Реферат

2*D* ИГРА *«TERARIZATOR*» В ЖАНРЕ *SANDBOX* С ЭЛЕМЕНТАМИ СТРОИТЕЛЬСТВА НА ПЛАТФОРМЕ *UNITY* ПОД ОС *WINDOWS*: дипломная работа / Ю.А. Живица – Гомель: ГГТУ им. П. О. Сухого, 2023. – Дипломная работа: 106 страниц, 27 рисунков, 17 таблиц, 13 источников, семь приложений.

Ключевые слова: игра, приложения, компьютерная, мир, персонаж.

Объектом разработки является компьютерное игровое приложение в жанре «*SandBox*».

Целью работы является разработка игрового приложения в жанре «*SandBox*».

Характеристика проделанной работы: при выполнении работы рассмотрены и проанализированы имеющиеся игровые приложения. Проведен обзор подобных приложений, после чего сформирован список требований, которым должен соответствовать разрабатываемый программный продукт. При разработке использованы объектно-ориентированный и структурный подходы программирования. В результате разработки спроектировано и реализовано компьютерное игровое приложение в жанре «*SandBox*».

Дипломная работа выполнена самостоятельно, приведенный в дипломной работе материал объективно отражает состояние разрабатываемого объекта, пояснительная записка проверена в системе «Антиплагиат.ру» (ссылка на систему: *https://www.antiplagiat.ru/*). Процент оригинальности составляет 86. Все заимствованные из литературных и других источников теоретические и методологические положения и концепции сопровождаются ссылками на источники, которые указаны в «Списке использованных источников».

Резюме

Темой работы является «2*D* игра *«Terarizator*» от третьего лица в жанре *SandBox* с элементами строительства на платформе *Unity* под ОС *Windows*».

Объектом исследования является игровое приложение для компьютеров.

Целью работы является разработка игрового приложения для компьютеров в жанре «*SandBox*» с элементами строительства.

Основным результатом работы является, компьютерное игровое приложение.

Рэзюме

Тэмай працы з'яўляецца «2D гульня «Terarizator» ад трэцяй асобы ў жанры SandBox з наступствамі будаўніцтва на платформе Unity пад АС Windows».

Аб'ектам даследавання з'яўляецца задача стварэння прыкладання для кампутараў.

Мэтай працы з'яўляецца распрацоўка гульнявога прыкладання ў жанры «SandBox» з наступствамі будаўніцтва.

Асноўным вынікам працы з'яўляецца справаздача аб праведзенай працы, камп'ютарнае прыкладанне.

Abstract

The theme of the work is «2D game «Terarizator» from a third person in the genre of SandBox with the consequences of construction on the Unity platform for Windows OS».

The object of research is a gaming application for computers.

The aim of the work is to develop a gaming application for computers in the «SandBox» genre with the consequences of construction.

The main result of the work is a report on the work done, a computer game application.

ОГЛАВЛЕНИЕ

Перечень условных обозначений и сокращений

В настоящей пояснительной записке применяются следующие термины, обозначения и сокращения.

Геймплей – компонент игры, отвечающий за взаимодействие игры и игрока.

Рендеринг – процесс получения изображения из 2*D* или 3*D* модели, компьютерный расчет и визуализация объектов и сцен со всей информацией о материалах, текстурах и освещении.

ОС – операционные системы.

ПК – персональный компьютер.

ПО – программное обеспечение.

2*D* – двухмерное пространство.

3*D* – трёхмерное пространство.

*F*2*P* – бесплатное игровое приложение.

ВВЕДЕНИЕ

Индустрия видеоигр очень сильно развивается в наше время, что позволяет ей быстро и легко предоставить людям игры любого жанра и тем самым удовлетворить предпочтения практически каждого игрока. Вычислительная мощность современных цифровых устройств способна воспроизводить компьютерную графику практически любой сложности, что помогает играм конкурировать с киноиндустрией в качестве изображения.

На данный момент, игры являются одним из наиболее популярных и легкодоступных источников удовольствий. Имеется возможность играть в них как одному, так и в компании нескольких людей, что помогает людям совершенствовать свои социальные навыки. Видеоигры доступны практически на всех современных гаджетах: компьютерах, смартфонах, телевизорах, консолях, электромобилях и даже на электронных часах. В видеоигры играют люди всех возрастов, так как видеоигры помогают провести досуг, и, к тому же, многие из них имеют достаточно низкий порог вхождения.

Игровое приложение «*Terarizator*» разработано на одном из самых продвинутых и современных движков для создания игр *Unity*, который в свою очередь, позволяет разрабатывать игры под все платформы и ОС.

Целью игры является добавление и разрушение в игровой мир блоков, передвижение игрока по миру, спуск в шахту, для нахождения основных ресурсов игры.

При проектировании дипломной работы задача была разбита на несколько подзадач:

– определить существующие методы реализации игрового приложения жанра «*SandBox*»;

– выбор движка, для создания игры;

– проектирование архитектуры игрового приложения;

– разработать игровое приложение, его структуру и иерархию классов;

– реализовать игровое приложение;

– произвести тестирование и верификацию игрового приложения.

При выборе движка *Unity,* который имеет большинство возможностей, которые не имеют другие движки. Из больших преимуществ, она хорошо оптимизирована под слабые устройства и обладает упрощенным интерфейсом. Отлично подходит для создания 2*D* игр, т.к. имеет различные компоненты, для реализации физики и взаимодействия игровых 2*D* объектов.

1 СУЩЕСТВУЮЩИЕ МЕТОДЫ РЕАЛИЗАЦИИ ИГРОВОГО ПРИЛОЖЕНИЯ ЖАНРА *SANDBOX*

1.1 Описание игровой предметной области

Игровая индустрия – наиболее распространённая современная сфера развлечений, набравшая популярность с развитием информационных технологий. Более 60% населения, пользующихся цифровыми устройствами, загружают игровые приложения и проводили за игрой своё время.

Зарождение данной сферы началось ещё в 1970 году, будучи движением энтузиастов, и за несколько десятилетий она достигла колоссальных объёмов продаж на рынке.

В 1970 году начали появляться первые игровые консоли, которые поставили рекордное количество продаж в игровой индустрии. Внешне приставка напоминает коробку, с панелью, с помощью которой, через коммутатор, который позволяет переключаться между каналами телевизора и игровой приставкой. Приставка *Magnavox Odyssey* была в дальнейшем модифицирована и на ее основе начали появляться более знаменитые консоли [1, с. 6].

На данный период времени, игроков, которые когда-либо сталкивались с игровой индустрией, насчитывается около двух миллиардов человек.

Перспективные направления развития игровой индустрии являются:

– мобильные игровые приложения;

– развитие облачного хранилища данных;

– ежедневная разработка и доработка *VR*/*AR*;

– использование игр в качестве проведения мероприятий;

– проведение масштабных турниров и соревнований по играм;

– киберспорт.

В настоящий момент самый легкодоступный сегмент игрового рынка приходятся на мобильные игровые приложения. На 2023 год приходится порядка 2,2 млрд. игроков, что означает низкий порог входа в игровую индустрию. И поскольку мобильные приложения начали адаптировать под различные платформы, их можно считать одними из наиболее популярных на сегодняшний день.

С развитием игровых приложений, развиваются виртуальные и социальные пространства. Например, если раньше игровые миры не могли создавать изображения, то на сегодняшний момент это все в открытом доступе.

Кроме того, начиная с 2016 года, происходит интеграция приложений с популярными соцсетями. Игроки могут пообщаться, выразить свои эмоции, посочувствовать событию, связанному в игре, делать скриншоты и размещать их в соцсетях.

Если говорить о киберспорте, как о профессии, есть большое количество плюсов и минусов в нем. Одним из самых главных плюсов, является демонстрирование себя, показать свои возможности с той или ной стороны, благодаря чему можно зарабатывать большое количество денег уже с раннего возраста. Киберспорт позволяет развивать мыслительные способности и улучшать навыки коммуникации. От игр растет интеллект, а также развивается аналитическое мышление. Еще одним преимуществом, является то, что это находится все еще на этапе развития и никто не может догадаться, что будет в дальнейшем. На рисунке 1.1 продемонстрирован рост киберспорта в игровой индустрии, в соответствии с временем.

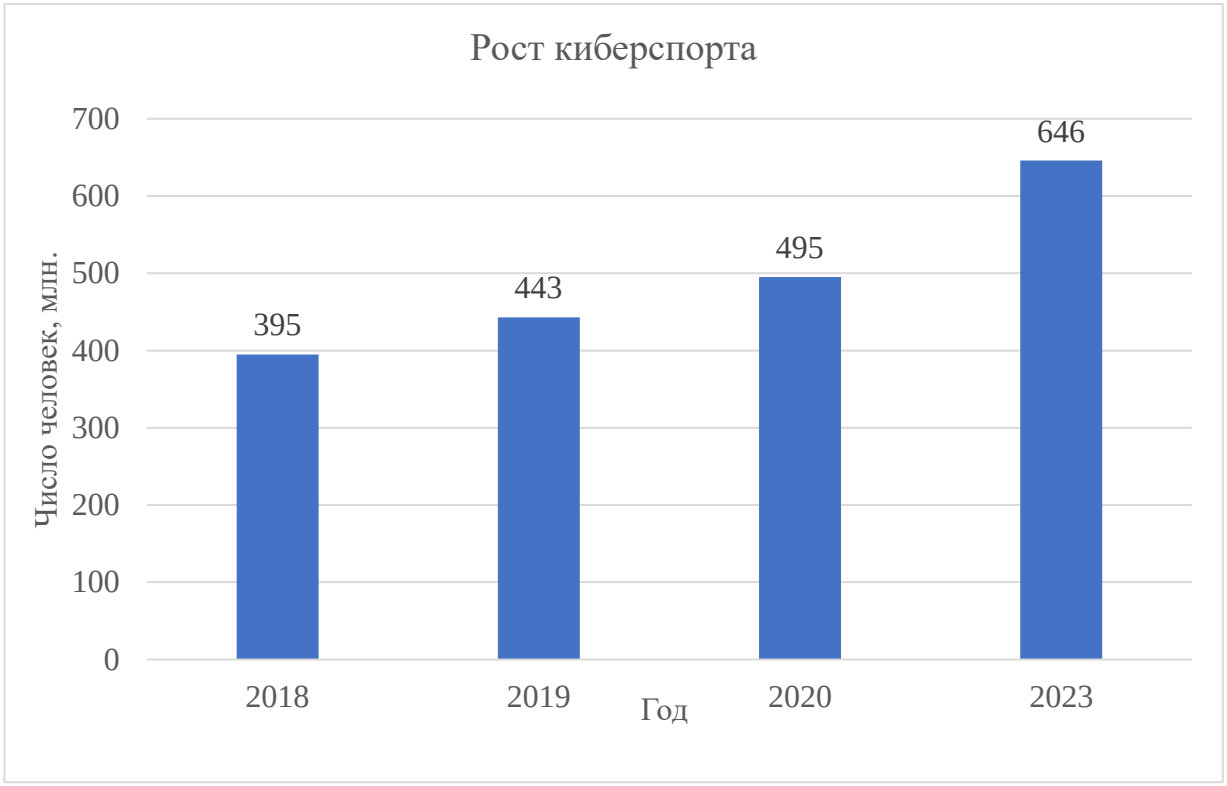

Рисунок 1.1 – Рост киберспорта в игровой индустрии

Существуют недостатки, которые играют большую роль в игровой индустрии:

– зависимость от игр;

– плохая оптимизации новых крупных проектов, под устаревшее ПО;

– проблема энергопотребления игровых приложений;

– сегмент рынка слишком большой, поэтому пробиться на рынок очень тяжело.

В целом, новые технологии, дают молодому поколению проявить свои возможности в какой-либо отрасли киберспорта, сделать огромные шаги в развитии компьютерных игр и игровой индустрии

1.2 Описание жанра *SandBox*

Для разработки игрового приложения, необходимо знать, общую информацию о жанре, к которому относится разрабатываемый продукт.

Жанр *SandBox* является аналогом игр, в котором нет основных целей для игрока и пользователю можно заниматься чем угодно.

Самой главной особенностью таких игр, является то, что главная цель может быть проигнорирована игроком или ее в принципе может и не быть [5].

К примеру, у самой популярной игры *Minecraft* в жанре *SandBox*, у игрока нет как таковой цели, что ему необходимо делать и чем заниматься, каждый сам для себя решает, что ему нужно делать, а что нет.

Основные механики, которые можно встретить в играх, подобного жанра:

– строительство базы или жилого дома;

– добыча ресурсов, чаще всего в шахте или на поверхности территории;

– соревнование между игроками;

– сражение между игроками, штурм городов и домов;

– прокачка персонажа.

В большинстве игр данного жанра игроку предоставлена свобода действий, благодаря чему игроку необходим огромный мир, для исследования. Скриншот из игры в жанре *SandBox* показан на рисунке 1.2.

Рисунок 1.2 – Пример игры в жанре *SandBox*

Одной из главной и немаловажной особенностью, является процедурно-генерируемый мир. При каждом заходе в игру, буквально создается новая карта,

что добавляет в игру эффект случайности. Чтоб игрок каждый раз не запоминал одну и ту же карту, а постоянно заново изучал и узнавал окружение.

Если взять еще один, приближенный прототип игрового приложения *Terarria*, в ней присутствует процедурно-генерируемый мир. К примеру, один игрок, который сильно развился, может присоединится к более слабому игроку и помочь в развитии ему.

1.3 Аналитический обзор существующих приложений жанра Sand-Box

1.3.1 Первым и самым главным проектом-аналогом является *Minecraft*. Перед началом игры необходимо выбрать параметры процедурной генерации мира. Игрок может ввести в специальное поле число, которое необходимо для генерации уникального мира. Если данные не вводить, игра автоматически сгенерирует мир. Игровой мир состоит из расставленных в фиксированном порядке кубов, блоков и практически не имеет ограничения. Игровой персонаж появляется на поверхности в случайном биоме – в горах, в лесу, в равнинах, пустыне и т.д.

По мере прохождения игры, игрок может делать предметы, для дальнейшего развития и достижения своих целей. Персонажем можно управлять от первого лица, но также добавлена механика переключения камеры от третьего лица на клавишу. У персонажа есть инвентарь, в котором он может хранить различные предметы. Игрок имеет здоровье, очки опыта и уровень, благодаря которому, он может улучшать предметы, для дальнейшего прохождения игры.

Достоинства игры *Minecraft*:

– безграничные возможности – *Minecraft* является игрой в жанре *SandBox*, в которой игроку вольно делать все, что угодно. Здесь нет какой-то конкретной цели, и игроку предоставляется большой открытый мир с полностью разрушаемым окружением;

– огромное сообщество – *Minecraft* имеет большую аудиторию и доступна на большинстве платформ: от мобильных до консолей и ПК

– регулярное развитие – разработчики почти каждый месяц выпускают масштабные обновления, в которых меняются механики и игрокам приходится заново привыкать к тем или иным обновлениям, что способствует появление интереса;

– кроссплатформенность – игра доступна на всех видах платформ. Исключением являются консоли предыдущего поколения;

– узнаваемый стиль – игру узнают все, от малого возраста люди, до пожилого.

На рисунке 1.3 показан скриншот из игры *Minecraft*.

Рисунок 1.3 – Интерфейс и игровой процесс приложения *Minecraft*

Недостатки приложения:

– высокая цена – если говорить о официальных ценах, то игра 10-летней давности стоит относительно дорого;

– микротранзакции – внутриигровые покупки существуют в игре для того, чтобы игроки тратили большое количество денег на игру, а не вкладывали в ее развитие;

– отсутствие обучения – в каждой масштабной игре, присутствует обучение, которые помогает игроку понять основы и базовые механики приложения;

– плохая репутация – из-за того, что игра очень популярная, в игре присутствуют неадекватные люди, которые всячески способствуют развитию отрицательных отзывов об игре;

– простота игры – некоторые люди считают, что игра слишком простая и ненатуральная, поэтому игроки сами делают все возможное, для того, чтоб сделать игру максимально реалистичнее

Minecraft – является одной и популярнейших игр в жанре *SandBox*. В приложение играют миллиарды людей на всей планете, что способствует развитию игровой индустрии. Игрокам предоставлен открытый код игры, с помощью которого делаются различные модификации для игры.

1.3.2 Следующей игрой, которая стала одной из самых популярных в своем жанре – *Terraria*, выпущенная широко известной компанией *Re-Logic* ещё в 2011 году. Данный проект является двухмерным одиночным и многопользовательским *SandBox* игры в жанре *RPG*, в котором игрок управляет персонажем,

который должен победить финального босса. Наглядный игровой процесс продемонстрирован на рисунке 1.4.

Рисунок 1.4 – Игровое приложение *Terraria*

Достоинства игрового приложения *Terraria*:

– богатый игровой мир – *Terraria* предполагает огромный, случайно генерируемый мир, полный разнообразных биомов, локаций и подземелий. Игроки могут исследовать его, открывать новые области и находить различные сокровища;

– множества предметов и рецептов – в игре присутствует огромное разнообразие предметов, оружия, доспехов и аксессуаров. Существует множество способов создания и улучшения предметов, а также система крафта, которая позволяет игрокам комбинировать различные материалы и создавать новые вещи;

– разнообразный геймплей – *Terraria* предлагает игроку разнообразные возможности геймплея. Игрок может сражаться с различными врагами и боссами, строить различные сооружения и дома, копать подземелья в поисках ресурсов, развивать персонажа и многое другое. Есть также режимы мультиплеера, позволяющие играть с друзьями;

– комплексность и глубина – в *Terraria* есть много сложных механик и систем, которые позволяют игрокам погрузиться в глубокий и сложный игровой мир. Это может быть особенно привлекательно для тех, кто любит глубокий геймплей и постоянное прогрессирование.

Недостатки игры Terraria:

– графика и анимации – хотя графика *Terraria* имеет свое очарование и уникальный стиль пиксельного искусства, некоторым игрокам может не нравиться ее простота и отсутствие детализации. Анимация тоже может быть несколько ограниченной по сравнению с другими современными играми;

– начальный уровень сложности – в начале игры *Terraria* может показаться сложной для новичков. Отсутствие явного обучения может быть вызовом для игроков, которые не знакомы с подобными жанрами игр или не имеют опыта в песочницах;

– ограниченность в вертикальном пространстве – *Terraria* ограничена двумерным пространством и не предлагает возможности для полетов или исследования в вертикальном направлении. Некоторым игрокам это может показаться ограничивающим или однообразным по сравнению с трехмерными играми;

– нет постоянного обновления контента – в отличие от некоторых других игр, *Terraria* не имеет регулярных обновлений контента. Хотя она имеет огромное количество контента и ресурсов для исследования, некоторым игрокам может не хватать новых обновлений, чтобы поддерживать интерес на протяжении длительного времени.

Несмотря на некоторые недостатки, *Terraria* остается популярной игрой, благодаря своим уникальным особенностям, глубине геймплея и богатому миру, который она предлагает.

1.3.3 Один из наиболее популярных проектов, является *Garry's Mod*. *Garry's Mod* – игра с видом от первого лица. С помощью *Garry's Mod* можно проводить любые действия с объектами и персонажами из игр на движке *Source* (*CS*:*S, CS*:*GO, TF*2*, HL*2 и других игр от *Valve*), например, менять текстуры, изменять размеры, копировать, соединять, взрывать, прикручивать колеса, изменять выражения лиц персонажей и т. д. Модификация предоставляет игроку огромные просторы для творчества.

Garry's Mod 9 предоставлял несколько сетевых режимов игры: игра в футбол с помощью гравипушки, стрельба из лазера, охота на птиц (одни игроки играют за охотников, другие – за птиц), гонки на арбузах и т. д. В *Garry's Mod* 10 доступны режимы: «песочница», *Ascension* (двумерная игра, задача в которой – добраться до определённой точки), *Dog Fight Arcade* (воздушные сражения), *Fretta*, *PropHunt*, *Trouble in Terrorist Town*. Ранее был доступен режим гонок на арбузах, но был убран в связи с его непопулярностью. Другие игровые режимы доступны в виде отдельных скачиваемых дополнений.

В *Garry's Mod* было создано множество игровых режимов, которые разрабатываются и по сей день. Простор для создания игровых режимов был обеспечен благодаря *SDK* движка *Source* и языку программирования *Lua*.

На рисунке 1.5 продемонстрирована часть уникального геймплея игр жанра *SandBox*.

Рисунок 1.5 – Игровой процесс *Garry’s Mod*

Проанализировав множество отзывов о данном нетипичном проекте, был составлен список достоинств и недостатков.

Достоинства *Garry’s Mod*:

– бесконечные возможности – *GMod* позволяет игрокам создавать свои собственные игровые механики, миссии, истории и многое другое. Игрок может строить различные объекты, сооружения и механизмы, а также изменять и модифицировать игровые элементы, используя широкий спектр инструментов и ресурсов;

– множество допустимых модификаций – *GMod* поддерживает модификации, которые позволяют игрокам расширить игровой опыт еще больше. Есть тысячи модов, добавляющих новые предметы, персонажей, карты, режимы игры и другие функции. Это обеспечивает большое разнообразие контента и постоянное обновление игры;

– мультиплеер и совместное взаимодействие – *GMod* предлагает множество режимов мультиплеера, включая кооперативную игру, *PvP*-сражения и совместное творчество. Вы можете играть с друзьями, создавать совместные проекты, проводить мероприятия и обмениваться контентом с другими игроками по всему миру;

– юмор и безумие – *Garry's Mod* известен своим юмором и неожиданными ситуациями. Игра позволяет создавать комические и необычные сцены,

провоцировать безумие и создавать смешные моменты. Это делает игру развлекательной и подходящей для проведения неформального времени в виртуальном мире.

Недостатки *Garry's Mod*:

– нет определенных целей или сюжета – *GMod* не имеет явной сюжетной линии или целей. Он полностью зависит от воображения и творчества игрока. Некоторым игрокам может не хватать структурированного игрового опыта или конкретных задач для выполнения;

– сложность для новичков – для новых игроков *GMod* может быть сложным и пугающим из-за большого количества инструментов и функций. Освоение всех возможностей и особенностей игры требует времени и изучения;

– графическое оформление – *GMod* основан на игровом движке *Source*, который был разработан довольно давно. Графика и визуальные эффекты могут показаться устаревшими по сравнению с современными играми. Однако, благодаря гибкости и возможностям модификаций, можно улучшить графический опыт.

В целом, *Garry's Mod* является популярной песочницей с бесконечными возможностями для творчества и экспериментов. Она предлагает уникальный игровой опыт и способствует развитию творческого мышления у игроков.

1.3.4 Последним популярным в больших кругах проектом является *Starbound*. *Starbound* – это 2*D*-песочница с элементами приключений и выживания, разработанная и выпущенная компанией *Chucklefish*. Она предлагает игрокам исследование космического мира, постройку баз, сражения с врагами и многое другое. На рисунке 1.6 продемонстрирована игра в жанре *SandBox* под названием *Starbound*.

Рисунок 1.6 – Игровое приложение *Starbound*

Достоинства *Starbound*:

– большой открытый мир – *Starbound* предлагает огромный открытый игровой мир для исследования. Игроку предстоит путешествовать по различным планетам, звездным системам и галактикам, каждый со своей уникальной средой, биомами и ресурсами. Мир генерируется случайным образом, что обеспечивает бесконечное разнообразие исследовательского опыта;

– глубокий геймплей и прогрессия – *Starbound* имеет разнообразные игровые механики, включая строительство баз, ресурсодобычу, крафтинг, сражения и выполнение квестов. Существует прогрессия персонажа, различные классы и умения, которые позволяют вам развивать своего персонажа в соответствии с вашими предпочтениями и стилем игры;

– множество разнообразных рас и персонажей – В *Starbound* игрок может выбрать одну из нескольких рас и настроить своего персонажа. Каждая раса имеет уникальные особенности, историю и стиль игры. Также встречаются многочисленные НИП, с которыми вы можете взаимодействовать, торговать и получать задания.

Недостатки игры *Starbound*:

– графика и анимации – некоторым игрокам может не нравиться пиксельная графика и стиль искусства *Starbound*. Она может показаться устаревшей по сравнению с современными играми;

– некоторая повторяемость – хоть *Starbound* предлагает огромный мир для исследования. Однако игрокам может надоесть повторение определенных игровых механик, особенно при длительных сессиях игры;

– технические проблемы – в прошлом *Starbound* имела некоторые технические проблемы, такие как низкая производительность на некоторых системах или проблемы с сетевым взаимодействием в мультиплеере. Хотя многие из них были исправлены патчами, они могут быть присутствовать в редких случаях.

В целом, *Starbound* предлагает глубокий и разнообразный игровой мир с огромными возможностями для исследования, творчества и совместной игры.

1.4 Инструменты и методы реализации игровых приложений

1.4.1 *GameMaker*: *Studio* – один из самых популярных движков для разработки игр. Он позволяет создавать приложения для различных платформ. Движок предоставляет возможности разработки без особых навыков программирования или знания большого количества языков. Данный движок использует систему *Drag and Drop*, когда разработчик использует действия и события над объектами, чтобы задать механику работы того или иного элемента игры.

Плюсом данного редактора является кроссплатформенность – возможность создавать собственные игры для персональных компьютеров, мобильных

устройств и игровых консолей. Также имеется встроенный магазин спрайтовой графики, звуков и анимаций. Это предоставляет возможность быстрой разработки, не занимаясь созданием вышеперечисленных частей помимо логики.

Интерфейс движка *GameMaker*: *Studio* представлен на рисунке 1.7.

