Реферат

ИГРОВОЕ 2*D* ПРИЛОЖЕНИЕ В ЖАНРЕ АРКАДНОГО СИМУЛЯТОРА ЖИЗНИ ФЕРМЕРА В *TOP-DOWN* ПРОЕКЦИИ НА ПЛАТФОРМЕ *UNITY*: дипломная работа / И.Д. Дубовцов – Гомель: ГГТУ им. П. О. Сухого, 2024 – Дипломная работа: 134 страницы, 26 рисунков, 26 таблиц, 10 источников, семь приложений.

Ключевые слова: игра, приложение, компьютерная, ферма, игровой процесс, предмет, инвентарь, чат, *NPC*, графика, пользователь, разработка, токен, *API*, игровой персонаж, игровой движок, *Unity*, функционал, архитектура, структура, класс, структура данных, параметр, взаимодействие, графический интерфейс, интерфейс, искусственный интеллект.

Характеристика проделанной работы: при выполнении работы рассмотрены и проанализированы конкурентноспособные игровые приложения в жанре аркадного фермерского симулятора. 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 аркадного симулятора жизни фермера.

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 81,76. 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 «игровое 2*D* приложение в жанре аркадного симулятора жизни фермера в *top-down* проекции на платформе *Unity*».

Объектом исследования является игровое приложение для компьютеров.

Целью работы является разработка игрового приложения для компьютеров в жанре аркадного симулятора жизни фермера.

Основным результатом работы является, компьютерное игровое приложение.

Рэзюмэ

Тэмай працы з'яўляецца «гульнявое 2D прыкладанне ў жанры аркаднага сімулятара жыцця фермера ў top-down праекцыі на платформе Unity». Аб'ектам даследавання з'яўляецца гульнявое прыкладанне для кампутараў. Мэтай працы з'яўляецца распрацоўка гульнявога прыкладання для кампутараў у жанры аркаднага сімулятара жыцця фермера. Асноўным вынікам працы з'яўляецца, камп'ютэрнае гульнявое прыкладанне.

Summary

The topic of the work is «2D game application in the genre of arcade simulatorof farmer's life in top-down projection on the Unity platform».The object of the study is a game application for computers.The purpose of the work is to develop a game application for computers in the genre of arcade farming simulator.The main result of the work is a computer game application.

СОДЕРЖАНИЕ

Перечень условных обозначений и сокращений 7
Введение 8
1 Существующие методы реализации игровых приложений в жанре аркадного симулятора жизни фермера 10
1.1 Описание предметной области 10
1.2 Жанр аркадного фермерского симулятора 11
1.3 Аналитический обзор представителей игр жанра аркадного фермерского симулятора 13
1.4 Игровой движок *Unity* 16
1.5 Игровой движок *Unreal Engine* 19
1.6 Сравнение *Unity* и *Unreal Engine* 21
1.7 Требования к проектируемому программному обеспечению 23
2 Архитектура игрового приложения «*Farmer's Valley*» 26
2.1 Общие положения об архитектуре игрового приложения 26
2.2 Основные функции игрового приложения «*Farmer's Valley*» 27
2.3 Структурные элементы игрового приложения «*Farmer's Valley*» 29
2.4 Устройство внутренней архитектуры и схема разделения данных 31
2.5 Архитектура пользовательского интерфейса 35
3 Программная реализация игрового приложения «*Farmer's Valley*» 38
3.1 Взаимодействие игровых элементов 38
3.2 Принцип работы модуля управления игровым миром и управления фермой 39
3.3 Принцип работы программных модулей управления инвентарем 42
3.3 Принцип работы модуля экономической и социальной системы 49
3.4 Паттерны в приложении «*Farmer's Valley»* 51
4 Валидация и верификацтя результатов работы игрового приложения «*Farmer’s Valley*» 53
4.1 Принципы работы пользовательского графического интерфейса и основные механики игрового приложения «*Farmer’s Valley*» 53
4.2 Виды тестирования игр 61
4.3 Функциональное тестирование 61
4.3 Юзабилити-тестирование 64
5 Экономическое обоснование игрового приложения «*Farmer’s Valley*» 66
5.1 Технико-экономическое обоснование целесообразности разработки программного продукта и оценка его конкурентоспособности 66
5.2 Оценка трудоемкости работ по созданию программного обеспечения 68
5.3 Расчёт затрат на разработку программного продукта 72

5.4 Расчёт договорной цены и частных экономических эффектов от производства и использования программного продукта 76
6 Охрана труда и техника безопасности 78
6.1 Требования к производственному освещению 78
7 Энергосбережение и ресурсосбережение при эксплуатации программного обеспечения 81
7.1 Энергосбережение и ресурсосбережение при внедрении и эксплуатации программного обеспечения 81
Заключение 84
Список использованных источников 85
ПРИЛОЖЕНИЕ А Программный код приложения 86
ПРИЛОЖЕНИЕ Б Руководство системного программиста 123
ПРИЛОЖЕНИЕ В Руководство программиста 124
ПРИЛОЖЕНИЕ Г Руководство пользователя 125
ПРИЛОЖЕНИЕ Д Формулы расчета экономической эффективности 126
ПРИЛОЖЕНИЕ Ё Результат опытной эксплуатации 133
ПРИЛОЖЕНИЕ Ж Список опубликованных работ 134

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.

ММО – массовая многопользовательская онлайн-игра.

ОС – операционные системы.

ПК – персональный компьютер.

ПО – программное обеспечение.

ИИ – искусственный интеллект.

Токены – это блоки, на которые система разбивает текст.

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

NPC – неигровой персонаж.

API – программный интерфейс, описание способов взаимодействия одной компьютерной программы с другими.

ChatGPT – чат-бот с генеративным искусственным интеллектом.

QA (*Quality Assurance*) специалист – это профессионал, отвечающий за обеспечение качества программного обеспечения в *IT*-компаниях.

2*D* – двухмерное пространство.

3*D* – трёхмерное пространство.

ВВЕДЕНИЕ

Индустрия видеоигр непрерывно развивается благодаря развитию новых технологий и накоплению пользовательского опыта. Современные игры сочетают в себе высококачественную графику, сложные игровые механики и интерактивные элементы, что позволяет привлечь и удержать внимание широкой аудитории. Одним из популярных жанров являются аркадные фермерские симуляторы, которые предлагают игрокам возможность управлять виртуальными фермами, развивая свои навыки стратегического планирования и управления ресурсами.

Целью данной дипломной работы является разработка игрового 2*D* приложения в жанре аркадного фермерского симулятора под названием «*Farmer's Valley*», выполненного в *top-down* проекции на платформе *Unity*. Уникальность проекта заключается в использовании 2.5*D* графики, которая сочетает элементы двухмерной и трехмерной графики, а также в реализации социального взаимодействия с неигровыми персонажами (*NPC*) при помощи *ChatGPT API*. Эта интеграция позволит создать более реалистичные и динамичные диалоги, значительно улучшая пользовательский опыт.

Актуальность разработки обусловлена растущим интересом к играм, сочетающим простоту управления с богатым игровым контентом и возможностями для социального взаимодействия. Введение 2.5*D* графики и интеграция современных технологий искусственного интеллекта, таких как *ChatGPT*, открывают новые горизонты для создания современных игровых продуктов.

Новизна проекта заключается в уникальной комбинации визуальных и технологических решений. Применение 2.5*D* графики позволяет создать визуально привлекательную и разнообразную игровую среду, а использование *ChatGPT API* для общения с *NPC* обеспечивает динамичное и реалистичное взаимодействие, что является редкостью в аркадных фермерских симуляторах.

