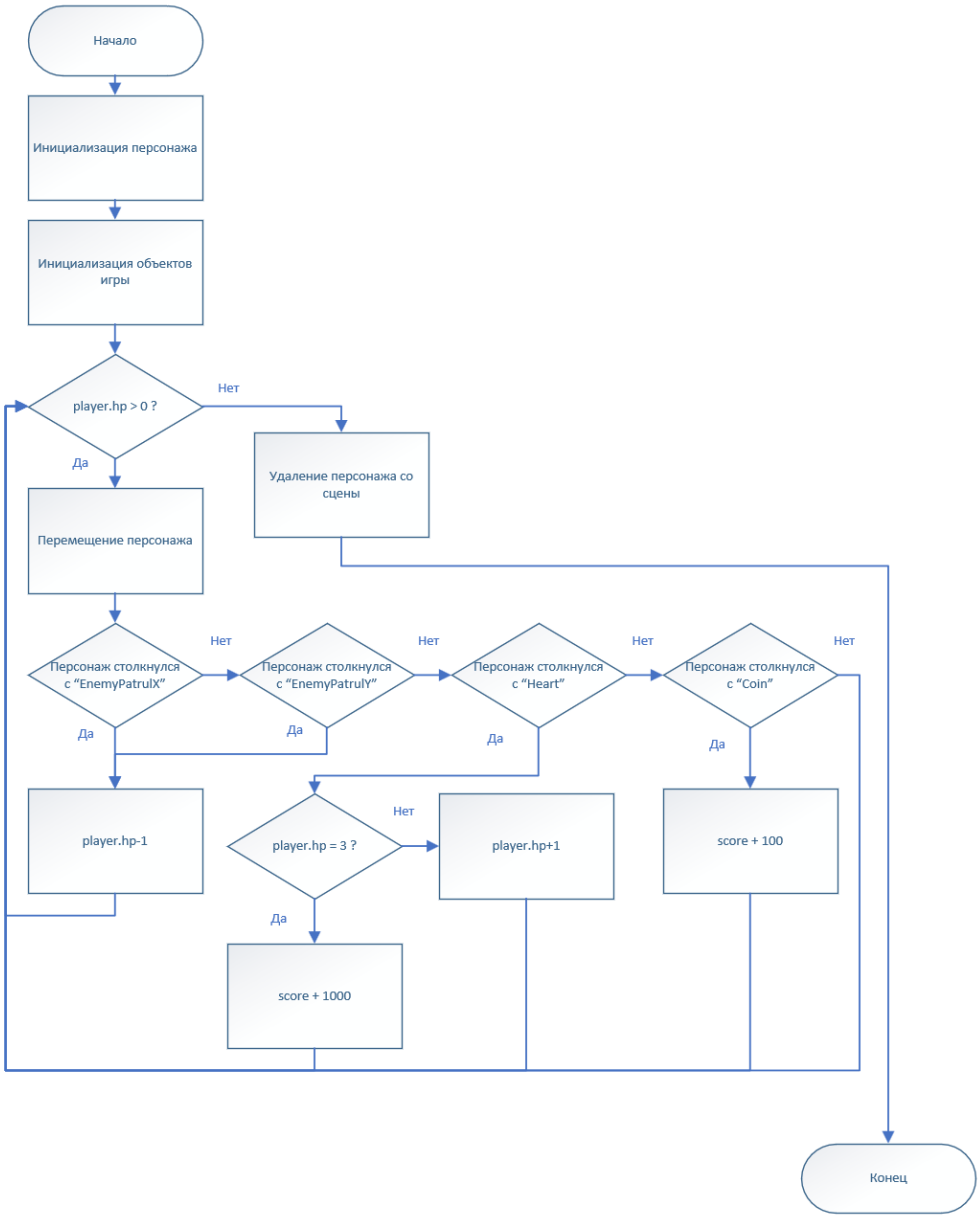


Рисунок 3.3 – Взаимодействие персонажа с объектами игры. Схема алгоритма

Основные шаги алгоритма:

– при столкновении игрока с объектом определить тип объекта;
– выполнить действие в зависимости от типа объекта;
– если объект — сундук, открыть его и выдать содержимое;
– если объект — сердце, восстановить здоровье игрока;
– если объект — ловушка, нанести урон игроку;
– удалить объект после взаимодействия, если это необходимо.

Разработка алгоритмов позволила детализировать функционал игрового процесса и взаимодействие между компонентами программного средства. Оптимизация алгоритмов обеспечит плавность работы игры и комфорт пользователя. Эти алгоритмы являются основой для реализации и тестирования всех модулей программного средства.

### 3.3 Разработка диаграммы классов программного средства

Диаграмма классов является важной частью проектирования архитектуры программного средства. Она отображает основные классы системы, их свойства, методы и взаимосвязи между ними. Диаграмма позволяет наглядно представить структуру программы и упрощает процесс её разработки и сопровождения. [15]

На основе анализа требований и функционала программного средства были выделены основные сущности, которые моделируются в виде классов. Каждый класс описывает объект или компонент системы и отвечает за определённый аспект работы программы.

Выделенные классы для разработки игрового программного средства:

– Player (представляет персонажа, управляемого пользователем);
– Enemy (описывает поведение противников);
– EnemyPatrolX (описывает поведение ловушек, который двигаются влево-вправо);
– EnemyPatrolY (описывает поведение ловушек, который двигаются вверх-вниз);
– Heart (представляет собой бонус, который восстанавливает жизнь);
– Area (отвечает за фрагменты игрового уровня);
– LevelManager (управляет генерацией уровней);
– LevelCounter (считает пройденное игроком расстояние и управляет счетчиком уровней);
– PlayerManager (обрабатывает состояние и действия игрока);
– HealthManager (управляет состоянием здоровья игрока);
– AudioManager (обеспечивает воспроизведение звуков);
– Sound (отвечает за аудио);
– MenuEvents (отвечает за работу с игровым меню);
– Parallax (обрабатывает эффект параллакса);
– PlayerShoot (генерирует пули, выпускаемые игроком);

– CoinChest (реализует функционал интерактивных сундуков).

Каждый класс содержит свойства (характеристики объекта) и методы (действия или операции, которые объект может выполнять).

1 Player:

Свойства:

– speed (скорость передвижения);
– jumpForce (мощность прыжка);
– groundCheck (объект для отслеживания коллизии с землей);
– playerRB (физика);
– animator (анимации);
– numberOfJumps (количество доступных прыжков);
– isGrounded (проверка нахождения на земле);
– isFacingRight (проверка, при которой персонаж разворачивается);
– deathLevel (смертельный уровень по Y для игрока);
– direction (позиция).

Методы:

– move() (перемещение персонажа);
– jump() (прыжок);
– flip() (разворот персонажа в другую сторону);
– takeDamage() (получение урона);
– death() (мгновенная смерть);
– getHurt() (неуязвимость после получения урона).

2 Enemy:

Свойства:

– player (отслеживание игрока);
– borderCheck (отслеживание границ платформы);
– enemyHP (количество здоровья);
– animator (анимации).

Методы:

– takeDamage() (получение урона);
– playerDamage() (нанесение урона).

3 EnemyPatrolX

Свойства:

– patrolSpeed (скорость движения);
– patrolDistance (радиус движения);
– movingRight (проверка на движение вправо).

Методы:

– patrol() (патрулирование).

4 EnemyPatrolY

Свойства:

– patrolSpeed (скорость движения);
– patrolDistance (радиус движения);
– movingUp (проверка на движение вверх).

Методы:

– patrol() (патрулирование).

5 Heart

Свойства:

– scoreReward (размер награды).

Методы:

– addHealth() (восполнение здоровья игрока).

6 Area

Свойства:

– begin (точка начала фрагмента);
– end (точка конца фрагмента).

7 LevelManager:

Свойства:

– [] AreaPrefabs (массив фрагментов);
– firstArea (стартовый фрагмент);
– player (отслеживание позиции игрока).

Методы:

– spawnArea() (спавн нового фрагмента);
– getRandomArea() (выбор случайного фрагмента, по вероятностям на данных координатах по оси X).

8 LevelCounter

Свойства:

– player (отслеживание позиции игрока);
– level (текущий уровень);
– lastCheckPointX (последняя достигнутая точка по X).

Методы:

– updateLevelUI() (повышение уровня на UI).

9 PlayerManager

Свойства:

– isGameOver (проверка на состояние игры);
– gameOverScreen (панель конца игры);
– numberOfPoints (счет);
– scoreText (UI для счета);
– pauseMenuPanel (панель паузы);
– enemy (отслеживание врага);
– player (отслеживание игрока);
– finalScoreText (UI для финального счета);
– highScoreText (UI для рекордного счета).

Методы:

– showGameOverScreen() (переход на панель конца игры);
– checkAndSaveHighScore() (хранение рекордного счета);
– getHighScore() (получение данных счета);
– replayLevel() (перезапуск уровня);

– exit() (выход в меню);
– pauseGame() (переход на панель паузы);
– resumeGame() (продолжение игры).

10 HealthManager:

Свойства:
– health (текущее здоровье игрока);
– maxhealth (максимальное здоровье игрока).

Методы:
– addHealth() (восполнение здоровья игрока).

11 AudioManager:

Свойства:
– [] sounds (массив аудио);
– sfxMixer (микшер звуков);
– musicMixer (микшер музыки).

Методы:
– playSFX() (воспроизведение звуков);
– playMusic() (воспроизведение музыки).

12 Sound

Свойства:
– name (название);
– clip (аудио);
– group (группа микшера);
– volume (громкость);
– pitch (усиление);
– loop (цикл).

13 MenuEvents

Свойства:
– volumeSlider (слайдер громкости звука);
– mixer (управляемый микшер);
– value (значение).

Методы:
– setVolume() (изменение громкости звука);
– loadGame() (переход на игровую сцену);
– exitGame() (выход из игры).

14 Parallax

Свойства:
– lenght (длина);
– mainCam (камера);
– middleBG (первый задник);
– sideBG (второй задник).

Методы:
– update() (обновить состояние).

15 PlayerShoot

Свойства:

– bulletForce (мощность выстрела);
– bulletHolle (генератор пуль);
– bullet (пуля).

Методы:

– fire() (выстрел).

16 CoinChest

Свойства:

– coinPrefab (монета);
– numberOfCoins (количество монет);
– spawnRadius (радиус спавна монет);
– chestAnimator (анимация открытия сундука).

Методы:

– spawnCoinsAfterAnimation() (спавн монет).

На диаграмме классов отображаются связи между классами. В системе используются следующие типы связей:

– ассоциация;
– агрегация;
– композиция;
– зависимость.

Ассоциация отражает, что один класс использует другой. Например, класс Player использует класс HealthManager для управления своим здоровьем.

Агрегация отражает, что один класс содержит другой, но не управляет его существованием. Например, LevelManager агрегирует фрагменты уровней (Area).

Композиция отражает, что один класс содержит другой и управляет его жизненным циклом. Например, PlayerManager включает в себя Player и отвечает за его создание и удаление.