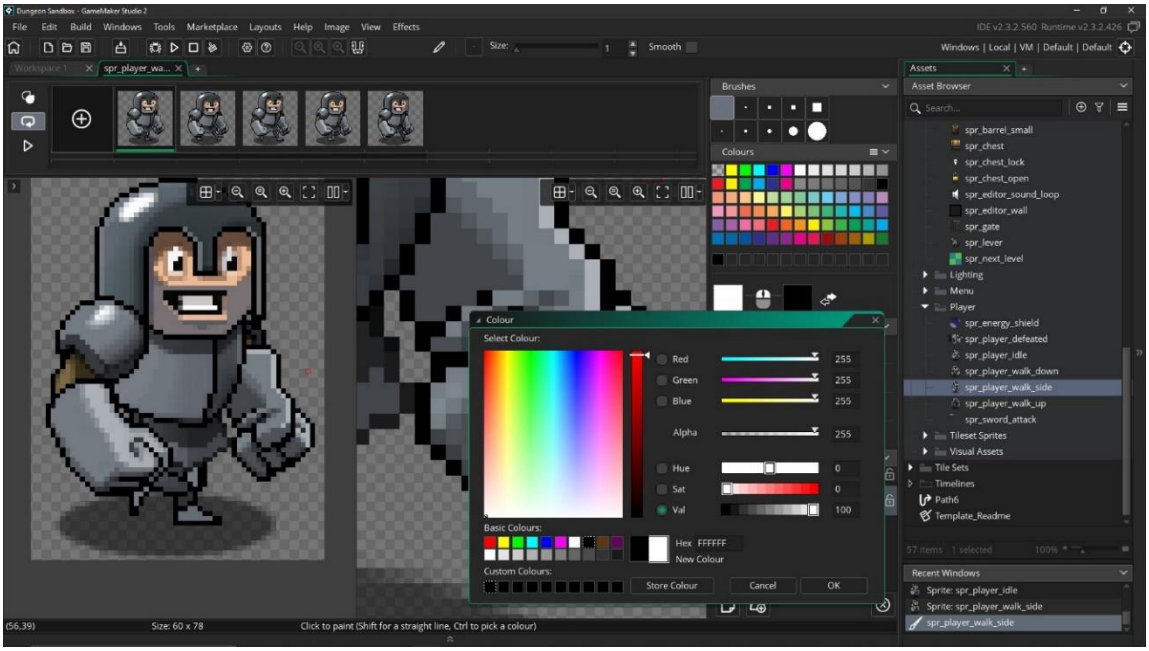

Рисунок 1.7 – Интерфейс игрового движка *GameMaker*: *Studio*

1.4.2 *Unreal Engine* (*UE*) – это движок для создания игр, один из двух наиболее популярных в мире. В ней можно работать с персонажами, логикой, физикой и графикой игры.

UE разработала компания *Epic Games* для своей игры под названием *Unreal*, и после этого движок стал популярен. Его основное отличие – хорошая оптимизация: *Unreal Engine* создавался не как отдельный коммерческий продукт, а как рабочий инструмент, и ориентирован он на *3D*-игры.

Наиболее известная сейчас версия – *Unreal Engine* 4, или *UE4*. Но в 2020 году вышел *Unreal Engine* 5, а некоторые игры до сих пор написаны на старых версиях 2 или 3. С каждым обновлением доступная графика становится все более мощной, возможности повышаются и позволяют создавать все более сложные и реалистичные игры.

Изначально движок создавался для внутренних нужд компании *Epic Games*. *Epic Games* разрабатывала на нем собственные игры, а ее проекты были трехмерными. Поэтому поддержка двумерных проектов была слабой. Но движок оказался таким удачным, что им начали пользоваться и другие игровые разработчики. Тем не менее ориентированность на 3*D*-игры сохранилась, и *Epic Games* начали добавлять больше возможностей для двумерных игр относительно недавно. Также немаловажной частью движка *Unreal Engine* является мощная

оптимизация и *C++*. Это мощный, быстрый, но довольно сложный язык, который непросто изучить с нуля. Тем не менее его применение позволяет хорошо оптимизировать игры. Это важное отличие *UE* от другого популярного движка, *Unity*: создать игру сложнее, но, если получится – она, скорее всего, будет быстрее и эффективнее.

Писать на *C++* сложно, а с движком работают не только программисты, но и, например, художники-аниматоры. Поэтому *Epic Games* разработали для *UE* внутренний язык визуального программирования, который называется *Blueprints*. Это способ программировать без написания кода – создавать программы из специальных визуальных блоков и связей между ними. Писать так игры легче, и способ подходит даже для тех, кто незнаком с *C++*. Однако сложную логику все же лучше реализовывать с помощью кода. *Blueprints* облегчает задачу, но он не универсален.

На рисунке 1.8 наглядно видно программную среду разработки *Unreal Engine*.

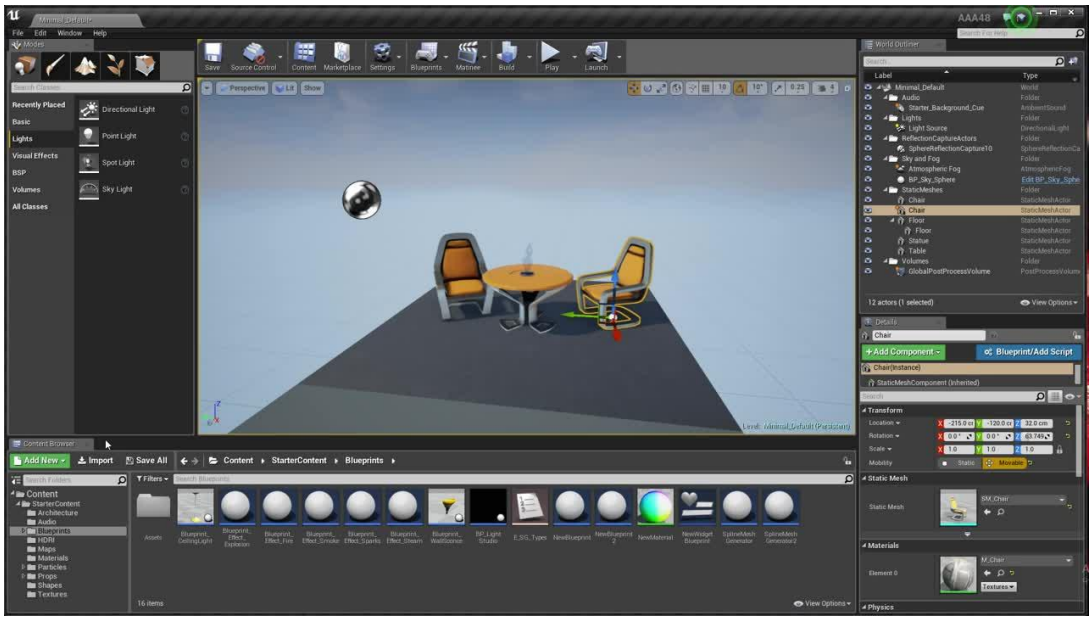

Рисунок 1.8 – Программная среда разработки *Unreal Engine*

В *Unreal Engine* огромное количество возможностей для создания фотореалистичной трехмерной графики. В нем множество текстур, визуальных эффектов и материалов, которые можно применить к объектам, чтобы изменить их внешний вид. Графика гибко настраивается, в результате можно создавать какие угодно материалы, поверхности и эффекты, задавать им различные параметры и смешивать друг с другом. Также есть функции, которые позволяют добавить графику на объекты или предметы за одно нажатие, при помощи панели инструментов.

1.4.3 *Unity* – межплатформенная среда разработки компьютерных игр. *Unity* позволяет создавать приложения, работающие под более чем 20 различными операционными системами, включающими персональные компьютеры, игровые консоли, мобильные устройства, интернет-приложения и другие. Выпуск *Unity* состоялся в 2005 году и с того времени идёт постоянное развитие.

Основными преимуществами *Unity*, являются наличие визуальной среды разработки, межплатформенной поддержки и модульной системы компонентов. К недостаткам относят появление сложностей при работе с многокомпонентными схемами и затруднения при подключении внешних библиотек [2, с. 12].

Редактор *Unity* имеет простой *Drag&Drop* интерфейс, состоящий из различных окон, которые легко настраивать, благодаря чему можно производить отладку игры прямо в редакторе. Движок поддерживает два скриптовых языка: *C#*, *JavaScript* (модификация). Ранее была поддержка *Boo* (диалект *Python*), но его убрали в 5-й версии. Расчёты физики производит физический движок *PhysX* от *NVIDIA*[6]. Среда разработки *Unity*, продемонстрирована на рисунке 1.9.

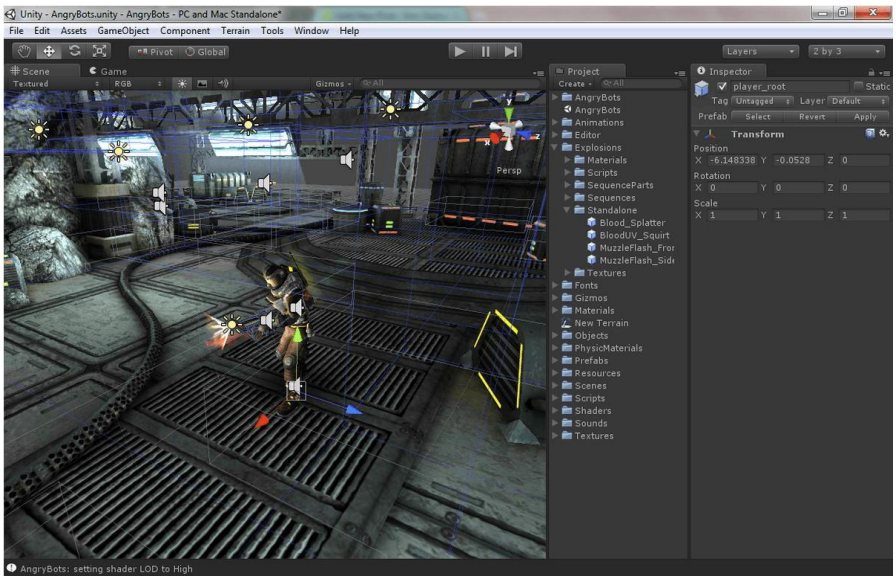

Рисунок 1.9 – Среда разработки *Unity*

Проект в *Unity* делится на сцены (уровни) – отдельные файлы, содержащие свои игровые миры со своим набором объектов, сценариев, и настроек. Сцены могут содержать в себе как, собственно, объекты (модели), так и пустые игровые объекты – объекты, которые не имеют модели («пустышки»). Объекты, в свою очередь содержат наборы компонентов, с которыми и взаимодействуют скрипты. У объектов есть название (в *Unity* допускается наличие двух и более объектов с одинаковыми названиями), может быть тег (метка) и слой, на котором он должен отображаться. Так, у любого объекта на сцене обязательно присутствует компонент *Transform* – он хранит в себе координаты местоположения, поворота и

размеров объекта по всем трём осям. У объектов с видимой геометрией по умолчанию присутствует компонент *Mesh Renderer*, делающий модель объекта видимой [7, с. 24].

Движок поддерживает множество популярных форматов. Модели, звуки, текстуры и материалы, скрипты можно запаковывать в формат *unityassets* и передавать другим разработчикам, или выкладывать в свободный доступ. Этот же формат используется во внутреннем магазине *Unity Asset Store*, в котором разработчики могут бесплатно и за деньги выкладывать в общий доступ различные элементы, необходимые при создании игр. Чтобы использовать *Unity Asset Store,* необходимо иметь аккаунт разработчика *Unity*. *Unity* имеет все нужные компоненты для создания мультиплеера. Еще одним преимуществом, является использование подходящей пользователю контроля версии. К примеру, *Tortoise SVN* или *Source Gear* [4, с. 67].

В *Unity* входит *Unity Asset Server* – инструментарий для совместной разработки на базе *Unity*, являющийся дополнением, добавляющим контроль версий и ряд других серверных решений [3].

Unity имеет большинство возможностей, которые не имеют другие движки. Из больших преимуществ, она хорошо оптимизирована под слабые устройства и обладает упрощенным интерфейсом. Имеет большую аудиторию, большое количество литературы, курсов, помощников, для создания игры. Отлично подходит для создания 2*D* игр, т.к. имеет различные компоненты, для реализации физики и взаимодействия игровых 2*D* объектов. Именно поэтому при сравнении движков был выбран *Unity* для разработки игрового приложения «*Terarizator*» под *ОС Windows*.

1.5 Требования к функционалу игрового приложения

Игровое приложение должно быть со свободным миром, случайной генерацией и обеспечивать свободный геймплей. У персонажа присутствует инвентарь, который представляет собой панель, на которой содержится картинка и пару маленьких панелей. Также у персонажа имеется пользовательский инвентарь, по которому он может выбрать предмет, а затем он должен появится у игрока в руке.

Мир необходимо генерировать путем добавления шума на текстуру, дальнейшем связывания текстуры с блоками. Мир полностью зависит от семечка, которое представляет собой случайное значение.

В игре должны присутствовать пещеры, а сделать это необходимо путем добавления на новую текстуру шума. Дальше добавить эту текстуру, перед генерацией мира.

Помимо пещер, игровое приложение должно содержать различные виды биомов, а именно:

– пустыня, которая представляет собой коричневый цвет и содержит собственную генерацию со своими блоками;

– зимний, который содержит блоки снега и растительность;

– поляна, биом, у которого шанс на появления травы больше, а шанс на деревья меньше;

– лес, в котором повышен шанс на появление деревьев.

Также с генерацией, должны генерироваться деревья, с определенными значениями высоты с шансом на их появление. К примеру, деревья должны сгенерироваться с шансом 5 процентов с высотой от 4 до 8 единиц. На макушке дерева должна генерироваться листва.

У каждого биома, должны быть свой набор текстур и спрайтов, а также блоки на заднем фоне.

Персонаж имеет три набора инструментов:

– каменная кирка, которая должна ломать только те блоки, который каким-либо образом связаны с шахтой;

– каменный топор, благодаря которому, можно ломать деревья, кактусы;

– каменный молот, которым можно ломать блоки, на заднем фоне.

Пользователь обладает камерой, которая следует за игроком, куда бы игрок не пошел, камера должна быть привязана к игроку.

Добавить автопрыжок, а именно проверить на соприкосновение игрока с блоком.

На рисунке 1.10 показан примерный набор блоков, для генерации мира.

Рисунок 1.10 – Пример набора блоков для генерации

Данные требования полностью раскрывают концепцию приложения и соответствуют поставленной задаче.

2 АРХИТЕКТУРА И ФУНКЦИОНАЛ ИГРОВОГО ПРИЛОЖЕНИЯ «*TERARIZATOR*»

2.1 Взаимодействие объектов на игровой сцене

В редакторе *Unity* для изменения свойств компонентов используется окно *Inspector*. К примеру, если необходимо изменить положения какого-либо объекта, компонента *Transform*, то следует изменить значение в *Inspector* или при помощи скрипта. Также можно изменить цвет объекта или физические его свойства. В основном скрипты изменяют все свойства, но разница в том, что *Script* изменяет на протяжении какого-то времени, в период одного кадра, а изменения в *Inspector* сработает в ту же секунду, как запускается игровое приложение.

Самый простой способ обратиться к компоненту на игровой сцене, это создание скрипта, который будет вытягивать из объекта какой-либо компонент. Функция *GetComponent* сохраняет переменную, которая содержит объект компонента. На рисунке 2.1 продемонстрировано схематично, как выполняется данная функция.

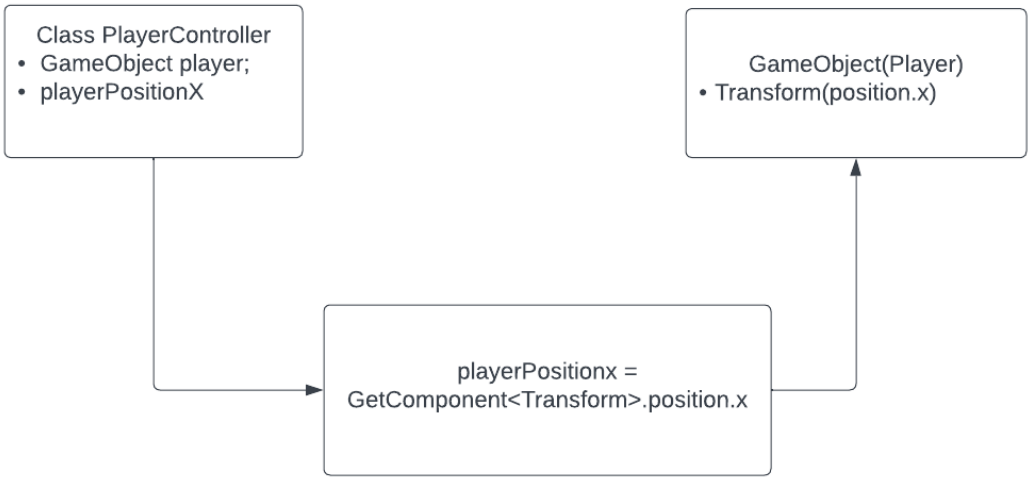

Рисунок 2.1 – Схема реализации функции *GetComponent*

Каждый объект имеет свой физический материал, который в свою очередь содержит коллайдеры. У объектов 2*D*-коллайдеры позволяют точно определить как примитивную, так и пользовательскую форму спрайтов. Если добавить к 2*D* объектам компонент *Rigidbody* 2*D*, предметы будут реагировать на гравитацию и вести себя как твердотельные объекты. *Capsule Collider*2*D* – это коллайдер, который используется при работе с 2*D* физикой. Форма коллайдера представлена в виде капсулы, которая помогает при проверке на соприкосновение с объектами

лучше, чем у других коллайдеров. Данная капсула напрямую привязана к координатам, ее края параллельны осям *X* и *Y* игрового пространства.

Существует еще один способ, как добавить физические свойства объекту. При помощи компонента 2*D Joints* можно добавить один *rigidbody* объект к другому или к фиксированной точке в пространстве. Благодаря данному компоненту, можно сделать ограничения, которые не позволяют объекту, к примеру, двигаться по оси *X*. Также есть *Hinge Joint*, которая позволяет осуществлять вращение вокруг заданной точки. А есть *Spring Joint*, который в свою очередь держит объекты раздельно, но расстояние между этими точками можно задавать вручную.

2.2 Архитектура игрового приложения «*Terarizator*»

Перед разработкой игрового приложения, необходимо разбить главную задачу на несколько подзадач.

На начальных этапах разработки и построения архитектуры было выделено несколько подзадач:

– случайная генерация ландшафта;

– логика приложения, а именно как объекты будут взаимодействовать на сцене;

– пользовательский интерфейс;

– графическая часть игрового приложение, а именно спрайты, текстуры и анимации;

– настройка анимации и аниматора персонажа;

– сцена и сами объекты.

Каждый из этих пунктов является отдельной веткой во всём дереве проекта. Разбиение процесса разработки на несколько подпроцессов гораздо облегчает выполнение поставленной задачи и облегчает доступ к необходимому разделу.

Рассмотрим эти подзадачи по отдельности. Случайная генерация происходит путем добавления шума на текстуру, которая выбирается автоматически. Белый цвет текстуры, отвечает за блоки, находящиеся в игровом мире, черный цвет, представляет собой пробелы в мире. Пробелы сделают мир более случайный и неровным, что соответствует ландшафту мира. Для генерации блоков земли, необходима переменная, которая будет отвечать за высоту блоков земли. Размер мира устанавливается пользователем путем изменения *worldSize*. Скрипт генерации мира содержит методы, который позволят удалять, добавлять и проверять блоки в мире. Создан метод, который реализует разделение мира, на отдельные фрагменты, для дальнейшего высвобождения ресурсов. Если игрок выйдет за какой-либо фрагмент, на сцене у этого объекта пропадает видимость и

игроку нельзя увидеть этот фрагмент до того момента, когда он снова не соприкоснётся с ним. Проверка на количество фрагментов мира, происходит при помощи деления размера мира на размер фрагмента. На рисунке 2.2 показана схема, по которой происходит генерация ландшафта.

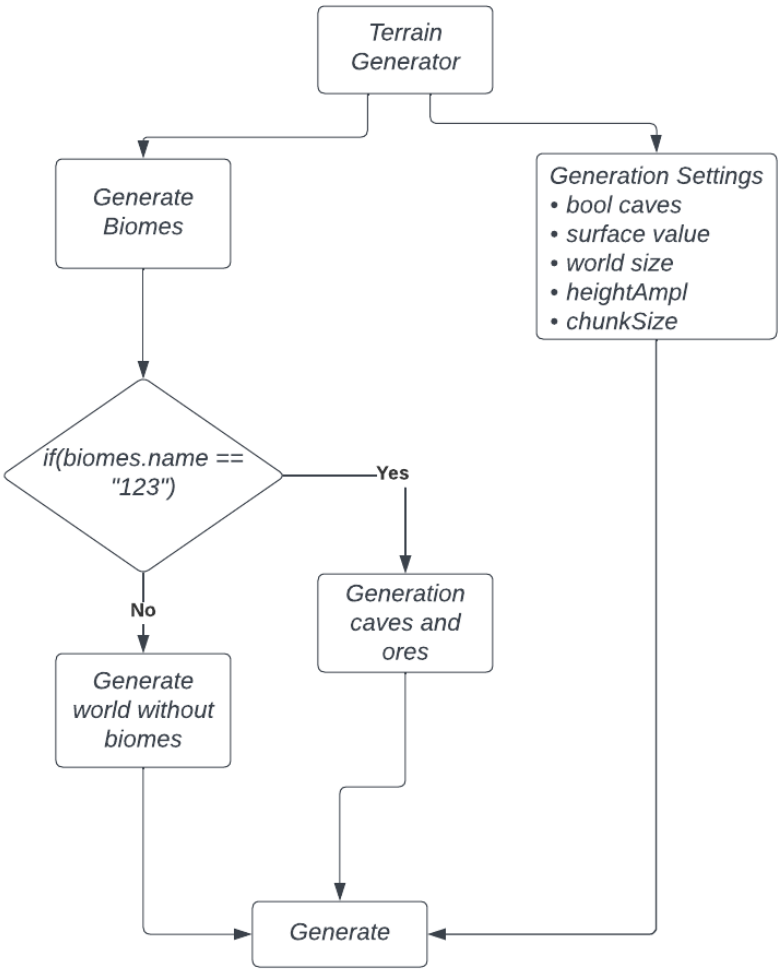

Рисунок 2.2 – Схема генерации ландшафта

Блоки в мире хранятся в массиве *GameObjects* . Блоки разделяются на те, который будут на заднем фоне и на те, которые будут на переднем. Также необходим метод, который будет проверять, находится ли плитка на заднем фоне, или на переднем.

Следующим методом, является обработка текстур и генерации отдельных биомов путем добавления текстур биомам, создании экземпляра текстуры, с полями ширины мира и дальнейшего добавления самих текстур.

Биомы созданы путем распознавания цветов градиента, который можно указать так, как выберет пользователь, что гораздо упрощает добавления новых

биомов и для корректной настройки всех данных. Было принято решение добавить переменную частоты появления биомов и дальнейшего изменения текстуры. Благодаря этой переменной, происходит наложение текстуры биома, на текстуру генерации мира и добавления методов, которые реализуют полученную текстуру. На рисунке 2.3 показана схема, по которой взаимодействуют текстуры в игре.

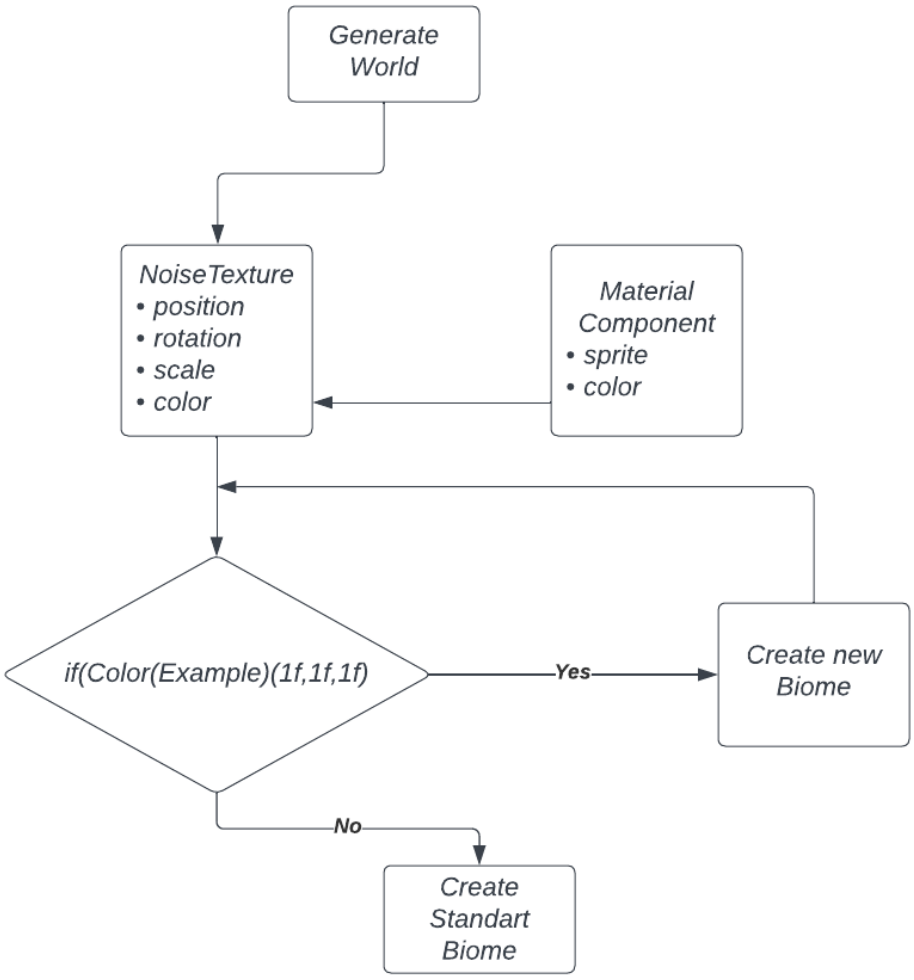

Рисунок 2.3 – Схема взаимодействия текстур в игре

Каждый биом, содержит в себе все блоки, находящиеся в нем. Для корректного распознавания биомов было принято решение использовать следующие цвета:

– белый – цвет, отвечающий за генерацию снежного биома со своими блоками и растительностью;

– зеленый – цвет, который относится к биому равнины, у которого в два раза повышен шанс на генерацию травы;

– темно-зеленый – лесной биом, в котором высота деревьев и шанс на появления деревьев выше, по сравнению с другими биомами;

– коричневый – цвет, отвечающий за биом пустыни. В этом биоме деревья заменены на кактусы, который так же, как и деревья имею шанс на появление. Трава заменена на мертвую. Шанс на появление руд увеличен.

