Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Уральский федеральный университет
имени первого Президента России Б.Н. Ельцина»

Институт радиоэлектроники и информационных технологий – РТФ
Школа профессионального и академического образования

ДОПУСТИТЬ К ЗАЩИТЕ ПЕРЕД ГЭК

РОП И.Н. Обабков

( подпись)

«29» мая 2023 г.

# ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

РАЗРАБОТКА ИГРЫ
«СИМУЛЯТОР ВЛАДЕЛЬЦА ПРОДУКТА»

Научный руководитель: Кошелев А. А. подпись
к.ф.-м.н., доцент
Нормоконтролер: Огуренко Е.В. подпись
Студент группы РИМ-210990 Коростелев Д. С. подпись

Екатеринбург
2023

Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования
«Уральский федеральный университет имени первого Президента России Б.Н. Ельцина»

Институт радиоэлектроники и информационных технологий – РТФ
Школа профессионального и академического образования
Направление подготовки 09.04.04 Программная инженерия
Образовательная программа Разработка и управление в программных проектах

**УТВЕРЖДАЮ**

РОП 09.04.04 Обабков И.Н.

«29» мая 2023 г.

# ЗАДАНИЕ

на выполнение ВКР

**студента** *Коростелева Дениса Станиславовича* **группы** *РИМ-210990*
(фамилия, имя, отчество)

**1. Тема выпускной квалификационной работы** *Разработка игры «Симулятор Владельца Продукта»*
Утверждена распоряжением по институту ИРИТ-РТФ от «____» _____ 2023 г. № _____

**2. Руководитель** *Кошелев Антон Александрович, к.ф.-м.н., доцент.*
(Ф.И.О., должность, ученое звание, ученая степень)

**3. Исходные данные к работе** *материалы, полученные в ходе преддипломной практики, техническая литература, техническая документация*

**4. Перечень демонстрационных материалов** *Презентация*

**5. Календарный план**

| № п/п | Наименование этапов выполнения работы | Срок выполнения этапов работы | Отметка о выполнении |
|---|---|---|---|
| 1. | *Разработка игры* | до 15 апреля 2023 г. | |
| 2. | *1 раздел (Анализ инструментов разработки)* | до 22 апреля 2023 г. | |
| 3. | *2 раздел (Подробный обзор возможностей выбранного фреймворка)* | до 07 мая 2023 г. | |
| 4. | *3 раздел (Описание разработки игры)* | до 21 мая 2023 г. | |
| 5. | *Подготовка полного текста ВКР* | до 26 мая 2023 г. | |

Руководитель _____ *Кошелев А.А.*
(подпись) Ф.И.О.

Задание принял к исполнению 01.04.2023г. _____
дата (подпись)

**6. Выпускная квалификационная работа закончена** «___» _____ 2023 г.
Считаю возможным допустить *Коростелева Д.С.. к* защите его ВКР в Государственной экзаменационной комиссии.
Руководитель _____ *Кошелев А.А.*
(подпись) Ф.И.О.

**7. Допустить** *Коростелева Д.С.* к защите магистерской диссертации в Государственной экзаменационной комиссии.

РОП _____ *Обабков И.Н.*
(подпись) Ф.И.О.

## РЕФЕРАТ

Выпускная квалификационная работа магистра 69 с., 24 рис., 29 источников.

УПРАВЛЕНИЕ ПРОДУКТОМ, МЕТОДОЛОГИЯ SCRUM, КОМПЬЮТЕРНЫЕ ИГРЫ, РАЗРАБОТКА ИГР-СИМУЛЯТОРОВ.

Цель командной работы – создание игры «Симулятор Владельца Продукта». Цель личной работы – разработка игры «Симулятор Владельца Продукта» с использованием современных инструментов разработки.

Объект исследования – компьютерные игры.

Методы исследования: анализ современных инструментов разработки игр, генерация идей для лучшей реализации игры, программная реализация игры, её тестирование на наличие ошибок, проведение апробации игры на 30 пользователях, сбор и систематизация обратной связи.

В результате командной работы создан продукт «Симулятор Владельца Продукта» – это компьютерная игра, доступная в браузере. Результатом личной работы является разработанная игра, доступная в браузере, описание реализованных сцен и скриптов.

Выпускная квалификационная работа выполнена в текстовом редакторе Microsoft Word.

**СОДЕРЖАНИЕ**

# ВВЕДЕНИЕ

## Актуальность темы исследования

Стремительное развитие IT индустрии приводит к тому, что каждый год появляются новые профессии. К сожалению, после появления новой специальности проходит много времени, прежде чем появится достаточное количество подходящих кандидатур. Кроме того, очень часто для новой вакансии сложно сформировать четкие требования, такие как необходимые компетенции и обязанности. Одной из таких профессий является владелец продукта. Данный термин пришёл из методики гибкого управления проектами Scrum, к которой современные компании прибегают всё чаще и чаще. Несмотря на то, что роль владельца продукта начала зарождаться в начале 2000-х, обязанности, которые необходимо выполнять человеку, берущему на себя эту роль, трактуются каждой компанией по-разному. Это приводит к тому, что требования к кандидатуре размываются, а все эти факторы собираются в неопределенность, которая отпугивает возможных соискателей – им непонятны ни трудовые функции, ни перспективы.

Для того, чтобы избежать неопределенности среди соискателей можно заняться популяризацией новых профессий среди студентов или людей, желающих сменить сферу деятельности. Для этого можно писать статьи в популярных научных журналах или в блогах на сайтах, таких как Habr или Яндекс.Дзен, но такой способ не даст человеку никакого опыта. Ещё один вариант заключается в том, чтобы создать игру-симулятор, в которой игрок сможет почувствовать на себе хотя бы часть тех обязанностей, которые ложатся на плечи владельца продукта.

Но, как и было сказано ранее, сфера IT развивается довольно быстро, и чтобы сделать актуальную игру необходимо использовать современные решения. Например, можно воспользоваться одним из популярных игровых фреймворков, которые не только позволяют сосредоточиться на создании логики игры, а не на технических аспектах, таких как нюансы вывода изображения на экран или воспроизведения аудио, но и создать

кроссплатформенное решение, которое можно быстро адаптировать под новую платформу, не переписывая код с нуля.

Современные инструменты разработки игр также предлагают множество других преимуществ: встроенные инструменты, ускоряющие процесс разработки, такие как редакторы игровых сцен, большую гибкость, позволяющая реализовать проект любой сложности, не жертвуя при этом оригинальными игровыми механиками, поддержка множества платформ, большое сообщество разработчиков, предоставляющих дополнительную поддержку, и современное качество графики и звука.

Актуальность позволила определить тему: «Разработка игры «Симулятор Владельца Продукта»».

**Гипотеза разработки** состоит в том, что с помощью игрового фреймворка Godot Engine можно создать кроссплатформенную игру – «Симулятор Владельца Продукта».

**Целью разработки** является разработка игры «Симулятор Владельца Продукта» с помощью игрового фреймворка Godot Engine.

**Для достижения поставленной цели необходимо решить следующие задачи**:

– исследовать и обосновать выбор игрового фреймворка для разработки игры

– разработать базовую логику игры, включающую в себя процесс выбора пользовательских историй в спринт;

– добавить новые механики для улучшения и разнообразия игрового процесса;

– настроить процесс CI/CD для быстрого развертывания игры на выбранной платформе.

**Объектом исследования** являются компьютерные игры.

**Предметом исследования** является разработка компьютерных игр-симуляторов.

**Методы исследования** включают в себя:

– Анализ существующих инструментов разработки игр;

– Генерация идей для разработки игры;

– Программная реализация игры;

– Тестирование игры на наличие ошибок;

– Обработка результатов.

**Теоретической основой исследования стали** методологии по разработки ПО, техническая документация и статьи, описывающие лучшие практики разработки игр.

**Теоретическая и практическая значимость** работы заключается в предоставлении возможности обучения управлению для любого желающего, систематизации знаний и популяризации методологии SCRUM.

**База исследования**: УРФУ

**На защиту выносятся следующие положения:**

– Анализ игровых фреймворков и обоснование выбора;

– Описание процесса разработки, ключевые моменты и особенности реализации в выбранном фреймворке;

– Обоснование выбора платформы и описание реализованного процесса CI/CD

**Апробация результатов исследования и публикации.**

Результатом командной работы является продукт «Симулятор Владельца Продукта» – это компьютерная игра, доступная в браузере. По статистике было всего порядка тысячи игровых сессий и положительная обратная связь от игроков.

# 1 ОБЗОР ИНСТРУМЕНТОВ РАЗРАБОТКИ

## 1.1 Определение платформы

Первой задачей является выбор платформы, для которой будет разрабатываться игра. Для максимального охвата целевой аудитории необходимо выбрать наиболее универсальную платформу. Одной из таких платформ является разработка игры как веб-приложения, что обладает следующими преимуществами:

В игру доступную в Интернете можно играть на широком спектре устройств, включая настольные компьютеры, ноутбуки, планшеты и смартфоны, без необходимости каких-либо дополнительных загрузок или установок. Это означает, что игроки могут легко получать доступ к играм и играть в них без каких-либо барьеров, что может привести к увеличению аудитории и повышению вовлеченности.

Еще одним преимуществом разработки игр для Интернета является то, что ими можно легко делиться и продвигать через социальные сети, электронную почту и другие онлайн-каналы. Это может помочь повысить узнаваемость и популярность игры.

Веб-технологии, такие как HTML5 и WebGL, позволили создавать высококачественные, захватывающие игры с продвинутой графикой и интересным геймплеем, которые конкурируют с нативными или мобильными играми, не требуя того же уровня ресурсов, что и нативная игра.

## 1.2 Обзор фреймворков для создания игр

Разработка игры с нуля довольно сложный и комплексный процесс, потому что кроме разработки логики игры, необходимо предусмотреть следующие аспекты:

– Рендеринг – процесс генерации и вывода на экран двухмерного изображения, учитывающий положение и настройки виртуальной камеры, источники освещения, расположение объектов, находящихся в поле зрения камеры, их текстуры, тени, которые они отбрасывают и прочие аспекты [8].

– Симуляция физики, которая позволяет имитировать физическое поведение объектов, например, падение объектов в соответствии с гравитацией, учёт массы при ускорении, замедлении и столкновении объектов.

– Поддержка анимаций, которая позволяет использовать заранее подготовленные анимации в игре, отслеживать их состояние и переключаться между ними.

– Возможность проигрывания музыки и звуковых эффектов, наложения их друг на друга и проигрывание звуков с разной громкостью с целью имитации положения источника звука относительно персонажа игрока.

– Стабильность работы игры на разных платформах, что позволяет охватить большее количество потенциальных пользователей.

Исходя из этого к разработке игр с нуля прибегают только крупные компании или большие команды разработчиков. В качестве альтернативы для разработчиков с меньшим количеством ресурсов существуют инструменты, которые полностью или частично реализуют эти аспекты.

Данными инструментами могут выступать как фреймворки, разработанные для разных языков программирования, так и полноценные программы, сочетающие в себе вышеперечисленные аспекты и более специфичные особенности, упрощающие разработку и позволяющие разработчику реализовать более комплексные решения [1].

Кроме того, игровые фреймворки имеют следующие преимущества:

– Меньшее время разработки: игровые фреймворки предоставляют готовые инструменты и функциональность, которые могут сэкономить разработчикам время и усилия. Они предлагают такие функции, как системы симуляции физики, системы рендеринга и воспроизведения звуковых эффектов, которые можно использовать для быстрого создания игр.

– Кроссплатформенная поддержка: Многие игровые фреймворки поддерживают несколько платформ, таких как персональные компьютеры, мобильные устройства и консоли. Это означает, что разработчики могут

создавать игры для нескольких платформ, используя одну и ту же кодовую базу, что может сэкономить время и ресурсы.

– Поддержка сообщества: Популярные игровые фреймворки имеют большие сообщества разработчиков, которые делятся ресурсами, учебными пособиями и поддержкой.

– Масштабируемость: Игровые фреймворки предназначены для работы с большими, комплексными играми с большим количеством ресурсов и обширной кодовой базой. Они предлагают инструменты для управления этими ресурсами, оптимизации производительности и отладки игры, которые могут облегчить масштабирование по мере усложнения проекта.

Для выбора наиболее подходящего инструмента для создания игры был проведён анализ популярных решений, таких как Unity, Unreal Engine и Godot Engine. Кроме того, в качестве альтернативы большим игровым фреймворкам было решено также рассмотреть фреймворки для разработки игр – MonoGame для разработки на C# и PyGame для разработки на Python.

### 1.2.1 Unity

Unity – популярный кроссплатформенный игровой фреймворк, который широко используется для разработки игр и интерактивных интерфейсов. Он был создан в 2005 году датской компанией Unity Technologies и с тех пор стал одним из самых популярных игровых движков в индустрии. Основной задачей при создании игрового фреймворка Unity Techologies ставили для себя сделать создание игр более доступным [12].

Из преимуществ Unity можно выделить следующее:

– Кроссплатформенная поддержка: Unity поддерживает разработку игр для множества платформ, включая персональные компьютеры, мобильные устройства, устройства виртуальной реальности, консоли и веб-браузеры [24].

– Мощный визуальный редактор: Unity предоставляет ряд инструментов и функций для создания игр, включая систему анимации, систему освещения,

систему симуляции физики и систему выполнения скриптов, управляющих поведением игровых объектов [3;5].

– Активное сообщество: Unity располагает большим и активным сообществом, которое предоставляет разработчикам множество ресурсов, называемых ассетами, руководств и форумов поддержки.

– Unity Asset Store: Unity предоставляет ряд готовых ресурсов, таких как 3D-модели, звуковые эффекты и плагины, которые можно использовать для ускорения и упрощения процесса разработки [27].

– Ценообразование: Unity предлагает несколько вариантов оплаты, основываясь на потребностях разработчика и его доходах [29]. В том числе существует бесплатная персональная версия, что делает Unity более доступным.

К недостаткам Unity можно отнести следующее:

– Кривая обучения: из-за того, что Unity развивается уже больше 15 лет, он накопил довольно обширный набор функциональности, что может усложнить начинающему разработчику процесс ознакомления с возможностями игрового фреймворка.

– Проблемы с производительностью: как упоминалось ранее, Unity имеет довольно обширный набор инструментов и возможностей, что может отрицательно сказаться на производительности, особенно при разработке больших игр [21].

Важным фактором любого игрового фреймворка является стоимость его использования. Unity предоставляет следующие варианты предоставления лицензии:

– Unity Personal: бесплатный вариант использования Unity. Позволяет использовать большинство основных функций фреймворка и персональный доступ к некоторым облачным сервисам. Имеет ограничения по предоставляемым сервисам и может использоваться только если годовой доход от использования Unity составляет не более 100 тысяч долларов.

– Unity Plus: Подписка Unity Plus стоит 399$ в год или 40$ в месяц за каждое человека и предназначен для отдельных разработчиков и команд, которым необходим доступ к облачным сервисам с увеличенными лимитами и возможностью изменить заставку приложения. Но основная особенность Unity Plus в том, что он предназначен для тех команд, чей годовой доход от использования Unity составляет не более 200 тысяч долларов.

– Unity Pro: данный вариант подписки не имеет ограничения по годовому доходу и стоит 2040$ в год или 185 долларов в месяц. Unity Pro предназначен для профессиональных разработчиков и команд, которым требуется полная гибкость при создании коммерческих игр и интерактивного контента. Например, Unity Pro позволяет создавать сборки для таких закрытых платформ как Nintendo Switch, консоли PlayStation и Xbox и использовать более продвинутую систему симуляции физики.

– Unity Enterprise: данный вариант нужен только большим компаниям и кроме тех преимуществ, что предоставляет подписка Unity Pro, даёт доступ к исходному коду Unity, тем самым позволяя максимально оптимизировать свой проект [2; 23].

Существует множество популярных и известных игр 2D и 3D игр, разработанных на Unity как для мобильных устройств, так и для более серьёзных платформ, таких как персональные компьютеры или консоли. Среди них можно выделить Cuphead, Ori And The Blind Forest, Fe, Cities: Skylines, Pokemon Go, Subnautica, Hollow Knight, Beat Saber и Genshin Impact.

### 1.2.2 Unreal Engine

Unreal Engine – популярный игровой фреймворк, который широко используется для разработки высокобюджетных больших игр. Он был разработан компанией Epic Games в 1998 году и с тех пор стал одним из самых популярных игровых движков в индустрии. Несмотря на то, что Unreal Engine появился в 1998 году, он постоянно обновляется и в 2022 году уже вышла

пятая версия, позволяющая создать ещё более реалистичную графику в играх [11].

Преимуществами Unreal Engine являются:

– Реалистичная графика: Unreal Engine известен своими расширенными графическими возможностями, включая трассировку лучей в реальном времени, высококачественное освещение и динамические погодные эффекты.