Класс Player ассоциирован с классами HealthManager (управление здоровьем), PlayerShoot (стрельба) и PlayerManager (обработка состояния игрока).

Класс LevelManager агрегирует классы Player и Area (управление уровнем).

Класс MenuEvents ассоциирован с классами AudioManager (управление звуками).

Диаграмма классов была разработана на основе вышеуказанных описаний. Она включает:

– основные классы с их свойствами и методами;
– отображение типов связей между классами;
– логическую организацию компонентов игры.

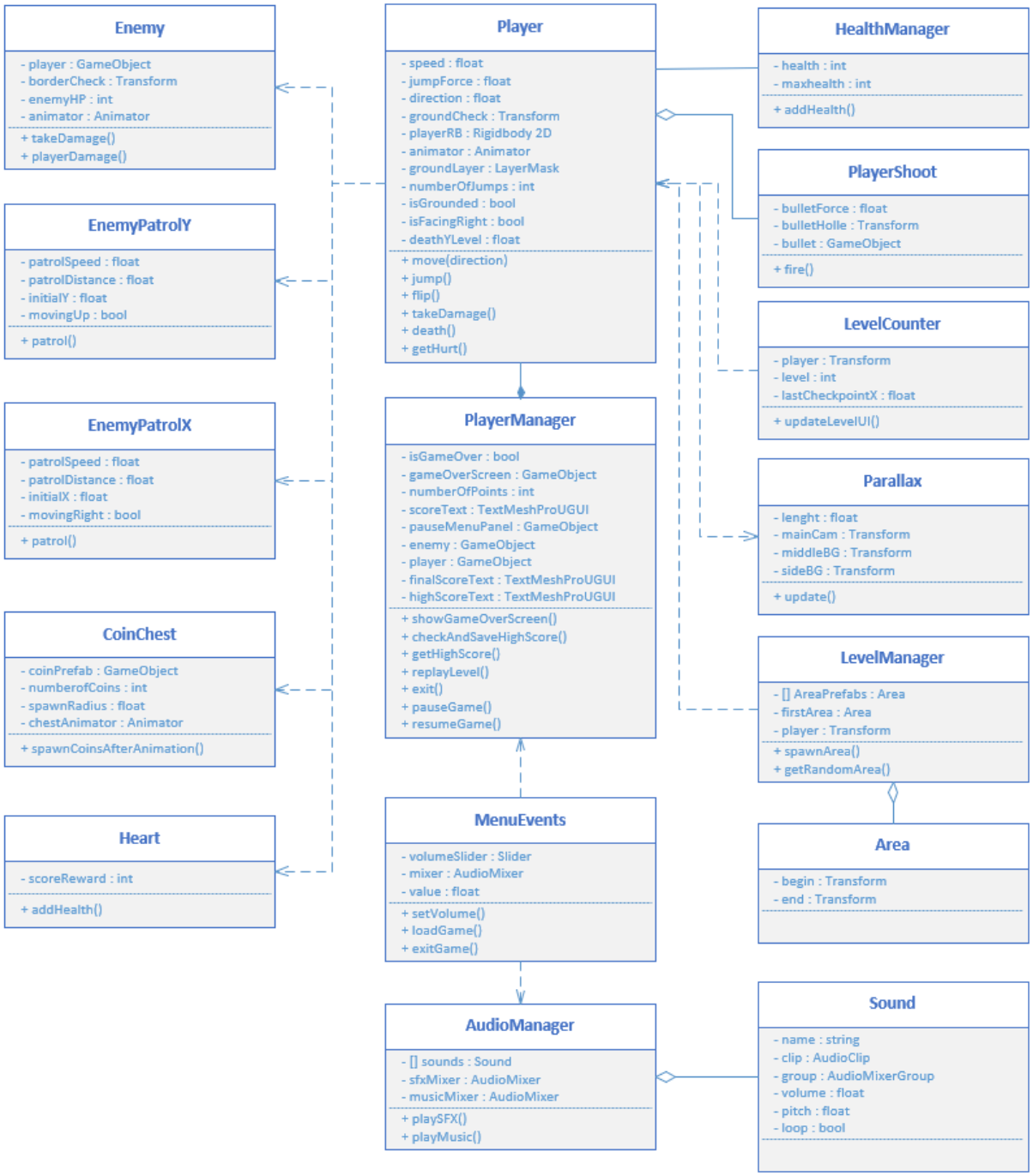


Рисунок 3.4 – Диаграмма классов

Диаграмма классов помогает систематизировать структуру проекта и упрощает процесс его разработки и сопровождения. [16]

### 3.4 Логика игровых сцен и переходов между ними

Логика игровых сцен и переходов между ними описывает последовательность действий, происходящих в игре, и определяет взаимодействие между различными состояниями игры. В проекте

предусмотрены несколько игровых сцен, каждая из которых выполняет свою функцию. [17]

Экран запуска игры:

– предоставляет игроку возможность выбрать действия (начать игру, просмотреть настройки или выйти из приложения).

Основная игровая сцена, где происходит игровой процесс:

– включает элементы управления персонажем, взаимодействия с объектами, генерации уровней и поведения врагов.

Переходы между сценами реализуются через систему управления состояниями, основанную на вызовах событий. Ниже представлены основные переходы между сценами:

Выбор игроком опции "Play" в главном меню запускает игровую сцену. При этом происходит загрузка всех игровых объектов и подготовка уровня.

После отображения итогов игры игрок может вернуться в главное меню для запуска новой игры или выхода.

Преимущества подхода:

– чёткая структура управления сценами;

– простота добавления новых состояний и переходов;

– повышение модульности и удобства сопровождения игры;

– возможность повторного использования кода для обработки переходов.

Для управления сценами используется UnityEngine.SceneManagement. Каждая сцена регистрируется и вызывается через функции LoadScene() и UnloadScene(). Переходы реализуются через обработку событий (например, нажатие кнопок интерфейса) и логические проверки (например, состояние игрока). Между сценами передаются данные о текущем прогрессе игрока, уровне или параметрах настроек. Это достигается через использование DontDestroyOnLoad для объектов. Сохранение данных в специальных менеджерах (например, PlayerManager).

### 3.5 Проектирование и разработка интерфейса ПС

Интерфейс пользователя (UI) является ключевым элементом взаимодействия игрока с программным средством. При проектировании интерфейса программного средства основное внимание уделяется его интуитивности, удобству и соответствию задачам игры. [18]

Цели проектирования интерфейса:

– интуитивность (минимальное количество действий для выполнения основных операций);

– функциональность (обеспечение всех необходимых возможностей для взаимодействия пользователя с игрой);

– эстетичность (визуальная привлекательность интерфейса для повышения уровня вовлечённости);

– универсальность (поддержка различных разрешений экранов и устройств);

– минимализм (отсутствие перегрузки информацией).

1 Главное меню — первая точка взаимодействия игрока с программным средством.


Рисунок 3.5 – Интерфейс меню

Элементы интерфейса:

– кнопка "Играть" для начала игры;

– кнопка "Настройки" для перехода к настройкам;

– кнопка "Выход" для завершения работы программы.

Особенности реализации:

– централизованное расположение кнопок для простоты восприятия;

– минимальное количество анимаций, чтобы не отвлекать игрока;

– использование нейтрального фона, связанного с общей стилистикой программного средства.

2 Интерфейс игрового процесса отображает текущий статус игрока и предоставляет вспомогательную информацию.

Элементы интерфейса:

– счётчик очков и уровней (показывает текущий прогресс);

– три сердца (отображает текущее состояние здоровья игрока);

– кнопка паузы (при нажатии открывает меню паузы).

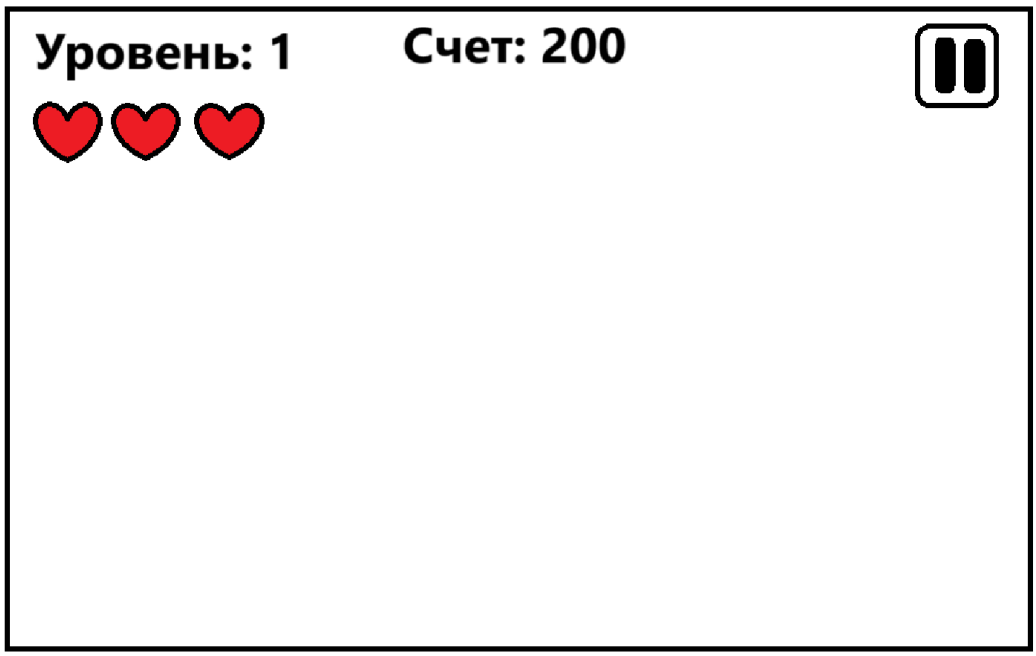


Рисунок 3.6 – Интерфейс игровой сцены

Особенности реализации:
– стабильное положение интерфейсных элементов;
– использование ненавязчивых цветов и минималистичного дизайна.
3 Меню паузы позволяет временно приостановить игровой процесс.


Рисунок 3.7 – Интерфейс меню паузы

Элементы интерфейса:
– кнопка "Да" для возврата к игре.
– кнопка "Нет" для возврата в главное меню.

4 Экран с итогами игры отображается после завершения игрового процесса.


Рисунок 3.8 – Интерфейс панели “Конец игры”

Элементы интерфейса:

– текстовое поле “Конец игры”;
– счетчик финальных очков;
– счетчик рекордных очков;
– текстовое поле “Начать новую игру?”;
– кнопка “Да” для перезапуска игры;
– кнопка “Нет” для выхода в главное меню.

Особенности реализации:

– близкое расположение кнопок для простоты восприятия;
– минимальное количество анимаций, чтобы не отвлекать игрока;
– использование нейтрального фона, связанного с общей стилистикой программного средства.

5 Меню настроек предоставляет возможность изменить параметры игры.