Сама сцена содержит компоненты, которые необходимы, для реализации игры, а именно:

– *terrain*;

– персонаж, со своими скриптами и пользовательскими настройками;

– пользовательский интерфейс;

– меню инвентаря.

На рисунке 2.4 продемонстрирован макет иерархии сцены.

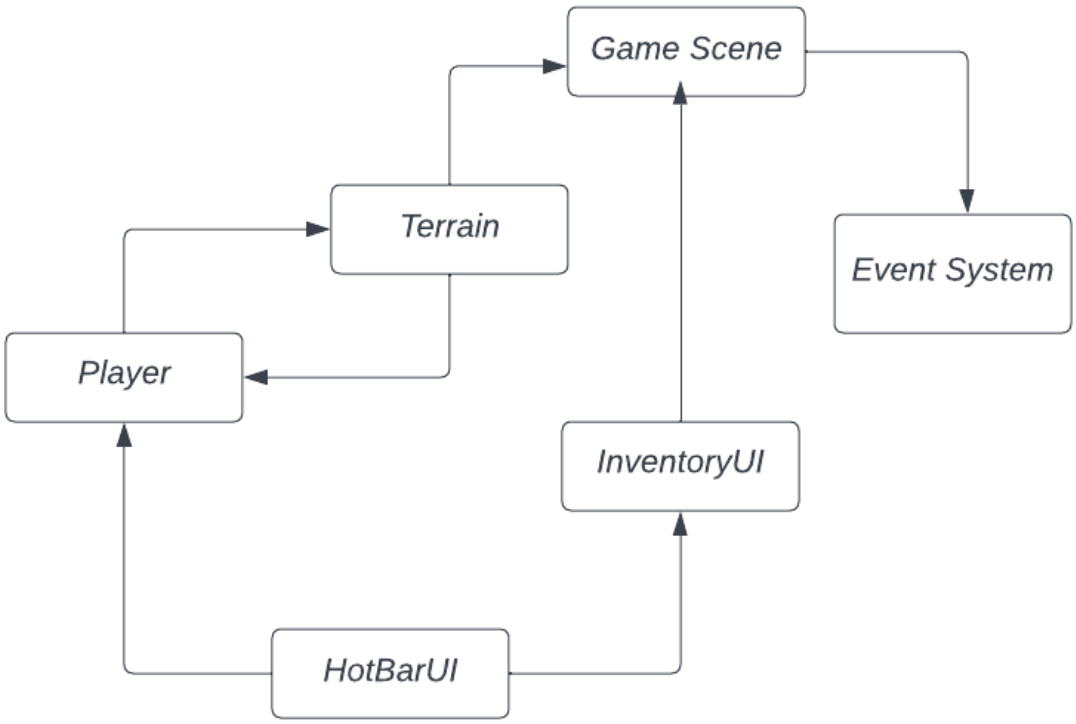

Рисунок 2.4 – Макет иерархии сцены

Пользовательский интерфейс состоит из трех компонентов. Первым компонентам является панель, которая является самой главной частью инвентаря, на котором будут находится слоты, для блоков и различных предметов. Второй компонент – это картинка, которая использует текстуру блоков, которые выпадают при их разрушении. Третий компонент также содержит панельки, которые отвечают за слоты в инвентаре, который при помощи кода должны быть размножены.

За графическую часть отвечают спрайты, который были нарисованы при помощи приложений *Paint* и *Aseprite,* одни из самых лучших 2*D* редакторов для создания спрайтов и текстур.

Наложение текстуры на *GameObject* происходит путем взятия компонента объекты *spriteRenderer* поля *sprite* и дальнейшего сравнения с переменной *Texture2D* которая содержит текстуру объекта. На рисунке 2.5 схематично показаны все свойства отдельного спрайта, наложенного на объект.

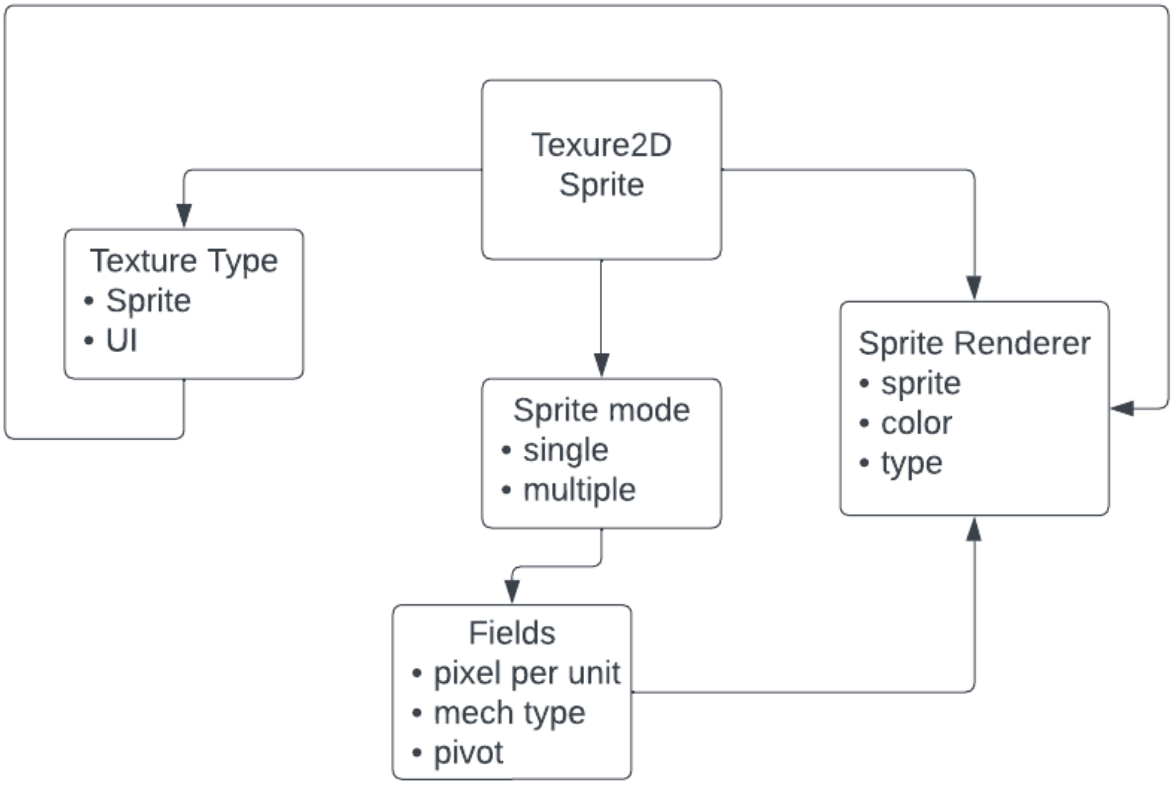

Рисунок 2.5 – Свойства спрайта

Игровые объекты, находящиеся на сцене, имеют свои коллайдеры и физические свойства, для взаимодействия на сцене с другими объектами. К примеру, игрок не сможет пройти сквозь блок, так как у него настроены компоненты *Rigidbody*2*D* и *Capsule Collider*2*D*.

После случайной генерации мира, происходит генерация биомов, которая генерируется путем считывания цветов с градиента.

Collider описывает взаимодействие объекта с окружающим миром. При разрушении объекта, его коллайдер отключается, тогда объект помечен как уничтоженный. *RigidBody2D* используется чтобы сделать объект физическим и дать ему соответствующие свойства. *Capsule Collider*2*D* содержит компоненты, который реализуют *collider,* размер его, поля для физических материалов, координат,

которые отвечают за смещение коллайдеры объекта. Связь игрового объекта и компонента происходит, как на рисунке 2.6.

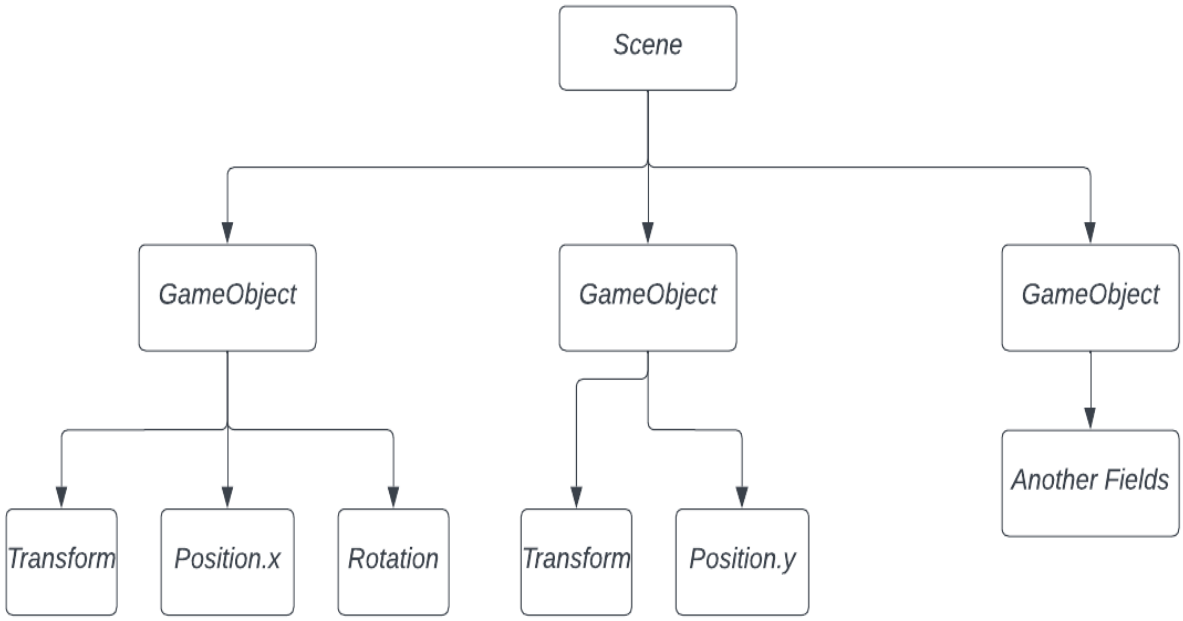

Рисунок 2.6 – Связь игрового объекта и компонента

Игрок взаимодействует с игровым миром с помощью интерфейса и главного героя. При взаимодействии с игровым миром, происходит два действия. Левая кнопка мыши, используется для разрушения объекта, в случае чего, выпадет маленькая моделька блока. Правая кнопка мыши, используется для добавления плитки в игровой мир. Можно установить объект на игровую сцену, который запишется в массив объектов, у которого все те же самые свойства, что и у обыкновенных объектов. У каждого блока существует поле, которое отвечает за то, может ли объект быть разрушаемым, или нет, в противном случае объект разрушить нельзя будет. Еще одно поле будет отвечать за то, может ли у объекта, который был разрушен, выпасть моделька его блока.

Еще одной главной задачей, является разбить все блоки на те, которые стоят на переднем фоне и те, которые стоят сзади. Отличие в том, что блоки, которые находятся позади, имеют оттенок в два раза темнее, обычного блока, что бы игрок не смог спутать объекты. Блоки на переднем плане, имеют все физические свойства и могу взаимодействовать с персонажем. Плитки на заднем фоне, никак не взаимодействуют и будут дополнять общую картину генерации ландшафта.

Главной механикой, является создание инвентаря персонажа, который будет открываться на определенную пользователем клавишу. Инвентарь представляет собой тот же самый пользовательский интерфейс, который содержит все те же самые компоненты. Отличием является надпись посреди экрана и слоты, которых будут в четыре раза больше, чем в пользовательском интерфейсе. У инвентаря содержаться несколько методов, реализующие добавление, проверку и удаление предметов из него. При помощи цикла, можно проверить каждый объект, его индекс, спрайт, название и сравнить с предметов в инвентаре, в случае успешного нахождения, инвентарь обновляется.

Каждый объект имеет представление анимации. Анимации всех объектов основаны на изменении позиции той или иной части тела персонажа. Частями являются отдельные *.png* изображения. С помощью скрипта перемещения персонажа задаётся движение частей тела, в результате чего создается анимация. Это относится только на движение рук, ног и разрушения объектов, или добавления их на сцену. Остальные анимации представляют собой набор изображений, размещённых и сменяющих друг друга согласно временной шкале. Пример аниматора персонажа представлен на рисунке 2.7.

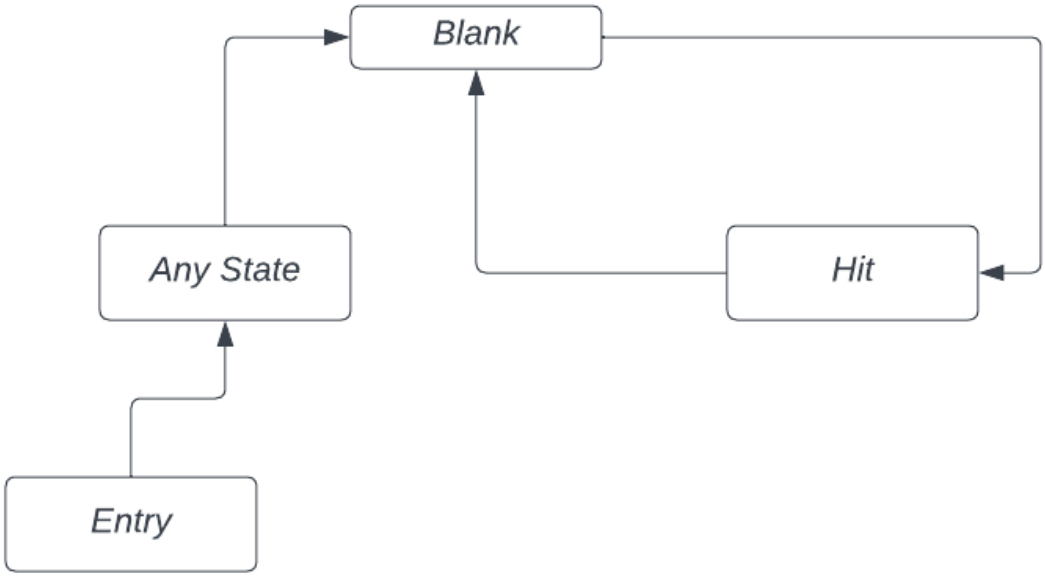

Рисунок 2.7 – Пример аниматора персонажа

Последней задачей необходимо представить, как реализовать автопрыжок. Это сделано при помощи *Raycasts*. Один луч находится в зоне ног персонажа, который проверяет, есть ли перед ним твердотельный объект, в случае чего его таков имеется, то производится автопрыжок. Второй находится возле головы персонажа, который проверяет обратную сторону, если возле головы персонажа

ничего не находится, то прыжок совершать не нужно. На рисунке 2.8 показан пример того, как при помощи лучей, можно проверить находится ли объект на земле, или нет.

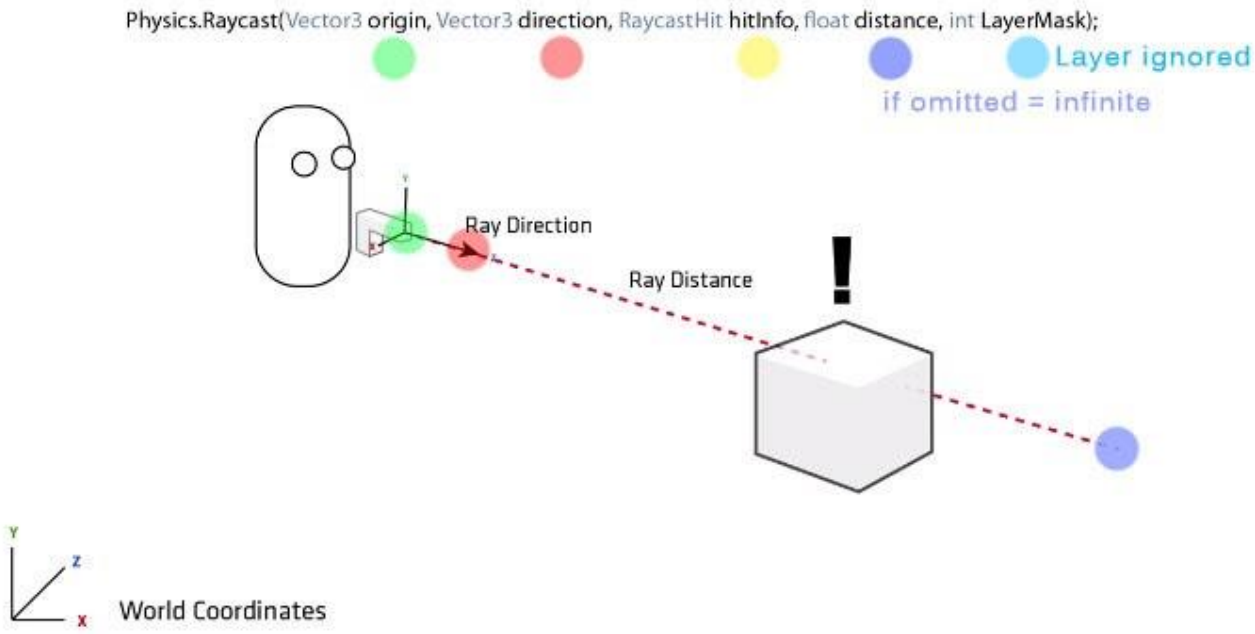

Рисунок 2.8 – Пример использования лучей

Если собрать все вместе, то можно сделать выводы, о том, что игра получилась довольно сложной, со своими уникальными механиками, которые реализованы в скриптах. В скриптах содержаться поля и методы, которые описывают тот или иной процесс в создании игры.

2.3 Архитектурная модель данных игрового приложения

Приложение включает в себя четыре блока с пользовательским интерфейсом:

– главное меню приложения;
– главная игровая сцена приложения;
– сцена, на которой происходит обучение;
– инвентарь.

Главное меню приложения включает в себя кнопку старта игры, которая должна перенести пользователя на другую сцену. Кнопку, переходя по которой, появляется новая сцен, на которой объясняется как пользователю играть. Последней кнопкой является кнопка, выхода из игры, в случае чего, если игрок захочет выключить приложение. Непосредственно на игровой сцене, есть также

сцена остановки игрового приложения, которая содержит кнопку «продолжить игру», кнопку «настроек» и кнопку «вернуться в главное меню», для дальнейшего выхода из игровой сцены. На рисунке 2.9 представлена обобщенная схема пользовательского интерфейса.

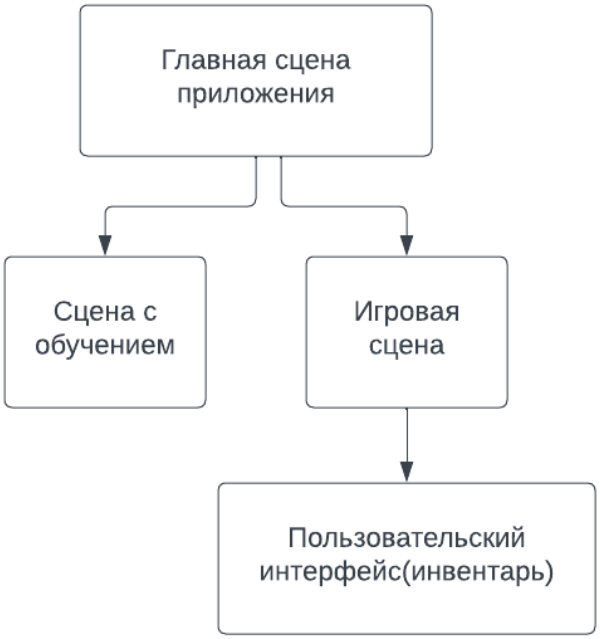

Рисунок 2.9 – Схема пользовательского интерфейса

Следующим этапом игрового приложения является наличие пользовательского интерфейса конкретно на игровой сцене, которая содержит в себе панель быстрого доступа.

В свою очередь панель быстрого доступа имеет следующие компоненты:

– слоты быстрого доступа, которые являются первыми слотами инвентаря и полностью идентичны;

– переключатель, который при прокручивании колеса мышки вниз и вверх меняет положение на слоте по оси *X*;

– поле *image*, которая в свою очередь подхватывает спрайт предмета или блока, который находится в мире или который можно добыть при помощи инструмента;

Компоненты игровой сцены, а именно логика взаимодействия пользователя с интерфейсом, способствуют управлять игровым процессом. Вся архитектура приложения представляет собой набор сцен, методов и классов, которые способствуют переключению между сценами и взаимодействование с окружающим миром, при этом сцены сохраняют свою уникальность. Сцены разбиты на несколько частей, которые дополняют друг друга.

3 СТРУКТУРА ИГРОВОГО ПРИЛОЖЕНИЯ «*TERARIZATOR*»

3.1 Иерархия игрового приложения «*Terarizator*»

Приложение разработано при помощи среды разработки *Unity*. Игровое приложение содержит три главных части, без которых оно не могло бы функционировать. Графическая часть представляет собой спрайты, нарисованные при помощи *Paint* и *Aceprite*. Игровая механика, включает в себя методы и классы, которые позволяют взаимодействовать с игровым миром. Логическая часть, позволяет объектам иметь все необходимые свойства, для реализации физической части составляющей игры.

Если говорить о взаимосвязи классов игры, то можно выделить классы, которые относятся к созданию игрового мира. Другие классы относятся к логической составляющей персонажа и все что с ним связано. Последняя группа классов, содержит в себе информацию о том, что представляет собой инвентарь. Пользователю, который захочет поиграть в игру, виден графический интерфейс, где происходят главные события в игре. Информация, которую пользователь получает, при помощи *Package Manager*, используется для пользовательского ввода и вывода, благодаря которому, он взаимодействует с игровым процессом. Вся логика, содержится в скриптах, благодаря чему, пользователю не нужно вводить никакие данные и как-либо взаимодействовать с кодом программы. На рисунке 3.1 продемонстрирована иерархия поведения событий игрового приложения.

Рисунок 3.1 – Иерархия событий игрового приложения

Помимо графической части приложения и ее логики, немаловажной частью является построение анимации при помощи временной шкалы. Аниматор игрока содержит некоторые переменные, которые позволяют связывать некоторые анимации логически, при помощи булевой операции. Происходят большое количество проверок, к примеру, если игрок пойдет влево, то анимация поменяется, а потом встанет в первоначальное положение.

3.2 Свойства объектов и их описание

Весь проект разбит на несколько важных компонентов, благодаря которым происходит дальнейший сбор проекта в одно целое. Эти части называются ресурсами проекта. Ресурсы представляют собой папки, находящиеся в проекте, а также все необходимые инструменты, для реализации тех или иных действий.

Существуют следующие типы ресурсов:

– папка, которая содержит всю необходимую информацию о ресурсах игры, а именно картинки и *tiles*;

– *Scenes*, представляет собой файлы сцен, на которых расположены все объекты, хранящиеся в папке *Sprites*;

– *Player*, ресурс, содержащий информацию о персонаже, физический материал персонажа, который в следующим понадобился для совершения анимации. Классы, описывающие передвижение игрока по земле и взаимодействия с окружающим миром;

– *Tiles*, папка, которая хранит в себе класс объектов, а именно сами объекты в виде классов. У каждого объекта присутствуют поля, которые относятся к конкретному блоку. Также хранит в себе текстуры и различные проверки, к примеру, относится ли блок к заднему фону;

– *Tools*, хранит в себе информацию о всех доступных инструментах в игре, которыми персонаж в праве управлять. Содержит в себе поля, которые передают спрайты, тип предмета и название;

– *Scripts*, содержит всю необходимую информацию о скриптах, сами скрипты, которую в свою очередь выполняют те или иные действия с объектами на сцене.

Главным является ресурс типа *Scene*, на которой происходит отрисовка объектов и генерация мира. Сцена представляет собой главный и единственный уровень, на котором генерируется случайный ландшафт, персонаж, пользовательский интерфейс, который взаимодействует с персонажем. Также в ресурсе *Scene* находятся все необходимые компоненты, для реализации меню. На рисунке 3.2 продемонстрированы все ресурсы, для разработки приложения.