Проект «*Farmer's Valley*» не только соответствует современным тенденциям в индустрии видеоигр, но и вносит вклад в развитие технологий взаимодействия человека и компьютера. Использование искусственного интеллекта для управления диалогами с *NPC* может быть применено в других областях, таких как образовательные программы и интерактивные обучающие системы. Это подчеркивает значимость и практическую применимость данной работы как для игровой индустрии, так и для научно-исследовательских задач.

Основной целью дипломной работы является создание игрового приложения, которое будет сочетать в себе современные графические и интерактивные решения. Для достижения данной цели необходимо решить следующие задачи:

– провести анализ существующих методов и технологий создания приложений в жанре аркадного симулятора жизни фермера;

– определить основные функциональные компоненты приложения и разработать его архитектуру;

– реализовать основные механики игрового процесса, такие как управление фермой, экономическая система, социальная система и смена дня и ночи;

– обеспечить сохранение и загрузку игрового состояния;

– провести тестирование и верификацию приложения для обеспечения его стабильной работы и соответствия поставленным требованиям.

Таким образом, разработка игрового приложения «*Farmer's Valley*» представляет собой актуальную и перспективную задачу, соответствующую современным тенденциям в индустрии видеоигр и научных исследованиях.

Проект не только демонстрирует возможности использования современных технологий в разработке игр, но и открывает пути для дальнейших исследований и применения полученных знаний в других областях.

1 СУЩЕСТВУЮЩИЕ МЕТОДЫ РЕАЛИЗАЦИИ ИГРОВЫХ ПРИЛОЖЕНИЙ В ЖАНРЕ АРКАДНОГО СИМУЛЯТОРА ЖИЗНИ ФЕРМЕРА

1.1 Описание предметной области

По степени влияния на потребителей и вовлеченности их в интерактивное окружение, предлагаемое видеоиграми, этот сегмент уже давно выделяется среди других видов развлечений.

Разработку игр невозможно рассматривать обособленно от индустрии компьютерных игр в целом. Непосредственно создание игр – это только часть комплексной «экосистемы», обеспечивающей полный жизненный цикл производства, распространения и потребления таких сложных продуктов, как компьютерные игры.

В структуре современной игровой индустрии можно выделить следующие уровни:

– платформы;

– игровые движки;

– разработка видеоигр;

– издание и оперирование;

– популяризация и потребление.

Платформы – аппаратно-программные системы, позволяющие запускать интерактивные игровые приложения. Среди основных видов платформ можно выделить следующие:

– персональные компьютеры на базе *Windows*, *Mac/OS X* или *Linux*;

– игровые консоли (специализированные устройства для игр, *XboxOne*, *PlayStation 4*, *NintendoSwitch*;

– мобильные устройства (*iOS*, *Android*, *Windows*).

Игровые движки – программная прослойка между платформой и собственно кодом игры. Использование готового игрового движка позволяет существенно упростить разработку новых игр, удешевить их производство и существенно сократить время до запуска. Кроме того, современные игровые движки обеспечивают кроссплатформенность создаваемых продуктов. Из наиболее продвинутых движков можно выделить: *Unity 3D*, *Unreal Development Kit*, *CryENGINE 3 Free SDK*.

Разработка игр. Большое количество компаний и независимых команд занимаются созданием компьютерных игр. В разработке участвует специалисты разных профессий: программисты, гейм-дизайнеры, художники, *QA* специалисты и др. К разработке крупных коммерческих игровых продуктов привлекаются большие профессиональные команды. Стоимость разработки

подобных проектов может составлять десятки миллионов долларов. Однако вполне успешные игровые проекты могут воплощаться и небольшими командами энтузиастов. Этому способствует присутствие на рынке большого количества открытых и распространенных платформ, качественных и практически бесплатных движков, площадок по привлечению «народных» инвестиций (краудфандинг) и доступных каналов распространения.

Издание и оперирование игр. Распространением игр или оперированием (в случае с *MMO*) занимаются, как правило, не сами разработчики, а издатели. При этом издатели (или операторы) локализуют игры, взаимодействуют с владельцами платформ, проводят маркетинговые компании, разворачивают инфраструктуру, обеспечивают техническую и информационную поддержку выпускаемым играм. Для средних и небольших игровых продуктов данный уровень практически не доступен. Такие продукты, как правило, сами разработчики выводят на рынок, напрямую взаимодействуя с платформами.

Популяризация. Специализированные средства массовой информации всегда являлись мощным каналом донесения информации до пользователей. Сейчас наиболее эффективным и широко представленным направлением СМИ являются информационные сайты, посвященные игровой тематике. Игровые журналы, долгое время выступавшие главным источником информации об играх, в настоящее время уступили свое место интернет-ресурсам. Специализированные выставки все еще остаются важным информационными площадками для игровой индустрии (*E3*, *GDC*, *Gamescom*, ИгроМир, *DevGamm*). Прямое общение прессы и игроков с разработчиками, обмен опытом между участниками рынка, новые контакты – вот то, что предлагают конференции и выставки в концентрированной форме. Еще один важный канал донесения полезной информации до игроков – это ТВ-передачи, идущие как в формате классического телевещания, так и на множестве каналов видео-контента.

Игроки – это основной источник прибыли для игровых продуктов. Но в современном мире наиболее активные игроки стали существенной движущей силой в популяризации игр и отчасти в расширении контента.

1.2 Жанр аркадного фермерского симулятора

Жанр игр про ферму включает в себя уникальные особенности и подходы к разработке. Создание игры в этом жанре требует не только технических навыков, но и понимания аспектов, характерных для фермерской тематики. Некоторые ключевые этапы разработки фермерской игры включают в себя концепцию, дизайн, программирование, тестирование и выпуск игры.

Разработчики фермерских игр должны учитывать разнообразные аспекты, включая управление фермой, выращивание культур, разведение скота, улучшение оборудования и многое другое. Важно создать баланс между реализмом и увлекательным игровым процессом, чтобы игроки могли насладиться аутентичным опытом фермерской деятельности.

Фермерские игры обычно имеют вид сверху вниз и позволяют игрокам управлять всеми аспектами фермерского хозяйства. Игроки могут посеять и ухаживать за растениями, строить и улучшать фермерские постройки, обрабатывать поля, управлять скотом и многое другое. Цель игры может варьироваться от создания прибыльного фермерского хозяйства до выживания в форс-мажорных обстоятельствах.

Разработка игр в жанре фермерских симуляторов требует специфических навыков и инструментов. Этот жанр игр подразумевает создание виртуальных ферм, где игроки могут управлять аспектами фермерского хозяйства, от посевов до ухода за животными.

Создание фермерской игры начинается с концепции, где разработчики определяют основные механики и особенности игрового процесса. Затем идет проектирование элементов игры, включая графику, интерфейс, уровни сложности и систему управления. Программирование игры включает в себя создание логики игровых механик, взаимодействия объектов на ферме и реализацию анимаций.

Для разработки фермерских игр часто используются специализированные игровые движки, такие как *Unity* или *Godot*. Эти инструменты предоставляют разработчикам возможность создавать детальные *2D* и *3D* модели ферм, животных, растений, а также реализовывать сложные игровые системы, например, учет погодных условий или сезонности.

Фермерские симуляторы предлагают игрокам широкий спектр деятельности, включая посев и уход за культурами, разведение и уход за животными, строительство и расширение фермерских построек, покупку нового оборудования и техники. Важным аспектом игры является реализм фермерской деятельности, отраженный в визуальных и звуковых эффектах, а также в поведении растений и животных.

Игры в жанре фермерских симуляторов часто представлены в виде сверху вниз, что позволяет игрокам получить обзор на всю ферму и ее окрестности. Важным аспектом таких игр является создание уникальной атмосферы и стиля фермы, чтобы игроки могли погрузиться в виртуальный мир сельского хозяйства.