Элементы интерфейса:

– ползунок регулировки громкости звука;
– текст для просмотра настроек управления;
– кнопка для возврата в главное меню.


Рисунок 3.9 – Интерфейс настроек

В проекте используется встроенный инструмент Unity для создания интерфейса. Его основные компоненты:

– Canvas: (контейнер для элементов интерфейса);
– RectTransform (управление положением и размерами элементов);
– EventSystem (обработка событий взаимодействия).

Используются относительные размеры элементов (в процентах) для обеспечения масштабируемости интерфейса на разных устройствах. Шрифты читабельные и подходящие стилю игры. Иконки используются для обозначения часто используемых функций (например, кнопка паузы).

Таким образом, разработанный интерфейс отвечает требованиям функциональности, эстетики и удобства, обеспечивая комфортное взаимодействие пользователя с программным средством.

# 4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

## 4.1 Выбор и обоснование видов тестирования

Тестирование программного средства является неотъемлемой частью его разработки, направленной на обеспечение качества и стабильности работы. Для тестирования программного средства используются различные подходы, которые обеспечивают проверку функциональности, производительности, удобства использования и устойчивости к ошибкам. В данном разделе описаны виды тестирования, которые были выбраны для проекта, а также обоснование их необходимости. [19]

Было выбрано функциональное тестирование. Функциональное тестирование позволяет убедиться, что каждая функция игры (например, запуск игры, пауза, взаимодействие с объектами) работает корректно. Этот вид тестирования выявляет ошибки в реализации игровых механик и пользовательского интерфейса.

Вторым было выбрано UI-тестирование. Пользовательский интерфейс является основным способом взаимодействия игрока с программой, поэтому его правильная работа критически важна. UI-тестирование гарантирует, что кнопки, панели и другие элементы корректно отображаются и масштабируются на устройствах с разными разрешениями.

Третьим было выбрано интеграционное тестирование. Проверка взаимодействия между различными компонентами системы. Игра состоит из множества взаимосвязанных модулей (например, игрок, враги, LevelManager, AudioManager), и важно убедиться, что их взаимодействие работает корректно. Интеграционное тестирование позволяет обнаружить ошибки, которые не проявляются на уровне отдельных модулей.

Далее выбор пал на регрессионное тестирование. Убедиться, что изменения в коде не привели к поломке ранее работающего функционала. В процессе разработки регулярно вносятся изменения и доработки, которые могут случайно нарушить существующий функционал. Регрессионное тестирование помогает избежать возврата уже исправленных ошибок.

Следующим стало нагрузочное тестирование. Проверка производительности игры при высоких нагрузках. Игра должна стабильно работать даже при большом количестве активных объектов (например, врагов, пуль или уровней). Нагрузочное тестирование позволяет определить пределы производительности и оптимизировать ресурсы.

Очередным выбранным видом тестирования стало тестирование удобства использования. Определение, насколько удобно и понятно пользователю взаимодействовать с программным средством. Для игр удобство использования напрямую влияет на уровень вовлечённости и удовлетворённости пользователя. Usability-тестирование помогает выявить сложные или запутанные элементы интерфейса.

Далее было выбрано модульное тестирование. Тестирование отдельных модулей или компонентов программы в изоляции от остальных. Модульное тестирование позволяет выявить ошибки в работе конкретных скриптов или классов. Оно сокращает затраты на исправление ошибок, так как их проще исправить на раннем этапе.

Следующим стало тестирование на различных устройствах и платформах. Убедиться, что игра корректно работает на всех целевых устройствах и платформах. Игра может запускаться на устройствах с разными разрешениями экранов, мощностью процессоров и операционными системами. Тестирование на устройствах помогает избежать проблем с адаптивностью и производительностью.

И последним стало тестирование на случайных ошибках. Проверка стабильности игры при случайных или хаотичных действиях игрока. Игрок может действовать непредсказуемо, что может привести к нештатным ситуациям. Monkey-тестирование помогает выявить ошибки, которые трудно предсказать заранее.

Выбранные виды тестирования охватывают все основные аспекты игрвого программного средства, включая функциональность, производительность и удобство использования. Это позволяет:

– своевременно выявить и исправить ошибки;
– обеспечить стабильную работу всех компонентов;
– предоставить пользователям качественный продукт.

Тщательно спланированное тестирование является залогом успешной разработки программного средства, минимизируя риски и увеличивая его надёжность.

### 4.2 Результаты тестирования

Тестирование программного средства является ключевым этапом разработки, который позволяет выявить ошибки и проблемы, а также оценить качество работы игры. В результате тестирования были получены данные о функциональности, производительности, удобстве использования и устойчивости игры. Все найденные проблемы были классифицированы и решены, что позволяет улучшить стабильность и удобство игры.

Все основные механики игры работают согласно требованиям: генерация уровней, взаимодействие персонажа с объектами, работа врагов и механик стрельбы. Игровой процесс стабилен, без сбоев или зависаний при выполнении ключевых операций. Были выявлены незначительные ошибки в взаимодействии персонажа с некоторыми объектами, которые были оперативно исправлены. Исправлены баги, связанные с некорректным отображением некоторых объектов на уровнях, а также ошибки в логике взаимодействия игрока с врагами и предметами.

Были проведено множество тестов (таблица 4.1).

Таблица 4.1 – Тест-кейс игровых механик

| Тест | Ожидаемый результат | Результат |
|---|---|---|
| 1 | 2 | 3 |
| Перемещение персонажа влево и враво при нажатии соответствующих клавиш<br>1 Начать игру<br>2 Нажать клавишу перемещения влево (A)<br>3 Нажать клавишу перемещения вправо (D) | Персонаж корректно перемещается влево и вправо. | Успех |
| Прыжок персонажа<br>1 Начать игру<br>2 Нажать клавишу прыжка (Space) | Персонаж прыгает. | Успех |
| Двойной прыжок персонажа<br>1 Начать игру<br>2 Нажать клавишу прыжка (Space) два раза | Персонаж совершил двойной прыжок. | Успех |
| Взаимодействие с сундуками<br>1 Начать игру<br>2 Выстрелить по сундуку | Сундук открылся и появились монеты. | Успех |
| Выстрел персонажа<br>1 Начать игру<br>2 Нажать клавишу выстрела (ЛКМ) | Персонаж стреляет. | Успех |
| Получение урона<br>1 Начать игру<br>2 Столкнуться с патрулирующим врагом или подойти к атакующему врагу | Персонаж получил урон и потерял одну жизнь. | Успех |
| Процедурная генерация уровня<br>1 Начать игру<br>2 Двигаться вперед по уровню | Уровень не закончился и генерация происходит корректно. | Успех |

Продолжение таблицы 4.1

| 1 | 2 | 3 |
|---|---|---|
| Система очков<br>1 Начать игру<br>2 Подобрать монету или нанести урон врагу. | Счет увеличился на 100. | Успех |
| Подбор сердца, если у персонажа не 3 жизни<br>1 Начать игру<br>2 Получить урон<br>3 Подобрать сердце | Персонаж получил одну жизнь. | Успех |
| Подбор сердца, если у персонажа 3 жизни<br>1 Начать игру<br>2 Подобрать сердце | Счет увеличился на 1000. | Успех |
| Смерть персонажа<br>1 Начать игру<br>2 Потерять все жизни | Персонаж умер и открылась панель конца игры. | Успех |
| Меню игры<br>1 Запустить программное средство<br>2 Проверить каждую кнопку меню | Кнопки выполняют свои функции. | Успех |
| Громкость звука<br>1 Перейти в меню<br>2 Нажать кнопку “Options”<br>3 Изменить положение ползунка | Звук изменяет свою громкость в соответствии с положением ползунка. | Успех |
| Смерть персонажа от падения в пропасть<br>1 Начать игру<br>2 Упасть с платформы | Персонаж потерял все жизни и открылась панель конца игры. | Успех |
| Убийство врага<br>1 Начать игру<br>2 Нанести урон врагу 4 раза | Враг воспроизвел анимацию смерти и пропал. | Успех |

Все модули, включая игрока, врагов, генерацию уровней, управление звуками, корректно взаимодействуют друг с другом. Были выявлены небольшие проблемы с синхронизацией звуковых эффектов с действиями игрока. Не было выявлено проблем с обменом данных между объектами и компонентами системы. Проведена оптимизация аудиофайлов для

синхронизации звуковых эффектов с действиями игрока. Устранены небольшие проблемы с задержками в обработке ввода в моменты интенсивного игрового процесса.

Все функции, которые были проверены ранее, продолжают работать корректно после внесения изменений. Ранее исправленные ошибки не возникли снова. Не были выявлены новые ошибки, связанные с внесёнными исправлениями. Все тесты прошли без сбоев. Внесённые изменения не вызвали новых ошибок в предыдущих частях проекта.

Игра стабильно работает даже при большом количестве объектов (например, врагов или пуль) на уровне, не теряя производительности. На слабых устройствах были выявлены проблемы с FPS при большом количестве объектов на экране, что потребовало оптимизации. Проведена оптимизация кода для улучшения производительности, включая использование объектных пулов и оптимизацию графики. Были минимизированы задержки при рендеринге объектов на слабых устройствах.

Интерфейс оказался интуитивно понятным, а управление — удобным для большинства пользователей. Некоторые игроки выразили пожелания по улучшению расположения кнопок в меню, а также изменению цвета текста в некоторых разделах меню. Были внесены изменения в расположение и стилизацию кнопок для улучшения пользовательского опыта. Также было улучшено отображение текста, чтобы повысить его читаемость на всех устройствах.

Все модули, такие как генерация уровня, управление персонажем, обработка столкновений и стрельбы, работали корректно и не выявили ошибок. Проблемы с обработкой столкновений были минимизированы и устранены, а код стал более стабильным. Были проведены дополнительные тесты для каждой функции, чтобы убедиться в её корректной работе. Модуль столкновений был улучшен для предотвращения ошибок при близких позициях объектов.

Все выбранные виды тестирования были выполнены успешно, что позволило обеспечить:

– высокое качество функциональности и производительности игры;
– удобство использования и стабильность интерфейса;
– плавную работу игры при любых нагрузках.

На основе результатов тестирования были приняты меры по исправлению ошибок и оптимизации, что позволило довести проект до финальной стадии разработки.

Результаты тестирования показали, что игра удовлетворяет функциональным, производительным и пользовательским требованиям.

# 5 РУКОВОДСТВО ПО ЭКСПЛУАТАЦИИ ПРОГРАММНОГО СРЕДСТВА

Это руководство по эксплуатации предназначено для пользователей, желающих освоить игровое программное средство с процедурной генерацией уровней.

Для корректной работы игры необходимо, чтобы система соответствовала следующим минимальным требованиям:

1 Операционная система:
– Windows 7/8/10;
– macOS 10.12 или выше.
2 Процессор:
– Intel Core i3 или аналогичный.
3 Оперативная память:
– 4 ГБ или более.
4 Видеокарта:
– NVIDIA GeForce GTX 660 / AMD Radeon HD 7850 или более новая.
5 Место на жестком диске:
– 300 МБ свободного пространства.