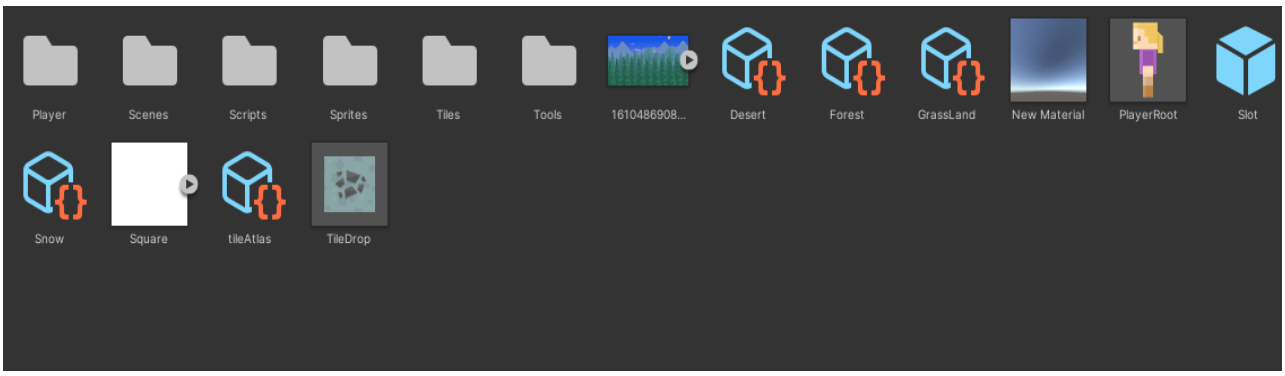

Рисунок 3.2 – Ресурсы игрового приложения

Спрайты и все что связано с графикой делают из приложения игровое. Все что связано с изображениями, хранятся в папке *Sprites*, а затем накладываются на игровые объекты. Все объекты имеют пиксельное разрешение 128 на 128 пикселей, что в *Unity* эквивалентно одному блоку.

Один из самых главных ресурсов являются *Scripts*. Они отвечают за все логическое поведение объектов на сцене.

Классы, реализуют скрипты, а именно:

– скрипт, реализующий процедурную генерацию ландшафта;

– скрипт, реализующий передвижение игрока, прыжок, взаимодействие с миром;

– скрипт, который реализует класс предмета;

– скрипт, содержащий информацию о конкретных биомах;

– скрипт, который отвечает за управление камерой, которая привязана к персонажу;

– скрипт, реализующий класс руды, генерируемой в мире;

– скрипт предметов, который содержит типы и свойственные поля;

– скрипт, благодаря которому происходит выпадение блока, в блоки с другим цветом и размером.

Каждый скрипт имеет свои уникальные поля или наследует их из других скриптов, благодаря чему, происходит вся логика взаимодействия. К примеру, происходит проверка, на заднем фоне находится ли предмет, или нет. Проверка на соприкосновение объектов с друг другом, распространяется это на все объекты. Все логические и физические свойства предметов, к примеру гравитация или изменение положения персонажа.

3.3 Классы разработки и их свойства

3.3.1 Класс передвижения персонажа *PlayerController* (Приложение А, листинг класса *PlayerController.cs*). Класс представляет скрипт, который отвечает за управление персонажем, содержит информацию о том, где появится игрок, его

начальные координаты. Позволяет игроку взаимодействовать с инвентарем и пользовательским интерфейсом, выбирать нужный предмет и ставить его на игровое поле. А также механику автопрыжка, реализованную при помощи лучей. Один луч проверяет соприкосновение с ногами персонажа, второй с головой, в случае чего, если персонаж соприкасается ногами с блоком, то происходит автопрыжок. В скрипте реализуется механика разрушение и добавления блоков в игровой мир. Все поля класса *PlayerController* (Приложение А, листинг класса *PlayerController.cs*) приведены в таблице 3.1.

Таблица 3.1 – Поля класса *PlayerController*

Название	Атрибут	Модификатор доступа	Тип
layerMask	–	*public*	*LayerMask*
selectedSlotIndex	–	*public*	*int*
hotbarSelector	–	*public*	*GameObject*
handHolder	–	*public*	*GameObject*
inventory	–	*public*	*Inventory*
inventoryShowing	–	*public*	*bool*
selectedItem	–	*public*	*ItemClass*
playerRange	–	*public*	*int*
moveSpeed	–	*public*	*float*
mousePos	–	*public*	*Vector2Int*
jumpForce	–	*public*	*float*
onGround	–	*public*	*bool*
rb	–	*public*	*Rigidbody2D*
anim	–	*private*	*Animator*
horizontal	–	*private*	*float*
hit	–	*public*	*bool*
place	–	*public*	*bool*
spawnPos	*HideInInspector*	*public*	*Vector2*
terrainGenerator	*HideInInspector*	*public*	*TerrainGenerator*

Класс также реализует следующие методы:

– *Spawn* (). Метод, который проверяет начальное положение персонажа, его координаты, принимает компоненты анимации, физики и реализует пользовательский интерфейс инвентаря;

– метод *FixedUpdate*. Базовый метод, который наследуется от базового класса *Unity*, под названием *MonoBehaviour*. Метод вызывается с фиксированной частотой обновления, которую можно настраивать в настройках. Если изображение обновляется с другой частотой, то приоритет идет на частоту обновления метода. Используется для обновления физики, связанной с игрой. К примеру, прыжок, совершенный с определенной силой, передвижение игроку по горизонтали, совершение автопрыжка;

– метод *Update*. В отличии от *FixedUpdate*, он срабатывает один раз за один кадр. Является главной функцией обновления кадров в *Unity*. Метод зависит от частоты обновления изображения на экране, что позволяет устройству адаптироваться под игру. В данном методе происходит разрушение блоков, взаимодействие со строительством плиток, для того, чтоб держать предметы в руках, используется игровой объект. Установка позиции персонажа и камеры.

– два метода, реализующие лучи у персонажа, для проверки автопрыжка. Методы *FootRaycast* и *HeadRaycast*.

3.3.2 *CameraController* (Приложение А, листинг класса *CameraController.cs*) является скриптом, который отвечает за управление камерой. Камера следует за персонажем и считывает данные о том, где находится персонаж каждый кадр. Класс имеет поля ограничивающие передвижение камеры в пределах координат положения персонажей. Поля класса *CameraController* (Приложение А, листинг класса *CameraController.cs*) продемонстрированы в таблице 3.2.

Таблица 3.2 – Поля класса *CameraController*

Название	Атрибут	Модификатор доступа	Тип
smoothTime	*Range* (0,1)	*public*	*float*
playerTransform	–	*public*	*Transform*
spawnPos	–	*public*	*Vector2*
worldSize	*HideInInspector*	*public*	*int*
orthoSize	–	*private*	*float*

Класс камеры реализует методы:

– *Spawn* (). Данный метод, позволяет камере принимать начальное положение на игровое сцене так, как и у игрока. Принимает размер камеры, по умолчанию ортографический режим;

– метод *FixedUpdate*. Как и у игрока, метод позволяет считывать положение камеры с определенной задержкой, а также при помощи переменной *smoothTime* настраивать камеру. К примеру, если игрок будет передвигаться, камеру будет двигаться за персонажем со скоростью персонажа, но останавливаться камера после передвижения будет с определенной задержкой. Метод

задает начальные координаты камеры *position.x* и *position.y* и позволяет при помощи функции *GetComponent*<> считывается положение игрока.

3.3.3 Главный класс, благодаря которому происходит генерация мира *TerrainGenerator* (Приложение А, листинг класса *TerrainGenerator.cs*). Для генерации мира необходимо было сгенерировать текстуру, которая под воздействием переменной *surValue* изменяется. На текстуре появляются пустые места, которые представляют собой пещеры в игровом мире. Также существует проверка на то, пещеры сгенерировались, или нет. В настройках *Inspector* можно указать размер создаваемого мира и высоту земли в мире. Класс хранит в себе все объекты на игровой сцене, которые на заднем фоне и на переднем. Объекты на заднем фоне, не имеют компонентов *Rigidbody2D* и отключается коллайдер, при создании мира. Они имеют цвет, который в два раза темнее, по сравнению с объектами на переднем фоне.

В мире присутствуют биомы, которые генерируются по похожей аналогии, как и игровой мир. На текстуру накладывается шум, благодаря чему текстура разбивается на четыре части. Каждая часть имеет свой цвет градиента. Цвет градиента сравнивается с переменной биома, что позволяет распознавать биомы.

Данный класс реализует добавление и разрушение плитки в игровой мир, путем чтения массива *GameObjects*[] и дальнейшего присвоения элемента массива.

Мир разделяется на несколько фрагментов, которые являются *Chunks*, создано это для того, чтоб высвобождать неиспользуемые ресурсы и для оптимизации игры. В таблице 3.3 приведены поля класса *TerrainGenerator* (Приложение А, листинг класса *TerrainGenerator.cs*).

Таблица 3.3 – Поля класса *TerrainGenerator*

Название	Атрибут	Модификатор доступа	Тип
player	–	*public*	*PlayerController*
camera	–	*public*	*CameraController*
tileDrop	–	*public*	*GameObject*
seed	*Header* (“*Tile Sprites*”)	*public*	*float*
tileAtlas	*Header* (“*Tile Sprites*”)	*public*	*TileAtlas*
biomes	*Header* (“*Tile Sprites*”)	*public*	*BiomeClass*[]
biomeFreq	*Header* (“*Biomes*”)	*public*	*flaot*
biomeGradient	*Header* (“*Biomes*”)	*public*	*Gradient*
biomeMap	*Header* (“*Biomes*”)	*public*	*Texture2D*
caves	*Header* (“*Generation Setting*”)	*public*	*bool*
surValue	*Header* (“*Generation Setting*”)	*public*	*float*

Продолжение таблицы 3.3

1	2	3	4
worldSize	*Header* (“*Generation Setting*”)	*public*	int
heightA	*Header* (“*Generation Setting*”)	*public*	*float*
chunkSize	*Header* (“*Generation Setting*”)	*public*	*int*
caveNoiseTexture	*Header* (“*Other Settings*”)	*public*	*Texture2D*
ores	*Header* (“*Ore Settings*”)	*public*	*Ores* []
curBiome	–	*private*	*BiomeClass*
worldChunks	–	*private*	*Gameobject*[]
world_ForegroundObjects	–	*private*	*Gameobject*[,]
world_BackgroundObjects	–	*private*	*Gameobject*[,]
world_BackgroundTiles	–	*private*	*Tile* [,]
world_ForegroundTiles	–	*private*	*Tile* [,]

Класс реализует следующие методы:

– метод *Start* (). В методе присваиваются массивам объектов значения размера мира *worldSize*. Проверяется, содержит ли мир ископаемые или нет. Начальная позиция камеры и появление персонажа также описываются в данном методе;

– метод *Update*. Метод обновляет фрагменты мира каждый кадр;

– *RefreshChunks*. Данный метод реализует расчет фрагментов мира и в случае чего, если мир может содержать фрагменты становятся активными, если не может, неактивными;

– метод *DrawCavesAndOres* содержит текстуру, которая потом генерирует пещеры и руды. В случае чего, если на текстуре появляется белый цвет, генерируется пещера, если черный, то генерируется обыкновенный камень;

– метод *DrawTexture* генерирует сами биомы;

– метод *CreateChunks* создает фрагменты мира;

– метод *GetCurrentBiome* проверяет значение цвета и сравнивает их со значениями градиента биома;

– главный метод *TerrainGenerate* генерирует сам мир. Для конкретного биома происходит собственная генерация благодаря значению *curBiome*. Происходит проверка, на каком уровне находится блоки земли, можно ли создавать деревья и траву, если над первым блоком находится пустой объект;

– метод *GenerateNoiseTexture* создает текстуру генерации мира. При успешной генерации мира происходит генерация руд и шахт. В случае чего, если на текстуре присутствует черный цвет, является камнем, если нет, то пещерой;

– методы *CactusGenerate* и *TreeGenerate* похожи друг на друга, отличаются только тем, что деревья могут появится во всех, кроме пустыни, а кактусы только в пустыни. В методе вызывается еще один метод *TilePlace* благодаря которому можно генерировать отдельные части мира;

– *BreakTile* метод, который позволяет разрушать предмет в том случае, если название инструмента соответствует блоку, который можно разрушить им;

– *RemoveTile* метод удаления плитки из массива *GameObjects* и проверка на то, может ли выпасть блок, который разрушил игрок;

– метод *CheckTile* проверяет находится ли плитка на заднем фоне в массиве объектов *world_BackgroundTiles* или нет, в противном случае, возвращает значение false;

– *TilePlace* метод, генерирующий блоки, которые не созданы при помощи генерации мира, а также добавляет эти блоки в массивы объектов;

– методы *AddTileToWolrld* и *AddObjectToWorld* проверяют, если объект или плитка, которая генерируется миром является на заднем фоне, в таком случае присваивается значение *tileObject* или *tile*. Те же самые действия происходят и с твердотельными объектами или плитками;

– методы *RemoveTileFromWorld* и *RemoveObjectFromWorld* присваивают значение удаленного объекта на игровой сцене *null*;

– методы *GetTileFromWorld* и *GetObjectFormWorld* читают массив объектов и возвращают индекс объекта или плитки.

3.3.4 Классы *Tile* (Приложение А, листинг класса *Tile.cs*), *Ores* (Приложение А, листинг класса *Ores.cs*) и *ToolClass* (Приложение А, листинг класса *ToolClass.cs)* имеют схожую структуру, но отличаются полями. Они являются классами объектов, при помощи которых, в обозревателе *Unity* можно создать их, установив значения, которые по умолчанию являются *none*. В таблице 3.4 содержаться все поля классов *Tile* (Приложение А, листинг класса *Tile.cs*), *Ores* (Приложение А, листинг класса *Ores.cs*) и *ToolClass* (Приложение А, листинг класса *ToolClass.cs).*

Таблица 3.4 – Поля классов *Tile, Ores* и *ToolClass*

Название	Атрибут	Модификатор доступа	Тип
tileName	–	*public*	*string*
wallVariant	–	*public*	*Tile*
tileSprites	–	*public*	*Sprite*[]
inBackground	–	*public*	*bool*
narturallyPlaced	–	*public*	*bool*
isStackable	–	*public*	*bool*

Продолжение таблицы 3.4

1	2	3	4
tileDrop	–	*public*	Tile
toolToBreak	–	*public*	*ItemClass.ToolType*
name	*Range* (0,1)	*public*	*string*
rariry	*Range* (0,1)	*public*	*float*
size	–	*public*	*float*
maxSpawnHeight	–	*public*	*float*
spreadTexture	–	*public*	*Texture2D*
nameTool	–	*public*	*string*
sprite	–	*public*	*Sprite*
toolType	–	*public*	*ItemClass.ToolType*

Класс *Tile* (Приложение А, листинг класса *Tile.cs*) содержит два метода, один из которых *CreateInstance*, который проверяет блок, который был создан при помощи правой кнопки мыши и метод *Init* присваивающий переменным класса тип *Tile*.

3.3.5 Классы *BiomeClass* (Приложение А, листинг класса *BiomeClass.cs*), *ItemClass* (Приложение А, листинг класса *ItemClass.cs*) и *TileAtlas* (Приложение А, листинг класса *TileAtlas.cs)* также схожи между собой и реализовывают классы, которые хранят значения биомов, предметов и блоков, содержащихся в конкретном биоме соответственно. В таблице 3.5 представлены поля классов *BiomeClass* (Приложение А, листинг класса *BiomeClass.cs*), *ItemClass* (Приложение А, листинг класса *ItemClass.cs*) и *TileAtlas* (Приложение А, листинг класса *TileAtlas.cs).*

Таблица 3.5 – Поля классов *BiomeClass*, *ItemClass* и *TileAtlas*

Название	Атрибут	Модификатор доступа	Тип
biomeName	–	*public*	*string*
biomeCol	–	*public*	*Color*
tileAtlas	–	*public*	*TileAtlas*
noise	*Header* (“*Other Settings*”)	*public*	*float*
caveNoiseTexture	*Header* (“*Other Settings*”)	*public*	*Texture2D*
terrainFreq	*Header* (“*Other Settings*”)	*public*	*float*
caves	*Header* (“*Generation Setting*”)	*public*	*bool*
dirtHeight	*Header* (“*Generation Setting*”)	*public*	*int*
surValue	*Header* (“*Generation Setting*”)	*public*	*float*
HeightM	*Header* (“*Generation Setting*”)	*public*	*float*
tree	*Header* (“*Trees*”)	*public*	*int*

Продолжение таблицы 3.5

1	2	3	4
minTreeHeight	*Header* (“*Trees*”)	*public*	*int*
maxTreeHeight	*Header* (“*Trees*”)	*public*	*int*
grassChance	*Header* (“*Other*”)	*public*	*int*
ores	*Header* (“*Ores*”)	*public*	*Ores*[]
itemType	–	*public*	*ItemType*
toolType	–	*public*	*ToolType*
tile	–	*public*	*Tile*
tool	–	*public*	*ToolClass*
itemName	–	*public*	*string*
sprite	–	*public*	*Sprite*
isStackable	–	*public*	*bool*
grass	–	*public*	*Tile*
dirt	–	*public*	*Tile*
stone	–	*public*	*Tile*
log	–	*public*	*Tile*
leaf	–	*public*	*Tile*
snow	–	*public*	*Tile*
sand	–	*public*	*Tile*
bedrock	–	*public*	*Tile*
coal	–	*public*	*Tile*
iron	–	*public*	*Tile*
gold	–	*public*	*Tile*
diamond	–	*public*	*Tile*
grass2	–	*public*	*Tile*

Класс *ItemClass* (Приложение А, листинг класса *ItemClass.cs*), содержит два метода, которые позволяют присваивать тип переменных классов *Tile* (Приложение А, листинг класса *Tile.cs)* и *ToolClass* (Приложение А, листинг класса *ToolClass.cs*). Остальные классы содержат поля, которые используются для создания класса объектов.

3.3.6 Класс *Inventory* (Приложение А, листинг класса *Inventory.cs*) и *InventorySlot* (Приложение А, листинг класса *InventorySlot.cs*) содержат в себе информацию о пользовательском интерфейса, а именно инвентаре. Существует два типа инвентаря, быстрого доступа и пользовательский инвентарь. У них схожие механики, за исключением того, что в инвентаре быстрого доступа хранятся значения первых 9 ячеек, а в пользовательском 36 ячеек. Поля классов *Inventory* (Приложение А, листинг класса *Inventory.cs*) и *InventorySlot* (Приложение А, листинг класса *InventorySlot.cs*) показаны в таблице 3.6.

Таблица 3.6 – Поля классов *Inventory* и *InventorySlot*

Название	Атрибут	Модификатор доступа	Тип
stackLimit	–	*Public*	*int*
start_Pickaxe	–	*Public*	*ToolClass*
start_Axe	–	*Public*	*ToolClass*
start_Hammer	–	*Public*	*ToolClass*
inventoryOffset	–	*Public*	*Vector2*
hotbarOffset	–	*Public*	*Vector2*
multiplier	–	*Public*	*Vector2*
inventoryUI	–	*Public*	*GameObject*
hotbarUI	–	*Public*	*GameObject*
inventorySlot	–	*Public*	*GameObject*
inventoryW	–	*Public*	*int*
inventoryH	–	*Public*	*int*
inventorySlots	–	*Public*	*InventorySlot*[,]
hotbarSlots	–	*Public*	*InventorySlot*[]
hotbarUISlots	–	*Public*	*GameObject*[]
uiSlots	–	*Public*	*GameObject*[,]
position	–	*Public*	*Vector2Int*
quantity	–	*Public*	*int*
item	–	*Public*	*ItemClass*

Класс *Inventory* (Приложение А, листинг класса *Inventory.cs*) содержит методы:

– метод *Start*. Данный метод позволяет задать ширину и высоту инвентаря и слотам инвентаря. А также обновлять инвентарь, когда это понадобится. Добавлять различные предметы в инвентарь;

– метод *SetupUI*. Метод, который описывает как инвентарь будет выглядеть в игре и на игровой сцене. Позволяет расширить интервал между слотами;

– метод *UpdateInventoryUI*. Реализует обновление инвентаря, после действий добавления и удаления предмета из инвентаря. Содержит в себе информацию о слотах, картинки и после обновляет их.

– *Add* метод, содержащий в себе алгоритм добавления предмета в инвентарь, путем сравнения позиции слота в инвентаре, название предмета, с названием предмета, созданного при помощи класса *Tile* (Приложение А, листинг класса *Tile.cs*);

– *Contains* метод, который проверяет при каждом обновлении инвентаря, находится ли предмет в инвентаре;

– *Remove* метод, реализует удаление объекта из инвентаря, в случае чего, если в инвентаре предмета больше, чем один.

4 ВЕРИФИКАЦИЯ И ВАЛИДАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ «TERARIZATOR»

4.1 Разновидности валидаций игровых приложений

Финальное игровое приложение было протестировано несколько раз с помощью различных методов тестирования.

Всего существует четыре вида тестирования игрового приложения:

– нагрузочное тестирование или тестирование производительности. Задачей тестирования производительности является определение масштабируемости приложения под нагрузкой, при этом нужно учесть несколько факторов. Первым является определение количества пользователей, одновременно работающих с приложением. При нагружении производительности операционной системы, проверятся работоспособность игрового приложения. Второй фактор – исследование производительности на разных уровнях нагрузки, расчет затрат времени на процессы при нагрузках. Тестировщик проверяет производительность системы в нагруженном состоянии или на слабых устройствах. К примеру, может быть такое, что разработчик оставил какие-либо данные перед загрузкой игры и не высвободил их вовремя. При запуске игры на слабом компьютере, игра начинает зависать. В результате чего данными тестами были проверены слабые места программного кода;

– конфигурационное тестирование – вид тестирования, который помогает проверить работу программного обеспечения при различных конфигурациях системы. У конфигурационного тестирования существуют две цели. Профилирование системы – определение оптимальной сборки конфигурации оборудования, обеспечивающие требуемые характеристики. Миграция системы с одного проекта в другой – проверка на то, как будут вести себя процессы, при совместимости с объявленной конфигурацией. На серверном уровне тестирования тестируется приложение с окружением, в котором оно будет загружено. На клиентском уровне проводится для того, чтоб понять, какое программное обеспечение максимально подходит для конкретного игрового приложения;

– функциональное тестирование – оно рассматривает поведение оборудование и основывается на спецификациях функциональности отдельных компонентов. Тесты основываются на функциях, которые обрабатывает система, а также могут проводится на разных уровнях тестирования приложения. Если рассматривать со стороны требований данных от приложения, то тесты используют спецификацию функциональных требований к системе. В этом случае нужно сделать список подзадач, которые необходимы для тестирования. Это позволяет не упускать при тестировании главный функционал;

– тестирование удобства пользователю, при использовании приложения.

Данное тестирование позволяет произвести оценку удобства продукта в использовании. Пользователи привлекаются в виду тестировщиков далее суммируются данные, которые были получены и делаются выводы о том, как лучше поменять игровое приложение для получения максимального удовольствия.

4.2 Верификация игрового приложения

Необходимо было создать игровое приложение, в которым можно управлять персонажем от третьего лица в мире, где нет какого-либо сюжета и хода игры. Игрок передвигается по миру, а для того, чтобы попасть в этот мир, необходимо в главном меню нажать кнопку «*NEW GAME*». На рисунке 4.1 продемонстрировано главное меню игрового приложения.

Рисунок 4.1 – Главное меню приложения

Для функционирования с программой, необходимо открыть приложение, которое находится в папке *Debug* проекта. После запуска игры, можно приступить к апробации игрового приложения.

После запуска игрового приложения, игроку дается выбор, на какую клавишу необходимо нажать. Кнопка «*Controls*» перекинет игрока в меню обучения, в котором игроку наглядно будет продемонстрировано, как передвигаться в игре, ломать блоки, ставить и совершать прыжок.

Игрок может передвигаться по игровому миру, прыгать по игровому ландшафту, строить и разрушать некоторые декоративные блоки. В самого мира, а именно самый последний нижний блок, игрок не сможет сломать, чтобы не

выпасть из игрового мира. Для перемещения игрока необходимо использовать клавиши:

– клавиша «*W*» – для совершения прыжка персонажа;

– клавиша «A» – позволяет персонажу пройти некоторое расстояние влево;

– клавиша «D» – с помощью данной клавиши, игрок может пройти вправо и поменять свое начальное положение аниматора.

Для того, чтоб поставить блок из пользовательского инвентаря игроку необходимо нажать на правую кнопку мыши. Для начала игроку необходимо разрушить блок в игровом мире, для того, чтоб он появился в игровом инвентаре. Затем при помощи выборщика выбрать в инвентаре быстрого доступа предмет и поставить его на игровое поле. На рисунке 4.2 продемонстрирована начальная игровая сцена.

Рисунок 4.2 – Начальная игровая сцена с инвентарем быстрого доступа

На рисунке можно увидеть инвентарь, в котором содержаться три типа инструментов. Первый инструмент является киркой, благодаря которой, игрок может разрушать блоки, которые как-либо связаны с шахтой. Вторым предмет – это топор, при помощи которого, можно разрушать деревья, листву и кактусы. Третий предмет представляет собой молоток, который разрушает задний фон, на котором располагаются некоторые блоки. К примеру, если игрок захочет при помощи кирки разрушить блок дерева или блок кактуса, у него ничего не получится. Если игрок переключится на пустой слот инвентаря, он сможет разрушать растительность в игровом мире. Каждый предмет, который пользователь выберет, будет показываться в руках игрока. Инструменты имеют стандартный

размер, а вот блоки, которые персонаж держит в руке, имеют модельку в два раза меньше.