Фермерские игры предлагают игрокам разнообразие сценариев и возможностей развития. Игроки могут выбирать, какие культуры выращивать, какими методами ухаживать за животными, как распределять ресурсы и многое

другое. Это позволяет каждой игре быть уникальной и дать разработчикам возможность создать идеальную ферму по своему вкусу.

1.3 Аналитический обзор представителей игр жанра аркадного фермерского симулятора

В этом подразделе представлен аналитический обзор наиболее значимых и популярных представителей жанра аркадного фермерского симулятора. Цель этого обзора – выявить ключевые черты и механики, которые способствовали успеху данных игр, а также понять, какие аспекты их дизайна и игрового процесса привлекают игроков. Рассмотрение успешных примеров позволит выделить лучшие практики и тенденции в жанре, что станет основой для дальнейшего развития собственной игры.

Серия игр *Farming Simulator* представляет собой значительное явление в жанре аркадных фермерских симуляторов. Начав своё существование в 2008 году, эта серия продолжает выпускаться и по сей день, демонстрируя высокий уровень популярности среди игроков. Основная концепция игр этой серии заключается в симуляции сельскохозяйственного труда, где игроку предоставляется возможность заниматься разнообразными видами деятельности, включая земледелие, животноводство, лесоводство и другие аспекты фермерской жизни.

Farming Simulator выделяется своей реалистичностью и широким спектром возможностей. Игроки могут выбирать разные направления в сельском хозяйстве, будь то выращивание сельскохозяйственных культур или разведение животных, а также приобретать и использовать разнообразное оборудование. Игра предлагает детализированную симуляцию, в которой каждое действие требует тщательного планирования и выполнения.

Обучение и интерфейс. Для новичков в игре предусмотрены обучающие задания, которые помогают освоить основные механики игры и понять, как эффективно управлять фермой. Ветераны серии, в свою очередь, могут использовать свой накопленный опыт для оптимизации процессов и максимизации урожайности.

Разнообразие деятельности. Игрокам предоставляется широкий спектр занятий, начиная от вспашки и посева полей и заканчивая уходом за животными и лесозаготовками. В игре представлено множество культур для выращивания, таких как пшеница, ячмень, кукуруза, картофель и другие. Каждая культура требует особого подхода к уходу и удобрению.

Экономические аспекты. Помимо земледелия, игроки также могут заниматься торговлей, покупая и продавая оборудование, а также продавать продукцию своей фермы. Важным элементом игрового процесса является

логистика, включающая перевозку урожая и управление запасами. В последних версиях игры были добавлены поезда, которые значительно облегчают транспортировку, но требуют дополнительных затрат и навыков управления.

Лесное хозяйство. В версии *Farming Simulator* была введена возможность заниматься лесоводством, что добавляет ещё один интересный элемент в игровой процесс. Этот аспект игры также требует специфических знаний и навыков для успешного ведения бизнеса.

Игры серии *Farming Simulator* учат игроков основам фермерства, предлагая погружение в процесс и понимание всех тонкостей сельскохозяйственного труда. Эти игры не только развлекают, но и предоставляют возможность получить полезные навыки, которые могут быть применимы и в реальной жизни.

Таким образом, серия *Farming Simulator* является не просто развлекательным продуктом, но и обучающим инструментом, который помогает игрокам лучше понять и оценить труд фермеров, предоставляя глубокий и увлекательный игровой опыт.

На рисунке 1.1 представлен кадр из игры *Farming Simulator*.

Рисунок 1.1 – Кадр игры из *Farming Simulator*

Игра *Stardew Valley*, созданная и разработанная всего одним человеком, Эриком Бароном, является значимым представителем жанра аркадных фермерских симуляторов. Вышедшая в 2016 году, она быстро завоевала популярность благодаря своему уникальному сочетанию различных игровых механик и богатого контента. Игра сочетает элементы фермерства, ролевых игр и симуляторов жизни, создавая уникальный и увлекательный игровой опыт.

Сюжет и персонажи. В основе сюжета лежит история персонажа, который получает в наследство ферму от своего дедушки. Игрок может настроить внешность и пол персонажа, однако эти параметры не влияют на игровой процесс. Главный герой свободен в своих действиях: можно заниматься фермерством, исследовать пещеры, общаться с местными жителями или выполнять квесты.

Игровая свобода. *Stardew Valley* предоставляет игрокам значительную свободу выбора. Игроки могут решать, каким образом управлять фермой, какие культуры выращивать, как ухаживать за животными и какие задания выполнять. В игре присутствуют элементы дэйтинг-симулятора, позволяющие строить отношения с другими персонажами и даже вступать в брак.

На рисунке 1.2 представлен кадр из игры *Stardew Valley*.

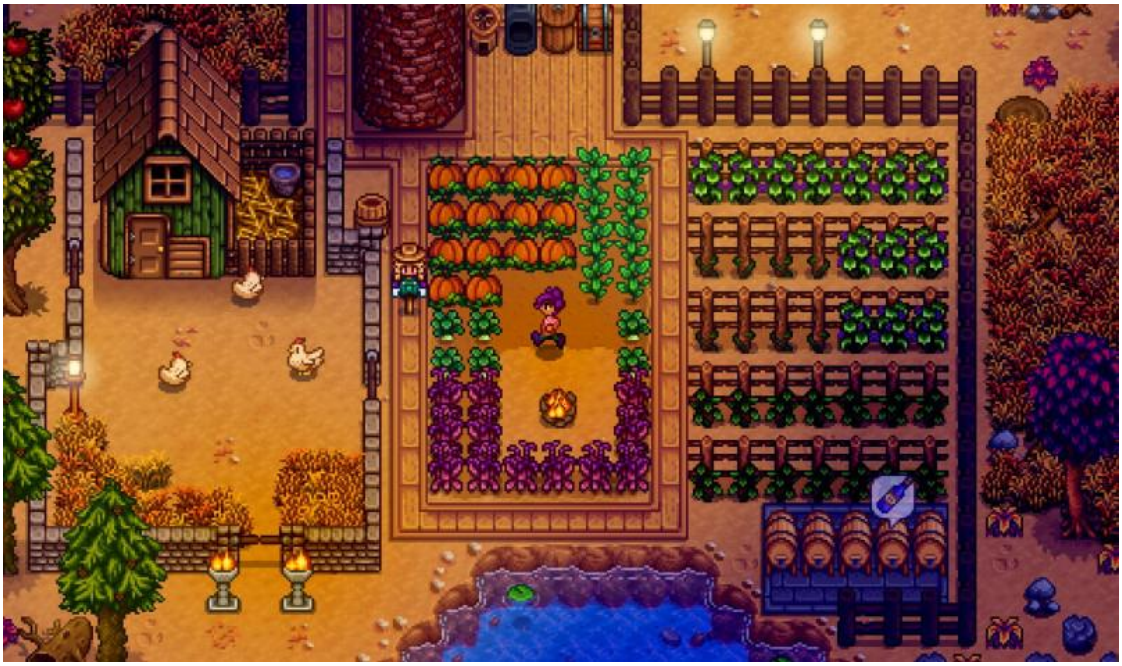

Рисунок 1.2 – Кадр из игры *Stardew Valley*

Ролевая система и развитие навыков. В отличие от многих других фермерских симуляторов, *Stardew Valley* включает ролевую систему, в которой игроки могут развивать навыки своего персонажа. Каждый вид деятельности, будь то использование мотыг или лейки, приносит очки опыта, которые улучшают соответствующие навыки и открывают новые возможности.

Сезонность и планирование. Игра симулирует смену времён года, что влияет на доступность различных культур и видов деятельности. Летом можно выращивать определенные растения и собирать мед, а зимой заниматься консервацией продуктов. Важно учитывать прогноз погоды и планировать свои действия, чтобы максимально эффективно использовать ресурсы и время.