Рекомендуемая настройка:
– Windows 10;
– 8 ГБ ОЗУ;
– NVIDIA GeForce GTX 1060 или аналогичная видеокарта.

Для установки игры скачайте распакуйте архив с программным средством. Запустите исполняемый файл и дождитесь появления меню программного средства.

После запуска программного средства пользователю будет предложено главное меню с несколькими опциями:

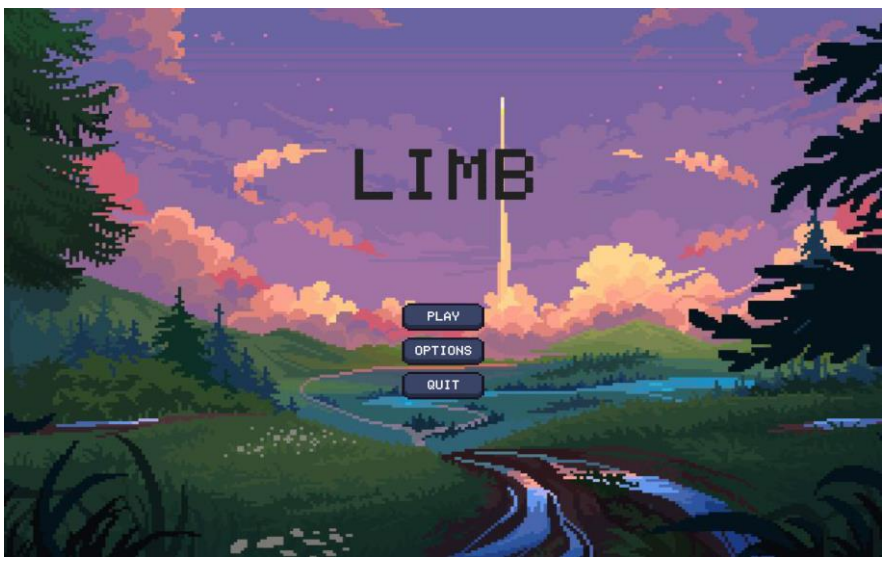


Рисунок 5.1 – Главное меню

После этого игрок может открыть настройки нажатием по кнопке “Options”, чтобы просмотреть управление персонажем и изменить уровень громкости.


Рисунок 5.2 – Панель настроек

Далее пользователь может закрыть панель настроек, нажав на крестик в правом верхнем углу панели настроек, и перейти к игровому процессу, нажав кнопку “Play”.

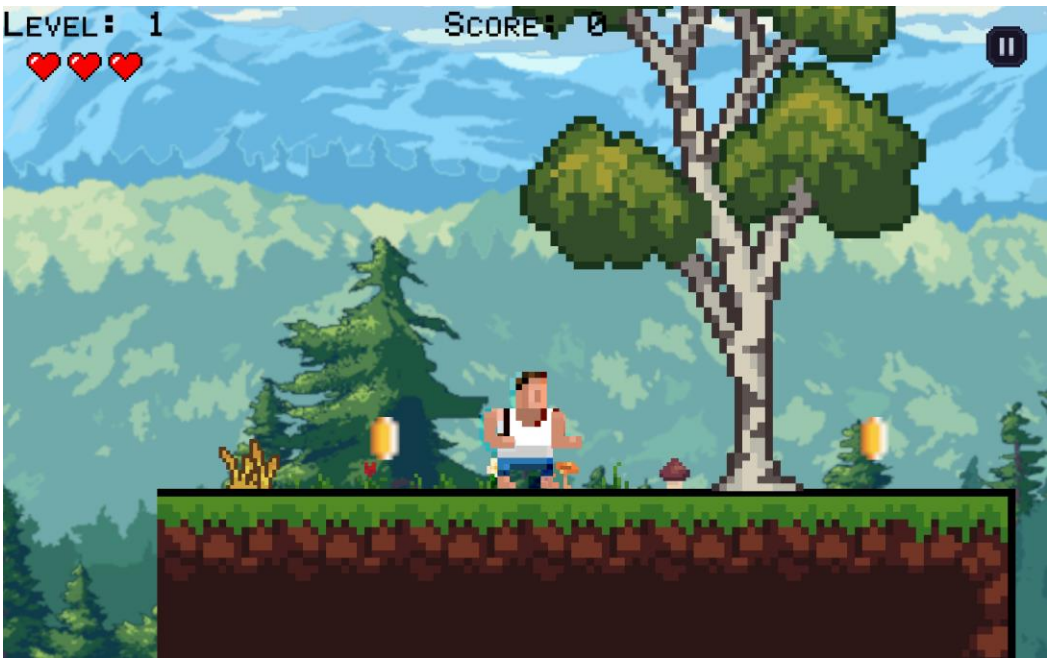


Рисунок 5.3 – Игровой процесс

Во время игрового процесса можно нажать на кнопку паузы, которая находится в правом верхнем углу экрана, чтобы приостановить игру.

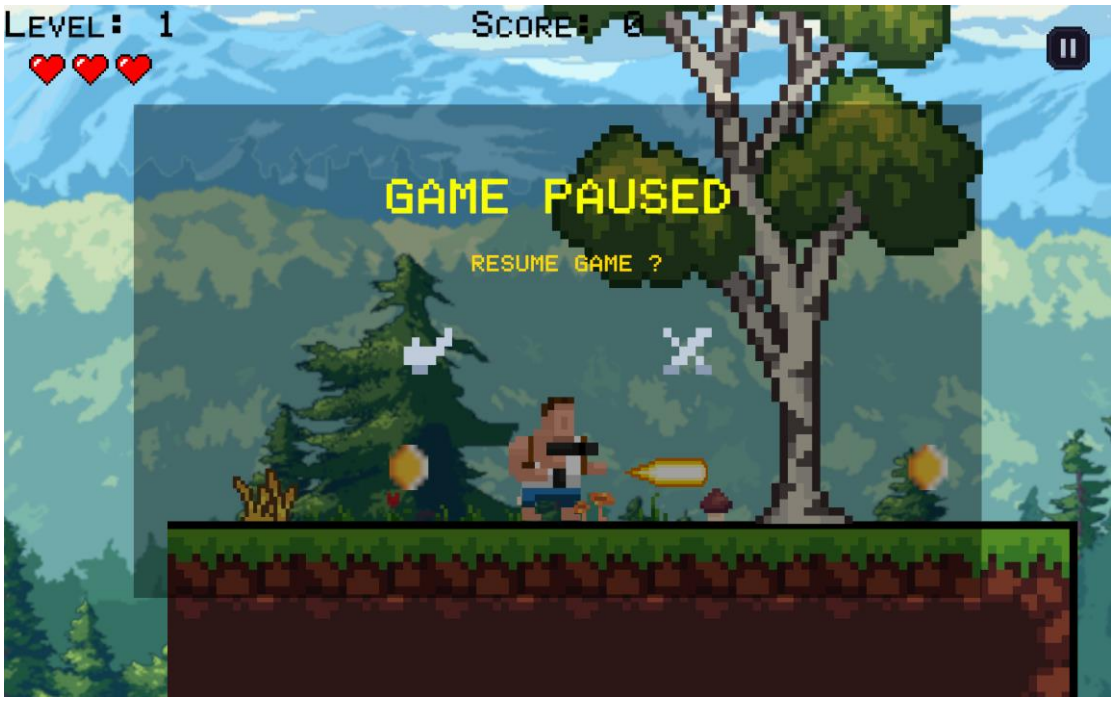


Рисунок 5.4 – Панель паузы

Пользователь может продолжить игру или вернуться в главное меню, выбрав соответствующий пункт на панели паузы, а после смерти персонажа на экране появится панель конца игры.


Рисунок 5.5 – Панель конца игры

На панели конца игры пользователь может просмотреть статистику, а именно финальный игровой счет текущей сессии и рекордный счет за все попытки. Ниже пользователь может выбрать один из пунктов, который либо начнет новую игру или вернет пользователя в главное меню.

Чтобы выйти из игры, необходимо нажать кнопку “Quit” в главном меню, после чего программное средство закроется и вернет пользователя в систему.

Игровой процесс:

1 Перемещение:

– Space (прыжок);
– A (движение влево);
– D (движение вправо).

2 Стрельба:

– "ЛКМ" (левая кнопка мыши) для стрельбы по врагам.

Каждый уровень генерируется случайным образом с использованием предустановленных фрагментов, что делает каждое прохождение уникальным.

На уровнях встречаются различные враги, с которыми необходимо сражаться. Для этого игрок может использовать оружие и собирать бонусы. Иногда игрок будет находить сундуки, которые можно разрушить и получить монеты.

Цель игры — получить как можно больше очков, собирая бонусы и побеждая врагов.

# 6 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ИГРОВОГО ПРОГРАММНОГО СРЕДСТВА С ПРОЦЕДУРНОЙ ГЕНЕРАЦИЕЙ УРОВНЕЙ НА ПЛАТФОРМЕ UNITY

## 6.1 Характеристика программного средства, разрабатываемого для реализации на рынке

В ходе дипломного проектирования будет разработано «Игровое программное средство с процедурной генерацией уровней на платформе Unity».

Целью разработки является создание игрового программного средства (2D-игры) с процедурной генерацией уровней для предоставления игрокам уникального игрового опыта, улучшения реиграбельности и расширения спектра интерактивных развлечений.

Разработка предназначена для использования в игровой индустрии, а именно для распространения на платформах цифровой дистрибуции (Steam, Epic Games Store). Игра может использоваться для досуга, развлечения, тренировки реакции и стратегического мышления.

Будет создан продукт, ориентированный на массовую игровую аудиторию, доступный для игроков с различным уровнем подготовки. Программное средство будет удовлетворять спрос на игры с высокой реиграбельностью и случайно генерируемыми уровнями. Использование алгоритмов процедурной генерации позволит минимизировать ручную работу разработчика при создании уровней. Программное средство генерирует уровни, комбинируя заранее подготовленные элементы, что обеспечивает уникальность каждого прохождения. Такая система процедурной генерации делает игровой процесс более разнообразным. Игрок сможет взаимодействовать с окружением, включая сундуки, которые можно разрушать для получения наград.

Разработкой программного средства занимается студент под руководством дипломного руководителя. Целевой аудиторией являются игроки, предпочитающие жанры платформеров, roguelike и инди-игры, в возрасте от 12 до 35 лет.

В игровой индустрии наблюдается устойчивый интерес к играм с элементами процедурной генерации уровней. Такие игры выделяются высокой реиграбельностью, благодаря уникальным уровням, а также интересом со стороны игроков, которые ценят непредсказуемость и вызов. Для разработчиков является плюсом уменьшение затрат на ручное проектирование уровней.

На рынке существует множество аналогов с различными геймплейными особенностями. Spelunky — культовый платформер с процедурной генерацией, известный своей сложностью и глубиной. Dead Cells — игра с

элементами roguelike, где каждый уровень создается заново при каждом прохождении. Terraria — песочница с процедурной генерацией мира, где игрок исследует, строит и сражается. Hollow Knight — хотя и с фиксированными уровнями, игра служит примером популярности платформеров.