– Визуальный язык программирования Blueprint: полноценная система сценариев игрового процесса, основанная на концепции использования интерфейса на основе узлов для создания элементов игрового процесса позволяет разработчикам создавать сложную игровую логику без необходимости написания кода [26].

– Мощный графический редактор: редактор Unreal Engine удобен в использовании и предоставляет ряд инструментов и функций для создания игр, включая систему анимации, систему симуляции физики и язык скриптов.

– Кроссплатформенная поддержка: как и подобает современному игровому движку, Unreal Engine поддерживает множество платформ для создания игр, такие как ПК, мобильные устройства, устройства виртуальной реальности и консоли [3].

– Активное сообщество: за 20 лет своего существования Unreal Engine успел обзавестись большим и активным сообществом, которое предоставляет разработчикам множество ресурсов и руководства для решения типовых задач.

К сожалению, Unreal Engine не идеален и имеет свои недостатки, из которых хотелось бы выделить следующие:

– Сложность: Unreal Engine имеет обширный функционал, накопленный за всё своё время существования, а также использует для написания скриптов C++ в качестве языка программирования, который является по мнению сообщества не самым легким в освоении, что может усложнить процесс изучение игрового фреймворка начинающими разработчиками.

– Производительность: реалистичная графика Unreal Engine приводит к тому, что игры могут иметь проблемы с производительностью.

– Стоимость: Лицензионные сборы за использование Unreal Engine могут быть довольно большими для крупных команд разработчиков.

Unreal Engine имеет довольно щадящую систему оплаты для проектов, чей суммарный доход не превышает 1 миллион долларов. В случае если продукт за всё время своего существования принёс более 1 миллиона долларов, то разработчики обязаны отчислять лицензионный сбор в размере 5% от прибыли [28].

На Unreal Engine разработано множество успешных проектов. Среди этих проектов хотелось бы упомянуть Fortnite, Yoshi’s Crafted World, Hellblade: Senua’s Sacrifice, Star Wars Jedi: Fallen Order, Sea Of Thieves и Borderlands 3.

### 1.2.3 Godot

Godot – популярный игровой фреймворк с открытым исходным кодом, разработка которого была начата в 2007 году Хуаном Линиецки и Ариэлем Манзуром. Первый релиз Godot был только в 2014 году, из тех пор он активно развивается, на данный момент уже выпущена 4 версия игрового фреймворка. Он был разработан как гибкий и простой в использовании игровой фреймворк, который можно было бы использовать для разработки 2D и 3D игр для различных платформ, включая ПК, мобильные устройства и веб-приложений [25].

Рассмотрим преимущества Godot, которые привлекают разработчиков:

– Простота в освоении: Godot предоставляет удобный интерфейс и мощный редактор, который позволяет разработчикам создавать игры быстро и эффективно. Он также предлагает ряд инструментов и функций для создания игр, включая систему симуляции физики, систему анимации и язык написания скриптов.

– Легковесность: в отличии от таких гигантов индустрии как Unity и Unreal Engine, Godot тратит очень мало ресурсов и занимает всего от 28 до 190 МБ, что открывает доступ к разработке игр большему количеству разработчиков [19].

– Гибкость: Godot легко настраивается, что позволяет разработчикам создавать игры, соответствующие их конкретным потребностям. Он также поддерживает несколько языков программирования, включая C++, C# и GDScript.

– GDScript: команда Godot разработала свой язык программирования, который не только имеет схожий с Python синтаксис, но и оптимизирован специально для разработки игр [15].

– Кроссплатформенность: как и все популярные игровые движки Godot позволяет разрабатывать игры под множество платформ, такие как персональные компьютеры, мобильные устройства, устройства виртуальной реальности, консоли и веб-браузеры.

Однако у Godot есть свои недостатки.

– Godot вышел в релиз относительно недавно, из-за чего его поддержка может быть не такой обширной как у более старых Unreal Engine и Unity. Кроме того, несмотря на то, что GDScript хорошо оптимизирован под нужды разработки игр, его возможности уступают таким языкам программирования как C++ и C# [3].

– Из-за того, что Godot появился намного позже, его функциональные и графические возможности сильно уступают Unity и Unreal Engine, поэтому это не лучший вариант для 3D игр с реалистичной графикой.

– Из-за открытого исходного кода игры разработанные на Godot не поддерживаются официально на закрытых платформах. Таковыми являются большинство консолей, из-за чего процесс выпуска игр на них довольно затруднён по сравнению с остальными игровыми фреймворками [10].

Если рассматривать ценовую политику Godot, то это наилучший вариант для независимых разработчиков, так как он бесплатен и не имеет никаких лицензионных сборов [13].

Из-за того, что Godot вышел в релиз относительно недавно, сейчас довольно мало известных игр, сделанных на данном игровом движке, но всё же можно выделить следующие проекты: версии Deponia для iOS и Android, переиздание Sonic Colors: Ultimate, Dome Keeper и Lumencraft.

### 1.2.4 MonoGame

MonoGame — это игровой фреймворк с открытым исходным кодом, который был создан в 2009 году группой разработчиков во главе с Шоном Харгривзом. Он разработан как кроссплатформенный фреймворк, который можно использовать для разработки игр для различных платформ, включая Windows, Mac OS, Linux, iOS, Android и другие [20].

Одним из ключевых преимуществ MonoGame и то, ради чего он был разработан, это его кроссплатформенность. Фреймворк предоставляет унифицированный API, который можно использовать для разработки игр для широкого спектра платформ, что может сэкономить разработчикам время и усилия. Он также предлагает ряд инструментов и функций для создания игр, включая мощный конвейер контента, обработку входных данных, аудио и графику [16].

Еще одним преимуществом MonoGame является ее гибкость. Фреймворк легко настраивается, что позволяет разработчикам создавать игры, соответствующие их конкретным потребностям. Он также поддерживает ряд языков программирования, включая C#, F# и Visual Basic.

Однако у использования MonoGame есть и некоторые недостатки. Одна из главных проблем заключается в том, что это относительно низкоуровневый фреймворк, а это значит, что для разработки игр с его помощью потребуется больше времени на написание кода и больше технических знаний, чем для игровых движков более высокого уровня, таких как Unity, Unreal Engine и

Godot. Кроме того, конвейер контента предоставляемый фреймворком может показаться менее удобным, чем аналогичные инструменты в других игровых фреймворках [7].

Огромным плюсом MonoGame является то, что, как и Godot, он абсолютно бесплатен и за разработанные на нём игры, не нужно платить никакие лицензионные сборы.

Несмотря на то, что MonoGame является просто фреймворком, на нём создано много популярных и успешных проектов, среди них Stardew Valley, Celeste, Streets of Rage 4 и FEZ.

### 1.2.5 Pygame

Pygame — это набор модулей Python, который используется для разработки 2D-видеоигр. Он был создан в 2000 году Питом Шиннерсом и был вдохновлен простой библиотекой DirectMedia Layer (SDL).

Преимущества использования Pygame включают в себя его простоту, удобство использования и возможность интеграции с Python, который является популярным и широко используемым языком программирования. У Pygame также есть большое сообщество разработчиков, которые вносят свой вклад в ее разработку, публикуют в сети Интернет множество ресурсов и руководств. Pygame также поддерживает множество платформ, включая Windows, Linux и macOS [17].

К недостаткам использования Pygame относится его ограниченная поддержка 3D, поскольку он в первую очередь предназначен для разработки 2D-игр. Производительность Pygame также может быть ограничена для более сложных игр, поскольку она не так оптимизирована, как другие игровые движки. Кроме того, Pygame имеет более ограниченный функционал по сравнению с большими игровыми фреймворками, такими как Unity, Unreal Engine и Godot [17].

Pygame полностью бесплатен в использовании, что делает его хорошим выбором для независимых разработчиков или команд с ограниченным бюджетом.

К сожалению, из-за ограниченности в функционале и существовании более продвинутых инструментов для разработки игр, существует довольно мало известных проектов разработанных на Pygame.

### 1.3 Выводы

Рассмотрев вышеперечисленные варианты, в качестве основного инструмента разработки был выбран Godot, потому что:

– Благодаря его легковесности можно быть уверенным, что игру сможет запустить как можно больше людей на самых разных устройствах.

– GDScript позволит быстро приступить к разработке игры, благодаря простоте и схожести синтаксиса с Python.

– Графический редактор сцен позволит быстрее компоновать сцен и редактировать анимации.

– При разработке на Godot мы не должны ничего выплачивать, а авторские права на игру полностью принадлежат нам.

– После разработки игры для запуска в веб-браузерах мы сможем быстро перенести игру на другие устройства, благодаря кроссплатформенности Godot, доработав только управление, которое может отличаться для каждой платформы.

## 2 ПОДРОБНЫЙ ОБЗОР ВОЗМОЖНОСТЕЙ GODOT ENGINE

Рассмотрим более подробно возможности выбранного игрового фреймворка.

### 2.1 Графический редактор

Godot, как и другие игровые движки, имеет единую среду разработки, в которой можно работать со всеми аспектами создания игры, такими как код, пользовательский интерфейс, визуальные эффекты и аудио. На Рисунок 1 представлен интерфейс Godot. В верхней части можно найти опции настройки проекта, создания новых сцен, кнопки переключения основной области и кнопки запуска проекта, отдельной сцены [18].

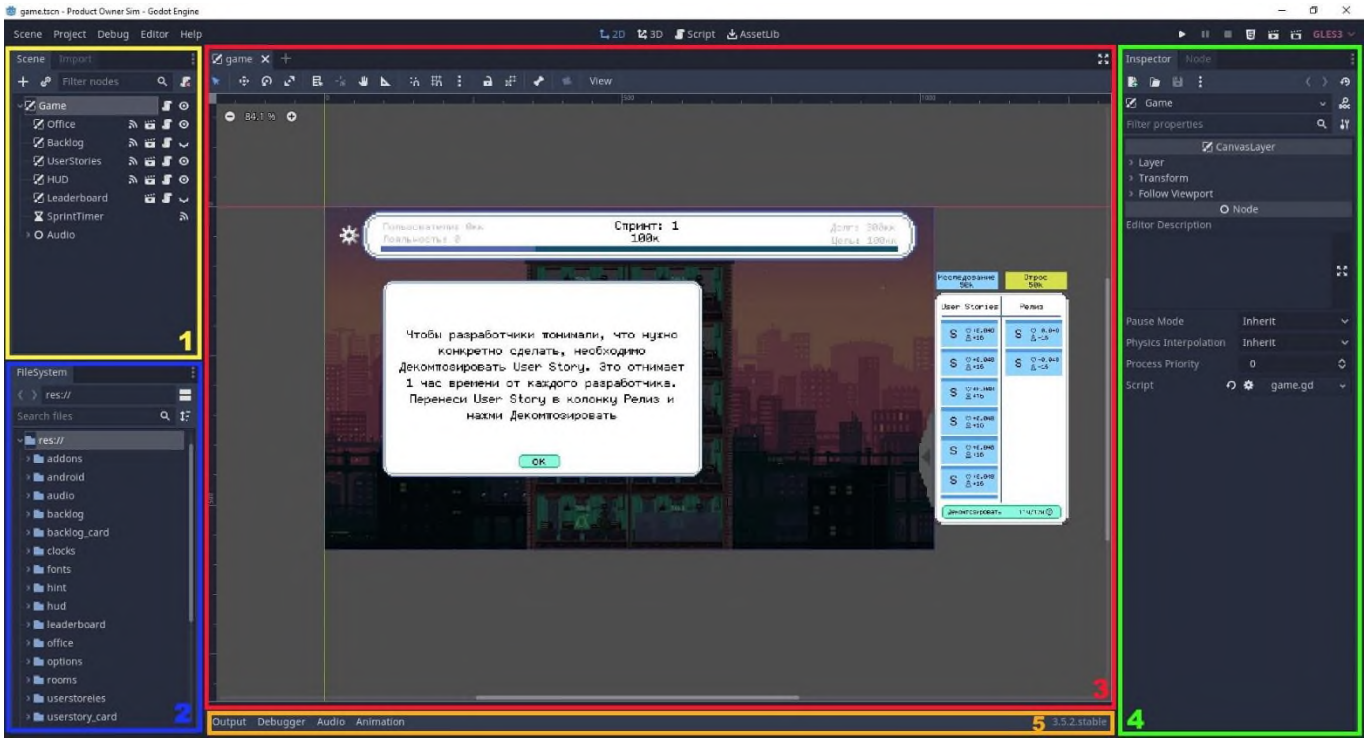


Рисунок 1 – Интерфейс Godot

В области 1 представлена иерархия открытой сцены, в которой можно видеть все добавленные узлы. В области 2 можно видеть все файлы и папки, в которых хранятся файлы сцен, аудиофайлы, графические спрайты и прочие ресурсы проекта. В области 3 выделено основное окно Godot. По умолчанию, оно представляет 2D или 3D вид открытой сцены. В верхней части данной области можно видеть инструменты, позволяющие перемещаться по сцене, передвигать объекты, скрывать их или изменять их поворот и масштаб. Кроме

того, в данной области может отображаться код скриптов или 2D или 3D вид сцены и библиотека ассетов. В области 4 представлен Инспектор, который позволяет редактировать параметры, выбранного игрового объекта. Если тип объекта наследуется от какого-нибудь другого, то параметры будут разделены по тем типам, к которым они относятся. Например, объект выделенный на Рисунок 1 имеет тип CanvasLayer, который наследуется от Node, поэтому в окне инспекторы представлены две вкладки с соответствующими названиями. В области 5 представлены кнопки, открывающие окно логирования, в котором выводятся ошибки, окно отладчика, позволяющее быстро найти проблему в коде игры, окно редактора анимации и окно настройки аудио эффектов.

### 2.2 Структура проекта

Говоря о структуре проекта Godot, нужно разобраться с двумя понятиями – сцена (Scene) и узел (Node).

В Godot узлы служат фундаментальными блоками для создания игр. Эти объекты способны выполнять ряд функций, таких как отображение спрайтов, воспроизведение анимации или представление 3D-модели объекта. Кроме того, узлы содержат свойства, которые можно изменять для настройки их поведения. Конкретные типы узлов, добавляемых в проект, будут зависеть от желаемой функциональности. Эта модульная система необходима для обеспечения гибкости при создании сложных объектов.

При добавлении узлов в Godot они упорядочиваются в иерархическую структуру, известную как дерево. Узлы добавляются как дочерние по отношению к другим узлам в дереве, и, хотя узел может иметь несколько дочерних узлов, у него может быть только один родительский узел. Эта совокупность узлов в дереве называется сценой [9].

В Godot сцены обычно используются для создания и упорядочивания игровых объектов в рамках проекта. Например, у вас может быть сцена, которая представляет персонажа, которым управляет игрок, и которая содержит все необходимые узлы и скрипты, позволяющие игровому персонажу отображаться в игре, передвигаться и сражаться. Эти различные

сцены могут быть объединены в финальную игру с помощью создания экземпляров.

### 2.3 Система симуляции физики

В основе симуляции физики лежит обнаружение столкновений, отслеживание скорости объектов, учёт их массы и силы, прилагаемой к ним, Raycast и система Ragdoll.

Обнаружение столкновений является важнейшим аспектом разработки игр, поскольку оно позволяет определить, когда два объекта пересекаются или вступают в контакт. Как только обнаруживается столкновение, обычно происходит соответствующее действие. Чтобы облегчить обнаружение столкновений и реагирование на них, Godot предоставляет множество объектов столкновения, таких как: Area, StaticBody, RigidBody, KinematicBody. Для разработки 2D игр в Godot есть аналогичные объекты, но с постфиксом «2D».

Объект Area позволяет определить, когда какой-либо объект входит или покидает область определенную, с помощью CollisionShape.

StaticBody представляет статичный объект, который не двигается в результате физических вычислений Godot. Но несмотря на это, он всё ещё участвует в проверке коллизий и может повлиять на скорость и вращение других объектов при столкновении с ними.

RigidBody является объектом, который двигается в соответствии с прилагаемыми к нему силами. Сила может быть приложена при выполнении скрипта или при коллизии с другим физическим объектом.

KinematicBody аналогичен StaticBody, потому что он не двигается в результате расчёта физики Godot, но в отличие от него движется по сцене в результате выполнения скриптом. Обычно KinematicBody представляет персонажа, которым управляет игрок.

Система Raycast или же проецирование лучей и проверка пересечений их с игровыми объектами – частая задача при разработке игр. Для проецирования лучей можно использовать такие узлы, как RayCast и

RayCast2D, однако во многих сценариях проецирование лучей требует более интерактивного подхода, который может быть достигнут с помощью кода, благодаря вызову методов PhysicsDirectSpaceState.intersect_ray() и Physics2DDirectSpaceState.intersect_ray(), которые могут учитывать не только начальную позицию и направление луча, но и маску, определяющую объекты, которые стоит игнорировать при проецировании луча.