В игровом мире существуют шахты, а в шахтах, если игрок прокопается чуть-чуть ниже, увидит разные руды. Появление руды, зависит от редкости ее. Редкая руда будет появляться в мире чуть-чуть реже, чем обыкновенная. Если прокопаться с разные стороны, можно заметить разные биомы, которые содержат разные блоки.

4.3 Функциональное тестирование игрового приложения

Целью функционального тестирования, как было выше приведено в примере, является проверка оборудования на реализуемость функциональных требований, а именно использовать все возможности устройства для решения перед ним поставленных задач. Тестирование проводилось поэтапно в несколько шагов разными тестировщиками. В процессе функционального тестирования исследуется игровое приложение, на наличие программных ошибок программиста и диагностикой требований к функционированию.

Следующие тесты позволяют выявить работоспособность игрового приложения.

Первая задача: проверка на запуск приложения.

Ожидаемый результат: игра должна запускаться на операционной системе *Windows* версии *XP* и выше.

Результат: ожидаемый и полученный результат совпали. Приложение корректно работает на операционных системах.

Вторая задача: проверить передвижение персонажа в игровом приложении.

Ожидаемый результат: главный персонаж должен двигаться при помощи клавиш «*W*», «*A*», «*S*», вверх, влево и вправо соответственно. Если персонаж каклибо взаимодействует с игровой сценой, у него должен присутствовать коллайдер и корректно отображаться на экран. Игровой персонаж должен обладать гравитацией, что способствует падению, если под ним не содержаться блоки. Прыжок осуществляется при помощи клавиши «*W*». При повороте направо или налево, персонаж должен поменять свою модельку и должен смотреть в ту сторону, в которую он направляется.

Результат: игрок плавно передвигается по игровому миру, совершает прыжок в размере двух блоков, ожидаемые и полученные результаты совпали.

Третья задача: проверить на наличие у персонажа автопрыжка.

Ожидаемый результат: при соприкосновении с блоком на расстоянии четверти блока, должны срабатывать *Raycast* лучи, которы в свою очередь провер проверяют автопрыжок. В случае успешной проверки, игрок должен

подпрыгнуть автоматически на высоту, равную одному игровому блоку. Высота прыжка выставляется автоматически.

Результат: при малейшем соприкосновении с блоком, игрок автоматически применяет прыжок, ожидаемые и полученные результаты совпали.

Четвертая задача: проверить корректную генерацию мира без выявления ошибок.

Ожидаемый результат: мир должен генерироваться случайно, без каких-либо повторений. Допускается несколько попыток генерации мира. В мире должны присутствовать:

– земля;
– камень и пещеры;
– растительность;
– деревья и кактусы;
– различные биомы;
– руды;
– нижний не разрушаемый блок.

Результат: игровой мир генерируется без каких-либо ошибок, полностью соответствует ожидаемым результатам.

Пятая задача: проверить работоспособность установки блоков в игровой мир.

Ожидаемый результат: блоки в игровой мир должны устанавливаться при помощи нажатия правой кнопки мыши. Объект должен записываться в массив объектов генерации мира.

Результат: при нажатии правой кнопки мыши блок устанавливается в заданную позицию мыши, что соответствует ожидаемому результату.

Шестая задача: проверить разрушение блоков в игровом мире.

Ожидаемый результат: разрушение блока в игровом мире должно осуществляться при помощи левой кнопки мыши. Курсор необходимо поставить в то место, в котором произойдет разрушение блока. После чего, блок должен разрушится, удалится из массива объектов и выпасть в виде уменьшенного в два раза размера блока.

Результат: при нажатии левой кнопки мыши блок разрушается, а также удаляется из списка объектов игрового мира, как описано в ожидаемом результате.

Седьмая задача: проверка на срабатывание анимации, при прыжке, ходьбе, установке и разрушении блока.

Ожидаемый результат: при совершении вышеперечисленных действий, у игрока должен срабатывать его *Animator*, а затем проигрывается соответствующая анимация. Должна происходить проверка, на то, может ли персонаж совершить анимацию, а также переключится на следующую.

Результат: при совершении действий, у игрока срабатывают анимации, что соответствует ожидаемому результату.

Восьмая задача: проверка на то, как игрок взаимодействует с блоками со стороны физики.

Ожидаемый результат: игрок при соприкосновении с блоком переднего плана, должен останавливаться и не сможет пройти дальше, пока не совершит прыжок. В случае чего, если блок является задним фоном, ничего происходить не должно.

Результат: у игрока срабатывает коллизия, при соприкосновении с объектами так, как описано в ожидаемом результате.

Девятая задача: проверить корректное отображение и изменения графического пользовательского интерфейса.

Ожидаемый результат: графический пользовательский интерфейс, включает в себя два инвентаря, быстрого доступа и обычный. Инвентарь должен содержать в себе слоты, в которых буду находится блоки или предметы.

Результат: графический пользовательский интерфейс отображается корректно и изменяется без каких-либо ошибок, ожидаемый и конечный результат совпадают.

Десятая задача: проверить работоспособность меню игрового приложения.

Ожидаемый результат: пользователь при взаимодействии с какой-либо кнопкой в игровом меню, должен совершать действия:

– запустить игровой мир;

– войти в меню обучения:

– выйти из игрового приложения.

Результат: при нажатии на кнопки, в меню интерфейса, происходят действия, которые описаны в ожидаемом результате в полном объеме.

Одиннадцатая задача: проверка работоспособности игрового инвентаря, а именно использование предметов.

Ожидаемый результат: при использовании начального предмета инвентаря топор, игрок вправе разрушать только блоки листвы, дерева и кактуса. Инструмент кирка, позволяет взаимодействовать игроку с шахтой и все, с чем она связана. Молот, позволяет персонажу разрушать плитки, которые находятся на заднем фоне.

Результат: после генерации мира, пользователю выдается набор инструментов, который позволяет выполнять действия, которые описаны в ожидаемом результате в полном объеме.

Двенадцатая задача: проверить звуковое сопровождение в игре.

Ожидаемый результат: звуковое сопровождение в игре должно функционировать исправно, срабатывать при разрушении блока и добавления его в игровой мир.

Результат: звуковые эффекты воспроизводятся корректно в необходимый интервал времени, ожидаемый и полученный результат совпадают.

4.4 Валидация на разных устройствах

В ходе тестирования игра проверялась на пяти различных устройствах, что способствует более тщательной проверки исходных данных приложения. Проблемы с загрузкой появились у третьего тестировщика, т.к. игра не рассчитана на слабые устройства.

В таблице 4.1 отображаются данные о тестировщиках, их устройствах и комплектов оборудования.

Таблица 4.1 – Устройства, которые использовались в тестировании

Устройство	ОС	*CPU*	Объем оперативной памяти
NVIDIA GeForce RTX 3070	*Windows* 11	*AMD Ryzen* 7 5700*G*	32 Гб
NVIDIA GeForce GTX 1060	*Windows* 10	*Intel Core i*5-7600	16 Гб
NVIDIA GeForce FX 5500	*Windows* 10	*Intel Pentium G*2130	4 Гб
NVIDIA GeForce GTX 750*TI*	*Windows* 10	*AMD Ryzen* 5 3600*G*	8 Гб
*Intel HD Graphics P*630	*Windows XP*	*Intel Pentium G*5400	4 Гб

Благодаря тестированию игрового приложения на несколько устройствах, удалось найти проблемы, которые связаны с быстродействием генерации игрового мира, а также выгрузки данных.

Удалось выявить следующие проблемы:

– адаптивность не была настроена под все используемые экраны;

– проблемы считывания коллайдеров и расчета коллизии при взаимодействии с объектами;

– обновление текстур каждый кадр, а не в частоту обновления экрана.

Проблема с адаптивностью была исправлена путем привязки пользовательского интерфейса с определенной частью на экране в *Inspector*. Расчет коллайдеров и соприкосновения с объектами на разных устройствах работает поразному, т.к. все действия производились в методе *FixedUpdate*, а не в *Update*.

По итогам тестирования и верификации игрового приложения, можно сделать выводы о том, что игра запускается стабильно и на хорошем уровне.

5 ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ДИПЛОМНОЙ РАБОТЫ

5.1 Оценка конкурентоспособности программного обеспечения

Игровое приложение предназначено для удовлетворения досуга пользователя и времяпровождения. В ходе использования пользователь погружается в процесс игры в песочницу-путешествие с элементами строительства и системой разрушения блоков.

Существуют другие крупные проекты, которые являются аналогом, но они имеют большие недостатки, выраженные низкой частотой обновлений, стоимостью, отсутствием возможности играть на различных операционных системах. Исходя из анализа существующих проектов можно сделать вывод о том, что разработка продукта целесообразна [10].

Техническая прогрессивность разрабатываемого ПП определяется коэффициентом эквивалентности ($К_{эк}$).

Расчет этого коэффициента осуществляется путем сравнения технического уровня товара-конкурента и разрабатываемого ПП по отношению к эталонному уровню ПП данного направления с использованием формулы:

$$К_{эк} = \frac{К_{т.н.}}{К_{т.б.}}, \quad (5.1)$$

где $К_{т.н.}, К_{т.б.}$ – коэффициенты технического уровня нового и базисного ПП, которые можно рассчитать по формуле:

$$К_{т} = \sum_{i=1}^{n} \beta \frac{P_i}{P_э}, \quad (5.2)$$

где β – коэффициенты весомости *i*-го технического параметра;

n – число параметров;

P_i – численное значение *i*-го технического параметра, сравниваемого ПП;

$P_э$ – численное значение *i*-го технического параметра эталона.

Расчет коэффициента эквивалентности приведен в таблице Д.1.

$$К_{эк} = \frac{2,3}{1,49} = 1,54.$$

Полученное значение коэффициента эквивалентности больше 1, следовательно, разрабатываемый ПП является технически прогрессивным.

Далее рассчитывается коэффициент изменения функциональных возможностей ($К_{ф.в.}$) нового программного продукта по формуле:

$$К_{ф.в.} = \frac{К_{ф.в.н.}}{К_{ф.в.б.}}, \quad (5.3)$$

где $К_{ф.в.н.}$, $К_{ф.в.б.}$ – балльная оценка неизмеримых показателей нового и базового изделия соответственно.

Расчет коэффициента изменения функциональных возможностей нового программного продукта приведен в таблице Д.2.

Новый ПП превосходит по своим функциональным возможностям базовый в 1,25 раза.

Конкурентоспособность нового ПП по отношению к базовому можно оценить с помощью интегрального коэффициента конкурентоспособности, по формуле 5.4, учитывающего все ранее рассчитанные показатели.

$$К_{и.} = \frac{К_{ф.в.} \cdot К_{н.} \cdot К_{эк.}}{К_{ц.}}, \quad (5.4)$$

где $К_н$ – коэффициент соответствия нового ПП продукта нормативам ($К_н$ = 1);

$К_ц$ – коэффициент цены потребления ($К_ц$ = 0,93).

Расчет уровня конкурентоспособности нового ПП приведен в таблице Д.3.

Коэффициент цены потребления рассчитывается как отношение договорной цены нового ПП к договорной цене базового ($К_ц$ = 0,93).

$$К_{и.} = \frac{1{,}33 \cdot 1{,}36 \cdot 1}{0{,}93} = 1{,}94.$$

Интегральный коэффициент конкурентоспособности ($К_и$) больше 1, т.е. новый программный ПП является более конкурентоспособным, чем базовый.

5.2 Оценка трудоёмкости работ по созданию программного обеспечения

В качестве единицы измерения объема ПО может быть использована строка исходного кода (*LOC*). Общий объем ПО (V_0) определяется исходя из количества и объема функций, реализуемых программой, по каталогу функций ПО по формуле:

$$V_{\text{о}} = \sum_{i=1}^{n} V_i \,, \tag{5.5}$$

где V_i – объем отдельной функции ПО;

n – общее число функций.

Уточненный объем ПО ($V_{\text{у}}$) определяется по формуле (5.6):

$$V_{\text{у}} = \sum_{i=1}^{n} V_{\text{у}i} \,, \tag{5.6}$$

где $V_{\text{у}i}$ – уточненный объем отдельной функции ПО в строках исходного кода.

Результаты расчетов представлены таблице Д.4.

Разработанное в ходе выполнения дипломной работы программное обеспечение по своим характеристикам относится ко второй категории сложности.

На основании принятого к расчету (уточненного) объема ($V_{\text{у}}$) и категории сложности ПО принимаем нормативную трудоемкость ПО выполняемых работ $\text{Т}_{\text{н}} = 213$ чел.-дн.

Дополнительные затраты труда, связанные с повышением сложности разрабатываемого ПО, учитываются посредством коэффициента повышения сложности ПО ($K_{\text{с}}$), который определяем по формуле:

$$\text{K}_{\text{с}} = 1 + \sum_{i=1}^{n} \text{K}_i \,, \tag{5.7}$$

где K_i – коэффициент, соответствующий степени повышения сложности;

n – количество учитываемых характеристик.

$$\text{K}_{\text{с}} = 1 + 0{,}12 = 1{,}12$$

Влияние фактора новизны на трудоемкость учитывается путем умножения нормативной трудоемкости на соответствующий коэффициент, учитывающий новизну ПО ($\text{K}_{\text{н}}$). Разработанная программа обладает категорией новизны Б, а значение $\text{K}_{\text{н}} = 0{,}72$.

Степень использования в разрабатываемом ПО стандартных модулей определяется их удельным весом в общем объеме ПО. В разработанном программном комплексе используется от 20% до 40% стандартных модулей, что соответствует значению коэффициента $\text{K}_{\text{т}} = 0{,}77$.

Программный модуль разработан на языке *C#*, что соответствует коэффициенту, учитывающему средства разработки ПО, $К_{у.р.}$= 0,55.

Значение коэффициентов удельных весов трудоемкости стадий разработки ПО определяются с учетом установленной категории новизны ПО и приведены в таблице Д.5.

Нормативная трудоемкость ПО ($Т_н$) выполняемых работ по стадиям разработки корректируется с учетом коэффициентов: повышения сложности ПО, учитывающих новизну ПО ($К_н$), учитывающих степень использования стандартных модулей ($К_т$), средства разработки ПО ($К_{у.р.}$) и определяются по формулам:

– для стадии технического задания по формуле:

$$Т_{у.т.з} = Т_н \cdot К_{т.з} \cdot К_с \cdot К_н \cdot К_{у.р}; \quad (5.8)$$

– для стадии технического задания по формуле:

$$Т_{у.э.п} = Т_н \cdot К_{э.п} \cdot К_с \cdot К_н \cdot К_{у.р}; \quad (5.9)$$

– для стадии технического проекта по формуле:

$$Т_{у.т.п} = Т_н \cdot К_{т.п} \cdot К_с \cdot К_н \cdot К_{у.р}; \quad (5.10)$$

– для стадии рабочего проекта по формуле:

$$Т_{у.р.п} = Т_н \cdot К_{р.п} \cdot К_с \cdot К_н \cdot К_{у.р}; \quad (5.11)$$

– для стадии ввода в действие по формуле:

$$Т_{у.в.н} = Т_н \cdot К_{в.н} \cdot К_с \cdot К_н \cdot К_{у.р}, \quad (5.12)$$

где $К_{т.з}$, $К_{э.п}$, $К_{т.п}$, $К_{р.п}$ и $К_{в.н}$ – значения коэффициентов удельных весов трудоемкости стадий разработки ПО в общей трудоемкости ПО.

Коэффициенты $К_н$, $К_с$ и $К_{у.р}$ вводятся на всех стадиях разработки, а коэффициент $К_т$ вводится только на стадии рабочего проекта.

$$Т_{у.т.з} = 213_{.} \cdot 0,1 \cdot 1,12_{.} \cdot 0,72 \cdot 0,55 = 9 \text{ чел.-дн.}$$

$$T_{у.э.п} = 213_{.} \cdot 0{,}2 \cdot 1{,}12_{.} \cdot 0{,}72 \cdot 0{,}55 = 19 \text{ чел.-дн.}$$

$$T_{у.т.п} = 213_{.} \cdot 0{,}3 \cdot 1{,}12_{.} \cdot 0{,}72 \cdot 0{,}55 = 28 \text{ чел.-дн.}$$

$$T_{у.р.п} = 213_{.} \cdot 0{,}2 \cdot 1{,}12_{.} \cdot 0{,}72 \cdot 0{,}77 \cdot 0{,}55 = 22 \text{ чел.-дн.}$$

$$T_{у.в.н} = 213_{.} \cdot 0{,}1 \cdot 1{,}12_{.} \cdot 0{,}72 \cdot 0{,}55 = 9 \text{ чел.-дн.}$$

Общая трудоемкость разработки ПО ($T_о$) определяется суммированием нормативной (скорректированной) трудоемкости ПО по стадиям разработки по формуле:

$$T_о = \sum_{i=1}^{n} T_{уi}, \qquad (5.13)$$

где $T_{уi}$ – нормативная (скорректированная) трудоемкость разработки ПО на *i*-й стадии, чел.-дн;

n – количество стадий разработки;

$$T_о = 9 + 19 + 28 + 22 + 9 = 87 \text{ чел.-дн.}$$

Результаты расчетов по определению нормативной и скорректированной трудоемкости программного обеспечения по стадиям разработки и общую трудоемкость разработки ПО ($T_о$) представлены в таблице Д.6.

Расчет ликвидной стоимости и приобретенных программных продуктов также не учитывается т.к. при внедрении игрового приложения, не нужно покупать лицензию на его использование.

Т.к в данный момент, приложению не нужны интернет-провайдера, а проект не является сетевым, прочие первоначальные капитальные вложения отсутствуют.

Расчет затрат на внедрение и адаптацию игрового приложения проводить нецелесообразно, т.к. затраты на внедрение равняются нулю.

5.3 Расчет затрат на разработку программного продукта

Суммарные затраты на разработку ПО ($З_р$) определяются по формуле:

$$З_р = З_{тр} + З_{эт} + З_{тех} + З_{м.в} + З_{мат} + З_{общ.пр} + З_{непр}. \quad (5.14)$$

Расчет производственных затрат на разработку приведены в таблице Д.7.

Расходы на оплату труда разработчиков с отчислениями ($З_{тр}$) определяются по формуле:

$$З_{тр} = ЗП_{осн} + ЗП_{доп} + ОТЧ_{зп}, \quad (5.15)$$

где $ЗП_{осн}$ – основная заработная плата разработчиков, руб.;

$ЗП_{доп}$ – дополнительная заработная плата разработчиков, руб.;

$ОТЧ_{зп}$ – сумма отчислений от заработной платы (социальные нужды, страхование от несчастных случаев), руб.

Основная ЗП разработчиков рассчитывается по формуле:

$$ЗП_{осн} = С_{ср.час} \cdot Т_о \cdot К_{ув}, \quad (5.16)$$

где $С_{ср.час}$ – средняя часовая тарифная ставка, руб./час;

$Т_о$ – общая трудоемкость разработки, чел.-час;

$К_{ув}$ – коэффициент доплаты стимулирующего характера $К_{ув}$ = 1,6.

Средняя часовая тарифная ставка определяется по формуле:

$$С_{ср.час} = \frac{\sum_i С_{чi} \cdot n_i}{\sum_i n_i}, \quad (5.17)$$

где $С_{чi}$ – часовая тарифная ставка разработчика *i*-й категории, руб./час;

n_i – количество разработчиков *i*-й категории.

Часовая тарифная ставка определяется путем деления месячной тарифной ставки на установленный при восьмичасовом рабочем дне фонд рабочего времени ($F_{мес}$):

$$С_ч = \frac{С_{м1} \cdot Т_{к1}}{F_{мес}}, \quad (5.18)$$

где $С_{м1}$ – базовая ставка специалиста;

$Т_{к1}$ – тарифный коэффициент.

$$С_ч = \frac{197 \cdot 1{,}21}{168} = 1{,}42 \text{ руб./ч.}$$

$$ЗП_{осн} = 1{,}42 \cdot 150 \cdot 8 \cdot 1{,}6 = 2726{,}4 \text{ руб.}$$

Дополнительная заработная плата рассчитывается по формуле:

$$ЗП_{доп} = \frac{ЗП_{осн} \cdot Н_{доп}}{100}, \quad (5.19)$$

где $Н_{доп}$ – норматив на дополнительную заработную плату разработчиков.

$$ЗП_{доп} = \frac{2726,4 \cdot 20}{100} = 545,28 \text{ руб.}$$

Отчисления от основной и дополнительной заработной платы (отчисления на социальные нужды и обязательное страхование) рассчитываются по формуле:

$$ОТЧ\ с.н. = \frac{(ЗП_{осн} + ЗП_{доп}) \cdot Н_{з.п.} \cdot Н_{н.с.}}{100} \quad (5.20)$$

где $Н_{з.п}$ – процент отчислений на социальные нужды и обязательное страхование от суммы основной и дополнительной заработной платы ($Н_{з.п}$ = 34%);

$Н_{н.с.}$ – процент обязательных отчислений на страхование от несчастных случаев ($Н_{н.с.}$ = 0.6%).

$$ОТЧ_{с.н} = \frac{(2726,4+545,28) \cdot 34,6}{100} = 1132 \text{ руб.}$$

$$З_{тр} = 2726,4 + 545,28 + 1132 = 4403,7 \text{ руб.}$$

Затраты машинного времени ($З_{м.в}$) определяются по формуле:

$$З_{м.в} = С_{ч} \cdot K_{т} \cdot t_{эвм}, \quad (5.21)$$

где $С_{ч}$ – стоимость 1 часа машинного времени, руб./ч;

$K_{т}$ – коэффициент мультипрограммности, показывающий распределение времени работы ЭВМ в зависимости от кол-ва пользователей ЭВМ; $K_{т}$ = 1;

$t_{эвм}$ – машинное время ЭВМ, необходимое для разработки и отладки проекта, ч.

Стоимость машино-часа определяется по формуле:

$$С_{ч} = \frac{З_{а.м} + З_{э.п} + З_{в.м} + З_{т.р} + З_{п.р}}{F_{эвм}}, \quad (5.22)$$

где $З_{ам}$ – амортизационные отчисления за год, руб./год;

$З_{э.п}$ – затраты на электроэнергию, руб./год;

$З_{в.м}$ – затраты на материалы, необходимые для обеспечения нормальной работы ПЭВМ (вспомогательные), руб./год;

$З_{т.р}$ – затраты на текущий и профилактический ремонт ЭВМ, руб./год;

$З_{пр}$ – прочие затраты, связанные с эксплуатацией ПЭВМ, руб./год;

$F_{эвм}$ – действительный фонд времени работы ЭВМ, час/год.

Такие коэффициенты как $ЗП_{об}$ (затраты на заработную плату обслуживающего персонала с учетом всех отчислений, руб./год) и $З_{ар}$ (стоимость аренды помещения под размещение вычислительной техники, руб./год). $З_{а}$ – аренда помещения равняется 25р в месяц.

Сумма годовых амортизационных отчислений ($З_{ам}$) определяется по формуле:

$$З_{ам} = \frac{\sum_{i} З_{прi}(1 + K_{доп}) m_i \cdot Н_{амi}}{Q_{эвм}}, \qquad (5.23)$$

где $З_{прi}$ – затраты на приобретение i-го вида основных средств, руб;

$K_{доп}$ – коэффициент, характеризующий дополнительные затраты, связанные с доставкой и наладкой оборудования, $K_{доп}$ = 13% от $З_{пр}$;

$З_{прi} / (1 + K_{доп})$ – балансовая стоимость ЭВМ, руб;

$Н_{амi}$ – норма амортизации, %.

$$З_{ам} = 2000 \cdot (1 + 0{,}13) \cdot 0{,}125 = 282{,}5 \text{ руб.}$$

За 86 дней разработки амортизационные отчисления составят 66,56 руб.

Стоимость электроэнергии, потребляемой за год, ($З_{эвм}$) определяется по формуле:

$$З_{э.п} = \frac{М_{сум} \cdot F_{эвм} \cdot С_{эл} \cdot A}{Q_{эвм}}, \qquad (5.24)$$

где $М_{сум}$ – паспортная мощность ПЭВМ, кВт; $М_{сум}$ = 0,41 кВт;

$С_{эл}$ – стоимость одного кВт-часа электроэнергии, руб;

A – коэффициент интенсивного использования мощности, A = 0,98…0,9.

Действительный годовой фонд времени работы ПЭВМ ($F_{эвм}$) рассчитывается по формуле:

$$F_{эвм} = (Д_{г} - Д_{вых} - Д_{пр}) F_{см} \cdot K_{см}(1 - K_{пот}), \qquad (5.25)$$

где $Д_г$ – общее количество дней в году; $Д_г$ = 365 дней;

$Д_{вых}$, $Д_{пр}$ – число выходных и празд-ых дней в году, $Д_{вых} + Д_{пр}$ = 112 дней;

$F_{см}$ – продолжительность 1 смены, $F_{см}$ = 8 часов;

$К_{см}$ – количество рабочих смен ЭВМ, $К_{см}$ = 1;

$К_{пот}$ – коэффициент, учитывающий потери рабочего времени, связанные с профилактикой и ремонтом ЭВМ, примем $К_{пот}$ = 0,2.

$$F_{эвм} = (365 - 112) \cdot 8 \cdot 1 \cdot (1 - 0{,}2) = 1619 \text{ ч.}$$

С учётом, что срок разработки программного продукта составляет 86 дней, действительный фонд времени работы ПЭВМ составляет 688 ч.

$$З_{э.п} = 0{,}41 \cdot 1619 \cdot 0{,}390852 \cdot 0{,}98 = 254{,}25 \text{ руб.}$$

Следовательно, за 86 дней разработки расходуется 59,90 руб.

