**Cодержание**

[Введение 4](#_Toc168331158)

[1. Алгоритмизация 6](#_Toc168331159)

[1.1. Постановка задачи 6](#_Toc168331160)

[1.1.1 Описание задачи 6](#_Toc168331161)

[1.2.1 Сравнительный анализ игр жанра «Rogue-Like» 8](#_Toc168331162)

[1.3.1 Общий алгоритм реализации проекта 14](#_Toc168331163)

[1.4.1 Анализ игровых движков. 15](#_Toc168331164)

[2. Программирование 20](#_Toc168331165)

[2.1. Начало работы 20](#_Toc168331166)

[2.2. Создание основных сцен 20](#_Toc168331167)

[2.2.1. Главное меню 20](#_Toc168331168)

[2.2.2. Генерация подземелий 21](#_Toc168331169)

[2.2.3. Функции Игрока 25](#_Toc168331170)

[2.3. Враги (Слаймы) 28](#_Toc168331171)

[2.3.1 Атака слайма и звуки 32](#_Toc168331172)

[2.4. Система лечения 35](#_Toc168331173)

[2.4.1. Система лута 37](#_Toc168331174)

[2.5 Камера 38](#_Toc168331175)

[2.5.1. События 39](#_Toc168331176)

[2.5.2. Звуки 41](#_Toc168331177)

[2.5.3. Пауза 42](#_Toc168331178)

[2.5.4. Факелы 44](#_Toc168331179)

[3.Тестирование 46](#_Toc168331180)

[Список использованных источников 57](#_Toc168331181)

[Приложение А 58](#_Toc168331182)

[Скриншоты игры 58](#_Toc168331183)

[Приложение Б 60](#_Toc168331184)

# **Введение**

Разработка игр – это очень интересное и прогрессивное направление в области IT. Ведь благодаря игровой индустрии многие специалисты из различных областей могут объединиться, для создания одного, масштабного проекта. Благодаря игровой индустрии различные производители компьютерного железа каждый год выпускают всё более мощное оборудование, что позже эволюционируют в аппаратные средства для серверных оборудований или мейнфреймы.

На текущий момент общество неразрывно связано с системами геоинформационных ресурсов или социальными сетями, мессенджерами, системами получения государственных услуг и т. д. Все ранее перечисленное существенно зависит от мощностей аппаратной составляющей серверного оборудования. С каждым годом, компьютерное железо становится всё отказоустойчивее и производительнее, что положительно сказывается для пользовательского опыта

Систем типа «клиент – сервер».

Закончив лирическое отступление и вернувшись к тебе проектирования игры, можно отметить, что современные игры, для различных платформ, необычайно сложны. Особенно, самый сложный и ответственный этап разработки – является именно планирование проекта. Ведь, проектирование включает в себя анализ рынка компьютерной и игровой индустрии, отбор оптимального игрового движка, разработку сюжета и визуальной составляющей, оценку экономической эффективности и планирование сроков реализации. Будучи студентом, изучающим информатику, можно изучать разработку игр, потому что эта область требует некоторых навыков, которые уже есть, и в то же время вы можете их улучшить.

В рамках текущего проекта, будет проектироваться компьютерная игра в жанре «Rogue - Like», так как, это очень интересный жанр в игровой индустрии и для потребителя требуется минимальный порог вхождения, не требующий особых знаний и навыков для комфортного игрового процесса, не как в играх серии Dota или Counter-Strike, где необходимо знать очень много тактик, наизусть знать локации и зоны игровых сражений, идеальное умение в использовании основных персонажей или типа игрового вооружения и снаряжения.

Для простых пользователей – это может вызовет затруднение и дискомфорт во время игрового сеанса, что оставит негативные эмоции и усилит усталость. Ведь игры, в первую очередь, должны помочь человеку расслабится и погрузится в игровой мир, для получения положительных эмоций и хорошего отдыха. Сегодня с вычислительными устройствами человек сталкивается практически постоянно.

Если еще полвека лет назад компьютеры можно было встретить только в научно-исследовательских центрах, институтах и на крупных предприятиях, то сейчас они проникли почти во все сферы жизни. К одной из таких сфер относится индустрия развлечений. Практически с самого своего зарождения, разработка компьютерных игр стала одной из самых трудоемких задач для различных видов специалистов из сферы информационных технологий и смежных областей.

Однако, как и разработка промышленного и другого программного обеспечения, создание компьютерных игр основывается на тех же самых циклах разработки, где нужен поэтапный подход, включающий сюда прогнозирование, работу с целевой аудиторией, экономические оценки, постановку задачи, разработку инструментария и самого продукта, создание мультимедийных ресурсов, контроль качества и прочее, которые в настоящее время не доведены до цикличности и структуризации. Данный аспект определил актуальность вопроса исследования выпускной квалификационной работы. Объектом исследования является проектирование компьютерных игр. Предмет исследования – технологии разработки компьютерной 2D игры жанра «Rogue – Like».

Цель работы – спроектировать концепт компьютерной игры жанра «Rogue - Like».

Для достижения поставленной цели были определены следующие задачи:

1. проанализировать предметную область проекта;

2. проанализировать программные средства и технологии создания игр;

3. сформировать требования к игровому ПО;

4. спроектировать эскиз визуальной составляющей игры Методы решения главной задачи работы – экспериментальный, методы системного анализа, моделирования, нормализации данных, теоретический.

Теоретико-методологической основой исследования послужили работы отечественных и зарубежных ученых в области IT – проектирования в сфере разработки инструментария, учебные пособия, монографии, научные статьи, ресурсы сети Интернет.

# **1. Алгоритмизация**

## 1.1. Постановка задачи

### 1.1.1 Описание задачи

Компьютерные игры в основном состоят из программного обеспечения, которое можно разделить на две широкие функции: игровой движок, управляющий ходом игры, и системы, поддерживающие живую работу.

В то время как небольшие независимые игры требуют всего понемногу, крупные игровые студии, как правило, имеют масштабы для поддержки собственных команд инженеров-программистов, которые разрабатывают различные элементы игры с нуля. В такой команде часто есть три основные роли:

- Программист геймплея – инженеры в этой роли сосредоточены на том, как игра выглядит с точки зрения пользователя, что включает в себя добавление элементов дизайна и обеспечение удобства системы.

- Графический программист – инженеры в этой роли отвечают за создание эффективного механизма рендеринга или настройку технических функций, чтобы игра хорошо работала на разных системах.

- Инженер-программист внутренних систем – инженеры в этой роли пишут компьютерный код для систем игры, которые игроки не видят напрямую; например, логика на стороне сервера и функции входа или выставления счетов в крупной онлайн-игре.

Помимо этих ролей программирования, есть должности, которые поддерживают живые операции, такие как инженеры по сетям, безопасности и эксплуатации (они относятся к системному администрированию), а также различные члены художественных команд, включая аниматоров, создателей моделей персонажей и художников по окружению. Благодаря развитию рынка инди-игр многие разработчики компьютерных игр получили возможность работать над своими игровыми проектами без финансовых и юридических обязательств перед издательскими компаниями. Инди-игры (англ. Indie games, от англ. independent video games - «независимые компьютерные игры») – это компьютерные игры, созданные отдельными разработчиками или небольшими коллективами без финансовой поддержки издателя компьютерных игр. Распространение осуществляется посредством каналов цифровой дистрибуции. Масштаб явлений, связанных с инди-играми, ощутимо возрастает со второй половины 2000-х годов, в основном ввиду развития новых способов онлайн дистрибуции и средств разработки.

Компьютерные игры оказали настолько значительное влияние на общество, что в последнее время в информационных технологиях наблюдается устойчивая тенденция к геймификации неигрового прикладного ПО. Так, в некоторых европейских школах для обучения стали использовать известную игру Minecraft, а для нужд армий стали создавать специальные программные тренажеры для обучения солдат. Компьютерные игры так же с 2011 года официально признаны правительством США и американским Национальным фондом отдельным видом искусства, наряду с театром и кино, а в России киберспорт официально был причислен к видам спорта. Из всего этого следует, что компьютерные игры плотно влились в нашу нынешнюю жизнь.

Более того, сфера их использования за последние 10 лет сильно выросла, теперь игры используют не только для развлечения и отдыха, но и для обучения людей и проведения научных исследований.

Компьютерные игры классифицируют по нескольким основным признакам:

− жанр;

− количество игроков;

− визуальное представление;

− платформа.

Жанр игры определяется целью и основной механикой игры. Основные жанры представлены в таблице 1.

Таблица 1.1 – Жанровая классификация

|  |  |  |
| --- | --- | --- |
| № п/п | Имя поля | Описание поля |
| 1 | id | Ключевое поле |
| 2 | Аркада (Arcade) | игры с нарочно примитивным игровым процессом |
| 3 | Приключенческая игра (Adventure) | главной частью игры является история |
| 4 | Ролевая игра (RPG) | главная особенность игры, игрок практически не ограничен в выборе игровых предметов, напарников и диалогов |
| 5 | Экшен (Action) | игра, характеризующая частым и активным нажатием кнопок управления |
| 6 | Стратегическая игра (Strategy) | основа игры – необходимость игроку делать нетривиальный выбор |
| 7 | Компьютерный симулятор (Simulator) | игрок делает множество упражнений и оттачивает свою технику |
| 8 | Головоломка (Puzzle) | требует аналитического мышления |
| 9 | Обучающая игра (Educational) | игрок обучается во время выполнения каких-либо действий в игре |
| 10 | Игрушки (Toys) | программы, взаимодействуя с которыми, игрок получает удовольствие |

Представленные здесь жанры нельзя назвать полными, так как в последнее время стали появляться игры собственных жанров, которые можно отнести как к одному из представленных здесь жанров, так и к самостоятельному отдельно от них.

Игры делятся на следующие виды.

По количеству игроков:

− однопользовательские;

− многопользовательские.

По визуальному представлению:

− текстовые – минимальное графическое представление, общение с игроком проходит с помощью текста;

− 2D – все элементы отрисованы в виде двумерной графики (спрайтов);

− 3D – все элементы отрисованы в виде трехмерной графики (3D – 9 − модели).

По типу платформы:

− персональные компьютеры;

− игровые приставки/консоли;

− мобильные телефоны.

Основываясь на данной классификации компьютерных игр, было принято решение разработать двумерный однопользовательский top down rogue-like для персональных компьютеров. Данное решение было принято потому, что top down rogue-like обладает простой и понятной механикой, что означает легкость в проектировании и разработке для начинающих игровых разработчиков. Данный проект относится к инди-разработке и развивается только за счет средств его разработчиков. Благодаря развитию ПО разработки компьютерных игр, команде разработчиков не требуется тратить несколько лет на разработку игрового движка. Это позволяет сразу же приступать к непосредственной работе над игровыми проектами значительно сокращает время его разработки

### 1.2.1 Сравнительный анализ игр жанра «Rogue-Like»

Среди всех типов игр «Rogue-Like» – довольно распространенная категория, даже если в течение некоторого времени она сокращалась. Популярность мобильных игр обновила жанр и открыла перед дизайнерами новые возможности.

Rogue–like (сленг «рогалик») – жанр компьютерных игр. Характерными особенностями классического roguelike являются генерируемые случайным образом уровни, пошаговость и необратимость смерти персонажа – в случае его гибели игрок не может загрузить игру и должен начать её заново. Многие roguelike выполнены в декорациях эпического фэнтези под сильным влиянием настольных ролевых игр наподобие Dungeons & Dragons.