В разработке игр система Ragdoll — это физическая система, которая имитирует движение и поведение безвольного, безжизненного тела. В Godot данная система предоставляет набор узлов и физических свойств, которые позволяют создавать объекты, похожие на тряпичную куклу, и управлять их перемещением. Ragdoll может взаимодействовать с другими физическими объектами в игровом мире и состоит из нескольких узлов, включая узлы RigidBody, CollisionShape и Joint, которые могут быть использованы для создания каркасоподобной структуры. Некоторые из особенностей системы Ragdoll включают в себя возможность прикладывать усилия и импульсы к определенным частям каркаса, возможность устанавливать ограничения на движения некоторых суставов, а также возможность сочетать анимацию и движения, основанные на физике, для более реалистичного поведения объекта.

### 2.4 Система анимации

Система анимации в Godot — это мощный инструмент, который позволяет разработчикам игр создавать сложные анимации для своих игровых объектов и управлять ими. Система анимации основана на системе ключевых кадров, где пользователь может определить серию этих кадров, которые представляют значения параметров, таких как положение, поворот или масштаб объекта в определенный момент времени. После этого значения указанные в этих кадрах интерполируются для проигрывания плавной анимации. Система анимации поддерживает различные типы анимации, включая 2D и 3D-анимацию, анимацию Ragdoll и анимацию частиц. Редактор анимации интуитивно понятен и предоставляет визуальный интерфейс для

создания и редактирования анимации. Кроме того, в ключевом кадре можно указать не только значение параметра, но и вызов метода объекта, определенного в скрипте.

Некоторые из ключевых особенностей анимационной системы Godot включают в себя возможность смешивать анимации, обеспечивая плавные переходы между различными анимациями. Система анимации также поддерживает деревья анимации, которые можно использовать для создания сложных анимаций, реагирующих на вводимые пользователем данные или другие события в игре.

### 2.5 Работа с аудио эффектами

В Godot существуют звуковые шины (audio buses) и аудиопотоки (audio streams) для воспроизведения звука и управления им в игре.

Звуковая шина — это виртуальный канал, по которому может передаваться аудио. Каждая звуковая шина имеет свой собственный набор свойств, включая громкость, высоту тона и звуковые эффекты. Направляя аудиопотоки на разные шины, можно управлять тем, как они смешиваются друг с другом, и настраивать их свойства независимо друг от друга. Например, можно создать отдельную шину для звуковых эффектов, отдельную – для музыки и отдельную – для закадрового голоса, а для изменения громкости или выключения той или другой звуковой дорожки можно просто выключить звуковую шину.

Аудиопоток — это фрагмент аудиоданных, который воспроизводится в режиме реального времени. Это может быть звуковой эффект, фрагмент фоновой музыки или любой другой тип аудио, который можно включить в свою игру. Аудиопотоки могут быть загружены из файла или сгенерированы процедурно.

Чтобы воспроизвести аудиопоток в Godot, необходимо подключить его к звуковой шине. Это можно сделать в окне инспектора, либо используя метод connect_to, передавая в качестве аргумента шину. Как только аудиопоток

подключен к шине, можно управлять его свойствами и воспроизведением, используя методы звуковой шины.

## 2.6 GDScript

Среди вариантов написания скриптов, управляющих поведением игровых объектов, в Godot существуют GDScript, C# и VisualScript, но разработчики игрового фреймворка советуют использовать GDScript, потому что он лучше всего внедрён в Godot и оптимизирован под его нужды.

### 2.6.1 Скрипты и функции обратного вызова

Хотя узлы обладают множеством свойств и функций, их поведение и возможности можно расширить, прикрепив к ним скрипт. Это позволяет написать код, который будет управлять узлом и его дочерними объектами во время игры.

В Godot есть несколько встроенных функций обратного вызова, которые используются для управления поведением узлов и скриптов. Эти функции автоматически вызываются движком в определенные моменты игрового цикла. Вот некоторые из наиболее часто используемых обратных вызовов:

– _init() вызывается, когда объект инициализируется в памяти. Является аналогом конструкторов из других языков программирования и может быть определен для передачи параметров, которые используются при инициализации объекта.

– _enter_tree() вызывается, когда узел появляется в дереве сцены. Это может происходить при создании экземпляра объекта, смене сцены или после вызова метода add_child. Если у узла есть дочерние элементы, сначала будут вызваны методы _enter_tree у родительского объект, а затем уже методы дочерних объектов. Данная функция обычно используется для инициализации параметров объекта, которые не зависят от его дочерних объектов.

– _ready() вызывается, когда узел "готов", т.е. когда у узла и его дочерних элементов выполнился метод _enter_tree. Если у узла есть дочерние элементы, их обратные вызовы _ready запускаются первыми. Таким образом в

данном методе можно определить инициализацию тех параметров узла, которые зависят от его дочерних элементов.

– _process(float delta) вызывается на этапе обработки основного цикла игры. Обработка происходит в каждом кадре и настолько быстро, насколько это возможно, поэтому разница во времени с момента получения предыдущего кадра непостоянна и равна delta. Данный метод вызывается только если он переопределен в скрипте объекта, а для того, чтобы выключить вызов метода можно вызвать метод set_process.

– _physics_process(float delta) аналогичен методу _process и вызывается каждый кадр после обновления физического фреймворка. Он используется для реализации логики, основанной на физике, такой как перемещение персонажа или обнаружение столкновений. Условия вызова метода аналогичны методу _process, но для выключения вызова необходимо вызвать set_physics_process [4].

### 2.6.2 Сценарии автозагрузки

В Godot сценарии автозагрузки — это способ автоматической загрузки скрипта при запуске игры, что делает его глобально доступным для всех узлов и сцен в проекте. Предназначение данных сценариев похоже на популярный шаблон проектирования Одиночка (Singleton), но в отличие от него может быть создано больше одного экземпляра скрипта автозагрузки. Сценарий автозагрузки полезен для создания глобальных переменных и функций, к которым необходимо обращаться на протяжении всей игры, таких как сохранение состояния игры или хранение ресурсов, используемых несколькими узлами.

Чтобы создать сценарий автозагрузки в Godot, нужно перейти в меню "Project settings", выбрать вкладку "Autoload" и добавить путь к сценарию, который необходимо загрузить.

Одной из важных проблем является то, что автоматически загружаемые скрипты всегда загружены и выполняются в фоновом режиме, поэтому важно,

чтобы они были как можно менее нагруженными, чтобы избежать проблем с производительностью [22].

### 2.6.3 Система сигналов

Сигналы в Godot — это способ взаимодействия узлов друг с другом несвязанным и гибким способом. Они позволяют узлам посылать сигнал, когда происходит определенное событие, такие как падение персонажа с большой высоты или новый рекорд, и другие узлы могут подписываться на этот сигнал, чтобы узнать о произошедшем событии и обработать его.

Сигналы основаны на шаблоне Наблюдатель (Observer), который является популярным шаблоном программирования, допускающим слабую связь между объектами. Реализуя данный шаблон, объекты могут подписываться на события, которые генерируются другими объектами, и реагировать на эти события, не имея прямых сведений об объекте, который генерировал событие [2].

В Godot сигналы могут использоваться различными способами. Например, узел кнопки может выдавать сигнал при нажатии на него, и другие узлы могут подключаться к этому сигналу для выполнения действия при нажатии на кнопку. Аналогично, узел игрока может испускать сигнал при столкновении с вражеским узлом, и узел менеджера игры может подключиться к этому сигналу, чтобы уменьшить здоровье игрока.

Некоторые преимущества использования сигналов в Godot включают в себя:

– Слабую связь: сигналы позволяют узлам взаимодействовать друг с другом, не имея прямого представления о деталях реализации друг друга. Это может сделать код более модульным, менее зависимым от реализации остальных компонентов и более простым в обслуживании.

– Гибкость: несколько узлов могут подключаться к одному и тому же сигналу, но в то же время один узел может быть подключен к нескольким

узлам, а подключаться к сигналам и отключаться от них можно в момент выполнения программы.

В Godot не только существуют заготовленные сигналы, такие как «button_pressed» узла Button, но и возможность создавать собственные сигналы. Для этого в скрипте узла необходимо добавить ключевое слово signal, после этого имя этого сигнала и опциональные параметры, которые будут получать подписчики сигнала.

### 2.6.4 Система экспорта на разные платформы

Экспорт проекта Godot в формат, зависящий от платформы, необходим для создания распространяемой версии игры, которую можно запускать на других устройствах. В момент экспорта проекта Godot, создается упакованная версия игры, которая включает в себя все необходимые файлы и ресурсы, такие как скрипты, сцены, текстуры и звуки.

В отличие от компиляции проекта, его экспорт намного проще, так как разработчику не нужно разбираться в SDK каждой платформы, для которой нужно создать версию игры.

Кроме этого, у экспорта проекта есть следующие преимущества:

– Кроссплатформенность: Экспорт игры в формат, специфичный для целевой платформы, позволяет распространять игру на другие устройства. Кроме того, не нужно разбираться в ключевых особенностях той платформы, для которой нужно экспортировать проект. Из таких особенностей можно выделить сжатие текстур, которое отличается в зависимости от платформы – на персональных компьютерах это S3TC, на iOS – это PVRTC, а на Android – это ETC [14].

– Производительность: Экспорт игры более производителен по сравнению с компиляцией, за счёт оптимизации игры под конкретную платформу. Например, в процессе экспорта можно удалить неиспользуемые ресурсы и выбрать способ сжатия файлов, чтобы уменьшить размер файла игры и сократить время загрузки.

– Совместимость: Экспорт игры гарантирует, что она будет работать на целевой платформе без каких-либо проблем с совместимостью. Это особенно важно для мобильных устройств, которые могут иметь разные аппаратные характеристики или версии операционной системы.

– Распространение: Экспорт игры упрощает её распространение через магазины приложений.

# 3 РАЗРАБОТКА ИГРЫ

## 3.1 Краткое описание игры

Игроку предстоит взять на себя роль владельца продукта и заработать один миллион долларов. Для этого ему необходимо принимать важные решения:

– Какие пользовательские истории стоят реализовать в текущим спринте?

– Стоит ли потратить деньги на исследование рынка или лучше узнать мнение пользователей о продукте?

– Пришло ли время расширяться и стоит ли нанять новых роботов?

– Стоит ли сосредоточиться на исправлении багов и технического долга или же стоит добавить в продукт новый функционал?

Для того, чтобы принять правильное решение ему необходимо учитывать следующие факторы:

– Чтобы реализовать пользовательскую историю, нужно её сначала декомпозировать – разделить на конкретные задачи, которые роботы смогут реализовать.

– Деньги на развитие продукта были взяты в кредит и каждый спринт необходимо выплачивать его часть.

– На работу роботов нужно также тратить некоторую часть доходов каждый спринт.

– Выгоду от пользовательских историй нельзя точно оценить заранее, поэтому количество пользователей и лояльности может отличаться от тех, что были показаны после исследования.

– Если долго не выпускать обновления для продукта, то пользователи перестанут им пользоваться, а лояльность будет падать.

Основной экран игры представлен на Рисунок 2. В области 1 выделен HUD, в котором пользователь может видеть количество пользователей, их лояльность продукту, номер спринта, текущее количество денег, текущее

значение долга и сумму, которую нужно набрать для победы. Кроме того, слева расположена шестеренка, по нажатию на которую открывается окно настроек.

В области 2 показан игровой офис – большинство комнат выглядят пустыми, так как они только доступны для покупки. В единственной открытой комнате видны роботы, которые и выполняют все поставленные задачи.

В области 3 можно видеть «блокнот» – список пользовательских историй, находящихся в разработке. Элементы в области 4 отвечают за появление новых пользовательских историй и выбор тех, которые нужно декомпозировать.

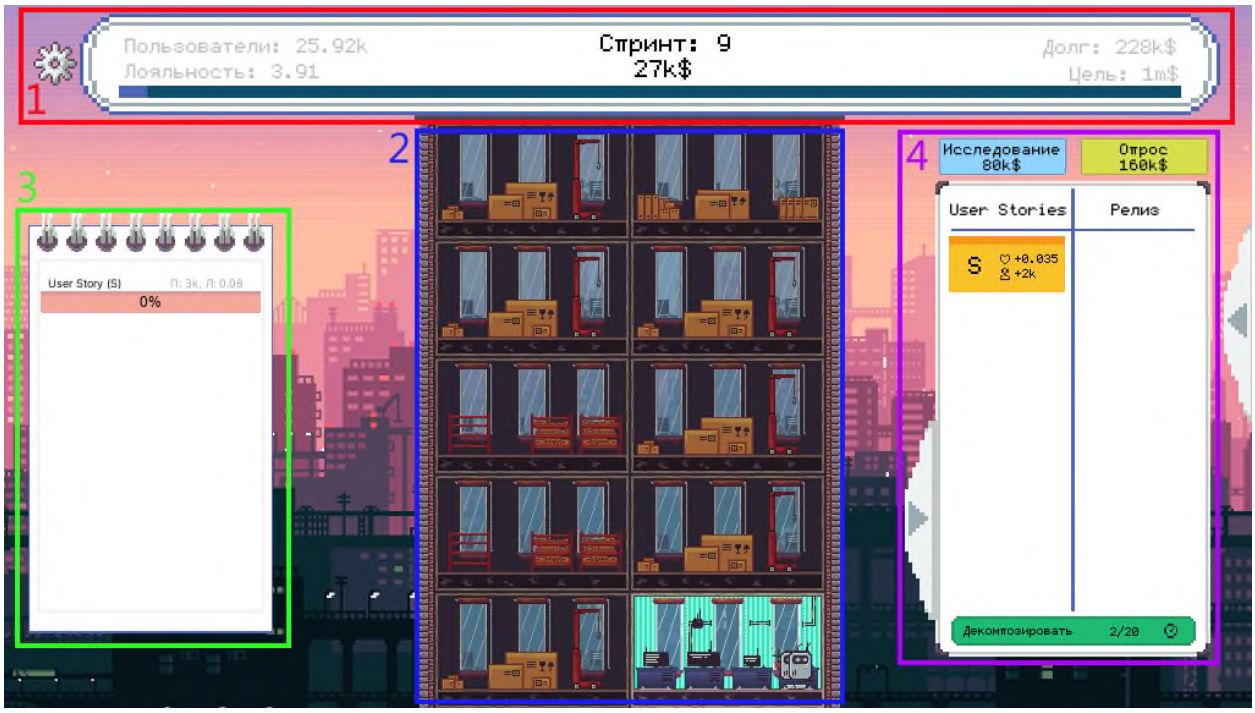


Рисунок 2 – Основной экран игры

### 3.2 Обучение

Для того, чтобы ввести пользователя в курс дела, игра начинается с обучения. В нем посредством всплывающих окон игроку предлагается перейти по ссылке на обучающий курс и ввести название своего продукта. Название продукта используется при добавлении записи в таблицу рекордов в случае победы. Далее каждое всплывающее окно, описывающее игровой процесс, сопровождается подсветкой определённых элементов интерфейса.

Подсказки продолжают появляться до тех пор, пока пользователь не доведет до релиза первую пользовательскую историю. После этого подсказки будут появляться только при появлении новых игровых элементов. Это происходит при полной выплате долга и после исправления первого бага.

Рассмотрим элементы, участвующие в обучении игрока. Большая часть обучающих подсказок находится в отдельном узле HUD, который отвечает за интерфейс, отображающийся постоянно и предоставляющий информацию о прогрессе игрока.

Элементы в данной иерархии можно разделить на две группы – элементы-подсказки, которые появляются в определенные моменты игры для описания той или иной игровой механики и функциональные элементы.

К элементам-подсказкам относятся FirstMvp, StartMiniHints, WorkersInstructions, ReleaseCongratulations, UserMiniHints, UserIconsInstruction, EmptySprintInstruction, CreditPaidCongratulations, BugInstructions, TechDebtInstructions. Данные элементы либо являются экземплярами узла Hint, либо используют экземпляр MiniHint.

Hint и MiniHint очень похожи, но отличаются тем, как отображаются подсвечиваемые элементы. Рассмотрим узел Hint. Сам узел и его иерархия представлены на рисунках Рисунок 3 и Рисунок 4 соответственно.