Стратегическое планирование. Успешное ведение фермы требует тщательного планирования. Игрокам нужно учитывать бюджет, выбирать и подготавливать участки для посадки, а также защищать урожай от вредителей с помощью пугал. В игре существует ограничение по времени и энергии персонажа, что добавляет элемент стратегического планирования в повседневные задачи.

Многопользовательский режим. Игра поддерживается и развивается своим разработчиком, и в последние годы была добавлена возможность многопользовательской игры.

Stardew Valley предлагает глубокий и захватывающий игровой опыт, сочетая элементы фермерства, ролевой игры и симулятора жизни. Игра позволяет игрокам чувствовать себя настоящими профессиональными садоводами, наслаждаться процессом своей деятельности и достигнутыми результатами. Благодаря своему уникальному подходу к жанру и постоянной поддержке со стороны разработчика, *Stardew Valley* продолжает оставаться одной из самых популярных и любимых игр в жанре аркадных фермерских симуляторов.

1.4 Игровой движок *Unity*

Unity – это профессиональный игровой движок, позволяющий создавать видеоигры для различных платформ.

Любой игровой движок предоставляет множество функциональных возможностей, которые задействуются в различных играх. Реализованная на конкретном движке игра получает все функциональные возможности, к которым добавляются ее собственные игровые ресурсы и код игрового сценария.

Приложение *Unity* предлагает моделирование физических сред, карты нормалей, прегражденне окружающего света в экранном пространстве (*Screen Space Ambient Occlusion*, *SSAO*), динамические тени. Список можно продолжать долго. Подобные наборы функциональных возможностей есть во многих игровых движках, но *Unity* обладает двумя основными преимуществами перед другими передовыми инструментами разработки игр. Это крайне производительный визуальный рабочий процесс и сильная межплатформенная поддержка. Визуальный рабочий процесс – достаточно уникальная вещь, выделяющая *Unity* из большинства сред разработки игр. Альтернативные инструменты разработки зачастую представляют собой набор разрозненных фрагментов, требующих контроля, а в некоторых случаях библиотеки, для работы с которыми нужно настраивать собственную интегрированную среду разработки (*Integrated Development Environment, IDE*), цепочку сборки и прочее

в этом роде. В *Unity* же рабочий процесс привязан к тщательно продуманному визуальному редактору. Именно в нем будут компоновать сцены будущей игры путем связывания игровых ресурсов и код в интерактивные объекты. *Unity* позволяет быстро и рационально создавать профессиональные игры, обеспечивая невиданную продуктивность разработчиков и предоставляя в их распоряжение исчерпывающий список самых современных технологий в области видеоигр.

Редактор особенно удобен для процессов с последовательным улучшением, например, циклов создания прототипов или тестирования. Даже после запуска игры остается возможность модифицировать в нем объекты и двигать элементы сцены. Настраивать можно и сам редактор. Для этого применяются сценарии, добавляющие к интерфейсу новые функциональные особенности и элементы меню.

На рисунке 1.3 представлен интерфейс игрового движка *Unity*.

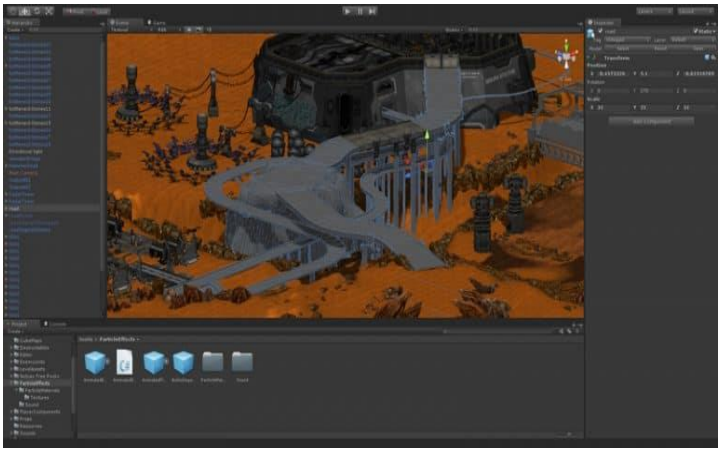

Рисунок 1.3 – Интерфейс движка *Unity*

Дополнением к производительности, которую обеспечивает редактор, служит сильная межплатформенная поддержка набора инструментов *Unity*. В данном случае это словосочетание подразумевает не только места развертывания (игру можно развернуть на персональном компьютере, в интернете, на мобильном устройстве или на консоли), но и инструменты разработки (игры создаются на машинах, работающих под управлением как *Windows*, так и *Mac OS*) [1]. Эта независимость от платформы явилась результатом того, что изначально приложение *Unity* предназначалось исключительно для компьютеров *Mac*, а позднее было перенесено на машины с операционными системами семейства *Windows*. Первая версия появилась в 2005 году, а к настоящему моменту вышли уже пять основных версий (с множеством небольших, но частых обновлений). Изначально разработка и развертка поддерживались только для машин *Mac*, но через несколько месяцев вышло

обновление, позволяющее работать и на машинах с *Windows*. В следующих версиях добавлялись все новые платформы развертывания, например межплатформенный веб-плеер в 2006-м, *iPhone* в 2008- м, *Android* в 2010-м и даже такие игровые консоли, как *Xbox* и *PlayStation*. Позднее появилась возможность развертки в *WebGL* – новом фреймворке для трехмерной графики в веб-браузерах. Немногие игровые движки поддерживают такое количество целевых платформ развертывания, и ни в одном из них развертка на разных платформах не осуществляется настолько просто. Дополнением к этим основным достоинствам идет и третье, менее бросающееся в глаза преимущество в виде модульной системы компонентов, которая используется для конструирования игровых объектов. «Компоненты» в такой системе представляют собой комбинируемые пакеты функциональных элементов, поэтому объекты создаются как наборы компонентов, а не как жесткая иерархия классов. В результате получается альтернативный (и обычно более гибкий) подход к объектно-ориентированному программированию, в котором игровые объекты создаются путем объединения, а не наследования [2].

Оба подхода схематично показаны на рисунке 1.4.

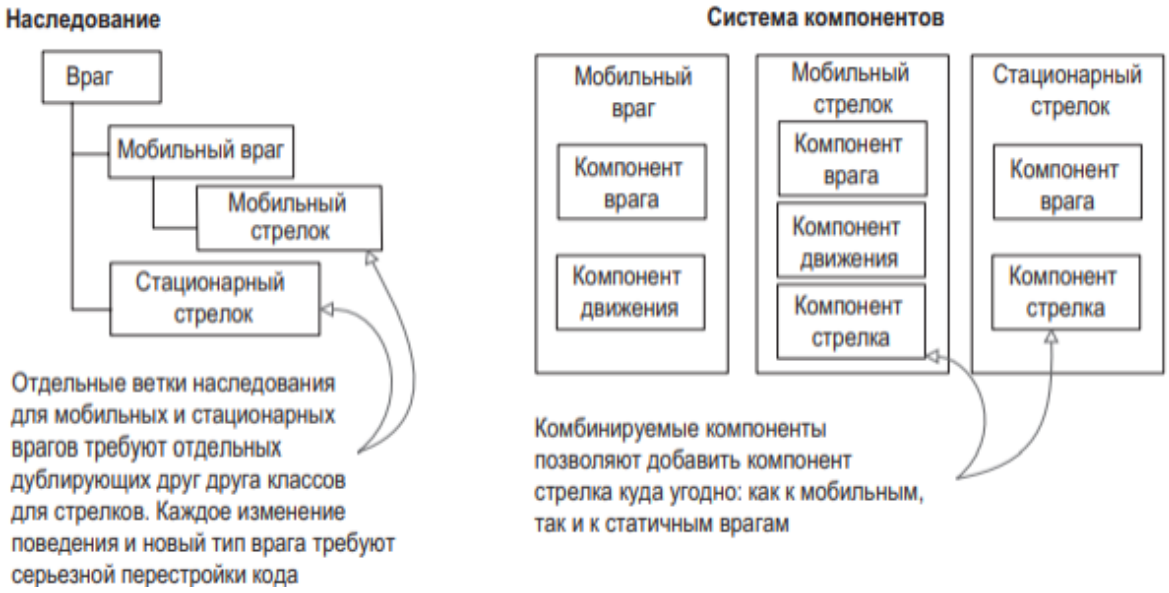

Рисунок 1.4 – Сравнение наследования с компонентной системой

Каждое изменение поведения и новый тип врага требуют серьезной перестройки кода. Комбинируемые компоненты позволяют добавить компонент стрелка куда угодно: как к мобильным, так и к статичным врагам. В компонентной системе объект существует в горизонтальной иерархии, поэтому объекты состоят из наборов компонентов, а не из иерархической структуры с наследованием, в которой разные объекты оказываются на разных ветках дерева. Разумеется, ничто не мешает написать код, реализующий вашу