Игровой процесс игр жанра roguelike в целом вдохновляется Rogue (компьютерная игра, написанная в 1980 году. Её основной темой является исследование подземелий. Она была необычайно популярной на университетских Unix-системах в начале 1980-х годов и породила целый жанр) однако отдельные игры могут значительно отличаться друг от друга. В 2008 году на конференции International Roguelike Development Conference 2008 в Берлине участники попытались определить ключевые факторы, выделяющие roguelike среди других игр; их определение известно, как «Берлинская интерпретация» Берлинская интерпретация выделила ряд ключевых факторов, позволяющих определить игру как roguelike:

– Игра должна быть пошаговой, каждая команда должна соответствовать одному действию и одному ходу;

– Игровые уровни должны генерироваться случайным образом, будучи уникальными для каждого прохождения;

− Игра должна содержать «перманентную смерть», не позволяя игроку продолжить прохождение после гибели персонажа;

– Игра должна иметь единый режим и единый набор команд для всех игровых ситуаций, не допуская каких-либо дополнительных меню, головоломок или мини-игр;

– Игра должна предоставлять игроку не какой-то единый линейный путь, а свободу со множеством вариантов прохождения;

– Игрок должен самостоятельно исследовать найденные предметы и открывать их свойства.

Большая часть этапа игрового дизайна состоит в том, чтобы свободно использовать эти возможности, чтобы заставить игрока находить решения проблем. Решение их заставит приключение продолжиться. В некотором смысле игры типа «Rogue–Like» похожи на игры-головоломки. Игровой процесс включает в себя процедурную генерацию уровней и перманентную смерть персонажа, что создает уникальные и сложные ситуации для каждого прохождения. Количество взаимодействий и стратегий довольно ограничено, но их разнообразие отличает «Rogue−Like» от других жанров, таких как аркады, где требуется быстрый рефлекс, и ролевых игр, где могут преобладать нарративные элементы.

Игры, такие как NetHack и Rogue, часто считаются наиболее важными в истории жанра «Rogue–Like». Эти игры создавались с использованием процедурной генерации и ASCII-графики. Они носили описательный характер, позволяя геймдизайнерам, сценаристам и разработчикам создавать сложные, многослойные уровни и механики. Благодаря процедурной генерации и глубоким механикам, «Rogue–Like» игры предоставляли игрокам уникальные и разнообразные испытания в каждом прохождении без необходимости постоянного вмешательства программистов на каждом этапе разработки.

В наши дни, благодаря популярности инди-игр и их казуальному характеру, такие игры, как «Rogue–Like», становятся всё более популярными. Эти игры:

* состоят из процедурно сгенерированных уровней и стратегических решений;
* используют мало системных ресурсов;
* не требуют специальных устройств;
* не требуют специальных навыков исполнения, но предлагают глубокий геймплей.

У этого жанра всё больше и больше игроков, и довольные люди продолжают приходить к нему из других областей, таких как экшен и ролевые игры.

Самыми знаменитыми представителями жанра Rogue–Like можно назвать серии Dead Cells, The Binding of Isaac, Hades, Slay the Spire и другие. Остановимся и подробнее рассмотрим некоторые игры данного жанра:

«Dead Cells» (дата релиза 2018 г.) – представляет собой «Rogue–Like» -экшен-платформер. Геймплей заключается в исследовании процедурно сгенерированных уровней, сражении с врагами и сборе предметов.

Управление осуществляется при помощи клавиатуры и мыши или контроллера.

Игрок управляет главным героем, Безголовым, который может прыгать, атаковать, уклоняться и использовать различные способности. На каждом уровне игроку предстоит находить оружие и улучшения, побеждать боссы и находить скрытые проходы, чтобы продвигаться дальше. Каждое прохождение уникально благодаря процедурной генерации и перманентной смерти персонажа, что требует от игрока адаптироваться и разрабатывать новые стратегии.

В отличие от большинства игр этого жанра, инвентарь в «Dead Cells» разделен на две части: в одной хранятся активные предметы, такие как оружие и способности, а в другой – ячейки для улучшений и бонусов (мутаторы). Игрок может одновременно носить два оружия и две вспомогательные способности (например, гранаты или ловушки). Каждый предмет обладает своими характеристиками и стилем игры, что позволяет разнообразить геймплей и адаптироваться к разным ситуациям. Улучшения и бонусы, найденные во время прохождения, могут значительно усиливать героя, добавляя ему новые возможности и улучшая его выживаемость.

Также в инвентаре «Dead Cells» постоянно присутствуют ячейки для «Клеток» и «Свитков», которые герой собирает в течение игры. Клетки используются для постоянного улучшения снаряжения и навыков, а свитки улучшают характеристики героя на текущем этапе прохождения.

Некоторые механики игры связаны с использованием этих ресурсов: клетки нужно вкладывать в улучшения на специальных станциях, а свитки размещать стратегически, чтобы усилить определенные характеристики, такие как здоровье, урон от оружия или способности.

Эти ресурсы играют ключевую роль в прогрессе игрока и требуют грамотного управления для успешного прохождения уровней (см. рисунок 1–2).



Рисунок 1 – Скриншот из игры «Dead Cells»

Изображение выглядит как Компьютерная игра, Цифровая сборка, Программное обеспечение для видеоигр, снимок экрана

Автоматически созданное описание

Рисунок 2 – Скриншот из игры «Dead Cells»

Rogue (дата релиза 1980 г.) – это первая часть игры в жанре Rogue–Like с увлекательным геймплеем и процедурно сгенерированными уровнями, выпущенная в далёком 1980 году.

Проект сейчас может показаться устаревшим, поскольку графика представлена в виде символов ASCII, а звук и интерфейс выглядят очень древними.

Однако, попробовав хотя бы одну часть, очень легко стать поклонником этих захватывающих подземелий. Ведь здесь продуман каждый элемент игрового процесса, а случайно генерируемые события вызывают бурю различных эмоций.

И если вам всё же чужда такая графика, то сообщество уже позаботилось о различных ремейках и улучшенных версиях (обновление графической составляющей игры с сохранением использования исходного «движка» игры), благодаря которым познакомиться со знаменитой вселенной теперь может ещё больше людей (см. рисунок 3-4).

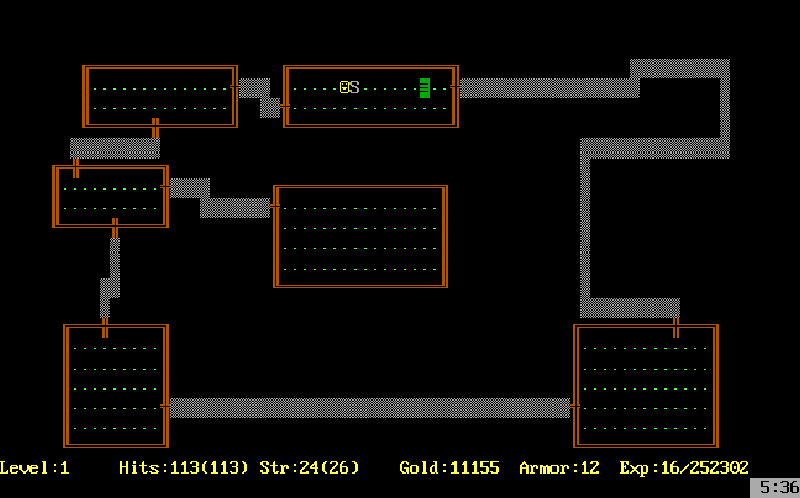


Рисунок 3 – Скриншот из игры «Rogue»

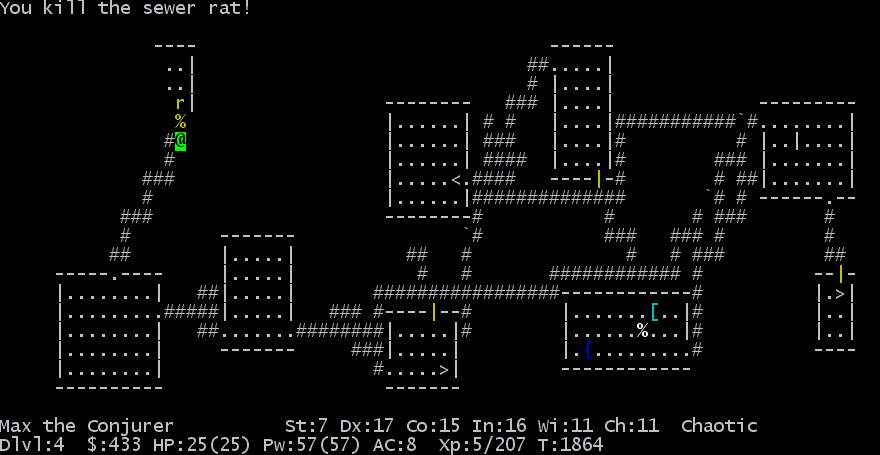


Рисунок 4 – Скриншот из игры «Rogue»

The Binding of Isaac (дата релиза 2011 г.) – это экшен-Roguelike, в котором игрокам предстоит управлять маленьким мальчиком по имени Айзек, сражаясь с различными монстрами в процедурно сгенерированных подземельях.

Игроки исследуют комнаты, находят и используют предметы с уникальными способностями, побеждают боссов и раскрывают тайны.

На рисунке 5 можно ознакомиться с визуальной частью игры, представляющей мрачный и причудливый стиль, который делает каждый забег уникальным и захватывающим.



Рисунок 5 – Скриншот из игры «The Binding of Isaac»

В The Binding of Isaac, игрок сталкивается с различными врагами и препятствиями в процедурно сгенерированных подземельях. Если игрок замечает несоответствие между типами врагов или подозрительные объекты в комнате, он может использовать специальные предметы или способности для проверки и выявления угроз. Например, при обнаружении скрытых ловушек или иллюзорных стен, игрок может использовать бомбы или специальные глаза, чтобы выявить их местоположение.

Если внешний вид врага вызывает сомнения, игрок может применять специальные предметы, такие как «X-Ray Vision» или «The Compass», чтобы получить дополнительную информацию о комнате и её содержимом. При нахождении опасности, такой как опасные предметы или ловушки, игрок может использовать предметы вроде «The Bible» или «Book of Revelations» для очищения комнаты.

Нарушителя, представленного в виде босса или мини-босса, можно побеждать с помощью найденного оружия и улучшений. После победы над врагом игрок может получить ключевые предметы и продвинуться дальше в своём приключении.



Рисунок 6 – Скриншот из игры «The Binding of Isaac»

Исходя из вышеперечисленного, игры жанра «Rogue-Like» действительно разнообразны и по-своему увлекательны. Однако многие из этих игр были выпущены довольно давно, соответственно, разработка происходила на актуальном программном обеспечении и аппаратно-технических средствах того времени, а иногда и более старых, поскольку разработка игр занимает длительное время. В момент релиза проекта на рынке игровой индустрии могут появиться более новые оборудование и программное обеспечение.

В рамках текущего проекта будет учитываться технологический скачок. В разработке будет использоваться современное программное обеспечение и более мощные аппаратно-технические средства, что позволит создать более сложные и увлекательные процедурно сгенерированные уровни, улучшенную графику и звук, а также более глубокую и разнообразную механику игры. Это обеспечит игрокам свежий и захватывающий опыт в жанре «Rogue–Like».

### 1.3.1 Общий алгоритм реализации проекта

Создание ассетов на этапе производства – это базовый этап, на котором разработчики игр создают различные ассеты, а затем используют их на этапе реализации игры. На этапе производства первым шагом является создание ресурсов для игры.

Создание раскадровки – самый важный этап производства игры; он включает в себя разработку игровых сценариев для решений уровней и включение методов планирования искусственного интеллекта для представления различных особенностей игр с помощью традиционной белой доски или блок–схемы.

В исследованиях, отнесенных к категории выбора платформы разработки, предлагались различные типы платформ для разработки игр. Платформы разработки предоставляют готовую архитектуру для подключения сервер-клиент и помогают разработчикам быстро создавать игры. Семантику игры можно отнести к формальному описанию языка для языков программирования; по этой классификации сообщалось только двух исследованиях.

Формальное языковое описание семантики игр дало возможность получить представление о дизайне языков программирования для разработки игр.

Сложность кода возрастает, особенно при разработке игр, из-за включения сложных модулей, методов искусственного интеллекта и разнообразного поведения. Наиболее распространенными языками программирования, используемыми в разработке игр, являются объектно-ориентированные структурированные языки, такие как Java, C и C++.