Затраты на материалы ($З_{в.м}$), необходимые для обеспечения нормальной работы ПЭВМ составляют около 1% от балансовой стоимости ЭВМ и определяются по формуле:

$$З_{в.м} = \sum_i З_{пр\,i} (1 + К_{доп}) \cdot m_i \cdot К_{м.з}, \qquad (5.26)$$

где $З_{пр}$ – затраты на приобретение (стоимость) ЭВМ, руб.;

$К_{доп}$ – коэффициент, характеризующий доп. затраты, связанные с доставкой, монтажом и наладкой оборудования, $К_{доп}$ = 12–13 % от $З_{пр}$;

$К_{м.з}$ – коэффициент, характеризующий затраты на вспомогательные материалы ($К_{м.з}$ = 0,01).

$$З_{в.м} = 2000 \cdot (1 + 0{,}13) \cdot 0{,}01 = 22{,}6 \text{ руб.}$$

Затраты на текущий и профилактический ремонт ($З_{т.р}$) принимаются равными 5% от балансовой стоимости ЭВМ и вычисляются по формуле:

$$З_{т.р} = \sum_i З_{пр\,i} (1 + К_{доп}) \cdot m_i \cdot К_{т.р}, \qquad (5.27)$$

где $К_{т.р}$ – коэффициент, характеризующий затраты на текущий и профилактический ремонт, $К_{т.р}$ = 0,05.

$$З_{т.р} = 2000 \cdot (1 + 0{,}13) \cdot 0{,}05 = 113 \text{ руб.}$$

Прочие затраты на эксплуатацию ПК ($З_{пр}$) состоят из амортизационных отчислений на здания, стоимости услуг сторонних организаций и составляют 5 % от балансовой стоимости. Вычисляются по формуле:

$$З_{пр} = \sum_i З_{прi}\left(1 + K_{доп}\right) \cdot m_i \cdot K_{пр}, \quad (5.28)$$

где $K_{пр}$ – коэффициент размера прочих затрат, связанных с эксплуатацией ЭВМ ($K_{пр}$ = 0,05).

$$З_{пр} = 2000 \cdot (1 + 0{,}13) \cdot 0{,}05 = 113 \text{ руб.}$$

Для расчета машинного времени ЭВМ ($t_{эвм}$ в часах), необходимого для разработки и отладки проекта, следует использовать формулу:

$$t_{эвм} = \left(t_{р.п} + t_{вн}\right) \cdot F_{см} \cdot K_{см}, \quad (5.29)$$

где $t_{р.п}$ – срок реализации стадии «Рабочий проект» (РП);

$t_{вн}$ – срок реализации стадии «Ввод в действие» (ВП); $t_{р.п} + t_{вн} = 31$;

$F_{см}$ – продолжительность рабочей смены, ч; $F_{см}$ = 8 ч;

$K_{см}$ – количество рабочих смен, $K_{см}$ = 1.

$$t_{эвм} = 31 \cdot 8 \cdot 1 = 248 \text{ ч.}$$

$$С_{ч} = \frac{66{,}56 + 59{,}9 + 22{,}6 + 113 + 113}{688} = 0{,}54 \text{ руб./ч.}$$

$$З_{м.в} = 0{,}54 \cdot 1 \cdot 248 = 133{,}92 \text{ руб.}$$

Расчет затрат на изготовление эталонного экземпляра ($З_{эт}$) осуществляется по формуле:

$$З_{эт} = \left(З_{тр} + З_{тех} + З_{м.в}\right) \cdot K_{эт}, \quad (5.30)$$

где $K_{эт}$ – коэффициент затрат на изготовление эталонного ПП, $K_{эт}$ = 0,05.

$$З_{эт} = \left(2478{,}12 + 0 + 141{,}44\right) \cdot 0{,}05 = 130{,}98 \text{ руб.}$$

Затраты на материалы (носители информации и прочее), необходимые для обеспечения работы ПЭВМ, рассчитываются по формуле:

$$З_{мат} = З_{приобр} \cdot (1 + K_{доп}) \cdot K_{м.з}, \quad (5.31)$$

где $З_{мат}$ – затраты на приобретение ЭВМ, руб.;

$K_{доп}$ – коэффициент, характеризующий доп. затраты, связанные с доставкой, монтажом и наладкой оборудования, $K_{доп}$ = 12–13 % от $З_{приобр}$;

$K_{м.з}$ – коэффициент, характеризующий затраты на вспомогательные материалы ($K_{м.з}$ = 0,01).

$$З_{мат} = 2000 \cdot (1 + 0{,}13) \cdot 0{,}01 = 22{,}6 \text{ руб.}$$

Общепроизводственные затраты ($З_{общ.пр}$) определим по формуле:

$$З_{общ.пр} = \frac{ЗП_{осн} \cdot Н_{общ.пр}}{100}, \quad (5.32)$$

где $Н_{общ.пр}$ – норматив общепроизводственных затрат.

$$З_{общ.пр} = \frac{1541{,}12 \cdot 10}{100} = 154{,}11 \text{ руб.}$$

Непроизводственные затраты рассчитываются по формуле:

$$З_{непр} = \frac{ЗП_{осн} \cdot Н_{непр}}{100}, \quad (5.33)$$

где $Н_{непр}$ – норматив непроизводственных затрат.

$$З_{непр} = \frac{1541{,}12 \cdot 5}{100} = 77{,}06 \text{ руб.}$$

Итого получаем суммарные затраты на разработку:

$З_р$ = 2478,12 + 130,98 + 0 + 154,11 + 22,6 + 133,92 + 77,06 = 2996,79 руб.

Результаты расчета суммарных затрат на разработку ПО представлены в таблице Д.8.

5.4 Расчет договорной цены разрабатываемого программного средства

Отпускная цена ПП ($Ц_{опт}$) определяется по формулам:

$$Ц_{отп} = З_р + П_р, \quad (5.34)$$

$$П_р = \frac{З_р \cdot У_р}{100}, \quad (5.35)$$

где $З_р$ – себестоимость ПО, руб.;
$П_р$ – прибыль от реализации ПП, руб.;
$У_р$ – уровень рентабельности ПП, % ($У_р$ = 30%).

$$П_р = \frac{2996{,}79 \cdot 30}{100} = 899{,}03 \text{ руб.}$$

$$Ц_{отп} = 2996{,}79 + 899{,}03 = 3895{,}82 \text{ руб.}$$

Прогнозируемая отпускная цена ПП рассчитывается по формуле:

$$Ц_{отп} = З_р + П_р + Р_{ндс}, \quad (5.36)$$

Налог на добавленную стоимость ($Р_{ндс}$) рассчитывается по формуле:

$$Р_{ндс} = \frac{(З_р + П_р) Н_{ндс}}{100}, \quad (5.37)$$

где $Н_{ндс}$ – ставка налога на добавленную стоимость, %, $Н_{ндс}$ = 20 %.

$$Р_{ндс} = \frac{(2996{,}79 + 899{,}03) \cdot 20}{100} = 779{,}16 \text{ руб.}$$

Отпускная цена с НДС составит:

$$Ц_{отп} = З_р + П_р + Р_{ндс} = 779{,}16 + 2996{,}79 + 899{,}03 = 4674{,}89 \text{ руб.}$$

Начальная отпускная цена рассчитывается как объем аудитории на цену отпуска и определяется по формуле:

$$Ц_н = Ц_{отп} / О_а,$$

где $Ц_н$ – итоговая начальная отпускная цена;

$Ц_{отп}$ – цена отпуска;

$О_а$ – объём аудитории, чел.

При стартовой аудитории в тысячу человек, сумма дохода будет полностью равна цене отпуска. При этом каждая последующая продажа будет приносить доходность с реализации. Ожидаемая цена за единицу лицензии приложения – 4,5 руб.

5.5 Расчет частных экономических эффектов от производства и использования программного продукта

Годовой экономический эффект от производства нового ПО ($Э_{пр}$) определяется по разности приведённых затрат на базовый и новый варианты:

$$Э_{пр} = (З_{пр.баз} - З_{пр.нов}) \cdot А_{пр.нов}, \quad (5.38)$$

где $З_{пр.баз}$, $З_{пр.нов}$ – приведённые затраты на ПО по базовому и новому вариантам, руб.;

$А_{пр.нов}$ – годовой объем выпуска в расчетном году для разрабатываемого ПО, ед.

В качестве модели распространения разрабатываемого ПО была принята модель *free-to-play*, следовательно цена на использование продукта составляет 0 руб. Для получения прибыли планируется использование так называемой рекламы в интернете. Средняя стоимость рекламы за 100 показов составляет 1ц,5 руб. Прибыль от перехода пользователя по баннеру варьируется в зависимости от региона: стоимость перехода от американского пользователя составляет 0,4 руб, стоимость же перехода в России и Беларуси составляет 0,09 руб. Следовательно, американские пользователи являются приоритетнее, так как разница в прибыли за переход по баннеру составляет примерно 0,31 руб.

Доход от реализации *free-to-play* игр рассчитывается по формуле:

$$П_р = К_{е.п.} \cdot П_{с.п.} \cdot К_д \cdot К_м = 150 \cdot 0{,}18 \cdot 30 \cdot 12 = 9720 \text{ руб}, \quad (5.39)$$

где $К_{е.п.}$ – количество пользователей в день, равняется 150;

$П_{с.п}$– средний ежедневных доход с пользователя, равняется 0,18 руб.;

$К_д$ – количество дней, равняется 30;

$К_м$– количество месяцев, равняется 12.

Прибыль за реализацию за 12 месяцев составит = 9720 руб.

Таким образом можно рассчитать сроки окупаемости проекта. Прибыль берется с учетом налога на прибыль в размере 20% (9720 · 0.8) = 7407 руб.,

$$Т_{пр} = \frac{5373,84}{7407} = 0,73 \text{ лет с учетом НДФЛ.}$$

Рентабельность проекта рассчитывается по формуле (5.40):

$$Р = \frac{П}{В} \cdot 100 = \frac{7407}{8456,4} \cdot 100 = 87,5\%, \quad (5.40)$$

где П – показатель прибыли в год;

В – показатель выручки.

Таким образом, по результатам рассчитанных показателей установлено, что реализация дипломной работы является экономически целесообразной.

Т.к. в ходе разработки игрового приложение было принято решение обеспечить его нормальным функционированием и адаптацией, то затраты на обеспечение функционирования приложения эквивалентны нулю.

Расчет годового экономического эффекта от производства ПП представлен в таблице Д.8.

В таблице Д.9 представлены значения прибыли, рентабельности и срока окупаемости программного продукта. Технико-экономические показатели приведены в таблице Д.10.

По результатам рассчитанных показателей установлено, что реализация проекта обоснована и является экономически целесообразной. Об этом свидетельствуют экономический эффект от производства продукта ($Э_{пр}$ = 1450,25 руб.).

6 ОХРАНА ТРУДА И ТЕХНИКА БЕЗОПАСНОСТИ

6.1 Механическая вентиляция

Механическая вентиляция – это система, которая обеспечивает постоянный обмен воздуха в помещении с помощью механических устройств, таких как вентиляторы, воздушные каналы и рекуператоры тепла. Она играет важную роль в обеспечении качества воздуха в зданиях и помещениях, а также в поддержании комфортных и безопасных условий для пребывания людей [9].

Основные преимущества механической вентиляции:

– обеспечение постоянного и контролируемого обмена воздуха. Механические системы вентиляции позволяют поддерживать постоянный поток свежего воздуха в помещении и эффективно удалить загрязненный воздух, пыль, запахи и вредные вещества;

– регулирование влажности и температуры воздуха. Многие механические системы вентиляции имеют функцию регулирования влажности и температуры воздуха, что позволяет создать комфортные условия в помещении вне зависимости от внешних климатических условий;

– энергоэффективность. Современные механические системы вентиляции часто оснащены рекуператорами тепла, которые позволяют восстановить и использовать тепло отработанного воздуха для подогрева входящего свежего воздуха. Это способствует снижению энергопотребления и экономии тепла;

– улучшение качества воздуха. Механическая вентиляция помогает избежать скопления в помещении вредных веществ, аллергенов, пыли и других загрязнений, что способствует улучшению качества воздуха и защите здоровья людей;

– контроль и мониторинг. Механические системы вентиляции могут быть управляемыми и мониторимыми, что позволяет контролировать и настраивать параметры воздухообмена в зависимости от потребностей помещения и количества присутствующих людей.

При проектировании, установке и эксплуатации механической вентиляции необходимо соблюдать требования и нормативы, установленные соответствующими законодательными актами и рекомендациями в области охраны труда и безопасности зданий [9].

Хотя механическая вентиляция имеет ряд преимуществ, она также имеет некоторые недостатки. Вот несколько из них:

– энергозатраты. Механическая вентиляция требует электроэнергии для работы вентиляторов и других компонентов системы. Это может привести к значительным энергозатратам, особенно если система работает непрерывно или

используется на крупных производственных объектах. Высокие энергозатраты могут отразиться на операционных расходах предприятия;

– зависимость от оборудования. Механическая вентиляция требует надежной работы оборудования, включая вентиляторы, фильтры, регулирующие клапаны и другие компоненты. Если оборудование выходит из строя или требует регулярного обслуживания, это может привести к проблемам с вентиляцией и нарушению рабочей среды;

– шум. Вентиляционные системы могут быть источником шума на производстве. Работа вентиляторов и поток воздуха через воздуховоды может создавать постоянный или интенсивный шум, который может быть раздражающим для работников и повлиять на их концентрацию и комфорт;

– распределение воздуха. Механическая вентиляция может иметь ограниченные возможности в распределении воздуха внутри помещений. В некоторых случаях, например, при наличии сложной архитектуры или больших площадей, может быть сложно достичь равномерного распределения свежего воздуха и поддержания комфортных условий во всех зонах;

– недостаточная эффективность фильтрации. Хотя механические вентиляционные системы обычно оснащены фильтрами для удаления загрязнений и вредных веществ из воздуха, они могут иметь ограниченную эффективность. Некоторые частицы или вредные вещества могут пройти через фильтры или накапливаться на них, требуя регулярную замену или обслуживание фильтров для поддержания эффективности системы.

Если говорить о механических вентиляциях на производстве, она способствует обеспечению комфортных и безопасных условий работы. На рисунке 6.1 схематично продемонстрирована механическая система вентиляции на производстве.

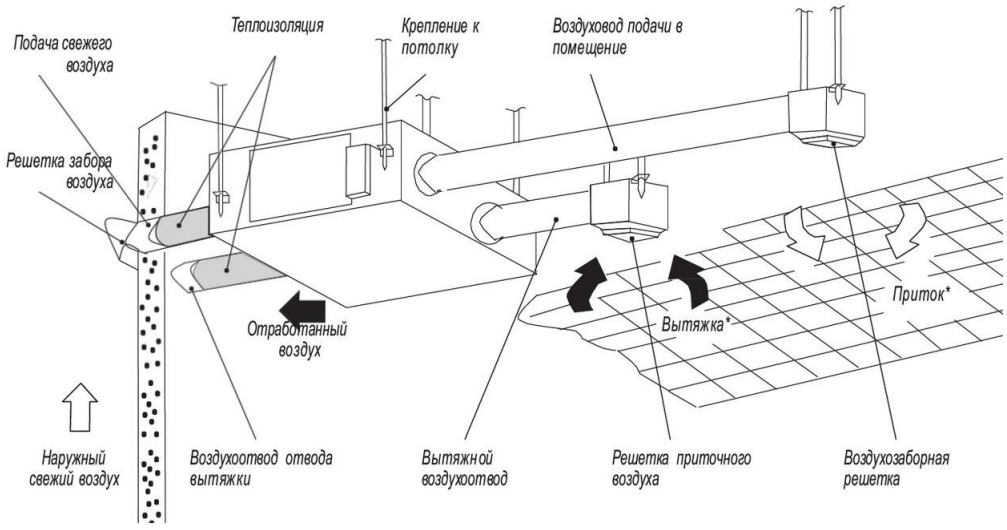

Рисунок 6.1 – Механическая система вентиляции

Существует несколько различных видов механической вентиляции, которые могут использоваться на производстве. Ниже перечислены некоторые из них:

– приточная вентиляция. Этот тип вентиляции обеспечивает подачу свежего воздуха в помещение. Вентиляционная система приточной вентиляции воздуха использует вентиляторы для притока свежего воздуха извне или из специально подготовленных зон, обеспечивая его распределение по всему помещению;

– вытяжная вентиляция. Вытяжная вентиляция предназначена для отвода загрязненного воздуха и выведения его наружу. Вентиляционная система с вытяжной вентиляцией устанавливает вытяжные вентиляторы или системы вытяжных шахт для эффективного удаления загрязненного воздуха из производственных зон;

– приточно-вытяжная вентиляция. Этот тип вентиляции сочетает в себе элементы приточной и вытяжной вентиляции. Он обеспечивает одновременный приток свежего воздуха в помещение и отвод загрязненного воздуха. Приточно-вытяжная вентиляция может быть осуществлена с помощью системы смешанного воздуха, в которой воздух смешивается внутри помещения перед отводом или после подачи;

– рециркуляционная вентиляция. В рециркуляционной вентиляции используется циркуляция воздуха внутри помещения без притока свежего воздуха извне. Воздух подается внутрь помещения, проходит через фильтры и системы очистки, а затем снова циркулирует. Этот метод может быть полезным в случаях, когда приток свежего воздуха ограничен или когда требуется контроль над температурой и влажностью воздуха;

– локальная вытяжная вентиляция. Этот вид вентиляции используется для отвода загрязнений непосредственно на рабочем месте. Он включает в себя установку вытяжных систем или вытяжных капюшонов непосредственно над источниками загрязнений, такими как машины, станки или рабочие столы.

Каждый вид механической вентиляции имеет свои особенности и применяется в зависимости от требований производственного процесса, размеров помещений, типа загрязнений и других факторов [10].

В целом, хотя механическая вентиляция является важным инструментом для обеспечения безопасной и комфортной рабочей среды на производстве, ее недостатки могут включать высокие энергозатраты, зависимость от оборудования, шум, проблемы с распределением воздуха и ограниченную эффективность фильтрации.

7 РЕСУРСО- И ЭНЕРГОСБЕРЕЖЕНИЕ ПРИ ИСПОЛЬЗОВАНИИ ПРОДУКТА

7.1 Вопросы ресурсосбережения, связанные с разработкой продукта

Ресурсосбережение – это организационная, экономическая, техническая, научная, практическая, информационная деятельность, методы, процессы, комплекс организационно-технических мер и мероприятий, сопровождающие все стадии жизненного цикла изделий и направленные на рациональное использование и экономию ресурсов.

Ресурсы – это природные или созданные человеком ценности, которые предназначены для удовлетворения производственных и непроизводственных потребностей. Из этого определения следует, что:

– материальные ресурсы – это комплекс вещественных элементов, предназначенных для обработки в процессе труда.

– ресурсосбережение – это процесс обеспечения роста объема полезных результатов при относительной стабильности материальных затрат.

Экономия материальных ресурсов – это экономическая категория, которая характеризуется снижением расхода материальных ресурсов на единицу продукции по сравнению с обычным или текущим периодом, но без снижения качества и технического уровня продукции.

Рациональный – разумный, целесообразный, обоснованный. Так что рациональное потребление материальных ресурсов является качественной характеристикой процесса разумного потребления материальных ресурсов.

Рационализация – усовершенствование, улучшение, введение более целесообразной организации чего-либо. Рационализация производства представляет собой комплекс мероприятий, направленный к более целесообразной организации производственного процесса с целью достижения наивысшей производительности труда при наименьших затратах производственных ресурсов.

Под рациональным потреблением обычно понимают процесс осознанного, общественно необходимого потребления материалов. Этот процесс – явление непрерывного характера, связанное с развитием человеческой мысли и деятельности. Поэтому то, что еще вчера было рациональным, сегодня может стать нерациональным в результате научных достижений.

Основной задачей ресурсосбережения, как науки, является экономия материальных ресурсов. Экономить материальные ресурсы можно, уменьшив затраты, установив определённые нормы, или можно внедрять новые технологии [11, с. 34].

Конкурентоспособность фирмы или предприятия, их способность удержаться на рынке товаров и услуг зависит, в первую очередь, от восприимчивости

производителей товаров к новинкам техники и технологии, позволяющим обеспечить выпуск и реализацию высококачественных товаров при наиболее эффективном использовании материальных ресурсов.

7.2 Экономия энергоресурсов в результате использования продукта

Энергосбережение – реализация организационных, правовых, технических, технологических, экономических и иных мер, направленных на уменьшение объема используемых энергетических ресурсов при сохранении соответствующего полезного эффекта от их использования (в том числе объема произведенной продукции, выполненных работ, оказанных услуг).

Энергосбережение – это рациональное энергоиспользование во всех звеньях преобразования энергии – от добычи энергоресурсов до потребления всех видов энергии конечными пользователями.

Эффекты от энергосбережения рассчитывают:

– как стоимость сэкономленных энергоресурсов или доля стоимости от потребляемых энергоресурсов, в том числе на единицу продукции;

– как количество тонн условного топлива сэкономленных энергоресурсов или доля от величины потребляемых энергоресурсов;

– в натуральном выражении (кВт·ч, Гкал и т. д.);

– как снижение доли энергоресурсов в валовом внутреннем продукте в стоимостном выражении, либо в натуральных единицах на 1 руб. валового внутреннего продукта.

На сегодняшний день существуют самые разнообразные пути экономии электроэнергии, которые могут оказаться либо эффективными, либо не очень. Для уменьшения потребляемой электроэнергии необходимо проводить процедуры:

– использовать персональные компьютеры лишь в дневное время, так как на многих предприятиях тариф электроэнергии в ночное время в несколько раз превышает тариф за неё в дневное время, что означает дополнительную трату средств;

– все ПК предприятия обновить, обращая на стандарт энергопотребления;

– заменить мониторы на аналоги, с меньшим энергопотреблением. Также при замене монитора нужно учитывать тот факт, что чем больше диагональ, тем выше потребление электроэнергии.

– при выборе нового принтера отдавать предпочтение струйному, т.к. он потребляет на 80–90% энергии меньше, чем лазерный;

– не оставлять ПК включённым на длительное время, если за ним никто не работает. Неиспользуемый два часа компьютер даже в «спящем режиме» потребляет 200–300 ватт, что за месяц составляет порядка 7,5 кВт·ч;

– рекомендуется всегда выключать периферийные устройства, если они не используются. Это позволит сэкономить порядка 2–3 кВт·ч за месяц.

Человеческое здоровье – самый невосполнимый и дорогостоящий ресурс [11, с. 72]. С целью уменьшения излучение с экрана, графическое приложение было сделано ярким, но простым для зрительного восприятия. Для графики были использованы цвета, легко различаемые человеческим глазом. При этом контрастность игры позволяет легко следить за картинкой на протяжении длительного времени. Пользователю не приходится всматриваться в мелкие детали, читать тексты, написанные мелким почерком на неконтрастном фоне. Таким образом, усталость глаз пользователя сводится к низкому уровню.

Графическое оформление разработанного игрового приложения достаточно оптимизировано, что никак не отразилось на конечной картинке, которую видит пользователь. Используя относительно небольшой набор расцветок, удалось получить вполне яркое и запоминающееся изображение. Это опять же отразилось на конечном объеме приложения и позволило уменьшить системные требования к нему. В результате процессор не перегружен, нагрузка на видеоускоритель и оперативную память небольшая, следовательно, потребляемая устройством в процессе игры энергия тоже небольшая.

ЗАКЛЮЧЕНИЕ

Результатом дипломной работы является компьютерное игровое приложение «*Terarizator*» с элементами строительства

В ходе выполнения работы был проведён анализ уже существующих продуктов, выявлены их достоинства и недостатки, на основе которых и разрабатывалась логика приложения.

В ходе разработки решены следующие задачи:

– произведен аналитический обзор доступных программных средств разработки, позволяющих разработать игровое приложение;

– разработан пользовательский графический интерфейс, реализующий взаимодействие пользователя с инвентарем и инвентарем быстрого доступа;

– разработана общая игровая структура и механики, позволяющие осуществлять игровой процесс;

– применены подходы проектирования на основе *Unity*2*D*, позволяющие разработать масштабируемое приложение;

– произведена верификация и тестирование конечного программного продукта.

Игровое приложение реализовано при помощи использования среды разработки *Unity*2*D*.

Приложение представляет из себя один игровой мир, который случайно генерируется, путем добавления шума, на текстуру. Пользователь управляет игроком, который свободно передвигается по игровому миру. Задачи и цели в игре отсутствуют.

Отличительной особенностью игрового приложения от аналогов является наличие разнообразия в игровых механиках.

Тестирование разработанного приложения показало, что игровое приложение выполняет свои функции, игровые механики функционируют должным образом.

Кроме того, система является масштабируемой, что позволяет впоследствии внедрять новую функциональность.

В ходе выполнения дипломной работы были опубликованы две работы: «Разработка открытого мира в 2*D*-играх» [5] и «Физические особенности для проектов *Unity* на основе *ECS*» [6].

Список использованных источников

1. Jessie Schell.: Game Design: How to create a game everyone plays/ Schell Jessie: Moscow, 2019. – 640 p.

2. Okita, A.: Learning C# Programming with Unity3D/ A.Okita. – Boca Raton, London, New York, 2019. – 686 p.

3. Unity Real-Time Development Platform. – Электронные данные. – Unity.com. – Режим доступа: https://unity.com/. – Дата доступа: 21.05.2023.