собственную компонентную систему, но в *Unity* уже существует вполне надежный вариант такой системы, органично встроенный в визуальный редактор. Эта система дает возможность не только управлять компонентами программным образом, но и соединять и разрывать связи между ними в редакторе. Разумеется, возможности не ограничиваются составлением объектов из готовых деталей; в своем коде вы можете воспользоваться наследованием и всеми наработанными на его базе шаблонами проектирования [3, с.283].

1.5 Игровой движок *Unreal Engine*

Unreal Engine (*UE*) – игровой движок, разрабатываемый и поддерживаемый компанией *Epic Games*.

Движком называют рабочую среду, позволяющую управлять всей системой элементов, из которых состоит игра.

Сегодня движок *Unreal Engine* активно применяется для разработки простых казуальных игр для смартфонов и планшетов, а также для создания полноценных высокобюджетных игр, рассчитанных на массовую аудиторию (их называют *AAA*-проектами). При этом не потребуется самостоятельно писать код, т. к. система визуального создания скриптов *Blueprints Visual Scripting* значительно упрощает задачу. Если же разработчик желает прописать игровую логику вручную, он может использовать язык программирования *C++*.

5 апреля 2022 года *Epic Games* порадовала пользователей, представив обновленный движок *Unreal Engine 5*, анонсированный два года назад. Среди главных достоинств – максимум фотореализма, увеличенная производительность и новый интерфейс.

Unreal Engine остается популярным более 20 лет, т. к. обладает следующими достоинствами:

– широкий функционал;

– визуальное программирование;

– бесплатная лицензия;

– возможность создать кросс-платформер;

– большая база пользователей.

Epic Games решила дать разработчикам больше, чем простой инструмент – в *UE* пользователи могут начать работу даже без узкоспециализированных знаний в области языков программирования.

Для тех, кто далек от написания программного кода, корпорация предложила простую и удобную в использовании систему *Blueprints Visual Scripting*. С ее помощью можно легко создать прототип любой игры, имея минимум теоретических знаний. Конечно, умение работать с функциональным

и объектно-ориентированным программированием будет плюсом, но начать разработку геймплея в *UE* можно и без него.

На рисунке 1.5 представлен интерфейс *Unreal Engine*.

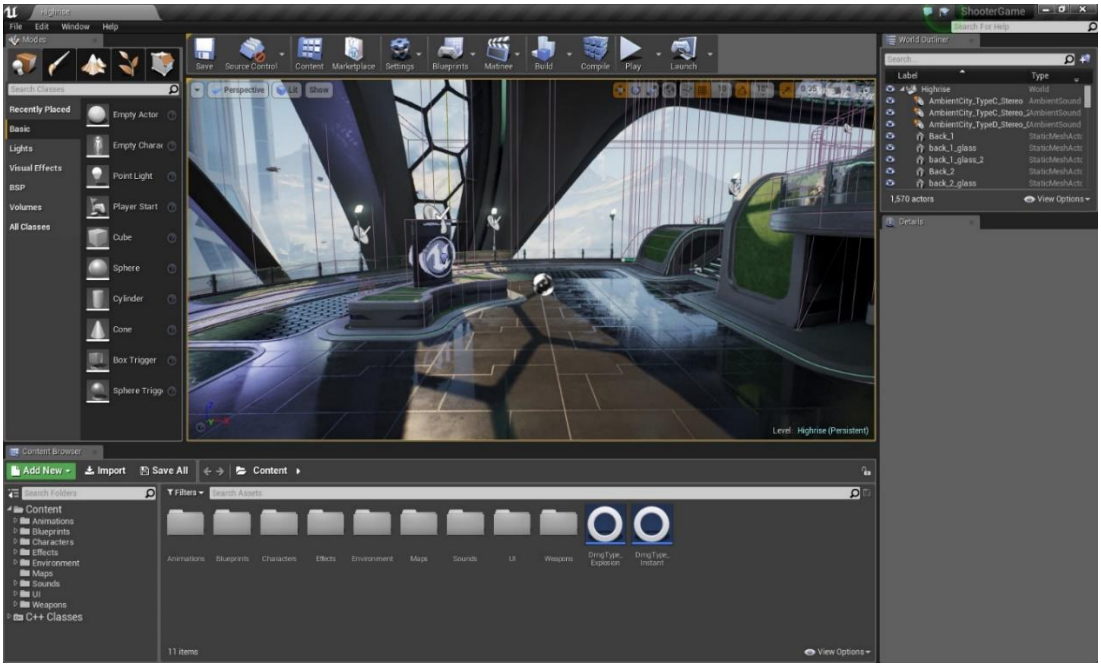

Рисунок 1.5 – Интерфейс *Unreal Engine*

Blueprints значительно проще для понимания и использования, чем *C++*, при этом их функции и возможности в большинстве случаев схожи. Однако иногда все же придется прибегнуть к кодингу: для произведения сложных математических расчетов, изменения исходного кода самого движка *UE* и ряда базовых классов проекта.

В игровом мире существуют объекты с уникальными оттенками, фактурами и физическими свойствами. В движке *UE* внешний вид зависит от настроек материалов. Цвет, прозрачность, блеск – задать можно практически любые параметры. При работе над игрой в *UE* материалы можно наносить на любые объекты, вплоть до мелких частиц. Отметим, что речь идет не просто о настройке текстур: материалы открывают более широкие возможности. К примеру, можно создавать необычные визуальные эффекты, причем *UE* позволяет делать это прямо в процессе игры.

Пользовательский интерфейс. Игроку важно не только видеть действия своего персонажа и карту, на которой он находится, но и иметь текстовую информацию, а также сведения о количестве очков, пунктах здоровья, инвентаре и т. д. С этой целью разработчики тщательно продумывают пользовательский интерфейс (*User Interface, UI*). В движке *Unreal* для создания

UI применяется *Unreal Motion Graphics* (*UMG*). Он позволяет выстраивать интуитивно понятный *UI*, выводить на экран необходимую пользователю информацию, а также менять положение кнопок и текстовых меток.

Анимация. Персонаж любой современной игры подвижен и гибок, умеет бегать и прыгать. Все это возможно благодаря анимированию. В *UE* начинающие разработчики могут импортировать уже готовые меши со скелетами персонажей и настройки анимации. Неопытных пользователей, которые желают познакомиться с ПО поближе, приятно удивит *Animation Blueprint* – скрипт, который значительно упрощает работу по созданию паттернов движений персонажа без использования кодинга.