Исследования, отнесенные к этой категории, изучали программный аспект разработки игр. Исследование влияние объектно-ориентированной технологии на производительность, размер исполняемого файла и методы оптимизации для мобильных игр и предположил, что объектно-ориентированную технологию следует использовать с большой осторожностью, поскольку структурное программирование в разработке игр очень конкурентоспособно.

Игровой движок – это своего рода специальная программная среда, которая используется на этапе производства для создания и разработки игр. Игровые движки состоят в основном из комбинации основных функций, таких как звук, физический движок или обнаружение столкновений, ИИ, сценарии, анимация, работа в сети, управление памятью и графы сцен. Основы теории игр используются при разработке игр, потому что это раздел теории принятия решений, описывающий взаимозависимые решения.

В большинстве исследований этой категории описывались различные аспекты технологий реализации игр на разных типах платформ. Они рассматривали возможность улучшения навыков программирования, 2D/3D-анимации и графики, звукорежиссуры, управления проектами, логического дизайна, дизайна интерфейса для написания историй и методов искусственного интеллекта.

### 1.4.1 Анализ игровых движков.

В настоящее время серьезные игры, а также разработка мобильных игр находятся на подъеме. День за днем настольные и консольные игры заменяются играми для мобильных телефонов и планшетов. Более того, люди, занимающиеся дизайном и разработкой серьезных игр, приходят из самых разных сфер.

Например, педагоги и эксперты в предметной области с ограниченными навыками программирования, если таковые имеются, участвуют в проектировании и разработке серьезных игр. Таким образом, выбор разработки игры с использованием исключительно языка программирования, такого как C++, C# или Java, на самом деле возможен не для всех. Это сделало использование игровых движков еще более важным.

Игровые движки ускоряют процесс разработки игры с помощью существующих шаблонов и ресурсов, которые можно использовать повторно, сводя к минимуму или полностью устраняя необходимость иметь глубокие знания в области программирования. Более того, игровые движки дают возможность один раз разработать игру и экспортировать ее на различные платформы, включая мобильные устройства, внеся всего несколько изменений в оригинальную версию.

Однако существует множество игровых движков, которые имеют некоторые общие черты, но также имеют много различий. На самом деле разные игровые движки имеют разную философию разработки игр и нацелены на широкий спектр различных потребностей: движки, не требующие знаний программирования; движки, основанные на популярных веб-технологиях; движки с открытым исходным кодом, которые могут быть настроены/расширены опытными пользователями; и профессиональные игровые движки.

Сделать осознанный выбор игрового движка – это решение, основанное на нескольких критериях, и это непросто. Хотя существует несколько обзорных статей об игровых движках, в большинстве случаев эти статьи посвящены определенному типу игровых движков или конкретной области (например, смоделированное хирургическое обучение).

Предоставление обзора игровых движков, которые охватывают широкий спектр различных профилей и потребностей пользователей на основе расширенного списка функций, считается важным. Насколько мне известно, такого обзора нет, и это послужило поводом для написания этого раздела в выпускной квалифицированной работе.

Следовательно, цель этого раздела – детально проанализировать и показать все функции, которые могут присутствовать в игровом движке, организованными в категории более высокого уровня; основные типы игровых движков; возможности, предлагаемые репрезентативными примерами игровых движков, попадающих в вышеупомянутые категории.

Конечная цель – определится в выборе наиболее подходящего игрового движка, исходя из потребностей конкретного игрового проекта и профиля человека (лиц), которые собираются его реализовывать.

Краткое описание игровых движков.

GameMaker был впервые выпущен под названием Animo в 1991 году и подходил для создания 2D-игр. Хотя игровой движок подходит для 2D-игр, он позволяет пользователю добавлять 3D-графику и физику. GameMaker прост в использовании для начинающих пользователей, так как не требует знаний в области программирования. Однако функциональные возможности 3D-камеры по сравнению с 3D-графикой ограничены.

Jmonkey был впервые выпущен в 2003 году. Это бесплатный игровой движок, который бесплатно экспортирует игры на все платформы, включая мобильные устройства. JMonkey поддерживает 3D-графику. Хотя необходимо иметь опыт программирования на Java, это движок с открытым исходным кодом, который позволяет пользователям расширять его и адаптировать к своим потребностям.

Первоначальная версия игрового движка Marmalade была выпущена в 2015 году. Этот движок был выбран, поскольку он бесплатно экспортирует игры на многие мобильные платформы и обеспечивает возможности 19 рендеринга 3D-графики. Кроме того, Marmalade способствует созданию настольных веб-приложений. Версия, которая анализируется в данном проекте – 8.5

Игровой движок OGRE 3D был впервые выпущен в ноябре 2013 года. Ogre3D – это бесплатный движок с открытым исходным кодом. Он позволяет экспортировать игру на iOS, Android и Windows Phone 8 и рендерить 3D-графику. Версия, которая анализируется в данном проекте – 2.1.

Игровой движок Shiva был впервые выпущен в июле 2007 года. Он обеспечивает визуализацию 3D-графики и использовался для разработки очень популярных игр, таких как Prince of Persia 2, воссозданная для мобильных устройств с использованием Shiva, и Babel Rising, был опубликован Ubisoft. Игровой движок Sio2 был впервые выпущен в 2009 году и уже через год стал одним из 3-х самых популярных игровых движков, используемых в App Store.

Sio2 – это игровой движок, который ранее был бесплатным и с открытым исходным кодом; однако теперь он доступен пользователям за плату. Можно также получить доступ к его исходному коду, купив версию сертифицированного разработчика. Пробная версия предоставляется пользователям с большинством доступных функций, таких как мобильный экспорт. Тем не менее, игровая симуляция предоставляется на различных устройствах iOS и является одним из самых популярных игровых движков для экспорта на платформы iOS, предоставляя расширенную 3D-графику.

Игровой движок Turbulenz был создан в апреле 2013 года и является единственным изучаемым нами движком, который позволяет экспортировать игры в мобильные браузеры на основе многообещающего подхода веб-технологий. В то же время Turbulenz является бесплатным, с открытым исходным кодом и поддерживает рендеринг 3D-графики.

Движок Unity Game был впервые публично анонсирован на Всемирной конференции разработчиков Apple в 2005 году и является одним из самых известных игровых движков. Unity бесплатно предоставляет расширенный 20 рендеринг 3D-графики и экспорт на мобильные устройства. Тем не менее, это один из самых популярных движков в игровой индустрии, с помощью которого было создано множество успешных игр, таких как Deus Ex: The Fall для мобильных телефонов или Assassin's Creed: Identity.

Первоначальный выпуск Unreal Engine состоялся в 1998 году, а Unreal Engine 4 впервые был выпущен в мае 2012 года. Этот движок подходит для рендеринга 3D-графики, является бесплатным и открытым исходным кодом, а также является одним из самых популярных игровых движков, как и многие игры. были созданы с его помощью и преуспели в такой отрасли, как Absolver.

Unity и Unreal Engine 4 поддерживают все основные функции и имеют поддержку PhysX SDK для физического моделирования. По результатам сравнительного анализа можно сделать несколько важных выводов относительно эффективности каждого игрового движка для категорий характеристик, исследованных в данном исследовании:

Было замечено, что Game Maker не подходит для разработки игр с 3D-графикой. Тем не менее, он идеально подходит для простого создания игр с 2D-графикой, обеспечивая удобную среду и возможность разрабатывать игры без какого-либо опыта программирования.

JMonkey не справляется с некоторыми аспектами аудиовизуальной точности, такими как отсутствие отбраковки окклюзии или анимации и инструментов 3D-звука. Более того, он не может экспортировать игры на 26 консоли. С другой стороны, это игровой движок с открытым исходным кодом, позволяющий пользователям расширять его функциональность.

Marmalade – это движок, которому не хватает нескольких функций в каждой категории, но у него есть преимущество в категории платформ развертывания, поскольку он может экспортировать игры на большинство игровых платформ.

Ogre3D, как и JMonkey, не поддерживает многие функции категории аудиовизуальной и функциональной точности. С другой стороны, Ogre3D также является игровым движком с открытым исходным кодом, поэтому его можно адаптировать к потребностям опытного пользователя. Кроме того, Ogre3D идеально подходит для создания теней и подходит для начинающих, предоставляя неплохую поддержку.

Шива не так доступна для пользователей, потому что она не бесплатна, а также не подходит для неопытных пользователей. Тем не менее, он содержит почти все аудиовизуальные функции, и в то же время превосходит в физическом подклассе функциональной точности, поддерживая все функции.

Sio2 подходит для создания приложений в программном обеспечении iOS, так как его ядро основано на этой операционной системе. Ему не хватает многих функций аудиовизуальной и функциональной точности. Он также не так доступен для пользователей, потому что, хотя раньше он был бесплатным и с открытым исходным кодом, сейчас это самый дорогой игровой движок.

Unity и Unreal Engine 4 – самые мощные игровые движки, которые поддерживают почти все функции, включенные в используемый фреймворк. Unreal Engine 4 обеспечивает лучшую аудиовизуальную производительность, чем Unity, поскольку он поддерживает больше функций и в то же время является движком с открытым исходным кодом.

Эмпирическое исследование Unity и Unreal Engine 4 были выбраны обзор и бенчмаркинг для дальнейшего сравнения, потому что они явно являются двумя наиболее развитыми игровыми движками. В контексте эмпирического исследования были разработаны два похожих настольных шутера на двух игровых движках с помощью руководств на их официальных веб-сайтах, чтобы наилучшим образом использовать различные функции двух движков. В частности, игра Survival Shooter была разработана в Unity, а Twin Stick Shooter – в Unreal Engine 4. Затем игры были обработаны для запуска на мобильных устройствах и экспортированы для тестирования на мобильном устройстве Android для получения результатов.

В ходе эмпирического исследования изучались такие характеристики, как удобство использования движка, кривая обучения, процесс экспорта для мобильных устройств, а также качество игры и размер приложения. Исходя из всего вышеперечисленного, оптимальным выбором для проектирования игры, будет Unity, так как имеет поддержку языка программирования С#.

# **2. Программирование**

## 2.1. Начало работы

Для работы мы выбрали движок Unity, теперь нам нужно создать проект и настроить его под себя, мы выбрали 2d проект, значит выбираем 2d проект, называем его Medievality.

## 2.2. Создание основных сцен

После того как создали проект необходимо создать сцены, у нас их будет две, первая сцена – это меню, в которой есть кнопка Play и кнопка Exit (см. рисунок 2) и еще одну сцену, в которой и будет проходить весь геймплей, это сцена с автоматической генерацией подземелья. (см. рисунок 3)

### 2.2.1. Главное меню

Сцену главного меню смотрите в приложении А (см. рисунок А3).