4. Mike Geig. Unity 2018 Game Development in 24 Hours – Sams Publishing: Amazon, 2018. – 464 p.

5. Живица, Ю. А. Разработка открытого мира в 2*D*-играх / Ю. А. Живица // Новые математические методы и компьютерные технологии в проектировании, производстве и научных исследованиях : материалы XXVI Республиканской научной конференции студентов и аспирантов, Гомель, 20–22 марта 2023 г. / ГГУ им. Франциска Скорины ; под ред: С. П. Жогаль [и др.] – Гомель, 2023. – 125 с.

6. Живица, Ю.А. Физические особенности для проектов *Unity* на основе *ECS* / Ю. А. Живица // Актуальные вопросы физики и техники: материалы XII Республиканской научной конференции студентов, магистрантов и аспирантов, Гомель, 20 апреля 2023 г. / ГГУ им. Франциска Скорины ; под ред: С. П. Жогаль [и др.] – Гомель, 2023. – 78 с.

7. Хокинг, Д. В. Unity в действии. Мультиплатформенная разработка на C# / Д. В. Хокинг – СПб.: Питер, 2019. – 215 с.

8. Кожевников, Е.А. Расчёт экономической эффективности разработки программных продуктов: метод. указания по подготовке организационно-экономического раздела дипломных работ для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» дневной формы обучения / Е.А. Кожевников, Н.В. Ермалинская. – Гомель: ГГТУ им. П.О. Сухого, 2012. – 68 с.

9. Кухаренко, С.Н. Электронный учебно-методический комплекс дисциплины «охрана труда» для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» дневной формы обучения / С.Н. Кухаренко, Д.В. Соболев. – Гомель: ГГТУ им. П.О. Сухого, 2013. – 93 с.

10. Типы систем вентиляции. – Электронные данные. – Режим доступа: https://www.rfclimat.ru/htm/vent_tp.htm. – Дата доступа: 24.05.2023.

11. Андрижиевский, А. А. Энергосбережение и энергетический ме-неджмент: учеб. пособие для студ. / А. А. Андрижиевский, В. И. Володин. – 2-е изд., испр. – Мн.: Вышэйшая школа, 2018. – 294 с.

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг программы «*Terarizator*»

Листинг класса *TerrainGenerator*:

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

public class TerrainGenerator : MonoBehaviour
{
    public PlayerController player;
    public CameraController camera;
    public GameObject tileDrop;

    [Header("Tile Sprites")]
    public float seed;
    public TileAtlas tileAtlas;

    public BiomeClass[] biomes;

    [Header("Biomes")]
    public float biomeFreq;
    public Gradient biomeGradient;
    public Texture2D biomeMap;

    [Header("Generation Settings")]
    public bool caves = true;
    public float surValue = 0.25f;
    public int worldSize = 100;
    public float heightA = 25f;
    public int chunkSize = 16;

    [Header("Other Settings")]
    public Texture2D caveNoiseTexture;

    [Header("Ore Settings")]
    public Ores[] ores;

    private BiomeClass curBiome;
    private GameObject[] worldChunks;

    private GameObject[,] world_ForegroundObjects;
    private GameObject[,] world_BackgroundObjects;

    private Tile[,] world_BackgroundTiles;
    private Tile[,] world_ForegroundTiles;

    // method of start any others methods
    private void Start()
    {
        world_ForegroundTiles = new Tile[worldSize, worldSize];
        world_BackgroundTiles = new Tile[worldSize, worldSize];
        world_ForegroundObjects = new GameObject[worldSize, worldSize];
```

```
        world_BackgroundObjects = new GameObject[worldSize, worldSize];

        for (int i = 0; i < ores.Length; i++)
        {
            ores[i].spreadTexture = new Texture2D(worldSize, worldSize);
        }

        seed = Random.Range(-10000, 10000);

        DrawCavesAndOres();
        DrawTextures();
        CreateChunks();
        TerrainGenerate();

        camera.Spawn(new Vector3(player.spawnPos.x, player.spawnPos.y, camera.transform.position.z));
        camera.worldSize = worldSize;
        player.Spawn();
        RefreshChunks();
    }
     // method of update
    private void Update()
    {
        RefreshChunks();
    }

    void RefreshChunks()
    {
        for(int i = 0; i < worldChunks.Length; i++)
        {
            if(Vector2.Distance(new Vector2((i * chunkSize) + (chunkSize / 2) , 0), new Vector2(player.transform.posi-
tion.x, 0)) > Camera.main.orthographicSize * 6f)
                worldChunks[i].SetActive(false);
            else
                worldChunks[i].SetActive(true);
        }
    }
    // generate ores and caves
    public void DrawCavesAndOres()
    {
        caveNoiseTexture = new Texture2D(worldSize, worldSize);
        for (int x = 0; x < caveNoiseTexture.width; x++)
        {
            for (int y = 0; y < caveNoiseTexture.height; y++)
            {
                curBiome = GetCurrentBiome(x, y);
                float v1 = Mathf.PerlinNoise((x + seed) * curBiome.noise, (y + seed) * curBiome.noise);
                if (v1 > curBiome.surValue)
                {
                    caveNoiseTexture.SetPixel(x, y, Color.white);
                }
                else
                {
                    caveNoiseTexture.SetPixel(x, y, Color.black);
                }
            }
        }
        caveNoiseTexture.Apply();
```

```
    for (int x = 0; x < worldSize; x++)
    {
        for (int y = 0; y < worldSize; y++)
        {
            curBiome = GetCurrentBiome(x, y);
            for (int i = 0; i < ores.Length; i++)
            {
                ores[i].spreadTexture.SetPixel(x, y, Color.black);
                if (curBiome.ores.Length >= i + 1)
                {
                    float v1 = Mathf.PerlinNoise((x + seed) * curBiome.ores[i].rarity, (y + seed) * curBiome.ores[i].rarity);
                    if (v1 > curBiome.ores[i].size)
                    {
                        ores[i].spreadTexture.SetPixel(x, y, Color.white);
                    }


                }
                ores[i].spreadTexture.Apply();
            }
        }
    }
}

// draw textures
public void DrawTextures()
{
    biomeMap = new Texture2D(worldSize, worldSize);

    for (int i = 0; i < biomes.Length; i++)
    {
        biomes[i].caveNoiseTexture = new Texture2D(worldSize, worldSize);
        for (int o = 0; o < biomes[i].ores.Length; o++)
        {
            biomes[i].ores[o].spreadTexture = new Texture2D(worldSize, worldSize);
        }

        GenerateNoiseTexture(biomes[i].noise, biomes[i].surValue, biomes[i].caveNoiseTexture);

        //ores
        for (int o = 0; o < biomes[i].ores.Length; o++)
        {
            GenerateNoiseTexture(biomes[i].ores[o].rarity, biomes[i].ores[o].size, biomes[i].ores[o].spreadTexture);
        }

    }

}

// method, which create chunks
public void CreateChunks()
{
    int numChunks = worldSize / chunkSize;
    worldChunks = new GameObject[numChunks];
    for (int i = 0; i < numChunks; i++)
    {
```

```
            GameObject newChunk = new GameObject(name = i.ToString());
            newChunk.name = i.ToString();
            newChunk.transform.parent = this.transform;
            worldChunks[i] = newChunk;
        }
    }

    public BiomeClass GetCurrentBiome(int x, int y)
    {
        //throw biomes
        for (int i = 0; i < biomes.Length; i++)
        {
            if (biomes[i].biomeCol == biomeMap.GetPixel(x, y))
            {
                return biomes[i];
            }
        }


        return curBiome;
    }

    public void TerrainGenerate()
    {
        Tile tileClass;
        for (int x = 0; x < worldSize - 1; x++)
        {
            float height;

            for (int y = 0; y < worldSize; y++)
            {

                curBiome = GetCurrentBiome(x, y);

                height = Mathf.PerlinNoise((x + seed) * curBiome.terrainFreq, seed * curBiome.terrainFreq) * curBi-
ome.heightM + heightA;

                if (x == worldSize / 2)
                    player.spawnPos = new Vector2(x, height + 2);

                if (y >= height)
                    break;
                if (y < height - curBiome.dirtHeight)
                {
                    tileClass = curBiome.tileAtlas.stone;

                    if (ores[0].spreadTexture.GetPixel(x, y).r > 0.5f && height - y < ores[0].maxSpawnHeight)
                        tileClass = tileAtlas.coal;
                    if (ores[1].spreadTexture.GetPixel(x, y).r > 0.5f && height - y < ores[1].maxSpawnHeight)
                        tileClass = tileAtlas.iron;
                    if (ores[2].spreadTexture.GetPixel(x, y).r > 0.5f && height - y < ores[2].maxSpawnHeight)
                        tileClass = tileAtlas.gold;
                    if (ores[3].spreadTexture.GetPixel(x, y).r > 0.5f && height - y < ores[3].maxSpawnHeight)
                        tileClass = tileAtlas.diamond;
                }
                else if (y < height - 1)
                {
```

```
            tileClass = curBiome.tileAtlas.dirt;
        }
        else
        {
            //top layer
            tileClass = curBiome.tileAtlas.grass;
        }

        if(y == 0)
            tileClass = tileAtlas.bedrock; // spawn layer bedrock


        if (caves && y > 0)
        {
            if (caveNoiseTexture.GetPixel(x, y).r > 0.5f)
            {
                TilePlace(tileClass, x, y, true);
            }
            else if(tileClass.wallVariant != null)
            {
                TilePlace(tileClass.wallVariant, x, y, true);
            }
        }
        else
        {
            TilePlace(tileClass, x, y, true);
        }
        if(y >= height - 1)
        {
            int t = Random.Range(0, curBiome.tree);
            if (t == 1)
            {
                //tree generator
                if (GetTileFromWorld(x,y))
                {
                    if(curBiome.biomeName == "Desert")
                    {
                        //generate cactus
                        CactusGenerate(curBiome.tileAtlas, Random.Range(curBiome.minTreeHeight, curBiome.maxTree-
Height), x, y + 1);
                    }
                    else
                    {
                        TreeGenerate(Random.Range(curBiome.minTreeHeight, curBiome.maxTreeHeight), x, y + 1);
                    }
                }
            }
            else
            {
                int i = Random.Range(0, curBiome.grassChance);
                //generate grass
                if (i == 1)
                {
                    if (GetTileFromWorld(x, y))
                    {
                        if(curBiome.tileAtlas.grass2 != null)
                            TilePlace(curBiome.tileAtlas.grass2, x, y + 1, true);
```

```
                }
            }
        }
    }
}

// generate texture of ground
public void GenerateNoiseTexture(float frequency, float limit, Texture2D noiseTexture)
{
    float v1;
    float b;
    Color col;
    for (int x = 0; x < noiseTexture.width; x++)
    {
        for (int y = 0; y < noiseTexture.height; y++)
        {
            v1 = Mathf.PerlinNoise((x + seed) * frequency, (y + seed) * frequency);
            b = Mathf.PerlinNoise((x + seed) * biomeFreq, (y + seed) * biomeFreq);
            col = biomeGradient.Evaluate(b);
            biomeMap.SetPixel(x, y, col);
            if (v1> limit)
            {
                noiseTexture.SetPixel(x, y, Color.white);
            }
            else
            {
                noiseTexture.SetPixel(x, y, Color.black);
            }
        }
    }

    noiseTexture.Apply();
    biomeMap.Apply();
}

void CactusGenerate(TileAtlas atlas,int treeHeight, int x, int y)
{
    //define our tree

    //generate log
    for (int i = 0; i < treeHeight; i++)
    {
        TilePlace(atlas.log, x, y + i, true);
    }

    TilePlace(atlas.leaf, x + 1, y + treeHeight, true);
    TilePlace(atlas.leaf, x + 1, y + treeHeight + 1, true);
}

void TreeGenerate(int treeHeight, int x, int y)
{
    //define our tree

    //generate log
    for (int i = 0; i < treeHeight; i++)
```

```
    {
        TilePlace(tileAtlas.log, x, y + i, true);
    }

    //generate leaves
        TilePlace(tileAtlas.leaf, x, y + treeHeight, true);
        TilePlace(tileAtlas.leaf, x, y + treeHeight + 1, true);
        TilePlace(tileAtlas.leaf, x, y + treeHeight + 2, true);

        TilePlace(tileAtlas.leaf, x - 1, y + treeHeight, true);
        TilePlace(tileAtlas.leaf, x - 1, y + treeHeight + 1, true);

        TilePlace(tileAtlas.leaf, x + 1, y + treeHeight, true);
        TilePlace(tileAtlas.leaf, x + 1, y + treeHeight + 1, true);
}

public bool BreakTile(int x, int y, ItemClass item)
{
    if (GetTileFromWorld(x, y) && x >= 0 && x <= worldSize && y >= 0 && y <= worldSize)
    {
        Tile tile = GetTileFromWorld(x, y);
        if (tile.toolToBreak == ItemClass.ToolType.none)
        {
            RemoveTile(x, y);
            return true;
        }
        else
        {
            if (item != null)
            {
                if (item.itemType == ItemClass.ItemType.tool)
                {
                    if (tile.toolToBreak == item.toolType)
                    {
                        RemoveTile(x, y);
                        return true;
                    }
                }
            }
        }
    }
    return false;
}

// remove the tile
public void RemoveTile(int x, int y)
{
    if (GetTileFromWorld(x, y) && x >= 0 && x <= worldSize && y >= 0 && y <= worldSize)
    {
        Tile tile = GetTileFromWorld(x,y);
        RemoveTileFromWorld(x, y);
        if (tile.wallVariant != null)
        {
            if(tile.naturallyPlaced)
                TilePlace(tile.wallVariant, x, y, true);
        }
```

```
        //drop tile
        if (tile.tileDrop)
        {
            GameObject newtileDrop = Instantiate(tileDrop, new Vector2(x, y + 0.5f), Quaternion.identity);
            newtileDrop.GetComponent<SpriteRenderer>().sprite = tile.tileDrop.tileSprites[0];
            ItemClass tileDropItem = new ItemClass(tile.tileDrop);
            newtileDrop.GetComponent<TileDropController>().item = tileDropItem;
        }

        Destroy(GetObjectFromWorld(x, y));
        RemoveObjectFromWorld(x, y);
    }
}

public bool CheckTile(Tile tile, int x, int y, bool isNaturallyPlaced)
{
    if (x >= 0 && x <= worldSize && y >= 0 && y <= worldSize)
    {
        if(tile.inBackground)
        {
            if (GetTileFromWorld(x + 1, y) ||
                GetTileFromWorld(x - 1, y) ||
                GetTileFromWorld(x, y + 1) ||
                GetTileFromWorld(x, y - 1))
            {

                if (!GetTileFromWorld(x, y))
                {
                    TilePlace(tile, x, y, isNaturallyPlaced);
                    return true;
                }
                else
                {
                    if (!GetTileFromWorld(x, y).inBackground)
                    {
                        TilePlace(tile, x, y, isNaturallyPlaced);
                        return true;
                    }
                }
            }
        }
        else
        {
            if (GetTileFromWorld(x + 1, y) ||
                GetTileFromWorld(x - 1, y) ||
                GetTileFromWorld(x, y + 1)||
                GetTileFromWorld(x, y - 1))
            {

                if (!GetTileFromWorld(x, y))
                {
                    TilePlace(tile, x, y, isNaturallyPlaced);
                    return true;
                }
                else
                {
                    if (GetTileFromWorld(x, y).inBackground)
```

```
                {
                    TilePlace(tile, x, y, isNaturallyPlaced);
                    return true;
                }
            }
        }
    }
    return false;
}

public void TilePlace(Tile tile, int x, int y, bool isNaturallyPlaced)
{
    if (x >= 0 && x <= worldSize && y>= 0 && y <= worldSize)
    {
        GameObject newTile = new GameObject();

        int chunkCoord = Mathf.RoundToInt(Mathf.Round(x / chunkSize) * chunkSize);
        chunkCoord /= chunkSize;
        newTile.transform.parent = worldChunks[(int)chunkCoord].transform;

        newTile.AddComponent<SpriteRenderer>();

        int spriteIndex = Random.Range(0, tile.tileSprites.Length);
        newTile.GetComponent<SpriteRenderer>().sprite = tile.tileSprites[spriteIndex];

        if(tile.inBackground)
        {
            newTile.GetComponent<SpriteRenderer>().sortingOrder = -10;
        }
        else
        {
            newTile.GetComponent<SpriteRenderer>().sortingOrder = -5;
            newTile.AddComponent<BoxCollider2D>();
            newTile.GetComponent<BoxCollider2D>().size = Vector2.one;
            newTile.tag = "Ground";
        }

        if(tile.name.ToUpper().Contains("WALL"))
        {
            newTile.GetComponent<SpriteRenderer>().color = new Color(0.5f, 0.5f, 0.5f);
        }

        newTile.name = tile.tileSprites[0].name;
        newTile.transform.position = new Vector2(x + 0.5f, y + 0.5f);

        Tile newTileClass = Tile.CreateInstance(tile, isNaturallyPlaced);

        AddObjectToWorld(x, y, newTile, newTileClass);
        AddTileToWorld(x, y, newTileClass);
    }
}

void AddTileToWorld(int x, int y, Tile tile)
{
    if(tile.inBackground)
    {
```

```
        world_BackgroundTiles[x, y] = tile;
    }
    else
    {
        world_ForegroundTiles[x, y] = tile;
    }
}

void AddObjectToWorld(int x, int y, GameObject tileObject, Tile tile)
{
    if (tile.inBackground)
    {
        world_BackgroundObjects[x, y] = tileObject;
    }
    else
    {
        world_ForegroundObjects[x, y] = tileObject;
    }
}

void RemoveTileFromWorld(int x, int y)
{
    if (world_ForegroundTiles[x,y] != null)
    {
        world_ForegroundTiles[x, y] = null;
    }
    else if (world_BackgroundTiles[x, y] != null)
    {
        world_BackgroundTiles[x, y] = null;
    }
}

void RemoveObjectFromWorld(int x, int y)
{
    if (world_ForegroundObjects[x, y] != null)
    {
        world_ForegroundObjects[x, y] = null;
    }
    else if (world_BackgroundObjects[x, y] != null)
    {
        world_BackgroundObjects[x, y] = null;
    }
}

Tile GetTileFromWorld(int x, int y)
{
    if (world_ForegroundTiles[x, y] != null)
    {
        return world_ForegroundTiles[x, y];
    }
    else if (world_BackgroundTiles[x, y] != null)
    {
        return world_BackgroundTiles[x, y];
    }
    return null;
}
```

```
    GameObject GetObjectFromWorld (int x, int y)
    {
        if (world_ForegroundObjects[x, y] != null)
        {
            return world_ForegroundObjects[x, y];
        }
        else if (world_BackgroundObjects[x, y] != null)
        {
            return world_BackgroundObjects[x, y];
        }
        return null;
    }
}
```

Листинг класса *Inventory*:

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

public class Inventory : MonoBehaviour
{
    public int stackLimit = 64;

    public ToolClass start_Pickaxe;
    public ToolClass start_Axe;
    public ToolClass start_Hammer;

    public Vector2 inventoryOffset;
    public Vector2 hotbarOffset;
    public Vector2 multiplier;

    public GameObject inventoryUI;
    public GameObject hotbarUI;
    public GameObject inventorySlotPrefab;

    public int inventoryW;
    public int inventoryH;
    public InventorySlot[,] inventorySlots;
    public InventorySlot[] hotbarSlots;

    public GameObject[] hotbarUISlots;
    public GameObject[,] uiSlots;

    private void Start()
    {
        inventorySlots = new InventorySlot[inventoryW, inventoryH];
        uiSlots = new GameObject[inventoryW, inventoryH];

        hotbarSlots = new InventorySlot[inventoryW];
        hotbarUISlots = new GameObject[inventoryW];

        SetupUI();
        UpdateInventoryUI();
        Add(new ItemClass(start_Pickaxe));
        Add(new ItemClass(start_Axe));
        Add(new ItemClass(start_Hammer));
    }

    void SetupUI()
```

```
    {
        //set up inventory
        for (int x = 0; x < inventoryW; x++)
        {
            for (int y = 0; y < inventoryH; y++)
            {
                GameObject inventorySlot = Instantiate(inventorySlotPrefab, inventoryUI.transform.GetChild(0).transform);
                inventorySlot.GetComponent<RectTransform>().localPosition = new Vector3((x * multiplier.x) + inven-
toryOffset.x, (y * multiplier.y) + inventoryOffset.y);
                uiSlots[x, y] = inventorySlot;
                inventorySlots[x, y] = null;
            }
        }

        //set up hotbar
        for (int x = 0; x < inventoryW; x++)
        {
            GameObject hotbarSlot = Instantiate(inventorySlotPrefab, hotbarUI.transform.GetChild(0).transform);
            hotbarSlot.GetComponent<RectTransform>().localPosition = new Vector3((x * multiplier.x) + hotbarOffset.x,
hotbarOffset.y);
            hotbarUISlots[x] = hotbarSlot;
            hotbarSlots[x] = null;
        }
    }

    void UpdateInventoryUI()
    {
        //update inventory
        for (int x = 0; x < inventoryW; x++)
        {
            for (int y = 0; y < inventoryH; y++)
            {
                if (inventorySlots[x, y] == null)
                {
                    uiSlots[x, y].transform.GetChild(0).GetComponent<Image>().sprite = null;
                    uiSlots[x, y].transform.GetChild(0).GetComponent<Image>().enabled = false;

                    uiSlots[x, y].transform.GetChild(1).GetComponent<Text>().text = "0";
                    uiSlots[x, y].transform.GetChild(1).GetComponent<Text>().enabled = false;
                }
                else
                {
                    uiSlots[x, y].transform.GetChild(0).GetComponent<Image>().enabled = true;
                    uiSlots[x, y].transform.GetChild(0).GetComponent<Image>().sprite = inventorySlots[x, y].item.sprite;
                    if (inventorySlots[x,y].item.itemType == ItemClass.ItemType.block)
                    {
                        if (inventorySlots[x, y].item.tile.inBackground)
                            uiSlots[x, y].transform.GetChild(0).GetComponent<Image>().color = new Color(0.5f, 0.5f, 0.5f);
                        else
                            uiSlots[x, y].transform.GetChild(0).GetComponent<Image>().color = Color.white;
                    }

                    uiSlots[x, y].transform.GetChild(1).GetComponent<Text>().text = inventorySlots[x, y].quan-
tity.ToString();
                    uiSlots[x, y].transform.GetChild(1).GetComponent<Text>().enabled = true;
                }
            }
        }

        //update hotbar

        for (int x = 0; x < inventoryW; x++)
```

```
        {
            if (inventorySlots[x, inventoryH - 1] == null)
            {
                hotbarUISlots[x].transform.GetChild(0).GetComponent<Image>().sprite = null;
                hotbarUISlots[x].transform.GetChild(0).GetComponent<Image>().enabled = false;

                hotbarUISlots[x].transform.GetChild(1).GetComponent<Text>().text = "0";
                hotbarUISlots[x].transform.GetChild(1).GetComponent<Text>().enabled = false;
            }
            else
            {
                hotbarUISlots[x].transform.GetChild(0).GetComponent<Image>().enabled = true;
                hotbarUISlots[x].transform.GetChild(0).GetComponent<Image>().sprite = inventorySlots[x, inventoryH -
1].item.sprite;

                if (inventorySlots[x, inventoryH - 1].item.itemType == ItemClass.ItemType.block)
                {
                    if (inventorySlots[x, inventoryH - 1].item.tile.inBackground)
                        hotbarUISlots[x].transform.GetChild(0).GetComponent<Image>().color = new Color(0.5f, 0.5f, 0.5f);
                    else
                        hotbarUISlots[x].transform.GetChild(0).GetComponent<Image>().color = Color.white;
                }

                hotbarUISlots[x].transform.GetChild(1).GetComponent<Text>().text = inventorySlots[x, inventoryH -
1].quantity.ToString();
                hotbarUISlots[x].transform.GetChild(1).GetComponent<Text>().enabled = true;
            }
        }
    }

    public bool Add(ItemClass item)
    {
        bool added = false;
        Vector2Int itemPos = Contains(item);
        if (itemPos != Vector2Int.one * -1)
        {
            inventorySlots[itemPos.x, itemPos.y].quantity += 1;
            added = true;
        }

        if (!added)
        {
            for (int y = inventoryH - 1; y >= 0; y--)
            {
                if (added)
                    break;
                for (int x = 0; x < inventoryW; x++)
                {
                    if (inventorySlots[x, y] == null)
                    {
                        inventorySlots[x, y] = new InventorySlot { item = item, position = new Vector2Int(x, y), quantity = 1 };
                        added = true;
                        break;
                    }
                }
            }
        }

        UpdateInventoryUI();
        return added;
    }
```

```
    public Vector2Int Contains(ItemClass item)
    {
        for(int y = inventoryH - 1; y >= 0; y--)
        {
            for(int x = 0; x < inventoryW; x++)
            {
                if (inventorySlots[x,y] != null)
                {

                    if (inventorySlots[x, y].item.itemName == item.itemName)
                    {
                        if(item.isStackable && inventorySlots[x, y].quantity < stackLimit)
                            return new Vector2Int(x, y);
                    }
                }
            }
        }
        return Vector2Int.one * -1;
    }

    public bool Remove(ItemClass item)
    {
        for (int y = inventoryH - 1; y >= 0; y--)
        {
            for (int x = 0; x < inventoryW; x++)
            {
                if (inventorySlots[x, y] != null)
                {
                    if (inventorySlots[x, y].item.itemName == item.itemName)
                    {
                        inventorySlots[x, y].quantity -= 1;

                        if (inventorySlots[x, y].quantity == 0)
                            inventorySlots[x, y] = null;

                        UpdateInventoryUI();
                        return true;
                    }
                }
            }
        }
        return false;
    }
}
```