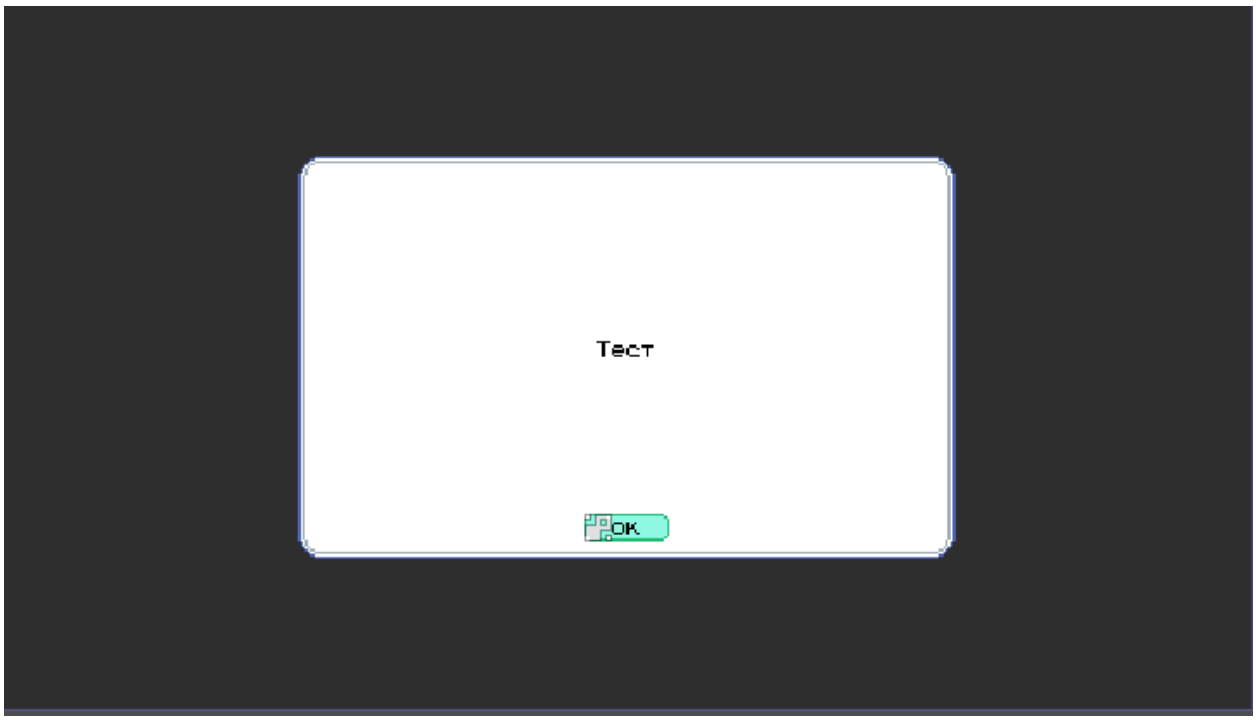


Рисунок 3 – Узел Hint

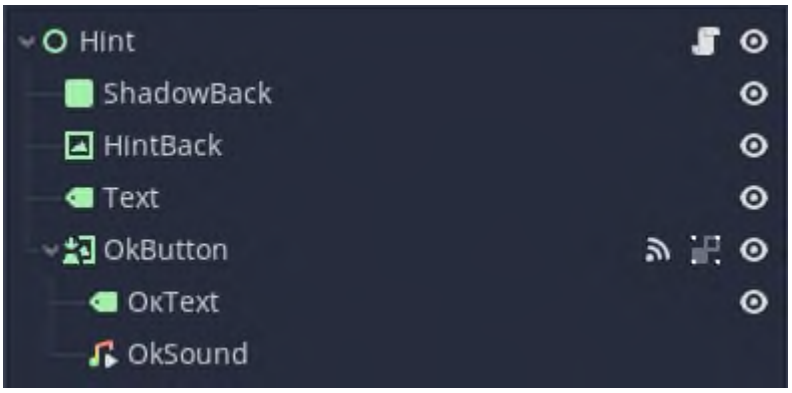


Рисунок 4 – Иерархия узла Hint

Узел ShadowBack играет роль затемнения, которое накладывается на весь экран, выделяя подсказку. HintBack это узел типа TextureRect и отображает текстуру заднего фона всплывающего окна. В узле Text типа Label отображается текст подсказки, который выставляется во время выполнения скрипта. Узел OkButton типа TextureButton является кнопкой подтверждения, в котором содержатся текст «ОК» и узел OkSound типа AudioStreamPlayer, который позволяет проигрывать звук при нажатии на кнопку «ОК».

Основной функционал данного узла заключен в скрипте. В начале скрипта объявлены переменные, которые позволяют настраивать подсказку через Инспектор:

– hide_button – позволяет указать нужно ли спрятать кнопку подтверждения при появлении подсказки на экране.

– text – массив строк, в который можно передать все строки, которые необходимо показать в рамках одного всплывающего окна.

– hl_nodes – массив идентификаторов объектов, которые необходимо подсветить, чтобы сфокусировать внимание пользователя.

Основой скрипта Hint является функция _on_OkButton_pressed, которая подключена к сигналу pressed элемента OkButton. Таким образом, при нажатии на кнопку вызывается _on_OkButton_pressed, которая в зависимости от того, остались ли ещё непоказанные строки, выводит на экран следующую подсказку или посылает сигнал «close_hint» и закрывается, если подсказок не осталось. Кроме того, при переключении текста, могут также переключаться подсвечиваемые элементы, если они были указаны в настройках узла.

Рассмотрим аналогичный узел MiniHint. Данный узел был создан, чтобы иметь возможность перемещать маленькое окно подсказки по экрану. Кроме того, в нём нет заранее добавленного затемнения. Это необходимо, чтобы ещё сильнее сфокусировать внимание игрока на важных игровых объектах. Пример перемещения подсказки представлен на рисунках Рисунок 5 и Рисунок 6.

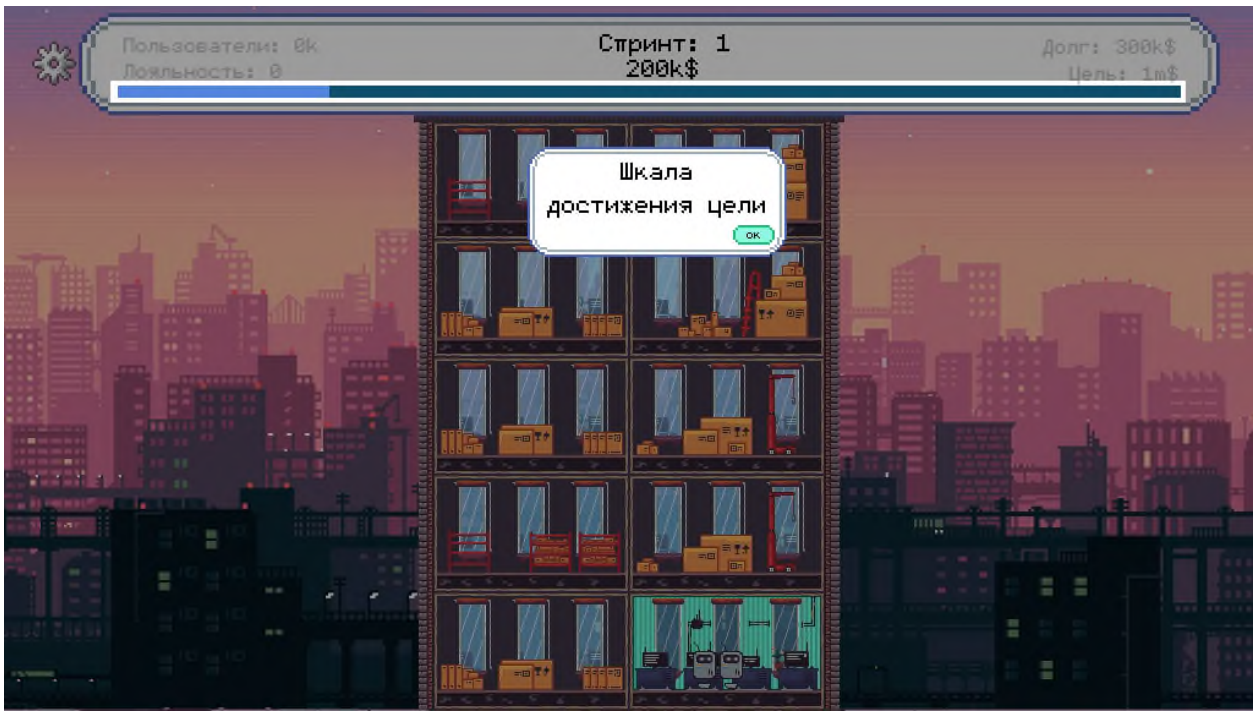


Рисунок 5 – Пример подсказки до перемещения

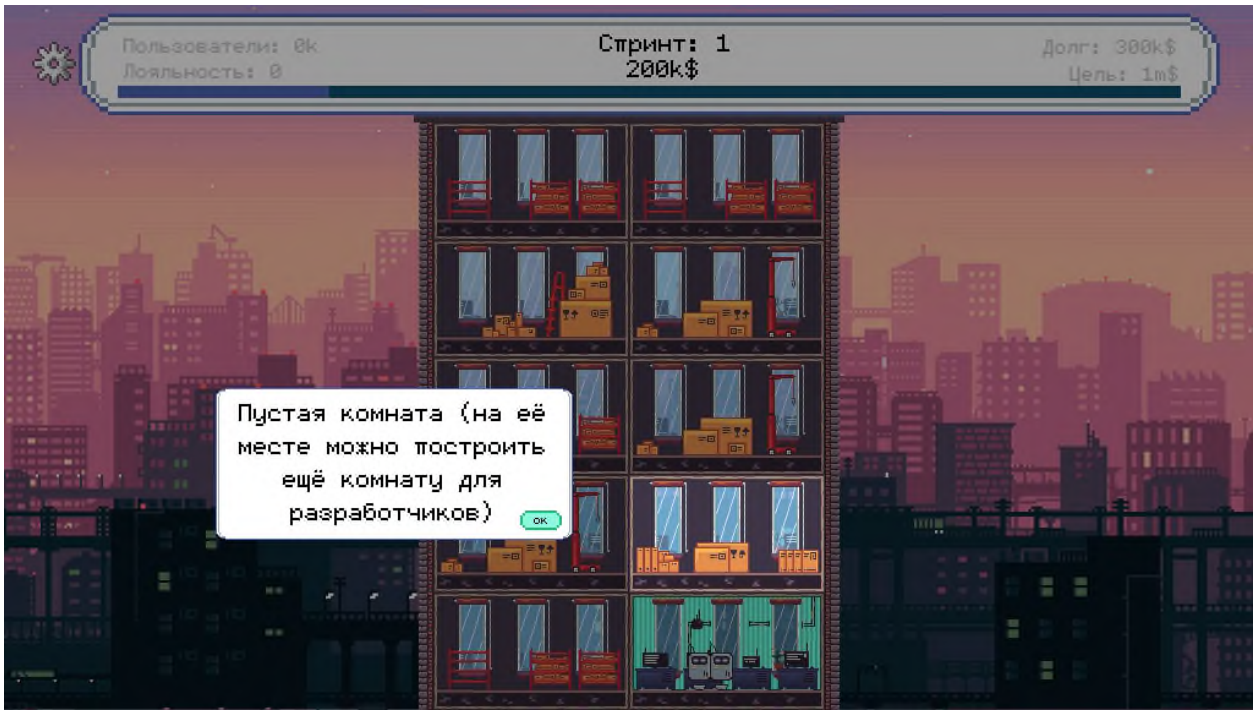


Рисунок 6 – Пример подсказки после перемещения

Рассмотрим реализацию узла MiniHint. Сцена в редакторе и иерархия узла представлены на рисунках Рисунок 7 и Рисунок 8 соответственно.


Рисунок 7 – Узел MiniHint

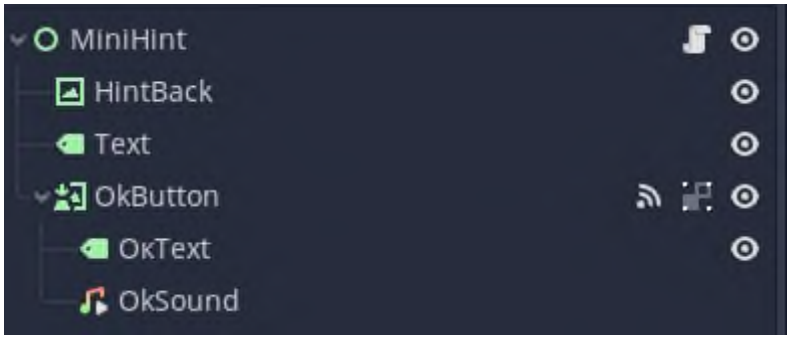


Рисунок 8 – Иерархия узла MiniHint

Назначение дочерних узлов MiniHint аналогично дочерним элементам узла Hint. Отличается только отсутствие какого-либо затемненного фона.

Одной из важных проблем, возникших во время разработки, являлась изменяющееся положение элементов на экране, которые нужно было подсветить. Проблема стала ещё более актуальна, при попытке запустить игру на мобильном устройстве, из-за того, что не только разрешение экрана может сильно отличаться, но и соотношение сторон может не совпадать с устройством, на котором разрабатывалась игра. Для того, чтобы решить эту проблему было решено добавить отдельный скрипт автозагрузки HighlightedNodes, задача которого хранить словарь идентификаторов объектов для подсветки и соответствующих им объектов Light2D. Таким образом, независимо от того, в каком узле находится объект, который нужно

подсветить, зная его идентификатор, мы можем получить этот объект и отобразить его на экране.

### 3.3 Основной игровой цикл

После обучения игра переходит в повторяющийся цикл, который состоит из следующих этапов:

– Выбор пользовательских историй для декомпозиции их на задачи.

– Выбор задач, которые роботы смогут выполнить в течение спринта.

– Ожидание окончания спринта.

– Окончание спринта и получение прибыли.

– Выпуск реализованных пользовательских историй.

Весь процесс управляется скриптом сцены Game, а состояние игры хранится в классе Global, который является скриптом автозагрузки и доступен всем классам во время выполнения.

#### 3.3.1 Скрипт автозагрузки Global

Начнём с описания скрипта Global, который используется почти всеми объектами игры для учёта состояния других объектов игры и своевременного реагирования на его изменения.

Скрипт Global является скриптом автозагрузки, из чего следует то, что он не привязан к какому-либо объекту или сцене. Он доступен постоянно всем объектам во время выполнения программы, из-за чего он должен быть легковесным, чтобы не потреблять слишком много ресурсов. Основным предназначением данного скрипта является хранение настроек игры в переменных, и активация сигналов для оповещения подписчиков об изменении тех или иных настроек.

Для начала рассмотрим сигналы, которые остальные объекты игры могут использовать для подписки на те или иные события:

– money_changed(money) – сигнал, активируемый при изменении количества доступных денег. В качестве параметра money передаётся текущее

значение. Например, на данный сигнал подписан объект HUD, который выводит важную для игрока информацию в интерфейсе.

– credit_changed(credit) – сигнал, активируемый при изменении остатка долга, который игроку необходимо выплатить. В качестве параметра credit передаётся текущее значение долга. На него также подписан объект HUD.

– customers_changed(customers) – сигнал, активируемый при изменении количества пользователей. В качестве параметра customers передаётся текущее количество пользователей. На него также подписан объект HUD.

– loyalty_changed(loyalty) – сигнал, активируемый при изменении лояльности. В качестве параметра loyalty передаётся текущая лояльность пользователей к продукту. На него также подписан объект HUD.

– sprint_changed(sprint) – сигнал, активируемый при изменении спринта. В качестве параметра sprint передаётся текущий спринт. На него также подписан объект HUD.

– rooms_changed(room) – сигнал, активируемый при покупке новой комнаты разработчиков. В качестве параметра room передаётся текущее количество комнат. На этот сигнал подписывается каждая не купленная комната для правильного отображения текущей стоимости.

– first_bug – сигнал, активируемый при появлении первого бага. Используется объектом HUD для своевременного отображения подсказок, объясняющих, что такое баги и на что они влияют.

– first_tech_debt – сигнал, активируемый при появлении первой задачи с техническим долгом. Используется объектом HUD для своевременного отображения подсказок, объясняющих, что это такое и что будет, если их не выполнять вовремя.

Кроме обычных переменных, в которых хранится состояние игры, в скрипте Global объявлены переменные, в которых хранятся настройки игры, такие как границы выбора случайных чисел, начальные значения денег, лояльности и пользователей и многое другое. К сожалению, пока что Godot не

позволяет добавлять экспортируемые переменные для скриптов автозагрузки, из-за чего все настройки приходится добавлять в код скрипта.

Рассмотрим некоторые из них:

– colors_for_use (Dictionary) – в данном словаре каждому из цветов перечисленных в UserCardColor сопоставлен объект CardColor, который хранит изображение определенного цвета для каждого типа задачи. Таким образом любой объект может по названию цвета и типу карточки получить нужное изображение. Например, чтобы получить фон для пользовательской истории синего цвета нужно выполнить следующий код: Global.colors_for_use[Global.UserCardColor.BLUE].long_color

– US_LTY (Dictionary) – в данном словаре в зависимости от размера пользовательской истории хранятся границы для выбора случайного числа, на которое умножается конечное количество получаемой лояльности за её выполнение.