Класс MainMenuGenerationController генерирует различные элементы главного меню, включая подземелья, пол, тени и факелы. Класс MainMenuMusicPlayer управляет воспроизведением музыки в главном меню и анимацией кнопки при нажатии.

namespace Game.UI.MainMenu.Scripts

{

public class MainMenuGenerationController : MonoBehaviour

{

[SerializeField] private SimpleRandomWalkDungeonGenerator rightCaveGenerator;

[SerializeField] private SimpleRandomWalkDungeonGenerator leftCaveGenerator;

[SerializeField] private RandomTorchesInDungeonGenerator torchesGenerator;

[SerializeField] private TilesSquareSpawner floorDestroyer;

[SerializeField] private TilesSquareSpawner wallShadowDestroyer;

[SerializeField] private TilesSquareSpawner foundationGenerator;

private void Start()

{

\_GenerateMainMenu();

}

public void \_GenerateMainMenu()

{

rightCaveGenerator.\_GenerateDungeon();

leftCaveGenerator.\_GenerateDungeon();

floorDestroyer.\_Spawn();

wallShadowDestroyer.\_Spawn();

foundationGenerator.\_Spawn();

torchesGenerator.\_GenerateTorches();

}

}

}

public class MainMenuMusicPlayer : MonoBehaviour

{

[SerializeField] private Button playButton;

[SerializeField] private AudioSource audioSource;

[SerializeField] private AudioClip mainMenuMusic;

private Animator animator;

private void Awake()

{

animator = GetComponent<Animator>();

playButton.onClick.AddListener(Disappear);

}

private void OnEnable()

{

}

private void OnDisable()

{

// playButton.onClick.RemoveListener(Disappear);

}

private void Start()

{

PlayMainMenuMusic();

}

private void PlayMainMenuMusic()

{

audioSource.clip = mainMenuMusic;

// audioSource.time = Mathf.Lerp(0, mainMenuMusic.length, Random.Range(0f, 1f));

audioSource.Play();

}

private void Disappear()

{

animator.SetTrigger("Disappear");

}

}

### 2.2.2. Генерация подземелий

Этот код реализует систему процедурной генерации подземелий. Абстрактный класс DungeonGeneratorBase управляет общей логикой генерации, очисткой тайлмапов и вызовом событий. Класс ProceduralGenerationAlgorithms предоставляет алгоритмы генерации, такие как случайная прогулка и генерация коридоров. Класс SimpleRandomWalkDungeonGenerator наследует DungeonGeneratorBase и реализует метод RunProceduralGeneration, генерируя пол и стены подземелья с использованием алгоритмов. Вспомогательный класс Direction2D содержит список направлений для случайного выбора.

public abstract class DungeonGeneratorBase : MonoBehaviour

{

public static event Action DungeonGeneratedEvent;

public static event Action DungeonDestroyedEvent;

[Header("Dungeon Generator Base")]

[SerializeField] protected bool clearBeforeGenerate;

[SerializeField] protected Vector2Int startPosition = Vector2Int.zero;

[SerializeField] protected Tilemap floorTilemap;

[SerializeField] protected Tilemap wallTilemap;

[SerializeField] protected Tilemap wallShadowTilemap;

[SerializeField] protected TileBase floorRuleTile;

[SerializeField] protected TileBase wallRuleTile;

[SerializeField] protected TileBase wallShadowRuleTile;

public void \_GenerateDungeon()

{

if (clearBeforeGenerate)

{

ClearDungeon();

DungeonDestroyedEvent?.Invoke();

}

RunProceduralGeneration();

DungeonGeneratedEvent?.Invoke();

}

private void ClearDungeon()

{

floorTilemap.ClearAllTiles();

wallTilemap.ClearAllTiles();

wallShadowTilemap.ClearAllTiles();

}

protected abstract void RunProceduralGeneration();

}

}

namespace Game.Dungeon.Scripts

{

public static class ProceduralGenerationAlgorithms

{

public static HashSet<Vector2Int> SimpleRandomWalk(Vector2Int startPosition, int walkLength,

int numberOfStepsForOneDirection, float chanceToTurn)

{

var path = new HashSet<Vector2Int> { startPosition };

var previousPosition = startPosition;

var numberOfStepsInOneDirection = 0;

var currentDirection = Direction2D.GetRandomCardinalDirection();

while (path.Count < walkLength)

{

var newPosition = previousPosition + currentDirection;

path.Add(newPosition);

previousPosition = newPosition;

numberOfStepsInOneDirection++;

if (Random.Range(0f, 1f) < chanceToTurn || numberOfStepsInOneDirection >= numberOfStepsForOneDirection)

{

var availableDirections = Direction2D.CardinalDirectionsList

.Where(direction => direction != currentDirection).ToArray();

currentDirection = availableDirections[Random.Range(0, availableDirections.Count())];

numberOfStepsInOneDirection = 0;

}

}

return path;

}

public static List<Vector2Int> RandomWalkCorridor(Vector2Int startPosition, int corridorLength)

{

var corridor = new List<Vector2Int>();

var direction = Direction2D.GetRandomCardinalDirection();

var currentPosition = startPosition;

corridor.Add(currentPosition);

for (var i = 0; i < corridorLength; i++)

{

currentPosition += direction;

corridor.Add(currentPosition);

}

return corridor;

}

}

public static class Direction2D

{

public static readonly List<Vector2Int> CardinalDirectionsList = new List<Vector2Int>

{

new Vector2Int(0, 1), // UP

new Vector2Int(1, 0), // RIGHT

new Vector2Int(0, -1), // DOWN

new Vector2Int(-1, 0) // LEFT

};

public static readonly List<Vector2Int> CornersList = new List<Vector2Int>

{

new(1, 1), // TOP RIGHT

new Vector2Int(1, -1), // BOTTOM RIGHT

new Vector2Int(-1, -1), // BOTTOM LEFT

new Vector2Int(-1, 1) // TOP LEFT

};

public static Vector2Int GetRandomCardinalDirection()

{

return CardinalDirectionsList[Random.Range(0, CardinalDirectionsList.Count)];

}

}

}

public class SimpleRandomWalkDungeonGenerator : DungeonGeneratorBase

{

[Space]

[SerializeField] private DungeonGenerationData dungeonGenerationData;

protected override void RunProceduralGeneration()

{

var floorPositions =

ProceduralGenerationAlgorithms.SimpleRandomWalk(startPosition, dungeonGenerationData.NumberOfSteps,

dungeonGenerationData.StepsForOneDirection, dungeonGenerationData.ChanceToTurn);

var wallPositions = GetWallPositions(floorPositions);

if (!clearBeforeGenerate)

{

wallPositions.ExceptWith(floorTilemap.GetCellPositionsOfAllTiles());

TilesPainter.PaintTiles(floorPositions.Where(pos => wallTilemap.HasTile((Vector3Int)pos)), wallTilemap, null);

}

TilesPainter.PaintTiles(floorPositions, floorTilemap, floorRuleTile);

TilesPainter.PaintTiles(wallPositions, wallTilemap, wallRuleTile);

TilesPainter.PaintTiles(floorPositions, wallShadowTilemap, wallShadowRuleTile);

}

private static HashSet<Vector2Int> GetWallPositions(HashSet<Vector2Int> floorPositions)

{

var leftTopWallPosition = new Vector2Int(floorPositions.Min(position => position.x),

floorPositions.Max(position => position.y)) + new Vector2Int(-10, 10);

var rightBottomWallPosition = new Vector2Int(floorPositions.Max(position => position.x),

floorPositions.Min(position => position.y)) + new Vector2Int(10, -10);

var wallPositions = new HashSet<Vector2Int>();

for (var x = leftTopWallPosition.x; x <= rightBottomWallPosition.x; x++)

for (var y = rightBottomWallPosition.y; y <= leftTopWallPosition.y; y++)

wallPositions.Add(new Vector2Int(x, y));

foreach (var floorPosition in floorPositions)

wallPositions.Remove(floorPosition);

return wallPositions;

}

}

### 2.2.3. Функции Игрока

Движение игрока реализовано в классе PlayerMovement который управляет движением игрока, используя LinearPhysicalMovement и PlayerInput. В Awake компоненты инициализируются, а в FixedUpdate, если введено направление движения, вызывается метод Move для перемещения игрока в этом направлении.

namespace Game.Entities.LivingEntities.Player.Scripts

{

[RequireComponent(typeof(LinearPhysicalMovement))]

[RequireComponent(typeof(PlayerInput))]

public class PlayerMovement : MonoBehaviour

{

private LinearPhysicalMovement linearPhysicalMovement;

private PlayerInput playerInput;

private void Awake()

{

linearPhysicalMovement = GetComponent<LinearPhysicalMovement>();

playerInput = GetComponent<PlayerInput>();

}

private void FixedUpdate()

{

if (playerInput.EnteredMovementDirection != Vector2.zero)

linearPhysicalMovement.Move(playerInput.EnteredMovementDirection);

}

}

}

Прицеливание оружием разработано в классе PlayerWeaponAim, который устанавливает начальную позицию объекта на центр коллайдера игрока при запуске. В методе Start позиция объекта transform устанавливается в центр границ playerCollider.

public class PlayerWeaponAim : MonoBehaviour

{

[SerializeField] private Collider2D playerCollider;

private void Start()

{

transform.position = playerCollider.bounds.center;

}

Атака главного персонажа реализуется в: классе PlayerAttack который управляет оружием игрока в игре. При включении он подписывается на событие нажатия кнопки атаки, а при отключении отписывается. В методе Awake инициализируется playerInput, а в методе Start активируется текущее оружие и деактивируются остальные. При нажатии кнопки атаки текущему оружию отправляется запрос на атаку.

namespace Game.Entities.LivingEntities.Player.Scripts

{

[RequireComponent(typeof(PlayerInput))]

public class PlayerAttack : MonoBehaviour

{

[SerializeField] private WeaponInput[] existingWeapons;

[SerializeField] private WeaponInput currentWeapon;

public WeaponInput CurrentWeapon => currentWeapon;

private PlayerInput playerInput;

private void OnEnable() => playerInput.AttackButtonClickedEvent += OnAttackButtonClicked;

private void OnDisable() => playerInput.AttackButtonClickedEvent -= OnAttackButtonClicked;

private void Awake()

{

playerInput = GetComponent<PlayerInput>();

}

private void Start()

{

foreach (var existingWeapon in existingWeapons)

existingWeapon.gameObject.SetActive(false);

currentWeapon.gameObject.SetActive(true);

}

private void OnAttackButtonClicked() => currentWeapon.SendAttackRequest();

}

}

Анимация игрока реализуется в классе PlayerAnimation который управляет анимациями игрока и реагирует на паузы в игре. Он подписывается и отписывается от событий изменения направления движения и паузы при активации и деактивации. В методе Awake инициализируются animator и playerInput. В методе Update проверяется направление взгляда игрока в зависимости от позиции мыши. Метод OnMovementDirectionChanged обновляет состояние анимации движения, а метод SetPaused контролирует скорость анимации при паузе.

namespace Game.Entities.LivingEntities.Player.Scripts

{

[RequireComponent(typeof(PlayerInput))]

[RequireComponent(typeof(Animator))]

public class PlayerAnimation : MonoBehaviour, IPauseHandler

{

[SerializeField] private SpriteRenderer spriteRenderer;

private PlayerInput playerInput;

private Animator animator;

private static readonly int IsMoving = Animator.StringToHash("IsMoving");

private void OnEnable()

{

playerInput.EnteredMovementDirectionChangedEvent += OnMovementDirectionChanged;

Pauser.Register(this);

}

private void OnDisable()

{

playerInput.EnteredMovementDirectionChangedEvent -= OnMovementDirectionChanged;

Pauser.UnRegister(this);

}

private void Awake()

{

animator = GetComponent<Animator>();

playerInput = GetComponent<PlayerInput>();

}

private void Update()

{

if (Pauser.IsPaused)

return;

spriteRenderer.flipX = MouseLocation.WorldPosition.x < transform.position.x;

}

private void OnMovementDirectionChanged(Vector2 enteredMovementDirection)

{

animator.SetBool(IsMoving, enteredMovementDirection != Vector2.zero);

}

void IPauseHandler.SetPaused(bool isPaused)

{

animator.speed = isPaused ? 0 : 1;

}

}

Смерть игрока реализуется в классе ShowWhenPlayerDie который управляет отображением UI при смерти игрока. При спавне игрока он подписывается на событие его смерти. Когда игрок умирает, вызывается метод Show, который включает указанные объекты и постепенно увеличивает прозрачность CanvasGroup за заданное время.