Разрабатываемое программное средство сочетает в себе динамичный геймплей и реиграбельность, что делает программное средство привлекательным для широкой аудитории.

Таким образом, разрабатываемое программное средство удовлетворяет существующую потребность в динамичных играх с уникальным контентом, предоставляя игрокам разнообразный опыт и высокий уровень вовлеченности.

На основании анализа рыночной ситуации и популярных игр в жанре платформеров с процедурной генерацией можно сделать предположения о цене копии (лицензии) и годовом объеме продаж. Целевая цена копии (лицензии) примерно 20-35 белорусских рублей. Такая цена является стандартной для инди-игр, ориентированных на платформы Steam и мобильные магазины приложений. Прогнозируемый годовой объем продаж при пессимистическом сценарии равен 10000 копий, при базовом сценарии равен 50000 копий, а при оптимистическом сценарии равен 100000 копий. Прогноз основывается на аналогичных продуктах, таких как Spelunky или Dead Cells, а также учитывает, что игра будет распространяться через популярные платформы цифровой дистрибуции.

Моделью монетизации выбрана единовременная покупка копии (лицензии) приложения.

Платная модель идеально подходит для игр, ориентированных на высокое качество и насыщенность контента.

Каналами продаж выбраны две платформы. Steam — ведущая платформа для дистрибуции ПК-игр с обширной аудиторией. Epic Games Store — платформа с растущей популярностью, ориентированная на инди-разработчиков.

Были выбраны стратегии продвижения, такие как маркетинг в социальных сетях, краудфандинг, сотрудничество с игровыми блогерами и стримерами, участие в выставках и конкурсах, акции и скидки. Публикация игровых трейлеров, демонстрация геймплея и закулисных материалов на платформах, таких как YouTube, Twitter, Instagram, TikTok.

Прогнозируемый доход за первый год (базовый сценарий):

Продажа 50000 копий по средней цене 25 белорусских рублей = 1250000 белорусских рублей.

Для организации-разработчика проект становится источником прибыли, которая может быть использована на дальнейшую разработку игр, расширение команды и поддержку продукта (обновления, патчи).

Таким образом, разрабатываемое программное средство обладает значительным потенциалом для коммерческого успеха благодаря

эффективной модели монетизации, выбору каналов продаж и стратегии продвижения.

Дата расчетов – ноябрь 2024 года. Ставка рефинансирования составляет 9,5 %.

## 6.2 Расчет инвестиций в разработку программного средства для его реализации на рынке

Основой для оценки инвестиций в программное средство для его реализации на рынке является расчёт затрат на заработную плату команде разработчиков.

Основной статьей расходов на создание ПО является заработная плата разработчиков (исполнителей) проекта, в число которых принято включать инженеров-программистов, участвующих в написании кода, руководителей проекта, тестировщиков и других. Заработная плата руководителей организации и работников вспомогательных служб (инфраструктуры) учитывается в накладных расходах.

Расчёт затрат на основную заработную плату команде разработчиков производится по формуле формуле 6.1 [20]:

$$З_о = К_{пр} \sum_{i=1}^{n} З_{чi} \times t_i, \quad (6.1)$$

где $К_{пр}$ - коэффициент премии и иных стимулирующих выплат;

$З_{чi}$ – часовой оклад исполнителя;

$n$ – общее число функций;

$t_i$ - трудоёмкость работ, выполненных исполнителем i-й категории.

Часовой оклад каждого исполнителя определяется путем деления его месячного оклада на количество рабочих часов в месяц.

Размер месячного оклада исполнителей каждой категории должен либо соответствовать установленному в организации разработчики фактического размера, либо сложившемуся на рынке труда размера заработной платы исполнитель, участвующих в разработке. Также рекомендуется принять коэффициент премии иных симулирующих выплат рамки единицам так как статистике среднемесячной зарплаты уже учитываются выплаты подобного рода.

Часовая тарифная ставка ($З_ч$) рассчитывается путем деления средней заработной платы в ИТ отрасли $З_м$ на установленную при 40-часовой недельной норме рабочего времени расчетную среднемесячную норму рабочего времени в часах ($Т_р$), определяется по формуле 6.2 [20]:

$$З_ч = З_м / Т_р, \quad (6.2)$$

где $Т_р$ – среднемесячная норма рабочего времени;

$З_м$ – средняя заработная плата в ИТ отрасли.

Расчет затрат на основную заработную плату представлен в таблице 6.1.

Таблица 6.1 – Расчет затрат на основную заработную плату команды разработчиков

| Категория исполнителя | Месячный оклад, р. | Часовой оклад, р. | Трудоём-кость работ, ч. | Итого, р. |
|---|---|---|---|---|
| Программист | 3000 | 17,9 | 200 | 3580 |
| Тестировщик | 1800 | 10,7 | 50 | 535 |
| Геймдизайнер | 2500 | 14,9 | 50 | 745 |
| Художник | 2200 | 13 | 50 | 650 |
| *Итого* | | | | 5510 |
| Премия и иные стимулирующие выплаты (40%) | | | | 2204 |
| *Всего* затрат на основную заработную плату разработчиков | | | | 7714 |

Общие затраты на основную заработную плату разработчиков составили 7714 рублей.

Помимо расчёта затрат на основную заработную плату команде разработчиков, необходимо рассчитать затрат на дополнительную заработную плату ($З_о$). Эти затраты включают выплаты, предусмотренные законодательством о труде (оплата отпусков, льготных часов, времени выполнения государственных обязанностей и других выплат, не связанных с основной деятельностью исполнителей), и определяется по нормативу в процентах к основной заработной плате. Данный расчёт производится по формуле 6.3 [20]:

$$З_д = \frac{З_о \times Н_д}{100} \quad , \tag{6.3}$$

где $Н_Д$ – норматив дополнительной заработной платы;

$З_о$ – основня заработная плата.

$З_д$= 7714 × 15 / 100 = 1157,1 руб.

Отчисления в фонд социальной защиты населения ($P_{соц}$) определяются в соответствии с действующими законодательными актами по нормативу в процентном отношении к фонду основной и дополнительной зарплаты исполнителей, определенной по нормативу, установленному в целом по организации, рассчитывается по формуле 6.4 [20]:

$$P_{соц} = \frac{(З_о + З_д) \times H_{соц}}{100} \quad , \tag{6.4}$$

где $Н_{соц}$ – норматив отчислений в фонд социальной защиты населения (34%).

$Р_{соц}$= (7714 + 1157,1) × 34 / 100 = 3016,2 руб.

Расходы по статье «Прочие затраты» ($Р_{пр}$) на конкретное ПО включают затраты на приобретение и подготовку специальной научно-технической информации и специальной литературы. Определяются по нормативу, разрабатываемому в целом по научной организации, в процентах к основной заработной плате, рассчитываются по формуле 6.5 [20]:

$$Р_{пр} = \frac{З_о \times Н_{пр}}{100}, \quad (6.5)$$

где $Н_{пр}$ – норматив прочих расходов для предприятия (35%).

$Р_{пр}$= 7714 * 35 / 100 = 2699,9 руб.

Расходы на реализацию программного средства рассчитываются по формуле 6.6 [20]:

$$Р_р = \frac{З_о \times Н_р}{100}, \quad (6.6)$$

где $Н_р$ – норматив расходов на реализацию (4%).

$Р_р$= 7714 * 4 / 100 = 308,6 руб.

Общая сумма затрат на разработку и реализацию данного программного средства по всем статьям затрат, рассчитывается по формуле 6.7 [20]:

$$З_р = З_о + З_д + Р_{соц} + Р_{пр} + Р_р , \quad (6.7)$$

$З_р$ = 7714 + 1157,1 + 3016,2 + 2699,9 + 308,6 = 14895,8 руб.

Расчёт инвестиций по статьям затрат представлено в таблице 6.2.

Таблица 6.2 – Расчёт инвестиций в разработку программного средства для его реализации на рынке

| Наименование статьи затрат | Формула/таблица для расчета | Значение, р |
|---|---|---|
| 1 Основная заработная плата разработчиков ($З_о$) | Формула (6.1), табл. 6.1 | 7714 |
| 2 Дополнительная заработная плата разработчиков ($З_д$) | Формула (6.3) | 1157,1 |
| 3 Отчисления на социальные нужды ($Р_{соц}$) | Формула (6.4) | 3016,2 |
| 4 Прочие расходы ($Р_{пр}$) | Формула (6.5) | 2699,9 |
| 5 Расходы на реализацию ($Р_р$) | Формула (6.6) | 308,6 |
| 6 Общая сумма затрат на разработку и реализацию | Формула (6.7) | 14895,8 |

## 6.3 Расчет экономического эффекта от реализации программного средства на рынке

На основании анализа рыночной ситуации и популярных игр в жанре платформеров с процедурной генерацией можно сделать предположения о цене копии (лицензии) и годовом объеме продаж. Целевая цена копии (лицензии) примерно 20-35 белорусских рублей. Такая цена является стандартной для инди-игр, ориентированных на платформы Steam и мобильные магазины приложений. Для нашего продукта, ориентированного на инди-сегмент, цена 20-35 белорусских рублей является оптимальной.

Опрос игроков (проведенный через опросники в социальных сетях, таких как VK и Instagram, а также на игровых форумах). 85% респондентов указали, что готовы заплатить за игру 20-35 белорусских рублей. 10% отметили готовность к покупке за 50+ белорусских рублей. 5% предпочли бы бесплатную модель с внутриигровыми покупками.

Это подтверждает обоснованность цены 25 белорусских рублей.

Ожидаемый объем продаж базируется на данных по успешным инди-играм.

Пессимистический сценарий (минимальный спрос): 10000 копий в первый год (минимальный охват через Steam, без массовой маркетинговой кампании).

Базовый сценарий (реалистичный спрос): 50000 копий в первый год (успешное продвижение через стримеров, социальные сети и распродажи).

Оптимистический сценарий (высокий спрос): 100000 копий в первый год (попадание в топы Steam, высокая лояльность аудитории).

На основе указанных данных и анализа спроса прогнозируем объем продаж в 50000 копий при цене 25 белорусских рублей в базовом сценарии.

Прирост чистой прибыли, полученной разработчиком от реализации программного средства на рынке, можно рассчитать по формуле 6.8 [20]:

$$\Delta П_{Ч}^{Р} = (Ц_{отп} \times N - НДС) \times Р_{пр} \times (1 - \frac{Н_{п}}{100}), \qquad (6.8)$$

где $Ц_{отп}$ – отпускная цена копии (лицензии) программного средства, р.; $N$ – количество копий (лицензий) программного средства, реализуемое за год, шт.; НДС – сумма налога на добавленную стоимость, р.; $Р_{пр}$ – рентабельность продаж копий (лицензий) (98,75%); $Н_{п}$ – ставка налога на прибыль согласно действующему законодательству, % (по состоянию на ноябрь 2024 г. – 20 %).

$$\Delta П_{Ч}^{Р} = (25 \times 50000 - 208333{,}33) \times 0{,}9875 \times 0{,}8 = 822916{,}67 \text{ руб.}$$

Налог на добавленную стоимость определяется по формуле 6.9 [20]:

$$НДС = \frac{Ц_{отп} \times N \times Н_{д.с}}{100\% + Н_{д.с}}, \qquad (6.9)$$

где $Н_{д.с}$ – ставка налога на добавленную стоимость в соответствии с действующим законодательством, % (по состоянию на ноябрь 2024 г. – 20 %).