– US_USR (Dictionary) – в данном словаре в зависимости от размера пользовательской истории хранятся границы для выбора случайного числа, на которое умножается конечное количество новых пользователей получаемых за её выполнение.

– US_FLOATING_PROFIT (Dictionary) – в данном словаре хранятся границы для выбора случайного числа, на которое умножается конечное количество новых пользователей и лояльности, получаемых за её выполнение в зависимости от того, сколько спринтов выполнялась эта история.

Остальные настройки, которые находятся в скрипте Global, будут рассмотрены по мере их использования.

### 3.3.2 Сцена Game

Рассмотрим сцену Game и скрипт Game.gd. Иерархия сцены и вид из редактора представлены на рисунках Рисунок 9 и Рисунок 10 соответственно.

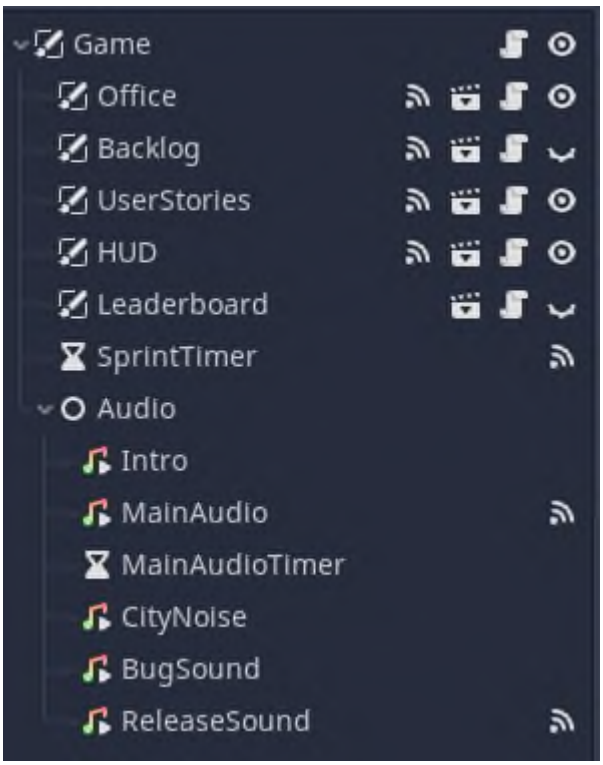


Рисунок 9– Иерархия сцены Game

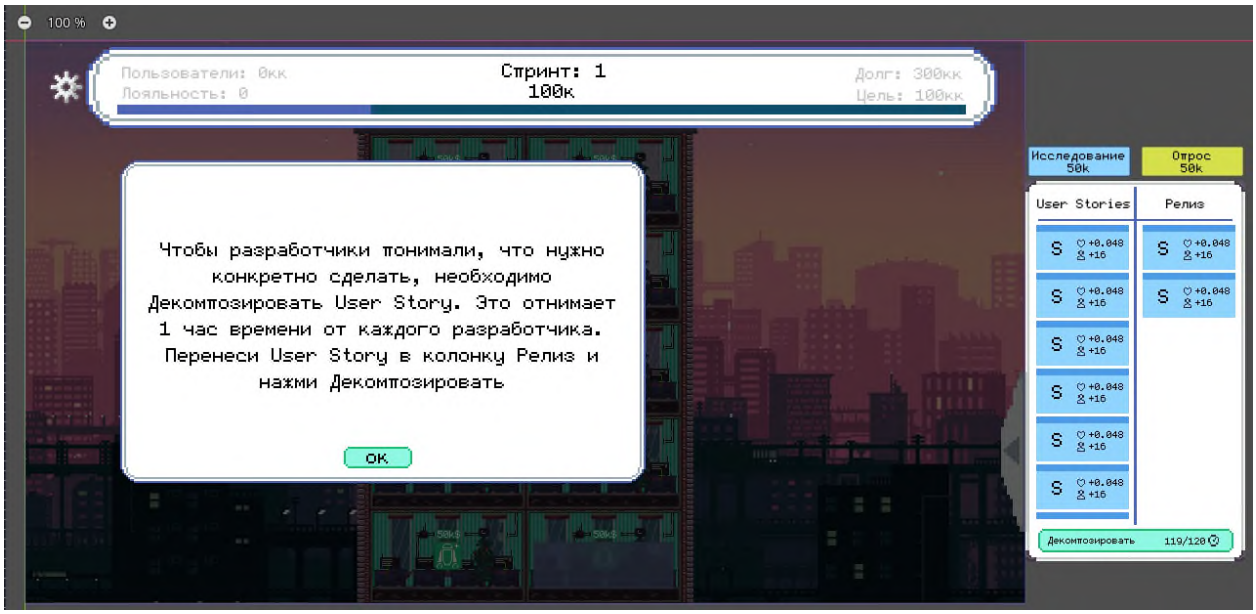


Рисунок 10 – Сцена Game

Сцена Game является основной сценой игры и в ней собраны экземпляры всех остальных игровых элементов:

– HUD – экземпляр одноименного узла, который отвечает за отображение большей части состояния игры.

– Office – экземпляр одноименного узла, который отвечает за отображение офиса, комнат и роботов.

– Leaderboard – экземпляр одноименного узла, который отвечает за отображение таблицы лидеров в конце игры.

– Backlog – экземпляр одноименного узла, который отвечает за отображение доступных задач и возможность их перемещать из доступных в текущий спринт.

– UserStories – экземпляр одноименного узла, который отвечает за отображение доступных пользовательских историй, их генерацию и возможность их декомпозировать.

– Audio – данный узел используется только для группировки объектов AudioStreamPlayer, которые проигрываются в определённые моменты игры.

Более подробное описание каждого узла будет представлено позже.

В сцене также видно на корневом элементе Game прикреплен скрипт Game.gd. Рассмотрим его поподробнее. Для того, чтобы игра не потребляла много ресурсов устройства, было решено как можно меньше использовать метод _process(), который выполняется при каждой отрисовке активной сцены. Вместо этого, большинство методов привязаны к определенным сигналам тех или иных объектов.

Рассмотрим некоторые методы, реализованные в скрипте Game.gd:

– _on_Backlog_start_sprint(cards_info) – данный метод привязан к сигналу «start_sprint» объекта Backlog. Данный метод вызывается, когда начинается спринт, и роботы должны начать имитировать работу, но только при условии, что в текущем спринте есть хоть одна задача или пользовательская история для декомпозиции. В противном случае роботы продолжают простаивать. Независимо от того, есть ли задачи в спринте или нет, включается таймер sprint_timer, а счётчик sprint_timer_count, показывающий сколько раз время таймера вышло, сбрасывается. Кроме того, на время спринта блокируются действия в окне выбора задач, в окне прогресса выполнения пользовательских историй и в окне бэклога.

– _on_HUD_begin_release() – этот метод привязан к сигналу «begin_release» объекта HUD, оповещающему, что обучающие подсказки вводящие игрока в курс дела закрыты. Данный метод открывает панель пользовательских историй, вызвав метод Userstories.show_panel, который будет рассмотрен ниже.

– _on_UserStories_start_release(cards_info, additional_cards) – этот метод связан с сигналом «start_release» объекта Userstories и вызывается, когда пользователь нажимает кнопку «Декомпозировать». В результате выполнения он выставляет флаг is_decomposited=true у всех объектов UserStoryCardInfo в списке Global.current_stories, обновляет индикаторы выполнения в HUD и вызывает метод show_backlog(), который открывает панель с задачами, полученными после декомпозиции.

– _on_SprintTimer_timeout() – этот метод связан с сигналом «timeout» объекта SprintTimer и вызывается, когда таймер заканчивает отсчёт. Данный метод вызывается несколько раз для того, чтобы во время спринта несколько раз изменить прогресс выполнения пользовательской истории. Спустя несколько выполнений метода _on_SprintTimer_timeout() вызывается метод next_sprint(), который проверяет, есть ли полностью выполненные пользовательские истории, обновляет показатели задач технического долга, подсчитывает прибыль за пройденный спринт, вычитает деньги затраченные на работу роботов и обновляет количество пользователей и их лояльность.

– _on_HUD_release_product() – этот метод связан с сигналом «release_product» объекта HUD. Когда вызывается этот метод, скрываются панели с задачами и пользовательскими историями, появляются новые баги и задачи с техническим долгом. Условия появления багов и задач технического долга будут рассмотрены позже. Также в этом методе вызывается update_profit() для получения пользователей и лояльности, который описан ниже.

– update_profit() – метод, вызываемый при релизе новых пользовательских историй. Для каждой пользовательской истории, находящейся в массиве Global.completed_us, подсчитывается получаемое количество пользователей и лояльности. И пользователи, и лояльность умножаются на случайный множитель, который зависит от того, как давно пользовательская история появилась в игре. Данный метод также убирает полосы загрузки выполненных историй и скрывает окно, в котором показывается прогресс текущих пользовательских историй.

– _on_Office_buy_robot() и _on_Office_buy_room() привязаны к сигналам «buy_robot» и «buy_room» соответственно и обновляют количество доступных часов, которые можно потратить на выполнение задач из пользовательских историй.

### 3.3.3 Сцена HUD

Объект HUD или же Head-up Display показывает большую часть важной для пользователя информации. Данный объект представляет собой интерфейс, состоящий из белого овала, на котором отображается прогресс игрока и выдвигающейся панели, похожей на блокнотный лист, в котором отображается прогресс каждой пользовательской истории.

Сцена HUD и её иерархия представлены на рисунках Рисунок 11 и Рисунок 12 соответственно.

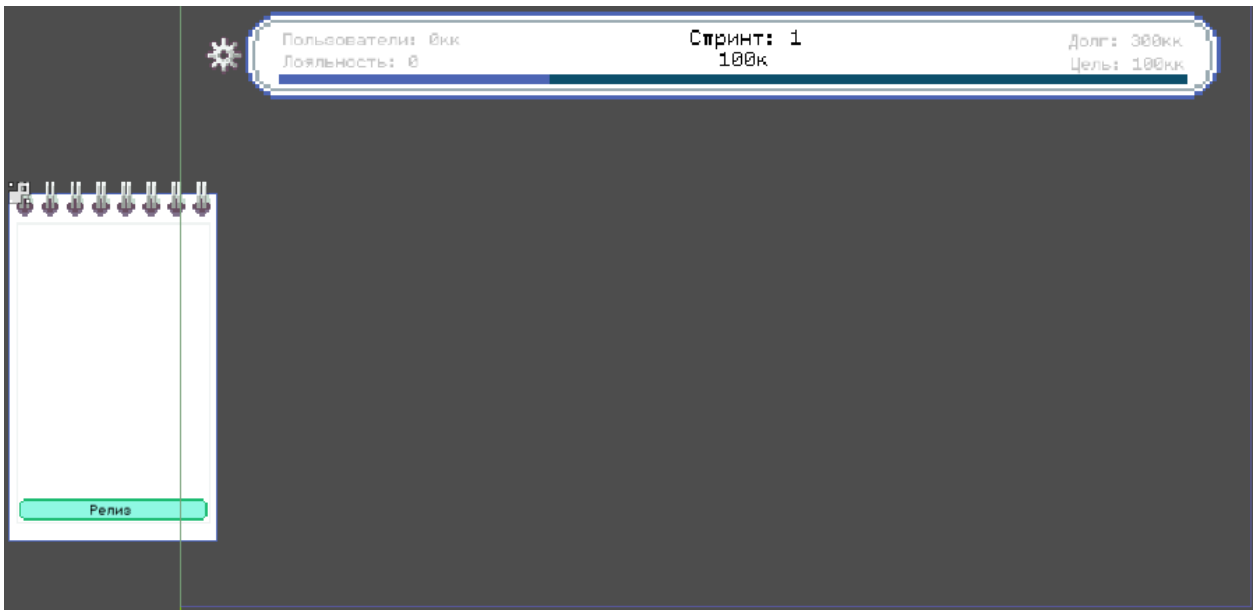


Рисунок 11 – Узел HUD

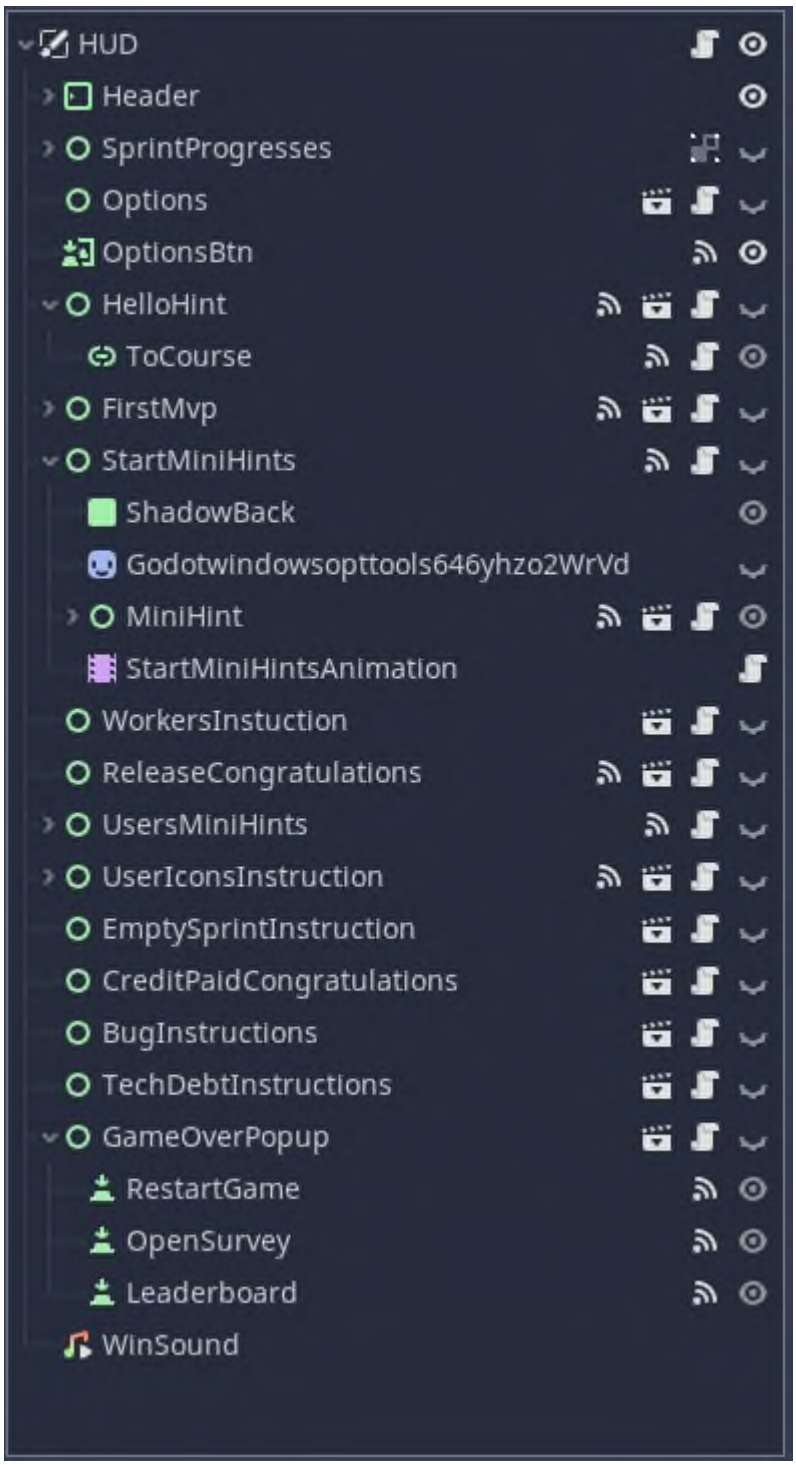


Рисунок 12 – Иерархия узла HUD

Кратко рассмотрим назначение каждого элемента:

– Header – элемент представляющий собой верхнюю часть интерфейса, в которой с помощью объектов Label выводится информация о текущем спринте, пользователях, лояльности, деньгах, долге и цели, к которой игроку нужно стремиться. Кроме того, у многих объектов также есть дочерний объект Light2D, который нужен для подсвечивания той или иной области во время

обучающих подсказок. Более подробная иерархия представлена на рисунке Рисунок 13.