[RequireComponent(typeof(CanvasGroup))]

public class ShowWhenPlayerDie : MonoBehaviour

{

[SerializeField] private float appearanceTime;

[SerializeField] private float delay;

[SerializeField] private GameObject[] objectsToEnable;

[SerializeField] private GameObjectSpawner playerSpawner;

private CanvasGroup canvasGroup;

private Damageable playerDamageable;

private void Awake()

{

canvasGroup = GetComponent<CanvasGroup>();

}

private void OnEnable() => playerSpawner.SpawnedEvent += OnPlayerSpawned;

private void OnDisable() => playerSpawner.SpawnedEvent -= OnPlayerSpawned;

private void OnPlayerSpawned()

{

playerDamageable = playerSpawner.SpawnedObject.GetComponent<Damageable>();

playerDamageable.DiedEvent += OnPlayerDied;

}

private void OnPlayerDied(object sender, Damageable.DiedEventArgs diedEventArgs) => Show();

private async void Show()

{

await UniTask.Delay((int)(delay \* 1000));

foreach (var objectToEnable in objectsToEnable)

objectToEnable.SetActive(true);

var appearanceSpeed = 1 / appearanceTime;

for (var elapsedTime = 0f; elapsedTime < appearanceTime; elapsedTime += Time.deltaTime)

{

canvasGroup.alpha += appearanceSpeed \* Time.deltaTime;

await UniTask.Yield();

}

}

## 2.3. Враги (Слаймы)

Противниками игрока будут выступать слизни.

Класс SlimeAI управляет поведением слизи, включая поиск цели и выполнение импульсов атаки. При запуске он асинхронно ищет цель и инициирует импульс атаки после случайного времени перезарядки. События ReadyToMakeImpulseEvent и StartedReloadingEvent оповещают о готовности к атаке и начале перезарядки соответственно. Методы IsSeeTarget и OnDrawGizmos проверяют видимость цели и рисуют линию к ней для отладки. Асинхронные методы работают в цикле, пока не будет запрошена отмена.

public class SlimeAI : MonoBehaviour

{

[SerializeField] private float minReloadTime;

[SerializeField] private float maxReloadTime;

[SerializeField] private float timeBetweenTargetSearch;

[SerializeField] private float timeBeforeMakeImpulse;

[SerializeField] private float rangeOfVision;

[SerializeField] private LayerMask canSee;

[SerializeField] private GameObject targetPrefab;

#region Events

public event EventHandler<ReadyToMakeImpulseEventArgs> ReadyToMakeImpulseEvent;

public class ReadyToMakeImpulseEventArgs : EventArgs

{

public Vector2 Direction;

public ReadyToMakeImpulseEventArgs(Vector2 direction)

{

Direction = direction;

}

}

public event EventHandler<StartedReloadingEventArgs> StartedReloadingEvent;

public class StartedReloadingEventArgs : EventArgs

{

public readonly float ReloadTime;

public StartedReloadingEventArgs(float reloadTime)

{

ReloadTime = reloadTime;

}

}

#endregion

private GameObject targetGameObject;

public GameObject TargetGameObject

{

get

{

if (targetGameObject != null) return targetGameObject;

targetGameObject = GameObject.FindWithTag(targetPrefab.tag);

return targetGameObject;

}

}

private Collider2D targetCollider;

public Collider2D TargetCollider

{

get

{

if (targetCollider != null) return targetCollider;

if (TargetGameObject == null) return null;

targetCollider = TargetGameObject.GetComponent<Collider2D>();

return targetCollider;

}

}

private Vector2 lastTargetPosition;

private CancellationTokenSource cancelSearchingTargetToken;

private void Awake()

{

cancelSearchingTargetToken = new CancellationTokenSource();

}

private void Start()

{

SearchTargetAsync(cancelSearchingTargetToken.Token);

MakeImpulseAndReloadAsync(cancelSearchingTargetToken.Token);

}

private void OnDestroy()

{

cancelSearchingTargetToken.Cancel();

}

private async void SearchTargetAsync(CancellationToken token)

{

while (true)

{

if (token.IsCancellationRequested) return;

if (Pauser.IsPaused)

{

await UniTask.Yield();

continue;

}

if (IsSeeTarget())

{

if (TargetCollider == null) continue;

lastTargetPosition = TargetCollider.bounds.center;

await UniTask.Yield();

}

else

await UniTask.Delay((int)(timeBetweenTargetSearch \* 1000));

}

}

private async void MakeImpulseAndReloadAsync(CancellationToken token)

{

while (true)

{

if (token.IsCancellationRequested) return;

if (Pauser.IsPaused)

{

await UniTask.Yield();

continue;

}

var randomReloadTime = Random.Range(minReloadTime, maxReloadTime);

StartedReloadingEvent?.Invoke(this, new StartedReloadingEventArgs(randomReloadTime));

await UniTask.Delay(TimeSpan.FromSeconds(randomReloadTime));

if (token.IsCancellationRequested) return;

Vector2 impulseDirection;

if (lastTargetPosition != Vector2.zero)

{

impulseDirection = (lastTargetPosition - (Vector2)transform.position).normalized;

lastTargetPosition = Vector2.zero;

}

else

impulseDirection = Random.insideUnitCircle.normalized;

await UniTask.Delay(TimeSpan.FromSeconds(timeBeforeMakeImpulse));

if (token.IsCancellationRequested) return;

ReadyToMakeImpulseEvent?.Invoke(this, new ReadyToMakeImpulseEventArgs(impulseDirection));

}

}

private void OnDrawGizmos()

{

if (targetCollider == null) return;

var myPosition = transform.position;

var directionToTarget = (targetCollider.bounds.center - myPosition).normalized;

Gizmos.DrawRay(myPosition, directionToTarget \* rangeOfVision);

}

private readonly RaycastHit2D[] slimeSees = new RaycastHit2D[10];

private bool IsSeeTarget()

{

if (TargetCollider == null) return false;

var myPosition = transform.position;

var directionToTarget = TargetCollider.bounds.center - myPosition;

var size = Physics2D.RaycastNonAlloc(myPosition, directionToTarget, slimeSees, rangeOfVision, canSee);

if (size <= 0)

return false;

if (slimeSees[0].collider.CompareTag(TargetCollider.tag))

return true;

else

return false;

// if (slimeSees.All(hit => hit.collider.CompareTag(TargetCollider.tag)))

// return true;

// else

// return false;

}

}

### 2.3.1 Атака слайма и звуки

Класс SlimeAttack отвечает за нанесение урона целям при столкновении с определенной скоростью. Содержит поля для урона и минимальной скорости атаки. При столкновении с целью проверяет скорость, и если она превышает порог, наносит урон цели. Класс SlimeMovement управляет движением слизи, применяя случайные импульсы при событии ReadyToMakeImpulseEvent. Также отслеживает столкновения и вызывает соответствующие события.

public class SlimeAttack : MonoBehaviour

{

[SerializeField] private float damage;

[Tooltip("The damage will be performed only if the slime speed is greater or equal than this.")]

[SerializeField] private float attackSpeed;

private SlimeAI slimeAI;

private SlimeMovement slimeMovement;

private new Rigidbody2D rigidbody2D;

private Damageable targetDamageable;

private Damageable TargetDamageable

{

get

{

if (targetDamageable != null) return targetDamageable;

if (slimeAI.TargetGameObject == null) return null;

targetDamageable = slimeAI.TargetGameObject.GetComponent<Damageable>();

return targetDamageable;

}

}

private Rigidbody2D targetRigidbody2D;

private Rigidbody2D TargetRigidbody2D

{

get

{

if (targetRigidbody2D != null) return targetRigidbody2D;

if (slimeAI.TargetGameObject == null) return null;

targetRigidbody2D = slimeAI.TargetGameObject.GetComponent<Rigidbody2D>();

return targetRigidbody2D;

}

}

private void Awake()

{

slimeAI = GetComponent<SlimeAI>();

slimeMovement = GetComponent<SlimeMovement>();

rigidbody2D = GetComponent<Rigidbody2D>();

}

private void OnEnable()

{

slimeMovement.TouchedOtherColliderEvent += OnTouchedOtherCollider;

}

private void OnDisable()

{

slimeMovement.TouchedOtherColliderEvent -= OnTouchedOtherCollider;

}

private void OnTouchedOtherCollider(Collision2D collision)

{

if (TargetDamageable == null) return;

if (collision.gameObject.CompareTag(slimeAI.TargetCollider.tag))

TryPerformDamageToTarget();

}

private void TryPerformDamageToTarget()

{

var targetVelocity = TargetRigidbody2D.velocity;

var slimeVelocity = rigidbody2D.velocity;

if ((targetVelocity - slimeVelocity).magnitude >= attackSpeed)

TargetDamageable.AddToHealth(transform, -damage);

}

}

}

public class SlimeAttack : MonoBehaviour

{

[SerializeField] private float damage;

[Tooltip("The damage will be performed only if the slime speed is greater or equal than this.")]

[SerializeField] private float attackSpeed;

private SlimeAI slimeAI;

private SlimeMovement slimeMovement;

private new Rigidbody2D rigidbody2D;

private Damageable targetDamageable;

private Damageable TargetDamageable

{

get

{

if (targetDamageable != null) return targetDamageable;

if (slimeAI.TargetGameObject == null) return null;

targetDamageable = slimeAI.TargetGameObject.GetComponent<Damageable>();

return targetDamageable;

}

}

private Rigidbody2D targetRigidbody2D;

private Rigidbody2D TargetRigidbody2D

{

get

{

if (targetRigidbody2D != null) return targetRigidbody2D;

if (slimeAI.TargetGameObject == null) return null;

targetRigidbody2D = slimeAI.TargetGameObject.GetComponent<Rigidbody2D>();

return targetRigidbody2D;

}

}

private void Awake()

{

slimeAI = GetComponent<SlimeAI>();

slimeMovement = GetComponent<SlimeMovement>();

rigidbody2D = GetComponent<Rigidbody2D>();

}

private void OnEnable()

{

slimeMovement.TouchedOtherColliderEvent += OnTouchedOtherCollider;

}

private void OnDisable()

{

slimeMovement.TouchedOtherColliderEvent -= OnTouchedOtherCollider;

}

private void OnTouchedOtherCollider(Collision2D collision)

{

if (TargetDamageable == null) return;

if (collision.gameObject.CompareTag(slimeAI.TargetCollider.tag))

TryPerformDamageToTarget();

}

private void TryPerformDamageToTarget()

{

var targetVelocity = TargetRigidbody2D.velocity;

var slimeVelocity = rigidbody2D.velocity;

if ((targetVelocity - slimeVelocity).magnitude >= attackSpeed)

TargetDamageable.AddToHealth(transform, -damage);

}

[RequireComponent(typeof(SlimeAI))]

[RequireComponent(typeof(Rigidbody2D))]

[RequireComponent(typeof(Collider2D))]

public class SlimeMovement : MonoBehaviour

{

[SerializeField] private float minForce;

[SerializeField] private float maxForce;

private SlimeAI slimeAI;

private new Rigidbody2D rigidbody2D;

public event Action<Collision2D> TouchedOtherColliderEvent;

private void Awake()

{

slimeAI = GetComponent<SlimeAI>();

rigidbody2D = GetComponent<Rigidbody2D>();

}

private void OnEnable()

{

slimeAI.ReadyToMakeImpulseEvent += OnReadyToMakeImpulse;

}

private void OnDisable()

{

slimeAI.ReadyToMakeImpulseEvent -= OnReadyToMakeImpulse;

}

private void OnReadyToMakeImpulse(object sender, SlimeAI.ReadyToMakeImpulseEventArgs e)

{}

## 2.4. Система лечения

Также в игре существует система лечения, она выполнена в роли зелий, которые выпадают с врагов в виде лута на землю, при подборе которых игрок вылечивает от 1 до 3 здоровья (в зависимости от вида зелья).