НДС = $\frac{25 \times 50000 \times 20\%}{100\% + 20\%} = \frac{250000}{1,2}$ = 208333,33 руб.

## 6.4 Расчет показателей экономической эффективности разработки и реализации программного средства на рынке

Расчет показателей экономической эффективности разработки и использования программного средства играет решающую роль в принятии стратегических решений компанией. Анализ полученных данных позволяет судить о финансовой целесообразности проекта, его влиянии на прибыльность.

Для определения экономической эффективности разработки программного средства необходимо сравнить с затратами на разработку программного средства и полученного экономического эффекта (годового прироста чистой прибыли). [20]

В данном случае годовой экономический эффект больше затрат, следовательно, необходимо воспользоваться формулой рентабельности инвестиций (Return on Investments, ROI):

ROI выражает отношение прибыли от инвестиций к изначальным инвестициям. Этот показатель позволяет оценить эффективность использования ресурсов в процессе разработки и эксплуатации программного средства. Для расчёта ROI, воспользуемся формулой 6.10 [20]:

$$ROI = \frac{\Delta П_{ч}^{р} - З_{р}}{З_{р}} \times 100\%, \quad (6.10)$$

где $\Delta П_{ч}^{р}$ – прирост чистой прибыли, полученной от реализации программного средства на рынке, р.; $З_{р}$– затраты на разработку и реализацию программного средства, р.

$ROI$ = (822916,67 – 14895,8) / (14895,8) ×100% = 5424,5%

Рассчитанный коэффициент возврата инвестиций свидетельствует о том, что проект обладает значительным потенциалом для обеспечения прибыли. Этот показатель поддерживает экономическую целесообразность разработки и продажи программного средства на рынке, а также показывает, что в течение года данное программное средство себя окупит на 5424,5%.

# ЗАКЛЮЧЕНИЕ

В рамках данного дипломного проекта было разработано игровое программное средство с процедурной генерацией уровней. Основной целью разработки являлось создание динамичной игры, в которой уровни формируются случайным образом из заранее заданных элементов, обеспечивая уникальный игровой опыт при каждом новом прохождении.

В ходе работы был проведен анализ предметной области, изучены существующие подходы к разработке игр с процедурной генерацией и определены ключевые особенности создания интерактивных и динамичных игровых механик. На основании этого анализа были выбраны инструменты и технологии, оптимальные для реализации проекта на платформе Unity.

В процессе проектирования была разработана архитектура программного средства, обеспечивающая стабильность работы алгоритмов процедурной генерации уровней, взаимодействие игровых объектов и обработку игровых событий. Реализация игрового процесса включала такие основные элементы, как управление персонажем, механика стрельбы, разрушение объектов (сундуков) и генерация уровней. Все механики были протестированы для обеспечения стабильной и корректной работы.

Итоговое программное средство соответствует поставленным требованиям и демонстрирует функциональность всех заявленных механик.

Программное средство обладает следующими особенностями:

– бесконечная генерация уровней с использованием заранее заданных фрагментов;

– динамичный игровой процесс с акцентом на управление персонажем и взаимодействие с окружением;

– подсчет очков и система рекорда, что повышает реиграбельность и интерес пользователей.

Разработанный проект имеет потенциал для дальнейшего развития, включая добавление новых типов врагов, объектов окружения, игровых механик и визуальных эффектов. Реализация таких улучшений позволит значительно расширить возможности игры и привлечь более широкую аудиторию пользователей.

Таким образом, поставленные задачи выполнены, а цели дипломного проекта достигнуты. Разработанное программное средство является готовым прототипом, который можно использовать как основу для коммерческой или любительской игры.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


[1] Антиплагиат – онлайн проверка текстов на заимствования [Электронный ресурс]. – Режим доступа: https://users.antiplagiat.ru.

[2] Литвинова, Т. В. Процедурная генерация в игровой индустрии. — М.: ИТМ, 2018. — 250 с.

[3] К. Нистром. Создание 2D-игр с использованием Unity: руководство по практическому проектированию игр. — Apress, 2018.

[4] Ковалев, А. В. Теория и практика разработки 2D-игр с использованием Unity. — М.: ВГУ, 2016. — 180 с.

[5] Стюарт, М. Программирование игр с Unity и C#: создание 2D-игр. — СПб: БХВ-Петербург, 2017. — 320 с.

[6] Щеклеин, И. В. Методы моделирования систем. — СПб.: БХВ-Петербург, 2008. — 304 с.

[7] М. Фаулер. UML. Основы. Краткое руководство. — 3-е изд. — Питер, 2017. — 208 с.

[8] Периклес Логотетис. UML и рациональный процесс разработки программного обеспечения. — ДМК, 2013. — 354 с.

[9] Диаграммы UML: описание и примеры [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/215015/.

[10] Что такое унифицированный язык моделирования? [Электронный ресурс]. – Режим доступа: https://www.lucidchart.com/pages/ru/uml.

[11] Е. В. Семенов, И. О. Петров. Проектирование и разработка игровых приложений на Unity. — Москва: Лаборатория знаний, 2019. — 210 с.

[12] О’Коннор, Дж. Алгоритмы для создания процедурных миров в играх. – М.: Вильямс, 2018. — 260 с.

[13] В. Ю. Петров. Методы создания искусственного интеллекта для компьютерных игр. — М.: ОСТ, 2019. — 250 с.

[14] Бенжамин Г. Изучаем игровые механики. – Санкт-Петербург: ДМК Пресс, 2021. – 250 с.

[15] OMG Unified Modeling Language (OMG UML). [Электронный ресурс]. – Режим доступа: https://www.omg.org/spec/UML/.

[16] А. Н. Тарасов. Диаграммы UML для анализа и проектирования программных систем. — М.: Горячая линия-Телеком, 2016. — 190 с.

[17] Кирсанова, Т. С. Разработка игр: от концепции до реализации. — Санкт-Петербург: Питер, 2020. — 400 с.

[18] Р. Шнайдерман. Интерфейс пользователя. Основы проектирования. — М.: Лори, 2016. — 640 с.

[19] И. Э. Миллер. Проектирование и разработка компьютерных игр. — К.: Диалектика, 2017. — 352 с.

[20] В. Г. Горовой [и др.]. Экономика проектных решений: методические указания по экономическому обоснованию дипломных проектов: учеб.-метод. пособие – Минск: БГУИР, 2021. – 107 с.

# ПРИЛОЖЕНИЕ А
## (обязательное)
## Листинг программного средства

```
using UnityEngine;

public class Area : MonoBehaviour
{
    public Transform Begin;
    public Transform End;

    public AnimationCurve AreaFromDistance;

    private void Start()
    {

    }
}

using System;
using UnityEngine;
using UnityEngine.Audio;

public class AudioManager : MonoBehaviour
{
    public static AudioManager instance;

    public Sound[] sounds; // Массив звуков
    public AudioMixer sfxMixer; // Микшер для звуковых эффектов
    public AudioMixer musicMixer; // Микшер для музыки

    void Awake()
    {
        if (instance != null)
        {
            Destroy(gameObject); // Если уже есть экземпляр, удаляем новый
        }
        else
        {
            instance = this; // Если нет, присваиваем текущий
            DontDestroyOnLoad(gameObject); // Оставляем объект между сценами
        }

        foreach (Sound s in sounds)
        {
            s.source = gameObject.AddComponent<AudioSource>(); // Добавляем компонент
AudioSource
            s.source.clip = s.clip; // Назначаем аудиофайл
            s.source.outputAudioMixerGroup = s.group; // Назначаем группу микшера
            s.source.volume = s.volume; // Настройка громкости
            s.source.pitch = s.pitch; // Настройка пича
            s.source.loop = s.loop; // Настройка зацикливания
        }
    }

    // Метод для воспроизведения музыки (останавливает предыдущую из той же группы)
    public void PlayMusic(string musicName)
    {
```

```
        Sound newMusic = Array.Find(sounds, item => item.name == musicName);
        if (newMusic == null)
        {
            Debug.LogWarning("Music not found: " + musicName);
            return;
        }

        // Останавливаем все звуки из той же группы микшера
        foreach (Sound s in sounds)
        {
            if (s.group == newMusic.group && s.source.isPlaying)
            {
                s.source.Stop();
            }
        }

        // Воспроизводим новую музыку
        newMusic.source.Play();
    }

    // Метод для воспроизведения звукового эффекта
    public void PlaySFX(string soundName)
    {
        Sound s = Array.Find(sounds, item => item.name == soundName);
        if (s == null)
        {
            Debug.LogWarning("Sound effect not found: " + soundName);
            return;
        }

        s.source.Play();
    }
}

using System.Drawing;
using UnityEngine;

public class Coin : MonoBehaviour
{
    public static int numberOfPoints;
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.transform.tag == "Player")
        {
            PlayerManager.numberOfPoints += 100;
            AudioManager.instance.PlaySFX("Coins");
            Destroy(gameObject);
        }
    }
}

using UnityEngine;
using System.Collections;

public class ChestWithCoins : MonoBehaviour
{
    [SerializeField] private GameObject coinPrefab; // Префаб монеты
    [SerializeField] private int numberOfCoins = 10; // Количество монет
    [SerializeField] private float spawnRadius = 1f; // Радиус, в котором появляются
монеты
```

```
    private Animator chestAnimator; // Ссылка на аниматор
    private bool isOpened = false; // Флаг, чтобы сундук открывался только один раз

    private void Start()
    {
        chestAnimator = GetComponent<Animator>();
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        // Проверяем, что объект, который столкнулся с сундуком — это пуля
        if (other.CompareTag("PlayerBullet") && !isOpened)
        {
            Destroy(other.gameObject); // Уничтожаем пулю
            isOpened = true;
            // Проигрываем анимацию открытия сундука
            chestAnimator.SetTrigger("Open");

            // Начинаем корутину, которая ждет окончания анимации
            StartCoroutine(SpawnCoinsAfterAnimation());
        }
    }

    private IEnumerator SpawnCoinsAfterAnimation()
    {
        // Получаем информацию о текущей анимации
        AnimatorStateInfo stateInfo = chestAnimator.GetCurrentAnimatorStateInfo(0);

        // Ждем длительность текущей анимации
        yield return new WaitForSeconds(stateInfo.length);

        // После окончания анимации спавним все монеты
        for (int i = 0; i < numberOfCoins; i++)
        {
            // Вычисляем случайную позицию для монеты в пределах радиуса
            Vector3 spawnPosition = transform.position + new Vector3(
                Random.Range(-spawnRadius, spawnRadius), // Случайное смещение по X
                Random.Range(0, spawnRadius), // Случайное смещение по Y (только
вверх)
                0f);

            // Спавним монету
            Instantiate(coinPrefab, spawnPosition, Quaternion.identity);
        }

        // Удаляем сундук
        Destroy(gameObject);
    }
}

using UnityEngine;
using UnityEngine.SceneManagement;

public class Heart : MonoBehaviour
{
    [SerializeField] private int scoreReward = 1000; // Очки за подбор, если здоровье
на максимуме

    private void OnTriggerEnter2D(Collider2D collision)
    {
```

```
        if (collision.CompareTag("Player")) // Проверяем, столкнулся ли игрок
        {
            // Проверяем, находится ли здоровье на максимуме
            if (HealthManager.health == HealthManager.instance.maxHealth)
            {
                // Если здоровье на максимуме, добавляем очки
                PlayerManager.numberOfPoints += scoreReward;
            }
            else
            {
                // Если здоровье не на максимуме, добавляем 1 здоровье
                HealthManager.instance.AddHealth(1);
            }

            // Уничтожаем объект сердца
            Destroy(gameObject);
        }
    }
}

using UnityEngine;
using TMPro;

public class LevelCounter : MonoBehaviour
{
    [SerializeField] private Transform player; // Ссылка на объект игрока
    private float lastCheckpointX = 0f; // Последняя точка обновления уровня
    private int level = 1; // Начальный уровень

    [SerializeField] private TMP_Text levelText; // UI-текст для отображения уровня

    void Start()
    {
        // Устанавливаем начальное положение игрока
        lastCheckpointX = player.position.x;

        // Обновляем UI
        UpdateLevelUI();
    }

    void Update()
    {
        // Проверяем, переместился ли игрок на 50 единиц по X
        if (player.position.x - lastCheckpointX >= 50f)
        {
            level++; // Увеличиваем уровень
            lastCheckpointX = player.position.x; // Обновляем контрольную точку
            UpdateLevelUI(); // Обновляем текст уровня в UI
        }
    }

    // Метод для обновления текста уровня
    private void UpdateLevelUI()
    {
        if (levelText != null)
        {
            levelText.text = "Level: " + level; // Устанавливаем текст уровня
        }
    }
}
```