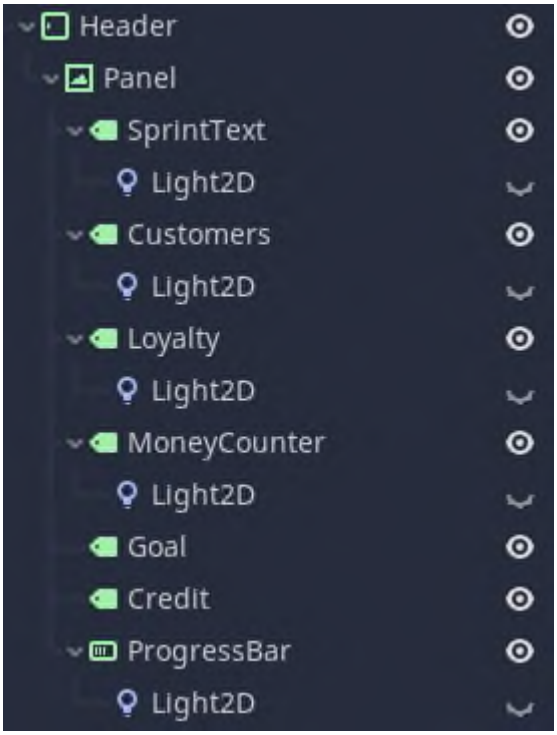


Рисунок 13 – Иерархия узла Header

– SprintProgresses – элемент представляющий собой выдвигающуюся панель, в которой отображаются индикаторы выполнения для каждой пользовательской истории, находящейся в разработке. В объект BarsContainer типа VBoxContainer, который представляет собой список элементов, располагаемых строго вертикально, помещаются объекты типа ProgressBar, которые представляют собой полосы загрузки с отображением параметров пользовательской истории. Более подробная иерархия узла SprintProgresses представлена на рисунке Рисунок 14.

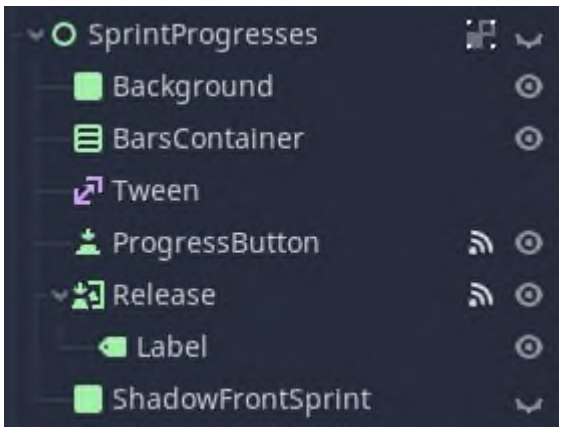


Рисунок 14 – Иерархия узла SprintProgresses

– Options и OptionsBtn – эти два элемента представляют собой окно с настройками, в котором можно выключить звуковые эффекты или фоновую музыку и кнопку, по нажатию на которую открывается это окно.

– HelloHint, FirstMvp, StartMiniHints, WorkersInstruction, ReleaseCongratulations, UserMiniHints, UserIconsInstruction, EmptySprintInstruction, CreditPaidCongratulations, BugInstructions и TechDebtInstructions являются экземплярами объектов Hint или MiniHint, рассмотренных ранее, которые нужны для того, чтобы вывести на экран поясняющие подсказки.

– GameOverPopup также является экземпляром объекта Hint, но выводится в конце игры и предлагает начать игру заново, оставить свой отзыв или посмотреть таблицу рекордов.

К корневому узлу сцены HUD прикреплен скрипт hud.gd, который управляет обновлением значений в интерфейсе и активирует некоторые сигналы, на которые может быть подписан его родительский объект.

Рассмотрим сигналы, объявленные в классе HUD:

– begin_release – этот сигнал активируется, когда закрываются первоначальные подсказки StartMiniHints.

– release_product – этот сигнал активируется, когда игрок нажимает на кнопку «Релиз».

– open_leaderboard – этот сигнал активируется, когда игрок нажимает на кнопку «Рекорды», которая появляется в конце игры. В результате его активации вызывается метод Game. _on_HUD_open_leaderboard, который открывает таблицу рекордов.

– change_intro_audio – этот сигнал активируется, когда начальные подсказки закрыты и нужно сменить фоновую музыку.

Рассмотрим некоторые методы, объявленные в классе HUD:

– _ready() – метод, который необходим для начальной настройки узла. Для начала он подписывает методы _on_money_changed, _on_credit_changed, _on_sprint_changed, _on_customers_changed, _on_loyalty_changed, _on_first_bug, _on_first_tech_debt на соответствующие сигналы скрипта Global, для отображения актуальной информации. Также он регистрирует все объекты типа Light2D в классе HighlightedNodes для того, чтобы их можно было подсвечивать в процессе обучения.

– update_current_bars(card_infos) – этот метод вызывается каждый новый спринт, убирает дочерние объекты из узла SprintProgresses/BarsContainer и добавляет в этот узел новые экземпляры класса USProgressBar для каждой пользовательской истории в массиве card_infos.

– remove_bar(user_story) – этот метод вызывается, когда пользовательская история вышла в релиз, и убирает соответствующую ей полоску загрузки из узла SprintProgresses/BarsContainer.

– show_progresses() и hide_progresses() имеют противоположный эффект. show_progresses() запускает процесс «выдвижения» той части интерфейса, где находятся все индикаторы выполнения. Для этого используется встроенный класс Tween, позволяющий плавно изменять с течением времени те или иные значения. В данном случае с помощью класса Tween изменяется значение rect_position с текущего на show_position или initial_position в зависимости от вызванного метода.

– _on_money_changed(money), _on_sprint_changed(sprint), _on_credit_changed(credit), _on_customers_changed(customers) и _on_loyalty_changed(loyalty) изменяют значения на актуальные в соответствующих объектах Label

– _on_first_bug() и _on_first_tech_debt() открывают окна с подсказками при появлении первого бага и задачи технического долга соответственно.

– increase_progress(part_of_sprint, cards_to_update) – данный метод выполняется при истечении времени таймера SprintTimer в сцене Game. При вызове этого метода, в каждой из пользовательской истории, задачи которых переданы в cards_to_update, обновляется выполненная часть на значение part_of_sprint. Кроме того, если есть хоть одна история, которая полностью выполнена, включается кнопка «Релиз».

– change_overlay(overlay_name) – этот метод используется для смены активного экрана, отображающегося поверх остального интерфейса, скрывая все остальные окна. Таковыми являются окна с подсказками, окно с настройками звука и окно, появляющееся в конце игры.

### 3.3.4 Сцена Office

Для имитации выполнения работы по реализации выбранных игроком пользовательских историй в середине экрана представлен офис, в котором передвигаются роботы. Данный объект является экземпляром сцены Office. Сама сцена и её иерархия представлены на рисунках Рисунок 15 и Рисунок 16.

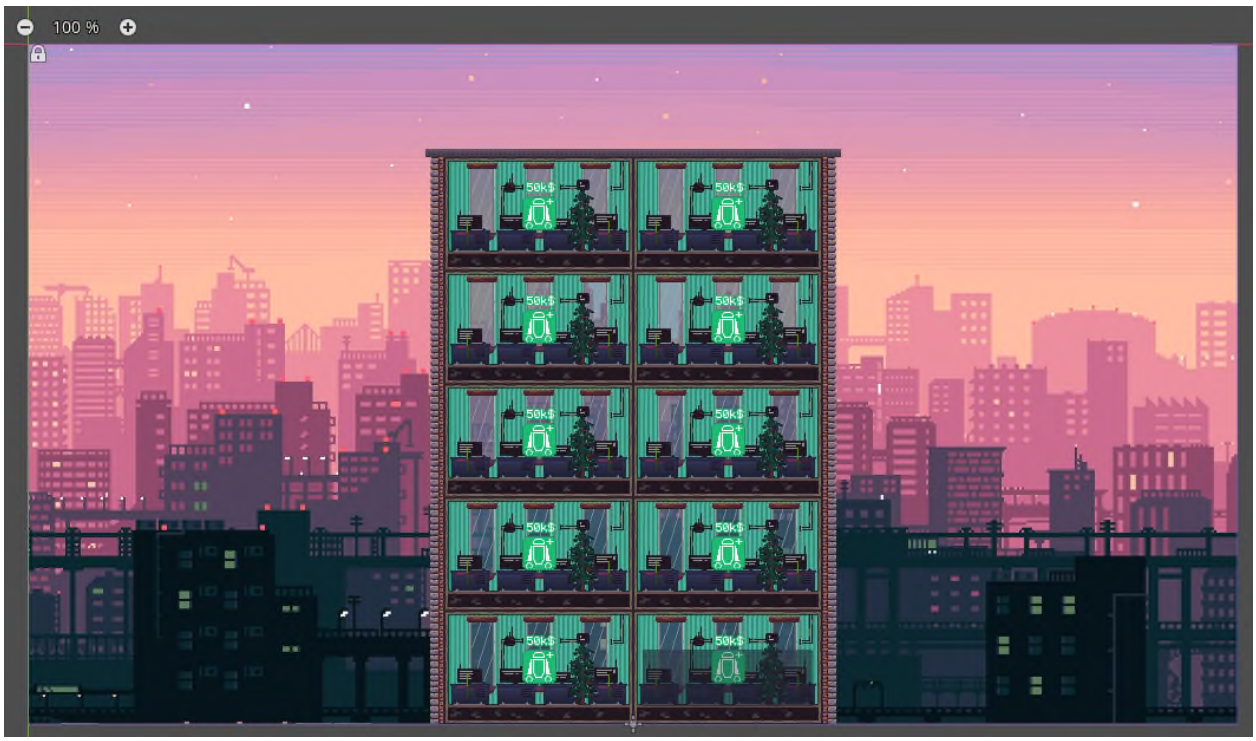


Рисунок 15 – Сцена Office

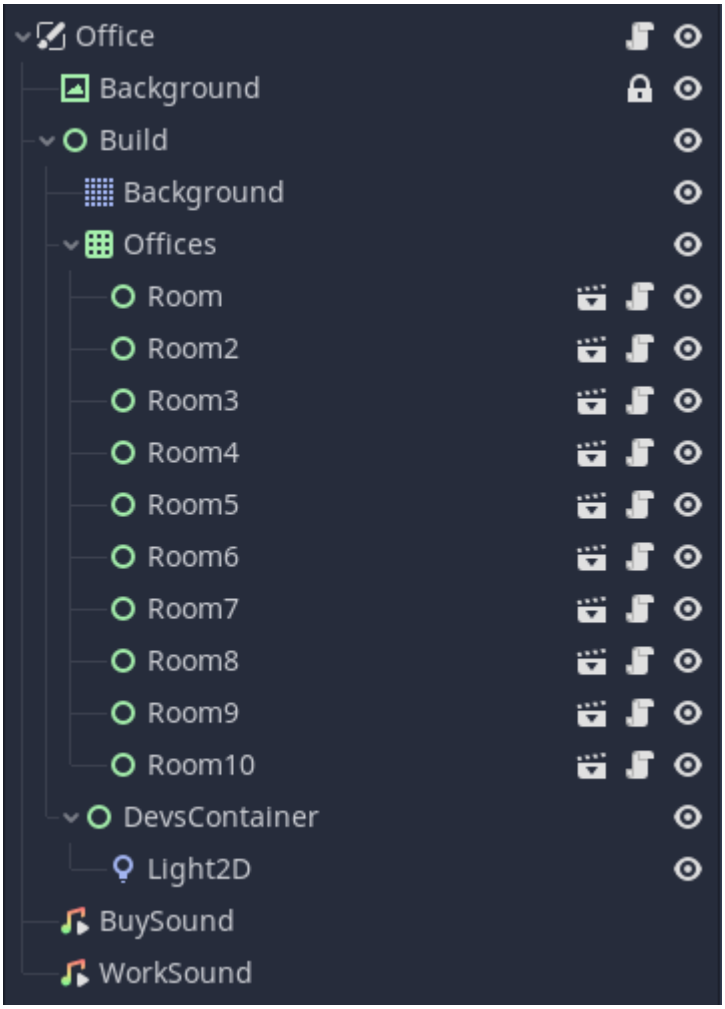


Рисунок 16 – Иерархия сцены Office

Данная сцена используется в основном только для группировки экземпляров сцены Room. Кроме того, в классе Office объявлены два сигнала buy_robot и buy_room, которые нужны для передачи соответствующих сигналов от экземпляров Room на уровень выше. Кроме того, здесь объявлены методы toggle_workers(are_working) и toggle_purchases(mode), которые вызывают соответствующие методы у всех экземпляров класса Room.

### 3.3.5 Сцена Room

Каждая комната, в которой находятся роботы, является экземпляром сцены Room. Вид сцены в редакторе и её иерархия представлены на рисунках Рисунок 17 и Рисунок 18.


Рисунок 17 – Сцена Room

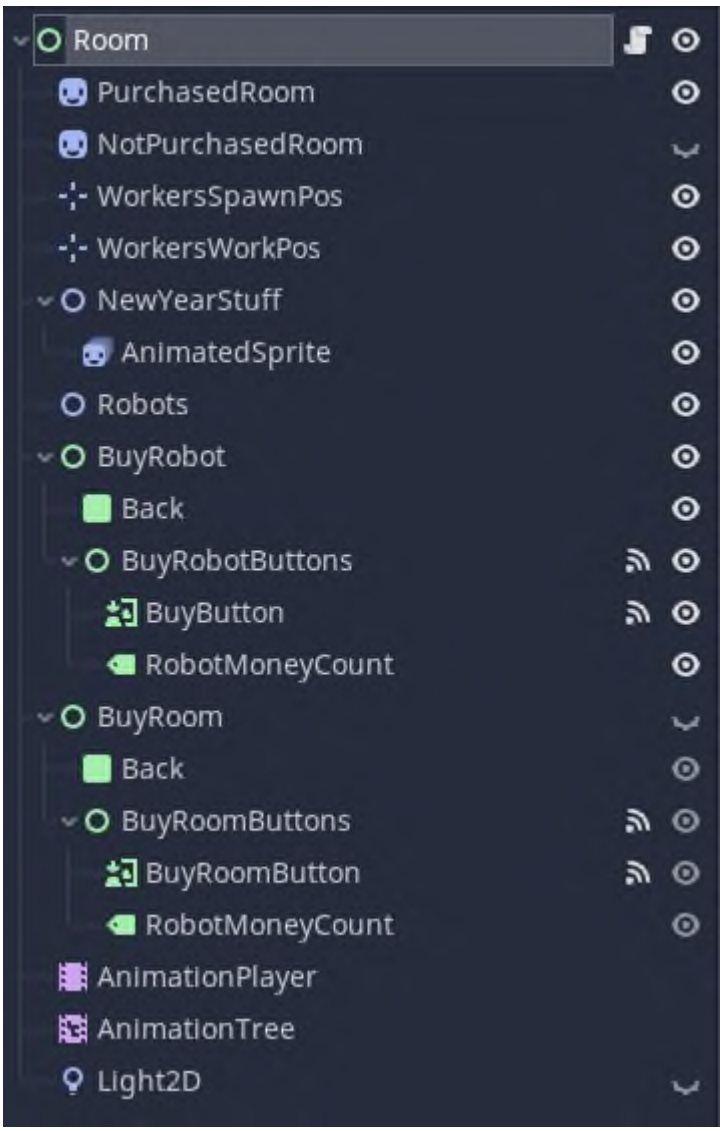


Рисунок 18 – Иерархия сцены Room

Рассмотрим назначение каждого элемента в иерархии:

– PurchasedRoom и NotPurchasedRoom представляют собой экземпляры класса Sprite и отображают фон купленной комнаты или не купленной соответственно.

– WorkerSpawnPos и WorkerWorkPos являются экземплярами класса Position2D. Их задача – задать позицию точки появления новых роботов и точки, в которую роботы приходят во время имитации работы.

– NewYearStuff является дополнительным спрайтом на новогоднюю тематику, отображающуюся с 20 декабря по 15 января.

– Robots – это экземпляр Node2D, который является контейнером, в который добавляются новые экземпляры класса Worker.

– BuyRobot и BuyRoom – два узла Control, в которых сгруппированы объекты участвующие в отображении цены и покупке новых комнат и роботов.

В качестве настройки класс Room имеет две переменные – worker_prefab – путь к сцене, которая представляет собой робота, и worker_count – количество роботов, которых необходимо создать в начале игры.

Рассмотрим некоторые методы, реализованные в классе Room:

– _ready() – в данном методе производится начальная настройка узла и его дочерних элементов: узел подписывает метод _on_rooms_changed на соответствующий сигнал скрипта Global. В случае, если в worker_count не указано количество роботов, которых нужно создать, комната считается не купленной, поэтому необходимо активировать спрайт NotPurchasedRoom, выключить узел BuyRobot, чтобы отключить возможность покупать роботов в этой комнате, и включить узел BuyRoom, дав возможность купить эту комнату. В обратном случае выключается узел BuyRoom и включается BuyRobot. Кроме того, в точке WorkerSpawnPos создаётся столько экземпляров сцены, указанной в worker_prefab, сколько роботов указано в worker_count.

– _on_BuyButton_pressed() – этот метод привязан к сигналу «pressed» элемента BuyButton, проверяет хватает ли у игрока денег, чтобы купить нового робота, создаёт новый экземпляр сцены, указанной в worker_prefab, в точке WorkerSpawnPos и активирует сигнал «buy_robot». В случае, если достигнуто максимальное количество роботов в комнате, указанное в Global.MAX_WORKER_COUNT, элемент BuyRobot выключается, тем самым отключая возможность докупать новых роботов в этой комнате.