Класс Healer отвечает за восстановление здоровья при столкновении с объектами, способными получать урон. При столкновении проверяет наличие среди столкнувшихся объектов тех, которые можно исцелить. Если такие найдены и их текущее здоровье не максимально, они получают увеличение здоровья и проигрывается звук исцеления. Класс HealingPotionData содержит данные о зелье исцеления, такие как количество исцеления и изображение зелья.

namespace Game.Entities.Consumables.Potions.HealingPotions.Scripts

{

[RequireComponent(typeof(Collider2D))]

public class Healer : MonoBehaviour

{

[SerializeField] private float healingAmount;

[SerializeField] private Sound healSound;

[SerializeField] private Damageable[] canHealPrefabs;

private GlobalAudioPlayer globalAudioPlayer;

private void Awake()

{

globalAudioPlayer = GameObject.FindWithTag(nameof(GlobalAudioPlayer)).GetComponent<GlobalAudioPlayer>();

}

private void OnCollisionEnter2D(Collision2D collision)

{

var foundDamageablePrefab = canHealPrefabs.FirstOrDefault(obj => obj.CompareTag(collision.gameObject.tag));

if (foundDamageablePrefab == null)

return;

var collidedDamageable = collision.gameObject.GetComponent<Damageable>();

if (collidedDamageable.Health == collidedDamageable.MaxHealth)

return;

collidedDamageable.AddToHealth(transform, healingAmount);

globalAudioPlayer.PlayAudio(healSound, transform.position);

LeanPool.Despawn(this);

}

}

}

namespace Game.Entities.Consumables.Potions.HealingPotions.Data

{

public class HealingPotionData : ScriptableObject

{

[SerializeField] private float healingAmount;

[SerializeField] private Sprite potionSprite;

public float HealingAmount => healingAmount;

public Sprite PotionSprite => potionSprite;

}

### 2.4.1. Система лута

Класс LootDropperWhenDie отвечает за сброс добычи при смерти живого существа. При смерти объекта выпадает случайная добыча из массива loots, если выпадение произошло в соответствии с вероятностью DropChance. Каждая добыча начинает движение со стартовой скоростью startSpeed, направленной к позиции убийцы. Класс Loot содержит информацию о добыче, включая объект-префаб для сброса, вероятность выпадения и начальную скорость.

namespace Game.Entities.LivingEntities.Scripts

{

[RequireComponent(typeof(Damageable))]

public class LootDropperWhenDie : MonoBehaviour

{

[SerializeField] private Loot[] loots;

private Damageable damageable;

private void Awake()

{

damageable = GetComponent<Damageable>();

}

private void OnEnable()

{

damageable.DiedEvent += OnDied;

}

private void OnDisable()

{

damageable.DiedEvent -= OnDied;

}

private void OnDied(object sender, Damageable.DiedEventArgs diedEventArgs)

{

foreach (var loot in loots)

{

if (Random.Range(0f, 1f) <= loot.DropChance)

{

var thisPosition = transform.position;

var spawnedLoot = LeanPool.Spawn(loot.DropPrefab, thisPosition, Quaternion.identity);

spawnedLoot.velocity = (thisPosition - diedEventArgs.Killer.position) \* loot.StartSpeed;

}

}

}

[Serializable]

private struct Loot

{

[SerializeField] private Rigidbody2D dropPrefab;

[SerializeField, Range(0, 1)] private float dropChance;

[SerializeField] private float startSpeed;

public Rigidbody2D DropPrefab => dropPrefab;

public float DropChance => dropChance;

public float StartSpeed => startSpeed;

}

}

## 2.5 Камера

Управление камерой происходит через ассет, и скрипты, мы старались создать плавную и приятную камеру

Класс CameraShaker отвечает за встряхивание камеры с помощью Cinemachine во время игры. Метод ShakeCameraAsync принимает интенсивность и время встряхивания, затем изменяет амплитуду тряски постепенно, пока не проходит указанное время. Класс CameraTarget служит для управления целью камеры. Метод LookAtMouseAsync выравнивает позицию объекта по координатам курсора мыши в режиме реального времени, а метод StopLookingAtMouse отменяет это выравнивание.

[RequireComponent(typeof(CinemachineVirtualCamera))]

public class CameraShaker : MonoBehaviour

{

[SerializeField] private CinemachineVirtualCamera cinemachineVirtualCamera;

private CinemachineBasicMultiChannelPerlin cinemachineBasicMultiChannelPerlin;

private void Awake()

{

cinemachineVirtualCamera = GetComponent<CinemachineVirtualCamera>();

cinemachineBasicMultiChannelPerlin =

cinemachineVirtualCamera.GetCinemachineComponent<CinemachineBasicMultiChannelPerlin>();

}

public async void ShakeCameraAsync(float intensity, float time)

{

cinemachineBasicMultiChannelPerlin.m\_AmplitudeGain = intensity;

for (var elapsedTime = 0f; elapsedTime < time; elapsedTime += Time.deltaTime)

{

cinemachineBasicMultiChannelPerlin.m\_AmplitudeGain -= (intensity / time) \* Time.deltaTime;

await UniTask.Yield();

}

cinemachineBasicMultiChannelPerlin.m\_AmplitudeGain =

}

public class CameraTarget : MonoBehaviour

{

[Range(2, 100)] [SerializeField] private float cameraTargetDivider;

private CancellationTokenSource lookingAtMouseCancellationToken;

private void Awake()

{

lookingAtMouseCancellationToken = new CancellationTokenSource();

}

private void OnDestroy()

{

lookingAtMouseCancellationToken.Cancel();

}

public void StopLookingAtMouse() => lookingAtMouseCancellationToken.Cancel();

public async void LookAtMouseAsync(Transform playerTransform)

{

while (true)

{

if (lookingAtMouseCancellationToken.IsCancellationRequested) return;

var cameraTargetPosition = ((Vector3)MouseLocation.WorldPosition + (cameraTargetDivider - 1) \*

playerTransform.position) / cameraTargetDivider;

transform.position = cameraTargetPosition;

await UniTask.Yield();

}

}

}

}

### 2.5.1. События

События определяют, что будет происходить, если например, игрок умер

Класс PlayerDiedEvent отслеживает смерть игрока в игре. При возникновении события смерти игрока (OnPlayerDied) класс выполняет следующие действия: останавливает музыку в подземелье (dungeonMusicPlayer.StopAbruptly()), прекращает выравнивание камеры по координатам мыши (cameraTarget.StopLookingAtMouse()), а также отображает окна меню и «Вы умерли» (menuWindow.ShowAsync() и youDiedWindow.ShowAsync() соответственно).

public class PlayerDiedEvent : MonoBehaviour

{

[SerializeField] private GameObjectSpawner playerSpawner;

[SerializeField] private DungeonMusicPlayer dungeonMusicPlayer;

[SerializeField] private CameraTarget cameraTarget;

[SerializeField] private Window menuWindow;

[SerializeField] private Window youDiedWindow;

private Damageable playerDamageable;

private void OnEnable()

{

playerSpawner.SpawnedEvent += OnPlayerSpawned;

if (playerDamageable != null)

playerDamageable.DiedEvent += OnPlayerDied;

}

private void OnDisable()

{

playerSpawner.SpawnedEvent -= OnPlayerSpawned;

if (playerDamageable != null)

playerDamageable.DiedEvent -= OnPlayerDied;

}

private void OnPlayerSpawned()

{

playerDamageable = playerSpawner.SpawnedObject.GetComponent<Damageable>();

playerDamageable.DiedEvent += OnPlayerDied;

}

private void OnPlayerDied(object sender, Damageable.DiedEventArgs e)

{

playerDamageable = null;

dungeonMusicPlayer.StopAbruptly();

cameraTarget.StopLookingAtMouse();

menuWindow.ShowAsync();

youDiedWindow.ShowAsync();

}

Или если умирают все противники

AllEnemiesDiedEvent – это компонент, который реагирует на событие смерти всех врагов в игре. Когда все враги умирают (OnAllEnemiesDied), компонент останавливает плавно музыку в подземелье (dungeonMusicPlayer.StopSmoothly()). Если игрок еще жив (его объект был создан), то отображается меню победы (winMenu.Show()) и асинхронно появляется окно меню (menuWindow.ShowAsync()).

public class AllEnemiesDiedEvent : MonoBehaviour

{

[SerializeField] private RandomEnemiesSpawner enemiesSpawner;

[SerializeField] private GameObjectSpawner playerSpawner;

[SerializeField] private DungeonMusicPlayer dungeonMusicPlayer;

[SerializeField] private WinMenu winMenu;

[SerializeField] private Window menuWindow;

private void OnEnable()

{

enemiesSpawner.AllEnemiesDiedEvent += OnAllEnemiesDied;

}

private void OnDisable()

{

enemiesSpawner.AllEnemiesDiedEvent -= OnAllEnemiesDied;

}

private void OnAllEnemiesDied()

{

dungeonMusicPlayer.StopSmoothly();

if (playerSpawner.SpawnedObject != null)

{

winMenu.Show();

menuWindow.ShowAsync();

}

}

}

Или позволяют регулировать кол-во противников

NumberOfEnemiesChangedEvent отслеживает изменения в количестве врагов, созданных enemiesSpawner. Когда это количество изменяется (OnNumberOfEnemiesChanged), метод ShowEnemiesCount объекта slimesCounter вызывается для отображения текущего количества врагов.

public class NumberOfEnemiesChangedEvent : MonoBehaviour

{

[SerializeField] private RandomEnemiesSpawner enemiesSpawner;

[SerializeField] private SlimesCounter slimesCounter;

private void OnEnable()

{

enemiesSpawner.SpawnedEnemies.CollectionChanged += OnNumberOfEnemiesChanged;

}

private void OnDisable()

{

enemiesSpawner.SpawnedEnemies.CollectionChanged -= OnNumberOfEnemiesChanged;

}

private void OnNumberOfEnemiesChanged(object sender, NotifyCollectionChangedEventArgs e)

{

slimesCounter.ShowEnemiesCount(enemiesSpawner.SpawnedEnemies.Count);

}

}

### 2.5.2. Звуки

Этот код реализует систему паузы для игры. Интерфейс IPauseHandler определяет метод SetPaused, который должен быть реализован всеми компонентами, реагирующими на паузу. Класс Pauser управляет состоянием паузы и списком зарегистрированных обработчиков паузы. Он предоставляет методы для регистрации и удаления обработчиков, а также для установки состояния паузы, которое применяется ко всем зарегистрированным обработчикам.

namespace Game.Pause

{

public interface IPauseHandler

{

void SetPaused(bool isPaused);

}

} namespace Game.Pause

{

public static class Pauser

{

private static readonly List<IPauseHandler> Handlers = new();

public static bool IsPaused { get; private set; }

public static void Register(IPauseHandler handler)

{

Handlers.Add(handler);

}

public static void UnRegister(IPauseHandler handler)

{

Handlers.Remove(handler);

}

public static void ClearRegisteredHandlers()

{

Handlers.Clear();

}

public static void SetPaused(bool isPaused)

{

IsPaused = isPaused;

foreach (var handler in Handlers)

{

handler.SetPaused(isPaused);

}

}

}

}

### 2.5.3. Пауза

Этот код реализует воспроизведение глобальных звуковых эффектов в игре. Класс GlobalAudioPlayer использует объект AudioSource для проигрывания аудио. Метод PlayAudio принимает аудиоклип, позицию и диапазон случайных высот тона, создавая аудиоустройство, которое воспроизводит звук с указанными параметрами, а затем удаляет его после завершения воспроизведения. Второй метод PlayAudio принимает объект Sound, который содержит набор аудиоклипов и диапазон случайных высот тона, выбирая случайный клип из этого набора для воспроизведения.

namespace Game.Audio.Scripts

{

public class GlobalAudioPlayer : MonoBehaviour

{

[SerializeField] private AudioSource audioUnitPrefab;

public void PlayAudio(AudioClip audioClip, Vector2 position, float minRandomPitch, float maxRandomPitch)

{

var spawnedAudioUnit = LeanPool.Spawn(audioUnitPrefab, position, Quaternion.identity, transform);

spawnedAudioUnit.pitch = Random.Range(minRandomPitch, maxRandomPitch);

spawnedAudioUnit.PlayOneShot(audioClip);

LeanPool.Despawn(spawnedAudioUnit, audioClip.length + 1);

}

public void PlayAudio(Sound sound, Vector2 position)

{

var spawnedAudioUnit = LeanPool.Spawn(audioUnitPrefab, position, Quaternion.identity, transform);

spawnedAudioUnit.pitch = Random.Range(sound.MinRandomPitch, sound.MaxRandomPitch);

var randomAudioClip = sound.RandomAudioClips[Random.Range(0, sound.RandomAudioClips.Length)];

spawnedAudioUnit.PlayOneShot(randomAudioClip);

LeanPool.Despawn(spawnedAudioUnit, randomAudioClip.length + 1);

}

}

}

Этот код определяет структуру Sound, которая представляет звук с несколькими параметрами для случайного воспроизведения. Она содержит массив аудиоклипов (randomAudioClips), объект источника звука (audioSource), а также минимальные и максимальные значения для случайной высоты тона (minRandomPitch и maxRandomPitch).