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

public class LevelManager : MonoBehaviour
{
    public Transform Player;
    public Area[] AreaPrefabs;
    public Area FirstArea;

    private List<Area> spawnedAreas = new List<Area>();

    private void Start()
    {
        spawnedAreas.Add(FirstArea);
    }

    private void Update()
    {
        if (Player.position.x > spawnedAreas[spawnedAreas.Count - 1].End.position.x -
15)
        {
            SpawnArea();
        }
    }

    private void SpawnArea()
    {
        Area newArea = Instantiate(GetRandomArea());
        newArea.transform.position = spawnedAreas[spawnedAreas.Count -
1].End.position - newArea.Begin.localPosition;
        spawnedAreas.Add(newArea);

        if (spawnedAreas.Count >= 3)
        {
            Destroy(spawnedAreas[0].gameObject);
            spawnedAreas.RemoveAt(0);
        }
    }

    private Area GetRandomArea()
    {
        List<float> areas = new List<float>();
        for (int i = 0; i < AreaPrefabs.Length; i++)
        {

areas.Add(AreaPrefabs[i].AreaFromDistance.Evaluate(Player.transform.position.x));
        }

        float value = Random.Range(0, areas.Sum());
        float sum = 0;

        for (int i = 0; i < areas.Count; i++)
        {
            sum += areas[i];
            if (value < sum)
            {
                return AreaPrefabs[i];
            }
```

```
        }

        return AreaPrefabs[AreaPrefabs.Length - 1];
    }
}

using UnityEngine.SceneManagement;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Audio;

public class MenuEvents : MonoBehaviour
{
    public Slider volumeSlider;
    public AudioMixer mixer;
    private float value;

    private void Start()
    {
        Time.timeScale = 1;
        mixer.GetFloat("volume", out value);
        volumeSlider.value = value;
        AudioManager.instance.PlayMusic("Main Menu");
    }
    public void SetVolume()
    {
        mixer.SetFloat("volume", volumeSlider.value);
    }

    public void LoadGame(int index)
    {
        AudioManager.instance.PlayMusic("Game");
        SceneManager.LoadScene(index);
    }

    public void ExitGame(int index)
    {
        Application.Quit();
    }
}

using UnityEngine;

public class Parallax : MonoBehaviour
{
    public Transform mainCam;
    public Transform middleBG;
    public Transform sideBG;

    public float length = 28.5f;

    void Update()
    {
        if (mainCam.position.x > middleBG.position.x)
            sideBG.position = middleBG.position + Vector3.right * length;

        if (mainCam.position.x < middleBG.position.x)
            sideBG.position = middleBG.position + Vector3.left * length;
```

```
        if (mainCam.position.x > sideBG.position.x || mainCam.position.x <
sideBG.position.x)
        {
            Transform z = middleBG;
            middleBG = sideBG;
            sideBG = z;
        }

    }
}

using UnityEngine.Audio;
using UnityEngine;

[System.Serializable]
public class Sound
{
    public string name;
    public AudioClip clip;
    public AudioMixerGroup group; // Группа для микшера (для музыки или эффектов)
    public float volume;
    public float pitch;
    public bool loop;

    [HideInInspector] public AudioSource source;
}

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

public class Bullet : MonoBehaviour
{
    void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.tag == "Enem")
        {
            AudioManager.instance.PlaySFX("Enemy");
            collision.GetComponent<Enemy>().TakeDamage(25);
            Destroy(gameObject);
            PlayerManager.numberOfPoints += 100;
        }
    }
}

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

public class HealthManager : MonoBehaviour
{
    public static HealthManager instance;
    public static int health = 3;
    public int maxHealth = 3;

    public Image[] hearts;
    public Sprite fullHeart;
    public Sprite emptyHeart;
```

```
    void Awake()
    {
        health = maxHealth;

        if (instance == null)
        {
            instance = this;
        }
        else
        {
            Destroy(gameObject);
        }
    }
    // Update is called once per frame
    void Update()
    {
        foreach (Image img in hearts)
        {
            img.sprite = emptyHeart;
        }
        for (int i = 0; i < health; i++)
        {
            hearts[i].sprite = fullHeart;
        }
    }

    public void AddHealth(int amount)
    {
        health += amount; // Увеличиваем здоровье
        if (health > maxHealth)
        {
            health = maxHealth; // Ограничиваем значение здоровьем
        }
    }
}

using UnityEngine;

public class PlayerMovement : MonoBehaviour
{
    PlayerControls controls;

    float direction = 0;
    public float speed = 400;
    public Rigidbody2D playerRB;
    public Animator animator;
    public bool isFacingRight = true;
    public float jumpForce = 5;
    bool isGrounded;
    public Transform groundCheck;
    public LayerMask groundLayer;
    int numberOfJumps = 0;
    private void Awake()
    {
        controls = new PlayerControls();
        controls.Enable();
        DontDestroyOnLoad(gameObject);

        controls.Land.Move.performed += ctx =>
        {
```

```
            direction = ctx.ReadValue<float>();
        };

        controls.Land.Jump.performed += ctx => Jump();
    }

    private void FixedUpdate()
    {
        isGrounded = Physics2D.OverlapCircle(groundCheck.position, 0.1f,
groundLayer);
        animator.SetBool("isGrounded", isGrounded);

        playerRB.velocity = new Vector2(direction * speed * Time.fixedDeltaTime,
playerRB.velocity.y);
        animator.SetFloat("speed", Mathf.Abs(direction));

        if (isFacingRight && direction < 0 || !isFacingRight && direction > 0)
            Flip();
    }

    void Flip()
    {
        isFacingRight = !isFacingRight;
        transform.localScale = new Vector2(transform.localScale.x * -1,
transform.localScale.y);
    }

    void Jump()
    {
        if (isGrounded)
        {
            numberOfJumps = 0;
            playerRB.velocity = new Vector2(playerRB.velocity.x, jumpForce);
            numberOfJumps++;
            AudioManager.instance.PlaySFX("Jump");
        }
        else
        {
            if(numberOfJumps == 1)
            {
                playerRB.velocity = new Vector2(playerRB.velocity.x, jumpForce);
                numberOfJumps++;
                AudioManager.instance.PlaySFX("Jump");
            }
        }
    }
}

using UnityEngine;
using System.Collections;
using System;
public class PlayerCollision : MonoBehaviour
{
    public bool isInvincible = false;
    private bool isDead = false;
    [SerializeField] private float deathYLevel = -5f;
    [SerializeField] private GameObject player;
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.transform.tag == "Enemy")
```

```
        {
            TakeDamage();
        }
    }
    private void Update()
    {
        if (!isDead && player.transform.position.y < deathYLevel)
        {
            GameOver();
        }
    }
    public void GameOver()
    {
        isDead = true;
        Death();
    }

    IEnumerator GetHurt()
    {
        Physics2D.IgnoreLayerCollision(3, 8);
        GetComponent<Animator>().SetLayerWeight(1, 1);
        isInvincible = true;
        yield return new WaitForSeconds(3);
        isInvincible = false;
        GetComponent<Animator>().SetLayerWeight(1, 0);
        Physics2D.IgnoreLayerCollision(3, 8, false);
    }
    public void TakeDamage()
    {
        HealthManager.health--;
        if (HealthManager.health <= 0)
        {
            PlayerManager.isGameOver = true;
            AudioManager.instance.PlayMusic("Game Over");
            gameObject.SetActive(false);
        }
        else
        {
            StartCoroutine(GetHurt());
        }
    }

    public void Death()
    {
        HealthManager.health-=3;
        if (HealthManager.health <= 0)
        {
            PlayerManager.isGameOver = true;
            AudioManager.instance.PlayMusic("Game Over");
            gameObject.SetActive(false);
        }
    }
}

using UnityEngine.SceneManagement;
using UnityEngine;
using Cinemachine;
using TMPro;
using Unity.VisualScripting;
```

```
public class PlayerManager : MonoBehaviour
{
    public static bool isGameOver;
    public GameObject gameOverScreen;
    public static int numberOfPoints;
    public TextMeshProUGUI scoreText;
    public GameObject PauseMenuPanel;
    public GameObject Enemy;
    public GameObject Player;
    public TextMeshProUGUI highScoreText; // Текст для рекорда
    public TextMeshProUGUI finalScoreText; // Текст для текущего счета

    private const string HighScoreKey = "HighScore"; // Ключ для рекорда


    private void Awake()
    {
        isGameOver = false;
    }

    void Update()
    {
        scoreText.text = "Score: " + numberOfPoints;
        if (isGameOver)
        {
            GameObject[] enemies = GameObject.FindGameObjectsWithTag("Enem");
            foreach (GameObject Enemy in enemies)
            {
                Destroy(Enemy);
            }