– _on_BuyRoomButton_pressed() – этот метод привязан к сигналу «pressed» элемента BuyRoomButton, проверяет хватает ли у игрока денег, чтобы купить новую комнату, меняет фон с NotPurchasedRoom на PurchasedRoom, создаёт новый экземпляр сцены, указанной в worker_prefab, в

точке WorkerSpawnPos и активирует сигнал «buy_room». Также выключается узел BuyRoom, убирая возможность повторно купить эту комнату.

### 3.3.6 Сцена Leaderboard

Для того, чтобы стимулировать игрока проходить игру заново, была добавлена таблица рекордов, в которой игрок может увидеть лучшие результаты других игроков. Эту таблицу можно открыть, когда игра заканчивается. Таблица рекордов реализована в сцене Leaderboard, которая представлена на рисунке Рисунок 19, а её иерархия – на рисунке Рисунок 20.

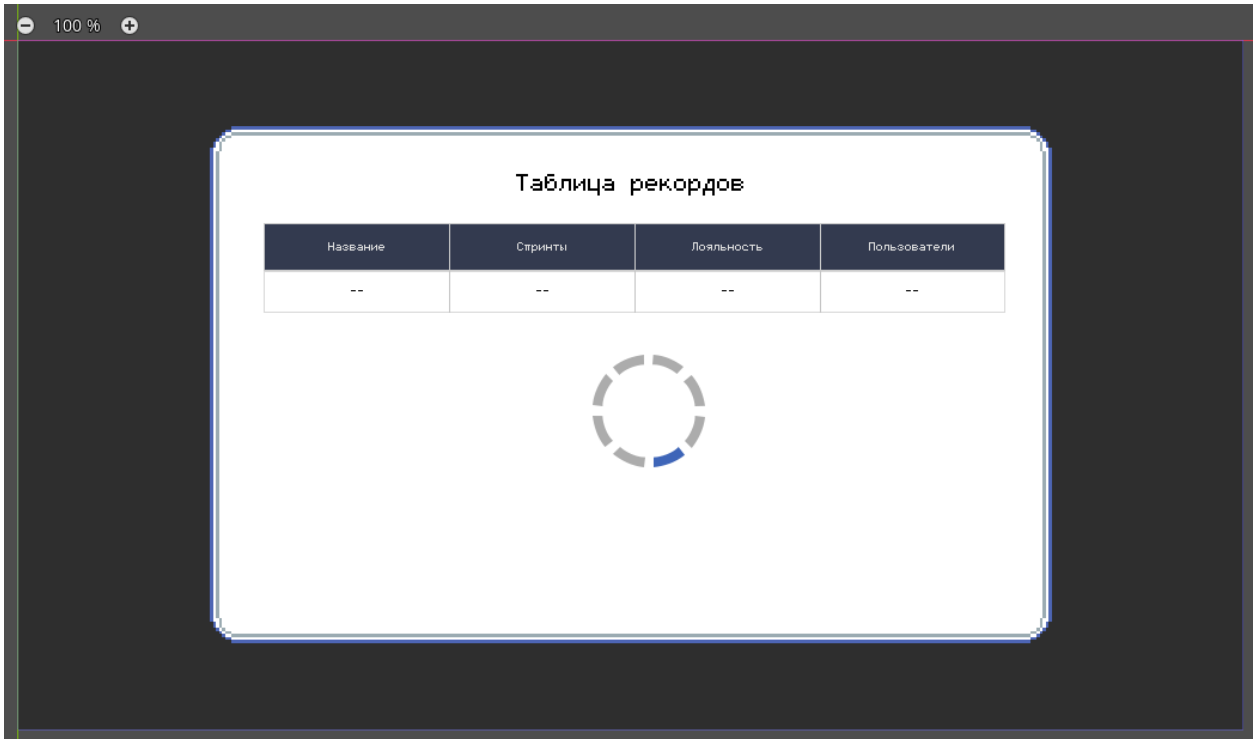


Рисунок 19 – Сцена Leaderboard

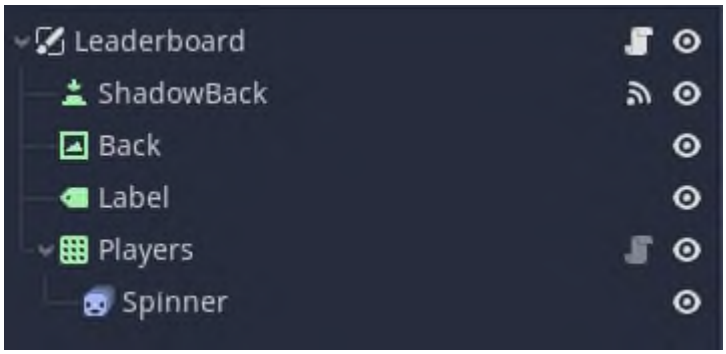


Рисунок 20 – Иерархия сцены Leaderboard

В основе данной сцены лежит узел Players, который является экземпляром импортированного класса Table, представляющего удобную для

настройки таблицу. Узел ShadowBack представляет затемненный фон, по нажатию на который таблица скрывается. Узел Back – это фоновое изображение таблицы, а Label – надпись сверху таблицы. Кроме того, у узла Table есть дочерний узел – экземпляр класса AnimatedSprite, который представляет собой анимированный загрузчик, показываемый в тот момент, когда подгружаются данные о рекордах.

Для хранения данных о рекордах игроков, в игре используется сторонний плагин SilentWolf. Данный плагин позволяет, не требуя от игроков никакой регистрации, сохранять их достижения. Перед тем, как начать работу с новой таблицей рекордов её необходимо создать на сайте проекта. После этого, при запуске игры необходимо вызвать метод SilentWolf.configure(), и передать в него ключ, сгенерированный для этого проекта [6].

В данном проекте конфигурация SilentWolf происходит в скрипте Global в методе _ready(), что позволяет быть уверенным, что к моменту открытия таблицы рекордов, плагин будет настроен. Для открытия таблицы рекордов, вызывается метод Leaderboard.show_leaderboard(), в котором очищается таблица Players и показывается загрузчик. После этого вызывается метод yield(SilentWolf.Scores.get_high_scores(20), "sw_scores_received"), который получает 20 лучших записей. После того, как данные получены, для каждой из полученных записей добавляется строка в таблицу Players.

### 3.3.7 Сцена UserStories

Каждый игровой цикл начинается с генерации новых пользовательских историй. Данный процесс полностью происходит в рамках узла UserStories. Рассмотрим его реализацию. На рисунках Рисунок 21 и Рисунок 22 представлены сам узел и его иерархия соответственно.

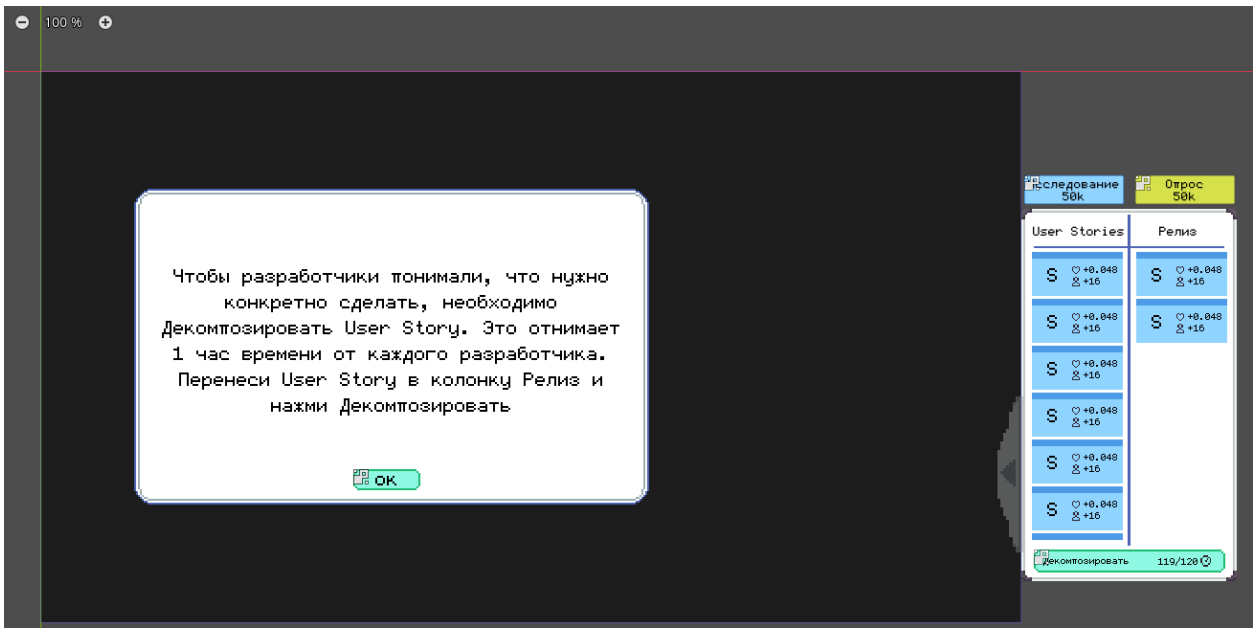


Рисунок 21 – Узел UserStories

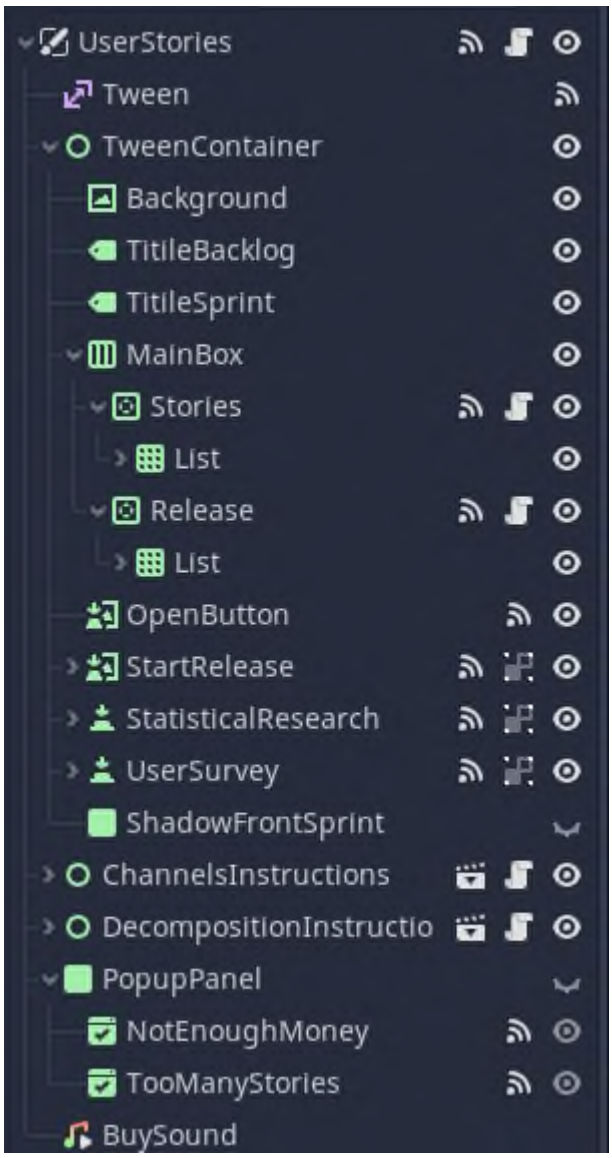


Рисунок 22 – Иерархия узла UserStories

Генерация обычных пользовательских историй происходит при проведении исследования рынка или опроса пользователя. Оба события настроены на соответствующие кнопки StatisticalResearch и UserSurvey, расположенные сверху выпадающей вкладки.

Для того, чтобы узел UserStories мог реагировать на события нажатия кнопок к их сигналам «button_pressed» привязаны функции _on_StatisticalResearch_pressed и _on_UserSurvey_pressed.

Методы _on_StatisticalResearch_pressed и _on_UserSurvey_pressed вызывают один и тот же метод generate_cards, но с разными параметрами:

– UserStoriesGenerator – генератор, определяющий вероятности появления пользовательских историй разного размера.

– Cost – сумма, которую должен заплатить игрок для получения новых пользовательских историй. Как и большинство настроек, необходимая сумма берётся из скрипта Global.

– Count – количество историй, которые должны появиться.

Далее объект UserStoriesGenerator передаётся в метод, который получает новые пользовательские истории из UserStoriesGenerator и добавляет их в соответствующий список на панели пользовательских историй.

После создания новых пользовательских историй в узле UserStories/TweenContainer/MainBox/Stories/List появляются элементы UserStoryCard. Для того, чтобы карточки можно было перемещать, в момент создания новой пользовательской истории узел Stories подписывается на сигнал «move_card» узла UserStoryCard посредством метода connect(). Сигнал «move_card» активируется при выполнении метода UserStoryCard._on_UserStoryCard_pressed только при условии, что панель не находится в процессе перемещения. Таким образом при нажатии на карточку, она удаляется из текущего узла UserStories/TweenContainer/MainBox/Stories/List и добавляется в UserStories/TweenContainer/MainBox/Release /List.

При перемещении карточек между списком доступных и тех, что пойдут в следующий релиз необходимо также отобразить количество часов, которое роботы потратят на декомпозицию пользовательских историй в задачи. Этот процесс тесно связан с перемещением карточек между списками. Как было сказано выше при перемещении карточки активируется сигнал «move_card», после чего карточка удаляется из старого узла и добавляется в новый. В момент добавления активируется сигнал «card_dropped», после чего вызывается либо метод UserStories._on_Stories_card_dropped, либо UserStories. _on_Release_card_dropped в зависимости от того, к какому списку была

добавлена пользовательская история. При выполнении метода UserStories._on_Release_card_dropped перемещенная история удаляется из списка доступных историй Global.available_stories, и добавляется в список текущих Global.current_stories. Кроме того, в этом же методе пересчитывается количество доступных часов для разработки и активируется/деактивируется кнопка декомпозиции в зависимости от того, есть ли истории в списке UserStories/TweenContainer/MainBox/Release/List. Метод UserStories._on_Stories_card_dropped, выполняет аналогичные действия, но удаляет историю из Global.current_stories и добавляет её в Global.available_stories.

После получения новых пользовательских историй, пользователю необходимо декомпозировать их на отдельные задачи, нажав кнопку «Декомпозировать».

Для того, чтобы игра могла реагировать на нажатие данной кнопки к сигналу «button_pressed» привязан метод UserStories._on_StartRelease_pressed. Вызов данного метода активирует сигнал «start_release» который также передаёт всем подписчикам список пользовательских историй, которые необходимо декомпозировать на задачи.

### 3.3.8 Сцена Backlog

После генерации новых пользовательских историй и выбора из них тех, которые будут реализованы в текущем спринте, открывается панель бэклога. Внешне и функционально данная панель похожа на панель с пользовательскими историями, но в отличие от неё в панели бэклога игрок управляет выбирает задачи, которые будут выполнены во время спринта. Кроме того, выбирая задачи игрок ориентируется уже не на показатели пользователей и лояльности, а на то, сколько часов займёт та или иная задача у роботов, чтобы максимально эффективно использовать время спринта. Рассмотрим реализацию данной сцены. Вид сцены в редакторе и её иерархия представлены на рисунках Рисунок 23 и Рисунок 24.