Метод Play внутри структуры устанавливает случайную высоту тона для audioSource в пределах заданного диапазона, а затем воспроизводит случайный аудиоклип из массива randomAudioClips.

namespace Game.Audio.Scripts

{

[Serializable]

public struct Sound

{

[SerializeField] private AudioClip[] randomAudioClips;

[SerializeField] private AudioSource audioSource;

[SerializeField] private float minRandomPitch;

[SerializeField] private float maxRandomPitch;

public AudioClip[] RandomAudioClips => randomAudioClips;

public AudioSource AudioSource => audioSource;

public float MinRandomPitch => minRandomPitch;

public float MaxRandomPitch => maxRandomPitch;

public void Play()

{

audioSource.pitch = Random.Range(minRandomPitch, maxRandomPitch);

audioSource.PlayOneShot(randomAudioClips[Random.Range(0, randomAudioClips.Length)]);

}

}

}

Этот код определяет класс RandomPitchOnAwake, который устанавливает случайную высоту тона для компонента AudioSource при запуске объекта.

Класс помечен атрибутом [RequireComponent(typeof(AudioSource))], что гарантирует наличие компонента AudioSource на объекте. Внутри метода Awake этот компонент инициализируется, а метод Start задает случайную высоту тона для аудиоклипа в диапазоне от minPitch до maxPitch.

namespace Game.Scripts

{

[RequireComponent(typeof(AudioSource))]

public class RandomPitchOnAwake : MonoBehaviour

{

[SerializeField] private float minPitch;

[SerializeField] private float maxPitch;

private AudioSource audioSource;

private void Awake()

{

audioSource = GetComponent<AudioSource>();

}

private void Start()

{

audioSource.pitch = Random.Range(minPitch, maxPitch);

}

}

}

### 2.5.4. Факелы

Этот скрипт вызывает эффект дрожания света для компонента Light2D в игре. Радиус внешней границы света меняется случайным образом через определенные промежутки времени, создавая реалистичный эффект колеблющегося света. Асинхронный метод ShakeLightAsync выполняет изменения радиуса, пока скрипт не уничтожится или не будет приостановлен.

namespace Game.Dungeon.Torches.Scripts

{

public class LightShaking : MonoBehaviour

{

[SerializeField] private float shakingOffset;

[SerializeField] private float secondsBetweenShakes;

private Light2D light2D;

private float originPointLightOuterRadius;

private void Awake()

{

light2D = GetComponent<Light2D>();

stopShakingLightToken = new CancellationTokenSource();

}

private void Start()

{

originPointLightOuterRadius = light2D.pointLightOuterRadius;

ShakeLightAsync();

}

private void OnDestroy()

{

stopShakingLightToken.Cancel();

}

private CancellationTokenSource stopShakingLightToken;

private async void ShakeLightAsync()

{

while (true)

{

if (stopShakingLightToken.IsCancellationRequested)

return;

await UniTask.Delay((int)(secondsBetweenShakes \* 1000));

if (Pauser.IsPaused)

continue;

var newPointLightOuterRadius = Random.Range(originPointLightOuterRadius - shakingOffset, originPointLightOuterRadius + shakingOffset);

light2D.pointLightOuterRadius = newPointLightOuterRadius;

}

}

}

}

Этот скрипт управляет анимацией факела в игре, реализуя интерфейс IPauseHandler для остановки и возобновления анимации при паузе. В методе Awake компонент Animator присваивается переменной и регистрируется в паузере. В методе SetPaused анимация останавливается при паузе и возобновляется при продолжении игры.

public class Torch : MonoBehaviour, IPauseHandler

{

private Animator animator;

private void Awake()

{

animator = GetComponent<Animator>();

Pauser.Register(this);

}

void IPauseHandler.SetPaused(bool isPaused)

{

animator.speed = Pauser.IsPaused ? 0 : 1;

}

}

# **3.Тестирование**

Тестирование игры в дипломном проекте необходимо для обеспечения качества и функциональности конечного продукта. Оно позволяет выявить и исправить ошибки, улучшить стабильность и производительность игры, а также убедиться, что все игровые механики и элементы интерфейса работают корректно. Тестирование помогает проверить, насколько игра соответствует требованиям и стандартам, заданным в проекте, и улучшить пользовательский опыт за счет выявления и устранения проблем, с которыми могут столкнуться игроки. Таким образом, тестирование является важным этапом в разработке, который способствует созданию качественного и надежного игрового продукта, соответствующего ожиданиям и требованиям.

В первом тесте в таблице 3.1. мы проверим работу кнопки в главном меню.

Таблица 3.1. – Тест-кейс «Войти в игру»

|  |  |
| --- | --- |
| Короткое описание | Протестировать кнопку в главном меню. |
| Проект | Medievality |
| Номер версии | 1 |
| Важность:   * S1 Блокирующая (Blocker) * S2 Критическая (Critical) * S3 Значительная (Major) * S4 Незначительная (Minor) * S5 Тривиальная (Trivial) | S5 |
| Приоритет:   * P1 Высокий (High) * P2 Средний (Medium) * P3 Низкий (Low) | P3 |
| Статус | Новая |
| Автор | Толстихин Артём |
| Шаги воспроизведения | 1. Зайти в проект. 2. Нажать на кнопку Play. |
| Ожидаемый результат | При нажатии кнопки нас перекинет на другую сцену |
| Фактический Результат | Ожидаемый результат выполнен. |

Кнопка PLAY работает исправно, переход на другую сцену работает корректно.

Во втором тесте мы проверим работу генерации подземелья.

Таблица 3.2. – Тест-кейс «Генерация подземелья»

|  |  |
| --- | --- |
| Короткое описание | После нажатия на кнопку играть, должно автоматически генерироваться подземелье |
| Проект | Medievality |
| Номер версии | 1 |
| Важность:   * S1 Блокирующая (Blocker) * S2 Критическая (Critical) * S3 Значительная (Major) * S4 Незначительная (Minor) * S5 Тривиальная (Trivial) | S5 |
| Приоритет:   * P1 Высокий (High) * P2 Средний (Medium) * P3 Низкий (Low) | P3 |
| Статус | Новая |
| Автор | Толстихин Артём |
| Шаги воспроизведения | 1) Переходим на сцену с подземельем. |
| Ожидаемый результат | При переходе на сцену генерируется подземелье |
| Фактический Результат | Ожидаемый результат выполнен |

После нажатии кнопки PLAY подземелье исправно генерируется.

В третьем тесте мы проверим атаку игрока, после того как игрок атакует должна воспроизводиться анимация, и, если он попал по противнику, то противник должен исчезнуть.

Таблица 3.3. – Тест-кейс «Атака персонажа»

|  |  |
| --- | --- |
| Короткое описание | После нажатия на левую кнопку мыши |
| Проект | Medievality |

Продолжение Таблицы 3.3

|  |  |
| --- | --- |
| Номер версии | 1 |
| Важность:   * S1 Блокирующая (Blocker) * S2 Критическая (Critical) * S3 Значительная (Major) * S4 Незначительная (Minor) * S5 Тривиальная (Trivial) | S5 |
| Приоритет:   * P1 Высокий (High) * P2 Средний (Medium) * P3 Низкий (Low) | P3 |
| Статус | Новая |
| Автор | Толстихин Артём |
| Шаги воспроизведения | 1)Нажимаем левую кнопку мыши для атаки.  2) Противник получает урон и исчезает |
| Ожидаемый результат | При переходе на сцену генерируется подземелье |
| Фактический Результат | Ожидаемый результат выполнен |

Тест выполнен. При нажатии левой кнопки мыши персонаж совершает атаку, противник получает урон.

В четвертом тесте протестируем получение урона от врагов.

Таблица 3.4. – Тест-кейс «Получение урона»

|  |  |
| --- | --- |
| Короткое описание | После того как нас атакует слайм, должно уменьшаться кол-во здоровья |
| Проект | Medievality |
| Номер версии | 1 |

Продолжение Таблицы 3.4

|  |  |
| --- | --- |
| Важность:   * S1 Блокирующая (Blocker) * S2 Критическая (Critical) * S3 Значительная (Major) * S4 Незначительная (Minor) * S5 Тривиальная (Trivial) | S5 |
| Приоритет:   * P1 Высокий (High) * P2 Средний (Medium) * P3 Низкий (Low) | P3 |
| Статус | Новая |
| Автор | Толстихин Артём |
| Шаги воспроизведения | 1. Ждем пока нас атакует слайм, чтобы получить урон 2. Получаем урон |
| Ожидаемый результат | После удара слайма здоровье отнимется |
| Фактический Результат | После удара здоровье отнимается |

Удар слайма работает исправно, после нанесения урона здоровье игрока уменьшается.

В пятом тесте протестируем восстановление здоровья от зелья

Таблица 3.5. – Тест-кейс «Зелье здоровья»

|  |  |
| --- | --- |
| Короткое описание | После того как мы потеряем здоровье, мы можем подобрать зелье здоровья, и после этого восстановить себе здоровье |
| Проект | Medievality |
| Номер версии | 1 |

Продолжение таблицы 3.5

|  |  |
| --- | --- |
| Важность:   * S1 Блокирующая (Blocker) * S2 Критическая (Critical) * S3 Значительная (Major) * S4 Незначительная (Minor) * S5 Тривиальная (Trivial) | S5 |
| Приоритет:   * P1 Высокий (High) * P2 Средний (Medium) * P3 Низкий (Low) | P3 |
| Статус | Новая |
| Автор | Толстихин Артём |
| Шаги воспроизведения | 1)Убиваем противников, чтобы с них выпало зелье лечения.  2)Подбираем зелье, чтобы здоровье восстановилось. |
| Ожидаемый результат | После подбора зелья здоровье восстановится |
| Фактический Результат | Ожидаемый результат выполнен |

Зелье здоровье работает исправно, здоровье восполняется.

В шестом тесте протестируем восстановление здоровья от Большого зелья

Таблица 3.6. – Тест-кейс «Большое зелье здоровья»

|  |  |
| --- | --- |
| Короткое описание | После того как мы потеряем здоровье, мы можем подобрать Большое зелье здоровья, и после этого восстановить себе здоровье, большое зелье падает намного реже чем маленькое, но и восстанавливает 3 здоровья вместо одного |
| Проект | Medievality |
| Номер версии | 1 |

Продолжение таблицы 3.6

|  |  |
| --- | --- |
| Важность:   * S1 Блокирующая (Blocker) * S2 Критическая (Critical) * S3 Значительная (Major) * S4 Незначительная (Minor) * S5 Тривиальная (Trivial) | S5 |
| Приоритет:   * P1 Высокий (High) * P2 Средний (Medium) * P3 Низкий (Low) | P3 |
| Статус | Новая |
| Автор | Толстихин Артём |
| Шаги воспроизведения | 1)Убиваем противников, чтобы с них выпало зелье лечения.  2)Подбираем зелье, чтобы здоровье восстановилось. |
| Ожидаемый результат | После подбора зелья здоровье восстановится |
| Фактический Результат | Ожидаемый результат выполнен |

Большое зелье здоровье работает исправно, здоровье восполняется на 3 единицы.