            ShowGameOverScreen(); // Показываем экран конца игры
        }
    }

    private void ShowGameOverScreen()
    {
        CheckAndSaveHighScore(); // Проверяем и обновляем рекорд
        int highScore = GetHighScore(); // Получаем рекорд

        // Отображаем текущий счет и рекорд на экране конца игры
        finalScoreText.text = "Final Score: " + numberOfPoints;
        highScoreText.text = "High Score: " + highScore;

        gameOverScreen.SetActive(true); // Показываем экран конца игры
    }

    private void CheckAndSaveHighScore()
    {
        int highScore = PlayerPrefs.GetInt(HighScoreKey, 0); // Загружаем рекорд
        if (numberOfPoints > highScore)
        {
            // Обновляем рекорд, если текущий счет больше
            PlayerPrefs.SetInt(HighScoreKey, numberOfPoints);
            PlayerPrefs.Save();
        }
    }

    private int GetHighScore()
    {
```

```
        return PlayerPrefs.GetInt(HighScoreKey, 0); // Получаем сохраненный рекорд
    }

    public void ReplayLevel()
    {
        AudioManager.instance.PlayMusic("Game");
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
        numberOfPoints = 0;
    }

    public void Exit(int index)
    {
        GameObject[] players = GameObject.FindGameObjectsWithTag("Player");
        foreach (GameObject Player in players)
        {
            Player.SetActive(false);
        }
        AudioManager.instance.PlayMusic("Main Menu");
        SceneManager.LoadScene(index);
        numberOfPoints = 0;
    }

    public void PauseGame()
    {
        AudioManager.instance.PlayMusic("Pause");
        Time.timeScale = 0;
        PauseMenuPanel.SetActive(true);
    }
    public void ResumeGame()
    {
        AudioManager.instance.PlayMusic("Game");
        Time.timeScale = 1;
        PauseMenuPanel.SetActive(false);
    }
}

using UnityEngine;

public class PlayerShoot : MonoBehaviour
{
    PlayerControls controls;
    public Animator animator;

    public GameObject bullet;
    public Transform bulletHole;
    public float force = 4000;


    void Awake()
    {
        controls = new PlayerControls();
        controls.Enable();

        controls.Land.Shoot.performed += ctx => Fire();
    }

    void Fire()
    {
        animator.SetTrigger("shoot");
        AudioManager.instance.PlaySFX("Shoot");
```

```
        GameObject go = Instantiate(bullet, bulletHole.position,
bullet.transform.rotation);
        if (GetComponent<PlayerMovement>().isFacingRight)
            go.GetComponent<Rigidbody2D>().AddForce(Vector2.right * force);
        else
            go.GetComponent<Rigidbody2D>().AddForce(Vector2.left * force);

        Destroy(go, 1.2f);
    }

    private void OnEnable()
    {
        controls.Enable();
    }
    private void OnDisable()
    {
        controls.Disable();
    }
}

using UnityEngine;
using UnityEngine.UI;

public class Enemy : MonoBehaviour
{
    GameObject player;
    public Transform borderCheck;
    public int enemyHP = 100;
    public Animator animator;

    void Start()
    {
        player = GameObject.FindGameObjectWithTag("Player");
        Physics2D.IgnoreCollision(player.GetComponent<Collider2D>(),
GetComponent<Collider2D>());
    }

    void Update()
    {
        if (player.transform.position.x > transform.position.x)
            transform.localScale = new Vector2(-2f, 2f);
        else
            transform.localScale = new Vector2(2f, 2f);

    }
    public void TakeDamage(int damageAmount)
    {
        enemyHP -= damageAmount;
        if (enemyHP > 0)
        {
            animator.SetTrigger("damage");
            animator.SetBool("isChasing", true);
        }
        else
        {
            animator.SetTrigger("death");
            GetComponent<CapsuleCollider2D>().enabled = false;
            this.enabled = false;
        }
```

```
    }

    public void PlayerDamage()
    {
        if (!player.GetComponent<PlayerCollision>().isInvincible)
            player.GetComponent<PlayerCollision>().TakeDamage();
    }
}

using UnityEngine;

public class IdleState : StateMachineBehaviour
{
    Transform player;
    Transform borderCheck;
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo,
int layerIndex)
    {
        player = GameObject.FindGameObjectWithTag("Player").transform;
        borderCheck = animator.GetComponent<Enemy>().borderCheck;
    }

    override public void OnStateUpdate(Animator animator, AnimatorStateInfo
stateInfo, int layerIndex)
    {
        if (Physics2D.Raycast(borderCheck.position, Vector2.down, 1) == false)
            return;

        float distance = Vector2.Distance(player.position,
animator.transform.position);
        if (distance < 3)
            animator.SetBool("isChasing", true);
    }
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo,
int layerIndex)
    {
        AudioManager.instance.PlaySFX("Enemy");
    }
}

using UnityEngine;

public class ChaseState : StateMachineBehaviour
{
    Transform player;
    public float speed = 3;
    Transform borderCheck;

    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo,
int layerIndex)
    {
        player = GameObject.FindGameObjectWithTag("Player").transform;
        borderCheck = animator.GetComponent<Enemy>().borderCheck;
    }
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo
stateInfo, int layerIndex)
    {
        Vector2 newPos = new Vector2(player.position.x,
animator.transform.position.y);
```

```
        animator.transform.position =
Vector2.MoveTowards(animator.transform.position, newPos, speed * Time.deltaTime);
        if (Physics2D.Raycast(borderCheck.position, Vector2.down, 1) == false)
            animator.SetBool("isChasing", false);

        float distance = Vector2.Distance(player.position,
animator.transform.position);
        if (distance < 1)
            animator.SetBool("isAttacking", true);
    }
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo,
int layerIndex)
    {

    }

}

using UnityEngine;

public class AttackState : StateMachineBehaviour
{
    Transform player;
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo,
int layerIndex)
    {
        player = GameObject.FindGameObjectWithTag("Player").transform;
    }
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo
stateInfo, int layerIndex)
    {
        float distance = Vector2.Distance(player.position,
animator.transform.position);
        if (distance > 1)
            animator.SetBool("isAttacking", false);
    }
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo,
int layerIndex)
    {

    }

}

using UnityEngine;

public class EnemyPatrolX : MonoBehaviour
{
    [Header("Patrol Settings")]
    public float patrolSpeed = 2f; // Скорость патрулирования
    public float patrolDistance = 3f; // Максимальное расстояние от начальной позиции

    private float initialX; // Начальная позиция по оси X
    private bool movingRight = true; // Направление движения: вправо или влево

    private void Start()
    {
        initialX = transform.position.x; // Сохраняем начальную позицию
    }
```

```
    private void Update()
    {
        Patrol();
    }

    private void Patrol()
    {
        // Двигаемся вправо или влево в зависимости от направления
        if (movingRight)
        {
            transform.Translate(Vector2.right * patrolSpeed * Time.deltaTime);

            // Если достигли правой границы, меняем направление
            if (transform.position.x >= initialX + patrolDistance)
            {
                movingRight = false;
            }
        }
        else
        {
            transform.Translate(Vector2.left * patrolSpeed * Time.deltaTime);

            // Если достигли левой границы, меняем направление
            if (transform.position.x <= initialX - patrolDistance)
            {
                movingRight = true;
            }
        }
    }

    private void OnDrawGizmos()
    {
        // Отображаем границы патрулирования в редакторе
        Gizmos.color = Color.blue;
        Gizmos.DrawLine(new Vector3(transform.position.x + patrolDistance,
transform.position.y, 0),
                        new Vector3(transform.position.x - patrolDistance,
transform.position.y, 0));
    }
}

using UnityEngine;

public class EnemyPatrulY : MonoBehaviour
{
    [Header("Patrol Settings")]
    public float patrolSpeed = 2f; // Скорость патрулирования
    public float patrolDistance = 3f; // Максимальное расстояние от начальной позиции

    private float initialY; // Начальная позиция по оси Y
    private bool movingUp = true; // Направление движения: вверх или вниз

    private void Start()
    {
        initialY = transform.position.y; // Сохраняем начальную позицию
    }

    private void Update()
    {
        Patrol();
```

```
    }

    private void Patrol()
    {
        // Двигаемся вверх или вниз в зависимости от направления
        if (movingUp)
        {
            transform.Translate(Vector2.up * patrolSpeed * Time.deltaTime);

            // Если достигли верхней границы, меняем направление
            if (transform.position.y >= initialY + patrolDistance)
            {
                movingUp = false;
            }
        }
        else
        {
            transform.Translate(Vector2.down * patrolSpeed * Time.deltaTime);

            // Если достигли нижней границы, меняем направление
            if (transform.position.y <= initialY - patrolDistance)
            {
                movingUp = true;
            }
        }
    }

    private void OnDrawGizmos()
    {
        // Отображаем границы патрулирования в редакторе
        Gizmos.color = Color.red;
        Gizmos.DrawLine(new Vector3(transform.position.x, transform.position.y +
patrolDistance, 0),
                        new Vector3(transform.position.x, transform.position.y -
patrolDistance, 0));
    }
}
```

| Обозначение | Наименование | Примечание |
|---|---|---|
| | Текстовые документы | |
| | | |
| БГУИР ДП 1-40 01 01 01 010 ПЗ | Пояснительная записка | 81 с |
| | Отзыв руководителя | |
| | Рецензия | |
| | Отчёт о проверке на заимствования | |
| | | |
| | Графические документы | |
| | | |
| ГУИР.181071.001 ПЛ | Диаграмма вариантов использования | Формат А1 |
| | Плакат | |
| ГУИР.181071.002 ПЛ | Диаграмма состояний | Формат А1 |
| | Плакат | |
| ГУИР.181071.003 ПЛ | Диаграмма классов | Формат А1 |
| | Плакат | |
| | | |
| ГУИР.181071.004 СА | Процедурная генерация уровня | Формат А1 |
| | Схема алгоритма | |
| ГУИР.181071.005 СА | Поведение врага | Формат А1 |
| | Схема алгоритма | |
| ГУИР.181071.006 СА | Взаимодействие персонажа с объектами | Формат А1 |
| | игры | |
| | Схема алгоритма | |
| | | |
| | | |
| | | |
| | | |
| | | |

| Изм | Лист | № докум. | Подп. | Дата | БГУИР ДП 1-40 01 01 01 010 Д1 | | |
|---|---|---|---|---|---|---|---|
| Разраб. | | Бабич Д.И. | | | Игровое программное средство с процедурной генерацией уровней на платформе Unity<br>Ведомость дипломного проекта | Лит. | Лист | Листов |
| Пров. | | Сицко А.Л. | | | | Т | 81 | 81 |
| Т.контр. | | Сицко А.Л. | | | | Кафедра ИСиТ, гр. 181071 | | |
| Н.контр. | | Шелягович А.С. | | | | | | |
| Утв. | | | | | | | | |