Звук. Для полного погружения в игру недостаточно просто собрать саундтрек из десятка файлов – музыку следует подобрать по тематикам сцен, настроить уровень ее громкости, прописать и расставить по нужным местам диалоги персонажей. В *UE* можно по-разному настраивать звуковые эффекты, зацикливать музыку и модулировать тон при каждом новом воспроизведении, а также работать с несколькими эффектами одновременно. За последнее отвечает ассет *Sound Cue*.

Система частиц. Данный компонент необходим для создания визуальных эффектов. Взрывы, брызги, искры, туман, снегопад или дождь – в *UE* все это можно создать, используя систему *Cascade*.

Искусственный интеллект. В компьютерной игре существуют не только главные, но и второстепенные персонажи. Искусственный интеллект (ИИ) отвечает за их решения (увидеть действие и среагировать). Настроить ИИ в *UE* можно, используя так называемые деревья поведения, *Behavior Trees*. В простые схемы закладываются алгоритмы действий и принятия решений. Здесь не только новичкам, но и профессионалам будет удобнее работать в *Blueprints Visual Scripting*, ведь все деревья визуально напоминают простые блок-схемы. Выстроить их гораздо быстрее и проще, чем писать длинный код.

1.6 Сравнение *Unity* и *Unreal Engine*

Первая область сравнения – редакторы для создания уровней, которые очень похожи. В них есть браузеры контента для ассетов, скриптов и других файлов проекта. Игровые объекты можно перетаскивать в область сцены и, таким образом, добавлять в её иерархию.

Объекты в редакторе сцены изменяются с помощью инструментов перемещения, поворота и масштабирования – они похожи в обоих движках. Свойства *Unity*-объектов отображаются в *Inspector*, а *UE* – в части *Details*. *Jayanam* также сравнивает возможности *Unity Prefabs* с *Blueprints*.

В обоих движках есть статические меши *(static meshes*) – их можно двигать, поворачивать и масштабировать – и скелетные меши *(skeletal meshes*) – геометрические объекты, привязанные к костям скелета и используемые для анимирования персонажей. Их можно создавать в программах вроде *Blender* или *Maya*.

Анимации, включённые для скелетных мешей, также можно импортировать. В *Unity* они прикрепляются к импортированному объекту, как клипы анимации (*animation clips*), а в *UE* называются последовательностями анимации (*animation sequences*). В первом движения управляются с помощью контроллеров анимации (*animation controllers*), а во втором – по тому же принципу действуют анимационные *Blueprints*.

В обоих движках есть стейт-машины, определяющие переходы из одного состояния ассета в другое.

В *UE* система называется *Persona*, а в *Unity – Mecanim*. В них возможно применение скелетных мешей одного скелета к другим, но в *Unity* это, в основном, используется для анимирования гуманоидов. В *UE* анимации можно редактировать, в *Unity* – практически нет, особенно плохо дело обстоит с движениями гуманоидов. Движки не подходят для профессионального анимирования персонажей – лучше использовать программы вроде *Blender* или *Maya*, а результат импортировать в виде *FBX*-файлов. Прикреплённый к объектам материал добавляется в проект, но его свойства вроде шейдера или текстур придётся применять вручную [4].

Для этого в *Unity* нужно задать материалу шейдер и добавить в его слоты текстуры – карты шероховатостей, нормалей или диффузии. Собственные шейдеры придётся писать самостоятельно или с помощью сторонних инструментов вроде *Shader Forge* или *ASE*. А в *UE* встроен очень мощный редактор материалов, основанный, как и система *Blueprints*, на нодах.

Для программирования в *UE* используется язык *C++*, который не все любят из-за сложности и продолжительности компилирования. Однако *Jayanam* считает, что у движка понятный *API* и приемлемый период компиляции. В *UE* очень мощная и проработанная система визуального скриптования – *Blueprints*, с помощью которой можно достичь практически тех же результатов, что и *с C++*.

Unity поддерживает языки *C#* и *UnityScript*. *API* и его концепт очень похож на аналог из *UE*. При использовании управляемого языка вроде *C#*, программист не обязан использовать указатели (*pointers*), компилирование происходит быстро. В *Unity* нет системы визуального «скриптования», и чтобы использовать что-то подобное, разработчик вынужден покупать сторонние дополнения вроде *Playmaker*.

Для *2D*-разработки в *Unity* есть великолепные инструменты – *sprite creator*, *sprite editor* и *sprite packer*. *UE* также поддерживает спрайты в *Paper 2d*, но решения из *Unity* мощнее, кроме того, в последнем есть отдельный физический движок для *2d*-объектов.

В *UE* встроен *постпроцессинг*. К сцене можно применять *bloom*-эффект, тонирование и антиалиасинг как глобально, так и к отдельным её частям (при помощи компонента *PostProcessVolume*).

В *Unity* есть стек постпроцессинга, который можно скачать из магазина ассетов движка. Система менее гибкая, чем в *UE* – эффекты применяются только стеком или скриптами к камере.

Sequencer в *UE* можно использовать для создания синематиков. Это мощный инструмент, работающий по принципу добавления объектов на временную шкалу. К настоящему моменту в *Unity* 5.6 нет системы для синематиков, но *timeline*-редактор добавили в *Unity* 2017.

Для реализации приложения был выбран игровой движок *Unity*, т. к. он обладает наибольшим инструментарием и информационной базой, которые в один момент являются удобными и простыми [4, с. 441].

1.7 Требования к проектируемому программному обеспечению

Целью разработки является создание программного обеспечения (ПО) для аркадного фермерского симулятора, который предоставит пользователю возможность управлять виртуальной фермой, заниматься сельскохозяйственными работами, взаимодействовать с другими персонажами и развивать свои навыки в различных аспектах фермерской деятельности.

Программное обеспечение должно включать следующие функциональные возможности:

– создание нового игрового состояния: возможность выбрать название нового игрового состояния;

– загрузка выбранного игрового состояния: возможность выбора сохраненного игрового состояния;

– управление фермой: посадка и уход за различными культурами, управление ресурсами и инвентарем;

– социальное взаимодействие: общение с *NPC* (неигровыми персонажами);

– экономическая система: покупка и продажа продуктов, управление бюджетом фермы;

– атмосферное составляющее: смена дня и ночи;

Игровые механики включают:

1. посадка и сбор урожая:

– механика вспашки земли;

– посадка семян различных культур;

– полив растений;

– сбор урожая по мере созревания;

2. механика загрузки и сохранения игрового состояния;

3. общение с *NPC*;

4. взаимодействие с активным, вспомогательным инвентарем, а также возможность хранения предметов в сундуке или сундуках;

5. экономика:

– продажа продукции;

– покупка семян;

6. смена дня и ночи;

7. возможности настройки игровой среды:

– смена разрешения экрана;

– смена полноэкранного режима на оконный и наоборот;

– управление громкостью музыки и звуковыми эффектами;

– смена управления;

8. сохранение и загрузка настроек игровой среды.

Графические и визуальные элементы включают следующие пункты:

1. стиль графики: 2.5*D* графика (создание 2*D*-спрайтов на основе 3*D* моделей с маской освещения и картами нормалей для достижения глубины и детализации, яркие насыщенные цвета для создания уютной и привлекательной атмосферы;

2. персонажи: спрайты персонажей с анимацией движения;

3. окружение: спрайты растительности, зданий и объектов;

4. интерфейс пользователя: интуитивно понятный и легко читаемый интерфейс;

5. эффекты и анимация:

– анимации различных действий персонажа (сбор урожая, вспашка, полив);

– визуальные эффекты для улучшения восприятия;

6. звуковые эффекты и музыка:

– эффекты для действий;

– фоновая музыка, создающая атмосферу спокойствия и умиротворенности.

Входные данные:

– данные об игровом состоянии (игровая валюта);

– команды от пользователя (например, действия по посадке, сбору урожая).