В седьмом тесте протестируем работу смерти персонажа.

Таблица 3.7. – Тест-кейс «Смерть игрока»

|  |  |
| --- | --- |
| Короткое описание | После того как мы потеряем всё здоровье, персонаж умрет, и высветится окно с уведомлением о смерти, где мы можем либо перезапустить, либо выйти в главное меню |
| Проект | Medievality |
| Номер версии | 1 |

Продолжение таблицы 3.7

|  |  |
| --- | --- |
| Важность:   * S1 Блокирующая (Blocker) * S2 Критическая (Critical) * S3 Значительная (Major) * S4 Незначительная (Minor) * S5 Тривиальная (Trivial) | S5 |
| Приоритет:   * P1 Высокий (High) * P2 Средний (Medium) * P3 Низкий (Low) | P3 |
| Статус | Новая |
| Автор | Толстихин Артём |
| Шаги воспроизведения | 1)Даём противникам нанести себе урон.  2)Когда кол-во здоровья опускается до нуля, вылезает окно. |
| Ожидаемый результат | После того как здоровье заканчивается, персонаж умирает и высвечивается окно с уведомление о смерти, где мы можем либо перезапустить, либо выйти в главное меню |
| Фактический Результат | Ожидаемый результат выполнен |

Механика смерти работает исправно, после того как игрок получает смертельный урон, он умирает, и после этого высвечивается окно с уведомлением о смерти игрока, где он может выбрать один из двух вариантов.

В восьмом тесте протестируем работу победы игрока над всеми противниками.

Таблица 3.8. – Тест-кейс «Победа игрока»

|  |  |
| --- | --- |
| Короткое описание | После того как мы убиваем всех противников на уровне, должно высветиться уведомление о победе. |
| Проект | Medievality |
| Номер версии | 1 |

Продолжение Таблицы 3.8

|  |  |
| --- | --- |
| Важность:   * S1 Блокирующая (Blocker) * S2 Критическая (Critical) * S3 Значительная (Major) * S4 Незначительная (Minor) * S5 Тривиальная (Trivial) | S5 |
| Приоритет:   * P1 Высокий (High) * P2 Средний (Medium) * P3 Низкий (Low) | P3 |
| Статус | Новая |
| Автор | Толстихин Артём |
| Шаги воспроизведения | 1)Убиваем всех противников на уровне.  2)Когда кол-во противников на уровне достигает нуля. |
| Ожидаемый результат | После того как все противники умирают, высвечивается уведомление о победе. |
| Фактический Результат | Ожидаемый результат выполнен |

Механика победы работает исправно, после того как игрок убивает всех противников, после этого высвечивается окно с уведомлением о победе игрока, где он может выбрать один из двух вариантов.

В девятом тесте мы проверим работу кнопки выхода в главном меню.

Таблица 3.9. – Тест-кейс «Выйти из игры»

|  |  |
| --- | --- |
| Короткое описание | Протестировать кнопку в главном меню. |
| Проект | Medievality |
| Номер версии | 1 |

Продолжение Таблицы 3.9

|  |  |
| --- | --- |
| Важность:   * S1 Блокирующая (Blocker) * S2 Критическая (Critical) * S3 Значительная (Major) * S4 Незначительная (Minor)   S5 Тривиальная (Trivial) | S5 |
| Приоритет:   * P1 Высокий (High) * P2 Средний (Medium) * P3 Низкий (Low) | P3 |
| Статус | Новая |
| Автор | Толстихин Артём |
| Шаги воспроизведения | 1. Зайти в проект. 2. Нажать на кнопку Exit. |
| Ожидаемый результат | При нажатии кнопки нас перекинет на другую сцену |
| Фактический Результат | Ожидаемый результат выполнен. |

Кнопка EXIT работает исправно, выход из игры работает корректно.

В десятом тесте протестируем работу скрипта на отображение кол-ва слаймов.

Таблица 3.10. – Тест-кейс «Кол-во соперников»

|  |  |
| --- | --- |
| Короткое описание | В этом тесте мы проверим работу скрипта, отображающего кол-во слаймов. |
| Проект | Medievality |
| Номер версии | 1 |
| Важность:   * S1 Блокирующая (Blocker) * S2 Критическая (Critical) * S3 Значительная (Major) * S4 Незначительная (Minor) | S5 |

Продолжение Таблицы 3.10

|  |  |
| --- | --- |
| * S5 Тривиальная (Trivial) |  |
| Приоритет:   * P1 Высокий (High) * P2 Средний (Medium) * P3 Низкий (Low) | P3 |
| Статус | Новая |
| Автор | Толстихин Артём |
| Шаги воспроизведения | 1) Запускаем уровень, проверяем скрипт кол-ва слаймов.  2)Количество слаймов не отображается.  3) Проверяем почему не работает, не работает из-за неправильного атрибута в скрипте .  4)Заменяем атрибут на нужный, скрипт начал работать. |
| Ожидаемый результат | Кол-во слаймов корректно отображается |
| Фактический Результат | Ожидаемый результат не выполнен. |

Скрипт количества слаймов не работал, потребовалось заменить атрибут в скрипте на нужный, так как текст был другой.

**Заключение**

Были проанализированы популярные средства разработки. В ходе анализа, было проведено их сравнение и выбраны наиболее актуальные средства разработки для начинающих разработчиков.

В ходе анализа стало ясно, что при разработке компьютерной игры с простой игровой механикой, стоит обратить внимание на дополнительные элементы игры, такие как сюжет и графическое оформление.

Это нужно для того, чтобы удержать потенциального игрока и продлить жизненный цикл разработки.

В проекте были рассмотрены и проведены анализы требований к программному продукту, проектированию. Для этих целей были использованы различные методы и средства, зарекомендовавшие себя в мировой практике разработки программных продуктов.

Основываясь на всей полученной в ходе исследования информации, было решено спроектировать прототип двумерного платформера для одного игрока на игровом движке Unity.

Такое решение было принято по нескольким причинам:

1. двумерная графика, в отличие от трехмерной легче в создании;

2. по игровой механике, игра жанра платформер проще реализуется;

3. игровой движок Unity распространяется бесплатно и позволяет разрабатывать приложения на языке программирования.

C# Освоение среды разработки Unity несет не маловажный характер, так как в современном мире индустрия разработки игр все сильнее распространяется в нашем обществе. Игры перестали быть лишь предметом для развлечений, и теперь используются и в других областях, например, в науке или в обучении пользователей. Поэтому развитие в данном направлении можно считать одним из самых важных в современном обществе. Задачи, в рамках текущей работы, были выполнены в полном объеме.

# **Список использованных источников**

Интернет источники:

1. Adobe Photoshop Elementshelp. Учебник по фотошопу [Электронный ресурс]. Режим доступа: https://helpx.adobe.com/pdf/photoshop\_reference.pdf - свободный;
2. bibliofond. Сайт с библиотекой шрифтов [Электронный ресурс]. Режим доступа. https://www.bibliofond.ru/view - свободный;
3. flossmanuals. Игровой движок [Электронный ресурс]. Режим доступа. https://fr.flossmanuals.net/creating-point-and-click-games-point-and-click-games/ - свободный;
4. GameMaker. Игровой Движок [Электронный ресурс]. Режим доступа. https://www.yoyogames.com/ - свободный;
5. Gamesisart. Игровой Движок [Электронный ресурс]. Режим доступа. https://gamesisart.ru/game\_.html#Game\_Create\_1 - свободный;
6. JMonkey. Игровой Движок [Электронный ресурс]. Режим доступа. https://jmonkeyengine.org/ - свободный;
7. Marmalade. Игровой Движок [Электронный ресурс]. Режим доступа. https://www.madewithmarmalade.com/ - свободный;
8. Mmo. Игровой Движок [Электронный ресурс]. Режим доступа. https://mmo13.ru/games/feature/ - свободный;
9. OGRE 3D. Игровой Движок [Электронный ресурс]. Режим доступа. http://www.ogre3d.org/ - свободный;
10. Shiva. Игровой Движок [Электронный ресурс]. Режим доступа. http://www.shiva-engine.com/ - свободный;
11. SIO2. Игровой Движок [Электронный ресурс]. Режим доступа. http://www.sio2 interactive.com/index.php - свободный;
12. Teletype. Библиотека Кистей [Электронный ресурс]. Режим доступа. https://teletype.in/@karaponich/4yLXgxa-5 - свободный;
13. Turbulenz. Игровой Движок [Электронный ресурс]. http://biz.turbulenz.com/home. - свободный.

# **Приложение А**

# **Скриншоты игры**



Рисунок А1– Скриншот главного меню.

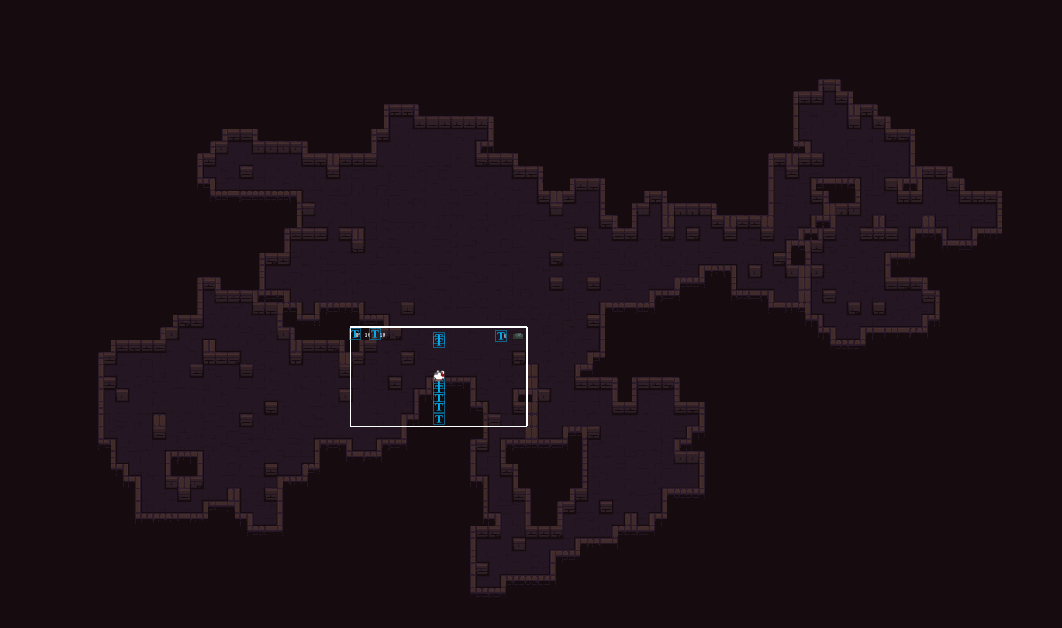


Рисунок А2 – Скриншот генерации подземелья.

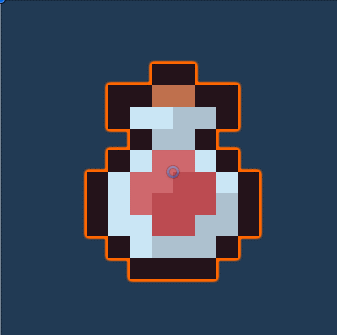


Рисунок А3 – Зелье лечения.

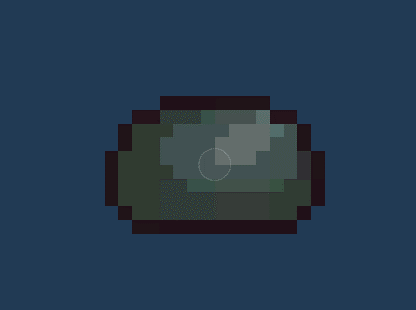


Рисунок А4 – Противник слайм.

# **Приложение Б**

**Содержание CD – диска**

1. Исходный код.
2. Пояснительная записка.
3. Исполняемые файлы программы.
4. Структура сцен в приложении.