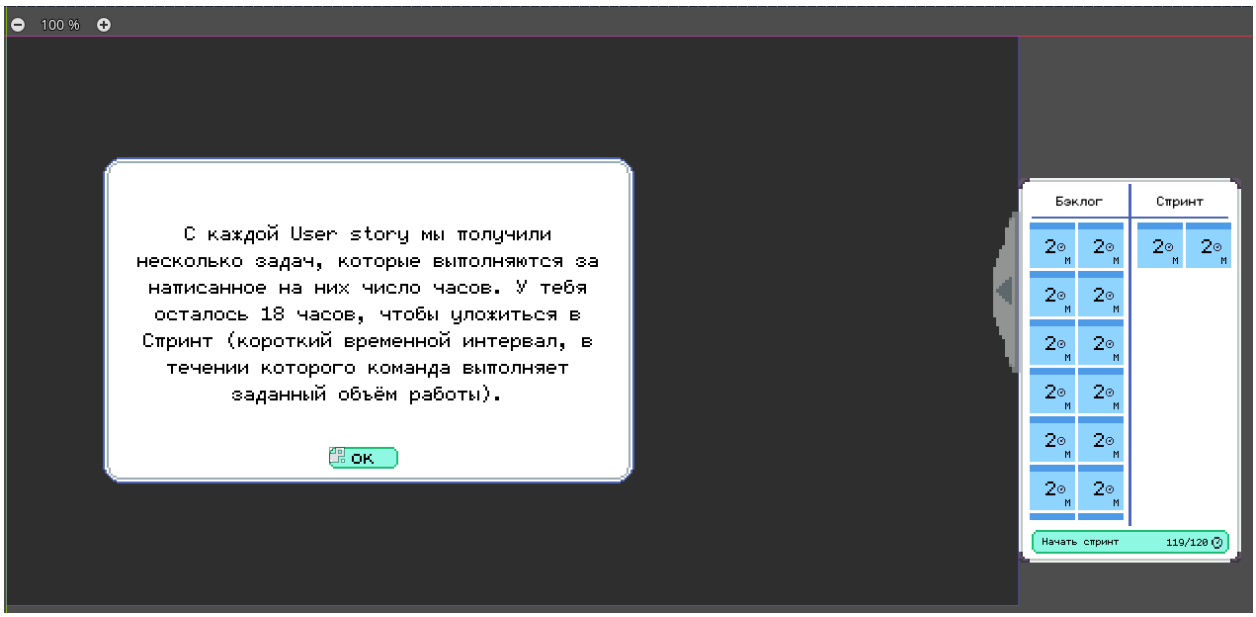


Рисунок 23 – Сцена Backlog

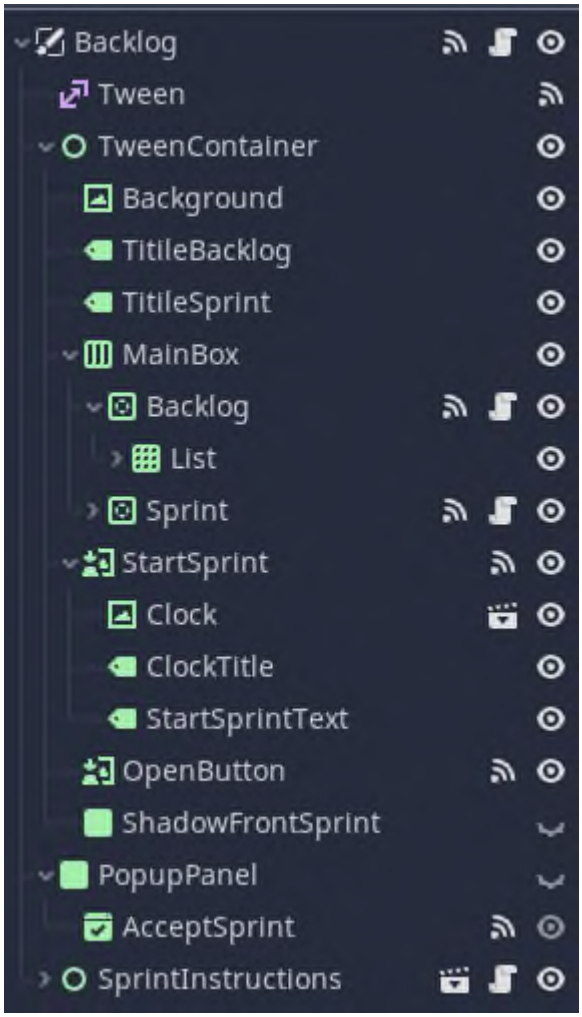


Рисунок 24 – Иерархия сцены Backlog

Основным узлом данной сцены является TweenContainer, в котором хранятся все остальные элементы. Внутри данного узла расположены:

– Background – фоновое изображение панели,

– TitleBacklog и TitleSprint – подписи сверху каждого списка с задачами,

– MainBox – экземпляр класса HBoxContainer, в котором хранятся оба списка с задачами,

– Backlog и Sprint – экземпляры класса ScrollContainer, между которыми перемещаются экземпляры класса BacklogCard, представляющие собой задачи,

– StartSprint – экземпляр класса TextureButton, по нажатию на который начинается текущий спринт,

– OpenButton – экземпляр класса TextureButton, по нажатию на который панель перемещается либо в пределах видимости, либо за границы экрана,

– SprintInstructions – узел, использующий экземпляр класса Hint, для отображения обучающих подсказок, касающихся спринтов.

Для начала рассмотрим сигналы класса Backlog, на которые можно подписаться из других классов:

– start_sprint(cards_info) – этот сигнал, активируется по нажатию на кнопку «Начать спринт», при условии, что игрок не превысил лимит по часам,

– open_backlog – этот сигнал активируется при нажатии на кнопку OpenButton для того, чтобы закрыть панель с пользовательскими историями.

Рассмотрим некоторые методы, реализованные в классе Backlog:

– _on_OpenButton_pressed() – данный метод вызывается при активации сигнала «pressed» кнопки OpenButton и в зависимости от того открыта панель или закрыта вызывается методы hide_panel() или show_panel() соответственно.

– show_panel() – при вызове этого метода удаляются все дочерние узлы элемента Backlog/List, пересчитываются доступные часы, отображаемые на кнопке StartSprint, а для каждой пользовательской истории, хранящейся в Global.current_stories, создаются экземпляры BacklogCard под каждую задачу. Также с помощью класса Tween рассмотренного ранее изменяется значение

rect_position узла TweenContainer от изначального до значения указанного в show_position.

– hide_panel() – при вызове этого метода с помощью помощью класса Tween изменяется значение rect_position узла TweenContainer от значения show_position до значения указанного в initial_position.

– _on_StartSprint_pressed() – данный метод вызывается при активации сигнала «pressed» узла StartSprint, проверяет хватает ли доступных часов на те задачи, которые игрок перенес в спринт, и в случае правильного выбора задач активирует сигнал «start_sprint», передавая задачи, находящиеся в узле MainBox/Sprint/List.

### 3.4 Генерация багов

Когда игрок выплачивает весь долг по кредиту, в игре появляется новый фактор, который стоит учитывать при планировании спринта – Баги. Эти задачи появляются с определенной вероятностью при выпуске в релиз той или иной пользовательской истории. Рассмотрим конкретный момент и условия появления новой задачи типа «Баг».

«Баг» может появиться только при выпуске в релиз новой пользовательской истории, то есть в момент выполнения метода Game._on_HUD_release_product(). В этом методе вызывается метод Game. check_and_spawn_bug(), который проверяет следующие условия:

– В массиве Global.current_bugs должно быть меньше 7 объектов,

– Кредит должен быть полностью выплачен, то есть значение Global.credit равно нулю,

– Кроме предыдущих двух условий должно быть выполнено одно из следующих – генератор случайных чисел RandomNumberGenerator выдал число меньше вероятности заданной в Global.BUG_SPAWN_PROBABILITY, либо на данный момент есть хотя бы одна задача технического долга.

Существующие «Баги» плохо сказываются как на лояльности продукта, так и на количестве активных пользователей. Для учёта этих штрафов каждый

спринт в методе Game.update_loyalty() вычитаются количество пользователей указанное в customers_debuff и лояльность указанная в loyalty_debuff. Кроме того, после вычета каждый из этих штрафов увеличивается на customers_increment и loyalty_increment соответственно.

Исходя из этого, чем дольше игрок держит в бэклоге задачи типа «Баг», тем сильнее упадёт лояльность продукта и тем больше пользователей от него отвернутся.

### 3.5 Генерация задач технического долга

При создании продукта иногда возникает возможность использовать более быстрое и простое решение, чем более надёжное. К сожалению, часто такое решение не выдерживает проверку временем и разработчикам приходится тратить время на замену этого решения более оптимальным и поддерживаемым в долгосрочной перспективе. Такие затраты времени и сил также называют техническим долгом. Игроку также приходится сталкиваться с такой проблемой, когда он полностью выплатил долг и уже исправил хотя бы один баг.

Рассмотрим условия появления этих задач. Как и «Баг» задача технического долга появляется в момент выпуска в релиз новой пользовательской истории при выполнении метода Game. check_and_spawn_tech_debt(). Для появления задачи технического долга должны быть выполнены все следующие условия: в массиве Global.current_tech_debt хранится меньше 7 объектов, игрок полностью выплатил долг, а генератор случайных чисел выдал число меньше заданной вероятности Global.TECH_DEBT_SPAWN_PROBABILITY.

Кроме того, невыполнение задач технического долга влечёт за собой тяжелые последствия, а именно задачи требуют всё больше и больше времени на выполнение. Подсчёт влияния текущего технического долга происходит при каждом выполнении новой задачи в методе Game.update_tech_debt_impact(). В этом методе подсчитывается количество выполненных задач, которые не относятся к техническому долгу, и

полученный штраф прибавляется к времени, которое необходимо потратить на задачи для реализации других пользовательских историй. Как и говорилось ранее, если в момент релиза пользовательской истории существует хотя бы одна задача технического долга, то в продукте обязательно появится «Баг».

### 3.6 Развертывание игры

На эффективность и скорость разработки любого программного проекта большое влияние оказывают технологии DevOps. Внедрение практик непрерывной интеграции и непрерывной доставки упрощают тестирование и развертывание новых релизов игры.

В качестве CI/CD системы используется Github Actions. Она позволяет относительно просто организовать автоматизацию рабочего процесса используя удаленные мощности, что для нашего проекта является весомым достоинством так как нет необходимости разворачивать собственные сервера для работы автоматизации.

Сборка игры производилась на базе официального образа Godot Engine с встроенными плагинами интеграций с популярными операционными системами.

# ЗАКЛЮЧЕНИЕ

Создание игры-симулятора для новой профессии может помочь в популяризации этой профессии и привлечении новых кандидатов. Человек, играя в такую игру, может получить представление о том, чем занимается владелец продукта, какие задачи ему ставятся и какие риски он принимает.

Создавать игру с помощью современных игровых фреймворков таких как Godot, позволило сэкономить большую часть времени, так как визуальный редактор сцен позволил настраивать расположение игровых элементов в реальном времени, а GDScript позволил не только быстро создать скрипты, контролирующие поведение игровых объектов, но и сократить время на разработку этих скриптов за счёт большого количества встроенных методов. Также нативная реализация в GDScript таких шаблонов проектирования как «Одиночка» и «Наблюдатель», позволила разработать менее связный и более поддерживаемый код.

Кроме того, система CI/CD основанная на GitHub Actions, позволила без лишних усилий со стороны разработчика, выпускать частые обновления игры. Также не стоит забывать, что разработанная игра, благодаря кроссплатформенности Godot, может быть выпущена на другие платформы в кратчайшие сроки.

Целью выпускной квалификационной работы было разработать игру «Симулятор Владельца Продукта».

Для достижения этой цели были выполнены следующие задачи:

– Рассмотрены возможности различных популярных игровых фреймворков и выбран подходящий для разработки игры-симулятора;

– Разработана логику игры, включающую в себя процесс выбора пользовательских историй в спринт и выбор задач с учетом доступных игроку ресурсов;

– Добавлены новые механики с появлением задач типа «Баг» и «Технический долг» для улучшения и разнообразия игрового процесса;

– Настроен процесс CI/CD для быстрого развертывания игры на платформе itch.io.

Результатом командной работы является продукт «Симулятор Владельца Продукта» — это компьютерная игра, доступная в браузере по адресу https://npg-team.itch.io/product-owner-simulator.

Сыграв в игру, игрок получит базовое представление об обязанностях владельца продукта, основах методологии SCRUM и сможет развить навыки стратегического планирования.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК


1. Сейдаметов Э. Э. Современные средства разработки игр / Э. Э. Сейдаметов, А. Э. Шабанов // Информационно-компьютерные технологии в экономике, образовании и социальной сфере. – 2020. – № 1 (27). – С. 54–62.

2. Эрих Г. Паттерны объектно–ориентированного проектирования / Г. Эрих, Х. Ричард, Д. Роберт, В. Джон – Санкт–Петербург: Питер, 2020. – 448 с.

3. Thorn A. Moving from Unity to Godot: An In-Depth Handbook to Godot for Unity Users / A. Thorn – Berkeley, CA: Apress, 2013. – 273 p.

4. Node [Электронный ресурс] // Godot Engine (3.5) documentation in English. – URL: https://docs.godotengine.org/en/3.5/classes/class_node.html ?highlight=node (дата обращения: 03.05.2023).

5. Polančec D. Developing MOBA games using the Unity game engine / D. Polančec, I. Mekterović // 2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) – 2017 – PP. 1510–1515.

6. SilentWolf – backend services for Godot Engine. [Электронный ресурс] // SilentWolf. – URL: https://silentwolf.com/ (дата обращения: 22.05.2023).

7. Sršen M. Developing a Game Engine in C# Programming Language / M. Sršen, T. Orehovački // 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO) – 2021 – PP. 1717–1722.

8. Akenine-Möller T. Real-Time Rendering / T. Akenine-Möller, E. Haines, N. Hoffman. – CRC Press, 2019. – 1045 p.

9. Bradfield C. Godot Engine Game Development Projects: Build five cross–platform 2D and 3D games with Godot 3.0. Godot Engine Game Development Projects / C. Bradfield – Packt Publishing Ltd, 2018. – 298 p.

10. Console support in Godot [Электронный ресурс] // Godot Engine (3.5) documentation in English. – URL: https://docs.godotengine.org/en/stable/tutorials/ platform/consoles.html (дата обращения: 07.05.2023).

11. Sinclair B. Epic launches Unreal Engine 5 [Электронный ресурс] / B. Sinclair // GamesIndustry. – URL: https://www.gamesindustry.biz/epic-launches-unreal-engine-5 (дата обращения: 06.05.2023).

12. Dealessandri M. What is the best game engine: is Unity right for you? [Электронный ресурс] / M. Dealessandri // GamesIndustry. – URL: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you (дата обращения: 06.05.2023).

13. License [Электронный ресурс] // Godot. – URL: https://godotengine.org/license/ (дата обращения: 07.05.2023).

14. Exporting projects [Электронный ресурс] // Godot Engine (3.5) documentation in English. – URL: https://docs.godotengine.org/en/stable/tutorials/export/exporting_projects.html (дата обращения: 13.05.2023).

15. GDScript reference [Электронный ресурс] // Godot Engine (3.5) documentation in English. – URL: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html (дата обращения: 07.05.2023).

16. Pavleas, J. Learn 2D Game Development with C# / J. Pavleas, J. Keng-Wei Chang, K. Sung, R. Zhu – Berkeley, CA: Apress, 2013. – 292 p.

17. Sloan K. Python, PyGame and Raspberry Pi Game Development / K. Sloan – Berkeley, CA: Apress, 2016. – 198 p.

18. Manzur A. Godot Engine Game Development in 24 Hours, Sams Teach Yourself: The Official Guide to Godot 3.0. / A. Manzur, G. Marques – Sams Publishing, 2018. – 432 p.

19. Releases godotengine/godot [Электронный ресурс] // GitHub – URL: https://github.com/godotengine/godot/releases (дата обращения: 07.05.2023).

20. Salin L. Game Development with MonoGame: Build a 2D Game Using Your Own Reusable and Performant Game Engine. Game Development with MonoGame / L. Salin, R. Morrar – Berkeley, CA: Apress, 2022. – 200 p.

21. Shahrabi S. Diagnosing performance issues in Unity [Электронный ресурс] /S. Shahrabi // Medium – 2019. – URL: https://shahriyarshahrabi.medium

.com/diagnosing-performance-issues-in-unity-c1fab81790b3 (дата обращения: 06.05.2023).

22. Singletons (AutoLoad) [Электронный ресурс] // Godot Engine (3.5) documentation in English. – URL: https://docs.godotengine.org/en/stable/tutorials/scripting/singletons_autoload.html (дата обращения: 09.05.2023).

23. Compare Unity plans: Pro vs Plus vs Free. Choose the best 2D-3D engine for your project! [Электронный ресурс] // Unity. – URL: https://unity.com/compare-plans (дата обращения: 06.05.2023).

24. Manual: System requirements for Unity 2021 LTS [Электронный ресурс] // Unity Documentation. – URL: https://docs.unity3d.com/Manual/system-requirements.html (дата обращения: 06.05.2023).

25. Tero S. Game development using the open–source Godot Game Engine [Электронный ресурс] / S. Tero // Theseus. – URL: http://www.theseus.fi/handle/10024/746943 (дата обращения: 05.05.2023).

26. Introduction to Blueprints [Электронный ресурс] // Unreal Engine Documentation. – URL: https://docs.unrealengine.com/5.0/en-US/introduction-to-blueprints-visual-scripting-in-unreal-engine/ (дата обращения: 06.05.2023).

27. Savov V. Unity is the little game engine that could revolutionize animated movies [Электронный ресурс] / V. Savov // The Verge. – URL: https://www.theverge.com/2017/6/30/15899446/unity-cinemachine-unite-europe-2017-animation (дата обращения: 06.05.2023).

28. Unreal Engine (UE5) licensing options [Электронный ресурс] // Unreal Engine. – URL: https://www.unrealengine.com/en-US/license (дата обращения: 06.05.2023).

29. What subscription tiers are available? [Электронный ресурс] // Unity Support. – URL: https://support.unity.com/hc/en-us/articles/208610336-What-subscription-tiers-are-available (дата обращения: 06.05.2023).