Условия эксплуатации:

– программное обеспечение должно работать на платформе *Windows*;

– поддержка разрешений экрана от *1024* пикселей *на 768* пикселей и выше;

– требования к минимальной производительности: 4 ГБ оперативной памяти, процессор с частотой 2 ГГц, видеокарта с поддержкой *OpenGL* 3.0;

– обеспечение возможности сохранения и загрузки игрового процесса.

Требования к надежности и быстродействию:

– программное обеспечение должно обеспечивать стабильную работу без сбоев и утечек памяти;

– быстродействие должно обеспечивать плавный игровой процесс с частотой не менее 30 кадров в секунду.

В качестве игрового движка был выбран *Unity*, так как он предоставляет мощные инструменты для разработки *2D* игр, обеспечивает поддержку множества платформ и имеет обширное сообщество разработчиков. Среда разработки *Visual Studio*, интегрированная в *Unity*, для удобства написания и отладки кода.

Инструментарий для разработки графики был выбран в пользу *Blender* для создания 3*D* моделей и *Adobe Photoshop* с *Aseprite* для обработки графических элементов.

Система контроля версий *Git* для отслеживания и сохранения изменений в процессе работы над проектом.

Таким образом, разработка программного обеспечения будет осуществляться с использованием проверенных и современных технологий, обеспечивающих надежность, быстродействие и удобство работы как для разработчиков, так и для конечных пользователей.

2 АРХИТЕКТУРА ИГРОВОГО ПРИЛОЖЕНИЯ «*FARMER'S VALLEY*»

2.1 Общие положения об архитектуре игрового приложения

Представление надежной и эффективной архитектуры приложения является критическим компонентом в разработке современных приложений. Понимание важности архитектуры и структуры приложений пришло к разработчикам игр не сразу. Это был пошаговый процесс, основанный на накопленном опыте, столкновениях с проблемами и поиске эффективных решений для их решения.

Главная цель разработки архитектуры приложения – определить структуру приложения, выделить его компоненты и определить взаимодействие между ними в приложении. Архитектура позволяет увидеть «общую картинку» приложения и оценить его целостность, а также обеспечить гибкость и расширяемость приложения в будущем.

Основные причины, по которым нужно разрабатывать архитектуру приложения, состоят в следующем:

– четкое определение структуры приложения и его компонентов;

– оценка сложности приложения и определение возможных рисков и проблем;

– обеспечение гибкости и расширяемости приложения в будущем;

– упрощение процесса разработки за счет более четкого понимания того, что нужно разрабатывать и как компоненты приложения взаимодействуют друг с другом;

– улучшение качества кода и снижение затрат на его сопровождение.

Разработка архитектуры может помочь определить, какие компоненты нужны для реализации игры, как они будут взаимодействовать друг с другом, какие алгоритмы нужны для обработки данных и как эти компоненты будут связаны с игровым движком. Это поможет упростить процесс разработки и обеспечить более гибкую и расширяемую архитектуру для будущих доработок и улучшений игры.

Чтобы правильно построить архитектуру приложения стоит использовать метод декомпозиции для правильной оценки задачи и последующем её разбиении на отдельные компоненты с целью оптимизации разработки приложения и дальнейшей поддержки.

Декомпозиция – это разделение большого и сложного на небольшие простые части. При постановке задач декомпозировать – значит разбить абстрактную большую задачу на маленькие задачи, которые можно легко оценить.

2.2 Основные функции игрового приложения «*FARMER'S VALLEY*»

Для разработки архитектуры приложения важно выделить его основные функции. С помощью разделения задачи методом декомпозиции можно выделить подзадачи, которые будет представлять приложение. На рисунке 2.1 представлен результат выделения функциональных задач.

Рисунок 2.1 – Основные функциональные задачи

Как видно из рисунка 2.1, удалось определить основной функционал приложения в качестве задач и подзадач с некоторой степенью конкретизации в определенных случаях.

Были выделены следующие задачи и подзадачи:

а) обновление игровых объектов;

б) отрисовка графики;

в) визуальные и звуковые эффекты;

г) социальное взаимодействие:

– общение с *NPC*;

д) экономическая система:

– покупка и продажа продукции;

е) управление фермой:

– посадка урожая;

– сбор урожая;

– уход за растениями;

ж) управление ресурсами:

– активный инвентарь;

– дополнительный или вспомогательный инвентарь;

– сундук или сундуки;

з) смена дня и ночи;

к) сохранение и загрузка игрового состояния;

л) настройка игровой среды;

м) сохранение и загрузка настроек игровой среды.

Описанные функции или подзадачи можно рассматривать в качестве компонентов, которые будут реализовывать ту или иную задачу.

За компоненты, описанные в пункте а и б, будет отвечать платформа *Unity*. Она будет обновлять игровые компоненты и рисовать графику.

Компоненты, описанные в пунктах б и в, будут представлять ресурсные компоненты. Они являются своеобразными исходными данными, которые будут находиться в рабочей области платформы и импортироваться в другие компонеты.

Схема инициализации компонентов, описанных в пунктах в и г показана на рисунке 2.2.

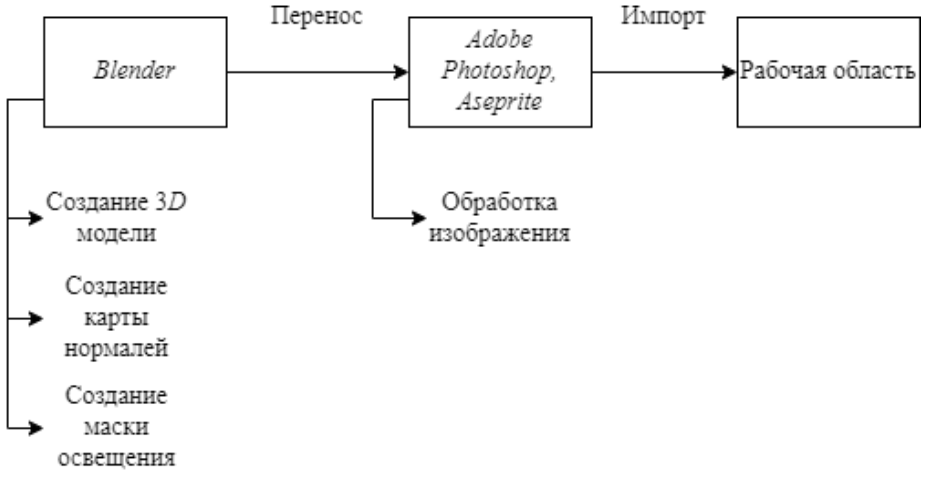

Рисунок 2.2 – Схема инициализации компонентов, описанных в пунктах б и в

Остальные функциональные компоненты следует выделять программными компонентами, которые будут реализовываться средствами написания кода.

Оставшиеся пункты можно разделить на компоненты, реализующие игровое составляющее игрового процесса или механики, другие компоненты будут отвечать за инициализацию и настройку игрового процесса.

2.3 Структурные элементы игрового приложения «*FARMER'S VALLEY*»

При проектировании игрового приложения, возникает необходимость определения основных структурных элементов приложения.

Основные структурные элементы, которые будут использоваться в проекте, включают следующие модули:

– модуль управления конструированием игрового мира;
– модуль управления фермой;
– модуль социального взаимодействия;
– модуль экономической системы;
– модуль управления инвентарем;
– модуль визуальных и звуковых эффектов;
– модуль времени;
– модуль сохранения и загрузки;
– модуль управления игровыми сценами;
– модуль инициализации.

Пользователь будет взаимодействовать с этими модулями путем взаимодействия с *GUI* (*Graphical User Interface*).

Хорошей практикой является разделять работу модуля на работу с *GUI* и логику его функционирования.

Схема модуля показана на рисунке 2.3.