Листинг класса *InventorySlot*:

```
using System.Collections;
using UnityEngine;

public class InventorySlot
{
    public Vector2Int position;
    public int quantity;
    public ItemClass item;
}
```

Листинг класса *BiomeClass*:

```
using System.Collections;
using UnityEngine;

[System.Serializable]
public class BiomeClass
{
    public string biomeName;

    public Color biomeCol;

    public TileAtlas tileAtlas;

    [Header("Other Settings")]
    public float noise = 0.05f;
    public Texture2D caveNoiseTexture;
    public float terrainFreq = 0.05f;

    [Header("Generation Settings")]
    public bool caves = true;
    public int dirtHeight = 5;
    public float surValue = 0.25f;
    public float heightM = 4f;

    [Header("Trees")]
    public int tree = 10;
    public int minTreeHeight = 6;
    public int maxTreeHeight = 9;

    [Header("Other")]
    public int grassChance = 10;

    [Header("Ore Settings")]
    public Ores[] ores;
}
```

Листинг класса *GroundCheck*:

```
using System.Collections;
using UnityEngine;

public class GroundCheck : MonoBehaviour
{

    public bool onground;
    private void OnTriggerStay2D(Collider2D col)
    {
        if (col.CompareTag("Ground"))
        {
            onground = true;
        }
        transform.parent.GetComponent<PlayerController>().onGround = onground;
    }

    private void OnTriggerExit2D(Collider2D col)
    {
        if (col.CompareTag("Ground"))
        {
            onground = false;
        }
        transform.parent.GetComponent<PlayerController>().onGround = onground;
    }
```

```
}
```

Листинг класса *ItemClass*:

```
using System.Collections;
using UnityEngine;

[System.Serializable]
public class ItemClass
{
    public enum ItemType
    {
        block,
        tool
    };

    public enum ToolType
    {
        none,
        axe,
        pickaxe,
        hammer,
        unbreakable
    };

    public ItemType itemType;
    public ToolType toolType;

    public Tile tile;
    public ToolClass tool;

    public string itemName;
    public Sprite sprite;
    public bool isStackable;

    public ItemClass(Tile _tile)
    {
        itemName = _tile.name;
        sprite = _tile.tileDrop.tileSprites[0];
        isStackable = _tile.isStackable;
        itemType = ItemType.block;
        tile = _tile;
    }

    public ItemClass(ToolClass _tool)
    {
        itemName = _tool.name;
        sprite = _tool.sprite;
        isStackable = false;
        itemType = ItemType.tool;
        toolType = _tool.toolType;
        tool = _tool;
    }
}
```

Листинг класса *Ores*:

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

```
[System.Serializable]
public class Ores
{
   public string name;
   [Range(0, 1)]
   public float rarity;
   [Range(0, 1)]
   public float size;
   public float maxSpawnHeight;
   public Texture2D spreadTexture;

}
```

Листинг класса *Tile*:

```
using System.Collections;
using UnityEngine;

[CreateAssetMenu(fileName = "tileClass", menuName = "TileClass")]
public class Tile : ScriptableObject
{
   public string tileName;
   public Tile wallVariant;
   public Sprite[] tileSprites;
   public bool inBackground = false;
   public bool naturallyPlaced = true;
   public bool isStackable = true;
   public Tile tileDrop;
   public ItemClass.ToolType toolToBreak;


   public static Tile CreateInstance(Tile tile, bool isNaturralyPlaced)
   {
      var thisTile = ScriptableObject.CreateInstance<Tile>();

      thisTile.Init(tile, isNaturralyPlaced);

      return thisTile;
   }

   public void Init (Tile tile, bool isNaturralyPlaced)
   {
      tileName = tile.tileName;
      wallVariant = tile.wallVariant;
      tileSprites = tile.tileSprites;
      inBackground = tile.inBackground;
      tileDrop = tile.tileDrop;
      naturallyPlaced = isNaturralyPlaced;
      toolToBreak = tile.toolToBreak;
   }

}
```

Листинг класса *TileAtlas*:

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

```
[CreateAssetMenu(fileName = "tileAtlas", menuName = "Tile Atlas")]
public class TileAtlas : ScriptableObject
{
   //terrain
   public Tile grass;
   public Tile dirt;
   public Tile stone;
   public Tile log;
   public Tile leaf;
   public Tile snow;
   public Tile sand;
   public Tile bedrock;

   //ores
   public Tile coal;
   public Tile iron;
   public Tile gold;
   public Tile diamond;

   //grass
   public Tile grass2;
}
```

Листинг класса *TileDropController*:

```
using System.Collections;
using UnityEngine;

public class TileDropController : MonoBehaviour
{
   public ItemClass item;

   private void OnTriggerEnter2D(Collider2D col)
   {

      if (col.gameObject.CompareTag("Player"))
      {
         //adds to the players inventory
         if(col.GetComponent<Inventory>().Add(item))
            Destroy(this.gameObject);

         //delete tile
      }
   }
}
```

Листинг класса *ToolClass*:

```
using System.Collections;
using UnityEngine;

[CreateAssetMenu(fileName = "ToolClass", menuName = "Tool Class")]
public class ToolClass : ScriptableObject
{
   public string name;
   public Sprite sprite;
   public ItemClass.ToolType toolType;
}
```

Листинг класса *PlayerController*:

```
using System.Collections;
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public LayerMask layerMask;
    public int selectedSlotIndex = 0;
    public GameObject hotbarSelector;

    public GameObject handHolder;

    public Inventory inventory;
    public bool inventoryShowing = false;

    public ItemClass selectedItem;


    public int playerRange;
    public Vector2Int mousePos;
    public float moveSpeed;
    public float jumpForce;
    public bool onGround;

    private Rigidbody2D rb;
    private Animator anim;
    public float horizontal;
    public bool hit;
    public bool place;

    [HideInInspector]
    public Vector2 spawnPos;
    public TerrainGenerator terrainGenerator;


    public void Spawn()
    {
        GetComponent<Transform>().position = spawnPos;

        rb = GetComponent<Rigidbody2D>();
        anim = GetComponent<Animator>();
        inventory = GetComponent<Inventory>();
    }

    private void FixedUpdate()
    {
        float jump = Input.GetAxisRaw("Jump");
        float vertical = Input.GetAxisRaw("Vertical");

        Vector2 movement = new Vector2(horizontal * moveSpeed, rb.velocity.y);

        if (horizontal > 0)
            transform.localScale = new Vector3(-1, 1, 1);
        else if (horizontal < 0)
            transform.localScale = new Vector3(1, 1, 1);

        //jumping
        if (vertical > 0.1f || jump > 0.1f)
        {
```

```
        if (onGround)
            movement.y = jumpForce;
    }
    //autojumping
    if(FootRaycast() && !HeadRaycast() && movement.x != 0)
    {
        if (onGround)
            movement.y = jumpForce * 0.7f;
    }

    rb.velocity = movement;
}

private void Update()
{
    horizontal = Input.GetAxis("Horizontal");
    hit = Input.GetMouseButtonDown(0);
    place = Input.GetMouseButton(1);

    //hotbatUI scroll
    if(Input.GetAxis("Mouse ScrollWheel") > 0)
    {
        //scroll up
        if (selectedSlotIndex < inventory.inventoryW - 1)
            selectedSlotIndex += 1;
    }
    else if(Input.GetAxis("Mouse ScrollWheel") < 0)
    {
        //scroll down
        if (selectedSlotIndex > 0)
            selectedSlotIndex -= 1;
    }

    //set selected slot UI
    hotbarSelector.transform.position = inventory.hotbarUISlots[selectedSlotIndex].transform.position;
    if (selectedItem != null)
    {
            handHolder.GetComponent<SpriteRenderer>().sprite = selectedItem.sprite;

        if (selectedItem.itemType == ItemClass.ItemType.block)
            handHolder.transform.localScale = new Vector3(-0.5f, 0.5f, 0.5f);
        else
            handHolder.transform.localScale = new Vector3(-1, 1, 1);

    }
    else
        handHolder.GetComponent<SpriteRenderer>().sprite = null;

    //set selected item
    if (inventory.inventorySlots[selectedSlotIndex, inventory.inventoryH - 1] != null)
        selectedItem = inventory.inventorySlots[selectedSlotIndex, inventory.inventoryH - 1].item;
    else
        selectedItem = null;

    if (Input.GetKeyDown(KeyCode.E))
    {
        inventoryShowing = !inventoryShowing;
    }

    if (Vector2.Distance(transform.position, mousePos) <= playerRange &&
        Vector2.Distance(transform.position, mousePos) >= 1f)
    {
```

```
            if (place)
            {
                if(selectedItem != null)
                {
                    if (selectedItem.itemType == ItemClass.ItemType.block)
                    {
                        if(terrainGenerator.CheckTile(selectedItem.tile, mousePos.x, mousePos.y, false))
                            inventory.Remove(selectedItem);
                    }
                }
            }

        }

        if(Vector2.Distance(transform.position, mousePos) <= playerRange)
        {
            if(hit)
            {
                terrainGenerator.BreakTile(mousePos.x, mousePos.y, selectedItem);
            }
                //terrainGenerator.RemoveTile(mousePos.x, mousePos.y);
        }

        //set mouse pos
        mousePos.x = Mathf.RoundToInt(Camera.main.ScreenToWorldPoint(Input.mousePosition).x - 0.5f);
        mousePos.y = Mathf.RoundToInt(Camera.main.ScreenToWorldPoint(Input.mousePosition).y - 0.5f);

        inventory.inventoryUI.SetActive(inventoryShowing);

        anim.SetFloat("horizontal", horizontal);
        anim.SetBool("hit", hit || place);
    }

    public bool FootRaycast()
    {
        RaycastHit2D hit = Physics2D.Raycast(transform.position - (Vector3.up * 0.5f), -Vector2.right * transform.lo-
calScale.x, 0.5f, layerMask);
        return hit;
    }

    public bool HeadRaycast()
    {
        RaycastHit2D hit = Physics2D.Raycast(transform.position + (Vector3.up * 0.5f), -Vector2.right * transform.lo-
calScale.x, 0.5f, layerMask);
        return hit;
    }
}
```

Листинг класса *CameraController*:

```
using System.Collections;
using UnityEngine;

public class CameraController : MonoBehaviour
{

    [Range(0,1)]
    public float smoothTime;

    public Transform playerTransform;
```

```
    public Vector2 spawnPos;

    [HideInInspector]
    public int worldSize;

    private float orthoSize;

    public void Spawn(Vector3 pos)
    {
        GetComponent<Transform>().position = pos;
        orthoSize = GetComponent<Camera>().orthographicSize;
    }

    public void FixedUpdate()
    {
        Vector3 pos = GetComponent<Transform>().position;

        pos.x = Mathf.Lerp(pos.x, playerTransform.position.x, smoothTime);
        pos.y = Mathf.Lerp(pos.y, playerTransform.position.y, smoothTime);

        pos.x = Mathf.Clamp(pos.x, 0.4f + (orthoSize * 2f), (worldSize + 0.4f) - (orthoSize * 2f));

        GetComponent<Transform>().position = pos;
    }
}
```

ПРИЛОЖЕНИЕ Б
(обязательное)
Руководство системного программиста

1. Общие сведения о программе

Разработанное игровое приложение предназначено для организации игрового процесса для одного игрока. Жанр игры – «*SandBox*». Данный жанр предусматривает возможность организации игрового процесса как для одного игрока, так и для нескольких. В данном случае для одного. Управление игроком осуществляется с помощью клавиатуры и мыши.

Для корректной работы ПО необходимы следующие требования:

– поддерживаемые операционные системы *Windows XP* и выше;

– наличие стандартной клавиатуры и мыши.

2. Структура программы

Структурно приложение разделено на две рабочие области – область игрового меню и область игрового поля.

Окно с настройками игры состоит из:

- кнопки начала игры;
- кнопки обучения игрока;
- кнопки выхода из игры.

При запуске появляется всего одно окно для пользователя. Данная область отвечает за инвентарь персонажа, а также за обзор самой игровой сцены.

3. Настройка программы

Настройка ПО осуществляется путем запуска исполняемого файла.

4. Проверка программы

Если все инструкции соблюдены и после запуска приложение не выдает никаких сообщений об ошибках, значит игровое приложение работает ис-правно.

5. Сообщения системному программисту

Игровое приложение «*Terarizator*» не выдает никаких сообщений системному программисту.

ПРИЛОЖЕНИЕ В
(обязательное)
Руководство программиста

1. Назначение и условия применения программы

Разработанное программное приложение предназначено для организации игрового процесса для одного игрока. Жанр игры – «*SandBox*». Данный жанр предусматривает возможность организации игрового процесса как для одного игрока, так и для нескольких. В данном случае для одного. Управление игроком осуществляется с помощью клавиатуры и мыши.

Для корректной работы программного средства необходимо соблюдение следующих требований:

- поддерживаемые операционные системы *Windows XP* и выше;
- наличие стандартной клавиатуры и мыши;
- наличие следующих устройств вывода: экран, подключаемый по *HDMI*.

2. Характеристики программы

При запуске открывается меню, где можно выбрать несколько действий, таких как запуск игры, меню обучения и выход. После нажатия запуска игры пользователь получает возможность управлять своим персонажем.

3. Обращение к программе

Приложение запускается путем открытия исполняющегося файла *terarizator.exe*, который находится в каталоге с игрой.

4. Входные и выходные данные

Входной информацией являются данные, которые предоставляет игрок путём нажатия на клавиши клавиатуры и кнопок мыши. Выходной информацией являются кадры на экране с результатом игрового процесса.

5. Сообщения

В ходе выполнения приложения никаких сообщений не предусмотрено.

ПРИЛОЖЕНИЕ Г
(обязательное)
Руководство пользователя

1. Введение

Руководство пользователя информирует пользователя базовыми знаниями по эксплуатации игрового приложения.

Разработанное программное приложение предназначено для организации игрового процесса для одного игрока. Игрок управляет персонажем, в возможности которого входят: передвижение, установка блоков, прыжок и удаление блоков. Управление игроком осуществляется с помощью клавиатуры и мыши.

Программное средство обладает следующим функционалом:

– наличие меню;

– игровая сцена;

– управления игровым персонажем от третьего лица;

– наличие генерации мира и щахт.

Для использования программного средство пользователь должен быть ознакомлен с:

– настоящим руководством пользователя;

– правилами использования ПК.

2. Назначение и условия применения

Разработанное программное приложение предназначено для одиночной игры. Данное приложение несёт развлекательный характер и не предназначено для азартных игроков.

Для корректной работы программного средства необходимо соблюдение следующих требований:

– поддерживаемые операционные системы *Windows XP* и выше;

– наличие стандартной клавиатуры и мыши;

– наличие следующих устройств вывода: экран, подключаемый по *HDMI*.

3. Подготовка к работе

Для запуска приложения необходимо запустить скомпилированный исполняемый файл *terarizator.exe* из каталога с игрой.

Если все инструкции соблюдены, и приложение не выдает никаких сообщений об ошибках, значит, программа работает исправно.

4. Описание операций

В ходе разработки игрового приложения было реализовано меню игры и игровая сцена, на которой происходят основные действия игры. При запуске

открывается меню, где можно выбрать несколько действий, таких как запуск игры, обучение и выход. Это изображает рисунок Г.1.

Рисунок Г.1 – Меню игрового приложения

Игрок может выйти из игры в меню при нажатии на кнопку *Exit*. Окно с игровой сценой при запуске игры продемонстрировано на рисунке Г.2.

Рисунок Г.2 – Игровая сцена приложения

Игровой мир случайным образом генерируется, а также все что находится в нем:

– шахты и руды в них;
– игровой персонаж;
– деревья и трава;
– пользовательский интерфейс.

В игре отсутствуют цели и задачи в соответствии и жанром игры. Игроку предоставляется открытый мир, где можно делать все что он захочет. Шахта и использование инвентаря продемонстрированы на рисунке Г.3.

Рисунок Г.3 – Шахты и инвентарь игрока

Управление игроком осуществляется при помощи клавиш *A* – для перемещения игрока влево, *D* – для перемещения игрока вправо, *W* – для совершения прыжка. Разрушение блока производится при помощи ЛКМ, а установка блока при помощи нажатии ПКМ.

5. Аварийные ситуации

Чтобы избежать ошибок при использовании данного игрового приложения, необходимо соблюдать порядок действий и условия использования, описанные в разделе 3 данного руководства пользователя.

В случае непредвиденного «зависания» программы рекомендуется завершить процесс в диспетчере задач и запустить снова. Или можно подождать некоторое время, так как существует возможность зависания программы за счет наличия библиотек, используемых в игровом приложении.

6. Рекомендации по освоению

Для запуска программы и корректной ее работы, желательно иметь операционную систему *Windows* 7/8/10. Так-же необходимо иметь исправную клавиатуру и мышь, для корректного управления героем во время игрового процесса.

ПРИЛОЖЕНИЕ Д

(справочное)

Результаты расчета экономического обоснования

Таблица Д.1 – Расчет коэффициента экономического обоснования

Наименование параметра	Вес параметра, β	Значение параметра			$\frac{P_б}{P_э}$	$\frac{P_н}{P_э}$	$\beta\frac{P_б}{P_э}$	$\beta\frac{P_н}{P_э}$
		$P_б$	$P_н$	$P_э$				
Объём памяти	0,3	9	7	7	1,29	1	0,39	0,3
Время обработки данных	0,3	0,5	0,8	0,3	1,67	2,67	0,5	0,8
Отказы	0,6	1	2	1	1	2	0,6	1,2
Итого							1,49	2,3
Коэффициент эквивалентности							2,3/1,49=1,54	

Таблица Д.2 – Расчет коэффициента изменения функциональных возможностей

Наименование показателя	Балльная оценка базового ПП	Балльная оценка нового ПП
Объём памяти	2	4
Быстродействие	3	4
Удобство интерфейса	4	5
Степень утомляемости	3	3
Производительность труда	4	4
Итого	16	20
Коэффициент функциональных возможностей	20/16 = 1,25	

Таблица Д.3 – Расчет уровня конкурентоспособности нового ПП

Коэффициенты	Значение
Коэффициент эквивалентности ($К_{эк}$)	1,36
Коэффициент изменения функциональных возможностей ($К_{ф.в.}$)	1,33
Коэффициент соответствия нормативам ($К_н$)	1
Коэффициент цены потребления ($К_ц$)	0,93
Интегральный коэффициент конкурентоспособности	$(1,33 \cdot 1,36 \cdot 1)/0,93 = 1,94$

Таблица Д.4 – Перечень и объем функций ПО

Код функций	Наименование (содержание) функций	Объем функции строк исходного кода	
		по каталогу (V_o)	уточненный (V_y)
101	Организация ввода информации	150	200
102	Контроль, предварительная обработка и ввод информации	688	500
305	Формирование файла	2460	400
303	Обработка файлов	1100	430
506	Обработка ошибочных сбойных ситуаций	1720	200
507	Обеспечение интерфейса между компонентами	1820	1000
702	Расчетные задачи (расчет режимов обработки)	1330	300
706	Предварительная обработка, печать	470	350
707	Графический вывод результатов	590	700
Итого		10328	4080

Таблица Д.5 – Значения коэффициентов удельных весов трудоемкости стадий разработки ПО в общей трудоемкости

Категория новизны ПО	Без применения CASE-технологий				
	Стадии разработки ПО				
	ТЗ	ЭП	ТП	РП	ВН
	Значения коэффициентов				
	$K_{т.з}$	$K_{э.п}$	$K_{т.п}$	$K_{р.п}$	$K_{в.н}$
Б	0,1	0,2	0,3	0,3	0,1

Таблица Д.6 – Расчет общей трудоемкости разработки ПО

Показатели	Стадии разработки					Итого
	ТЗ	ЭП	ТП	РП	ВН	
1	2	3	4	5	6	7
Общий объем ПО (V_o), кол-во строк (*LOC*)	–	–	–	–	–	10328

Продолжение таблицы Д.6

1	2	3	4	5	6	7
Общий объем ПО (*Vo*), кол-во строк (*LOC*)	–	–	–	–	–	10328
Общий уточненный объем ПО ($V_у$), кол-во строк (*LOC*)	–	–	–	–	–	4080
Категория сложности разрабатываемого ПО	–	–	–	–	–	2
Нормативная трудоемкость разработки ПО ($Т_н$), чел.-дн.	–	–	–	–	–	213
Коэффициент повышения сложности ПО ($К_с$)	1,12	1,12	1,12	1,12	1,12	–
Коэффициент, учитывающий новизну ПО ($К_н$)	0,72	0,72	0,72	0,72	0,72	–
Коэффициент, учитывающий степень использования стандартных модулей ($К_т$)	–	–	–	0,77	–	–
Коэффициент, учитывающий средства разработки ПО ($К_{у.р}$)	0,55	0,55	0,55	0,55	0,55	–
Коэффициенты удельных весов трудоемкости стадий разработки ПО ($К_{тз}$, $К_{эп}$, $К_{тп}$, $К_{рп}$, $К_{вн}$)	0,1	0,2	0,3	0,3	0,1	1
Распределение скорректированной (с учетом $К_с$, $К_н$, $К_{у.р}$) трудоемкости ПО по стадиям, чел.-дн.	9	18	28	22	9	–
Общая трудоемкость разработки ПО ($Т_о$), чел.-дн.	–	–	–	–	–	86

Таблица Д.7 – Параметры для расчета производственных затрат на разработку

Параметр	Единица измерения	Значение
1	2	3
Базовая ставка специалиста	руб.	195
Разряд разработчика	–	12
Тарифный коэффициент	–	1,21
Коэффициент $К_{ув}$	–	1,6
Норматив отчислений на доп. зар-плату разработчиков (Ндоп)	%	20

Продолжение таблицы Д.7

1	2	3
Численность обслуживающего персонала	чел.	1
Разряд обслуживающего персонала	–	8
Тарифный коэффициент	–	2,17
Стоимость одного кВт-часа электроэнергии (СЭЛ)	руб.	0,39
Коэффициент потерь рабочего времени ($К_{пот}$)	–	0,2
Премия	%	5
Доплата за стаж	руб.	19,5

Таблица Д.8 – Расчет суммарных затрат на разработку ПО, руб

Статья затрат	Итого
Затраты на оплату труда разработчиков ($З_{тр}$)	2478,12
Основная заработная плата разработчиков ($ЗП_{осн}$)	1541,12
Дополнительная заработная плата разработчиков ($ЗП_{доп}$)	308,22
Отчисления от основной и дополнительной заработной платы ($ОТЧ_{с.н.}$)	628,78
Затраты машинного времени ($З_{мв}$)	141,44
Стоимость машино-часа, руб./час ($С_{ч}$)	0,68
Стоимость электроэнергии, потребляемой за год ($З_{э.п.}$)	254,25
Затраты на текущий и профилактический ремонт	113
Прочие затраты, связанные с эксплуатацией ЭВМ	113
Машинное время ЭВМ, час	208
Затраты на изготовление эталонного экземпляра ($З_{эт}$)	130,98
Затраты на технологию ($З_{тех}$)	0
Затраты на материалы ($З_{мат}$)	22,6
Общепроизводственные затраты ($З_{общ.пр}$)	154,11
Непроизводственные (коммерческие) затраты ($З_{непр}$)	77,06
Суммарные затраты на разработку ПО ($З_{р}$)	2996,79

Таблица Д.9 – Расчет годового экономического эффекта от производства

Наименование параметра	Обозначение	Базовый вариант	Новый вариант
1	2	3	4
Оптовая цена, руб.	$Ц_i$	6345,5	4674,89
Норматив рентабельности	R_i	0,15	0,15

Продолжение таблицы Д.9

1	2	3	4
Себестоимость производства, руб.	$С_{прi}$	6345,5/(1+0,15)= =5525,7	4075,4
Удельные капитальные вложения, руб.	$К_р$	2000	
Нормативный коэффициент капитальных вложений		0,15	0,15
Расчет			
Удельные приведенные затраты на производство ПО, руб.	$З_{прi}$	5526+0,15·2000= =5825,65	4375,4
Годовой экономический эффект от производства ПП, руб.	$Э_{пр}$	5825,65 – 4375,4=1450,25	
Прирост прибыли, руб.	$\Delta П_{пр}$	[(4674,89 – 4075,4) – (6354,5 – – 5525,7)] · 1 = 205,66	

Таблица Д.10 – Технико-экономические показатели проекта

Наименование показателя	Единица измерения	Проектный вариант
Общая трудоемкость разработки ПО	чел.-дн.	150
Капитальные вложения в проект	руб.	2000,00
Затраты на разработку программы	руб.	2996,79
Затраты на оплату труда разработчиков	руб.	2478,12
Затраты машинного времени	руб.	141,44
Затраты на технологию	руб.	0
Затраты на материалы	руб.	22,60
Общепроизводственные затраты	руб.	154,11
Непроизводственные (коммерческие) затраты	руб.	77,06
Цена без НДС	руб.	3905,61
НДС	руб.	781,12
Цена с НДС	руб.	4674,89
Экономический эффект от производства нового ПП	руб.	1450,25

ПРИЛОЖЕНИЕ Ж
(рекомендуемое)

Копия справки о внедрении