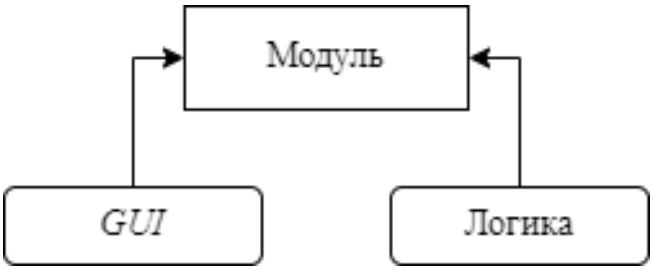

Рисунок 2.3 – Схема работы модуля

Пользователь будет взаимодействовать с вышеописанными модулями через графический пользовательский интерфейс (*GUI*). Разделение работы модуля на взаимодействие с *GUI* и логическую составляющую является хорошей практикой, позволяющей улучшить модульность и управляемость

кода. *GUI* будет служить мостом между пользователем и логикой приложения, предоставляя интуитивно понятные элементы управления и обратную связь.

Модуль управления конструированием игрового мира отвечает за создание и изменение игрового мира. За установку его границ и настройку положения визуальных игровых элементов.

Модуль управления фермой отвечает за следующие аспекты управления фермой:

– посадка урожая;

– сбор урожая;

– рыхление земли;

– полив растений;

– сбор продукции по мере роста.

Модуль социального взаимодействия отвечает за инициализацию параметров разговора и оптимизацию расходуемых ресурсов. Реализует общение с *NPC* в формате чата с помощью *GUI*.

Модуль экономической системы отвечает за управление экономическими аспектами игры, такими как покупка необходимой продукции и продажа продукции.

Модуль управления инвентарем отвечает за управлением инвентарем игрока, включая *drag and drop* систему, перемещение объектов между другими объектами инвентаря и сундуками.

Модуль визуальных эффектов отвечает за создание определенного эффекта при взаимодействии с игровыми объектами.

Модуль звуковых эффектов отвечает за воспроизведение звуковых эффектов в определенное время при определенном действии.

Модуль времени отвечает за смену времени суток и отображение текущего времени суток в привычном формате.

Модуль сохранения и загрузки отвечает за сохранение и загрузку игрового состояния. Сохраняет и загружает положение игровых элементов, а также состояние других модулей.

Модуль управления игровыми сценами отвечает за смену игровых сцен.

Модуль инициализации отвечает за распределение необходимых зависимостей для других модулей и является точкой входа при запуске сцены.

2.4 Устройство внутренней архитектуры и схема разделения данных

В проекте внутренняя архитектура основана на модульном подходе, где каждый модуль отвечает за определённую часть функциональности игры. Это позволяет улучшить модульность, тестируемость и расширяемость приложения.

В целях оптимизации программного продукта следует разделить игровое приложение на несколько сцен с сокращением количества игровых объектов на одной сцене.

Таким образом, игровое приложение будет запускаться с начальной сцены (главное меню) и из начальной сцены запускать игровую сцену. На рисунке 2.4 представлена схема сцен.

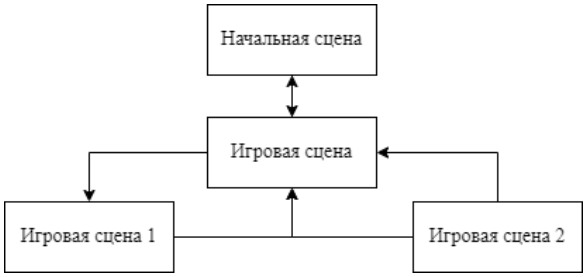

Рисунок 2.4 – Схема сцен

Приведенная схема запуска сцен показывает, что нахождение на сцене всех модулей совсем не обязательно.

На рисунке 2.5 представлена схема наличия модулей на начальной и игровой сцене.

Рисунок 2.5 – Схема наличия модулей на начальной и игровой сцене

После запуска сцены первым делом запускается модуль инициализации.

Каждая сцена инициализируется в своей точке входа, за которую отвечает модуль инициализации.

На рисунке 2.6 показана схема инициализации программных модулей после запуска сцены.

Рисунок 2.6 – Схема инициализации программных модулей после запуска сцены

Когда модуль инициализации закончит инициализировать остальные модули, то игровое состояние придет в рабочий режим, тогда работа модулей обновляется с помощью основного цикла игры. В случае, если модуль инициализации по каким-то причинам не сможет проинициализировать остальные модули, то запуск сцены будет отменен.

В этом разделе следует разобрать работу модуля инициализации, так как его функционал обусловлен использованием фрэймворка *Zenject*. Этот фреймворк содержит много полезного функционала, однако стоит обратить внимание только на его реализацию архитектурного паттерна проектирования *Dependency Injection*.

Перед тем, как перейти к описанию архитектурного паттерна проектирования *Dependency Injection*, стоит начать с описания понятия инверсии управления (*IoC*).

Инверсия управления (*Inversion of Control*, *IoC*) – это ключевой принцип объектно-ориентированного программирования, который применяется для снижения связности (зацепления) в программном обеспечении. Данный подход также представляет собой архитектурное решение для интеграции, которое упрощает расширение функциональности системы за счет того, что поток управления программы контролируется фреймворком.

Паттерн внедрения зависимостей (*Dependency Injection Pattern*) – это дизайн-шаблон, который использует принцип *IoC*, чтобы обеспечить, что класс

не принимает участие в создании экземпляра зависимого класса и не управляет его жизненным циклом. Иными словами, процесс создания класса и инициализации его переменных, поступающих через конструктора или методы, полностью возлагается на платформу, то есть на более высокий уровень.

В рамках реализации архитектурного паттерна *Dependency Injection* во фреймворке *Zenject* используется инструмент *Dependency Injection* (*DI*) *Container*.

Dependency Injection (*DI*) *Container* – это инструмент, который разрешает (*resolve*) зависимости для их внедрения. Проще говоря, это некий «черный ящик», в котором можно зарегистрировать классы (интерфейсы и их реализации) для последующего разрешения (*resolve*) в нужных местах, таких как конструкторы. Стоит отметить, что внедрение зависимостей возможно не только через конструкторов, но и через методы и свойства. Хотя внедрение через конструктор является самым распространённым.

Для чего нужен *DI*-контейнер:

– создание экземпляров объектов, включая разрешение зависимостей, в том числе и иерархических;

– управление жизненным циклом объектов (*lifetime*);

– доступ из любого места в программе, практически в любом конструкторе.

Таким образом, *DI*-контейнер играет важную роль в создании и управлении зависимостями, облегчая разработчикам задачу по обеспечению гибкости и модульности кода, а также упрощая процесс тестирования и сопровождения приложений.

Для обеспечения эффективного взаимодействия модулей используется схема разделения данных. Модули обмениваются данными через четко определенные интерфейсы, что позволяет поддерживать их независимость и облегчает тестирование и отладку.

Данные в приложении разделяются на несколько типов:

– игровые данные;

– экономические данные;

– визуальные и звуковые данные;

– временные данные;

– данные для загрузки и сохранения.

Игровые данные хранятся в оперативной памяти и относятся к текущему состоянию игрового мира, объектов и персонажей.

Экономические данные относятся к экономике игры и включают данные о предметах инвентаря и транзакции. К примеру, количество и типы ресурсов, стоимость покупки и продажи.

Визуальные данные относятся к визуальному представлению игры, к примеру, настройка графики и активные визуальные эффекты.

Звуковые данные относятся к звуковым эффектам и фоновой музыке.

Данные сохранений относятся к данным, которые используются для загрузки и сохранения игрового состояния.

В качестве модели данных определяются основные сущности и связи между ними, такие как игрок, *NPC*, предметы. Каждая сущность представлена отдельным классом или структурой данных.

На рисунке 2.7 представлена схема взаимодействия сущностей.

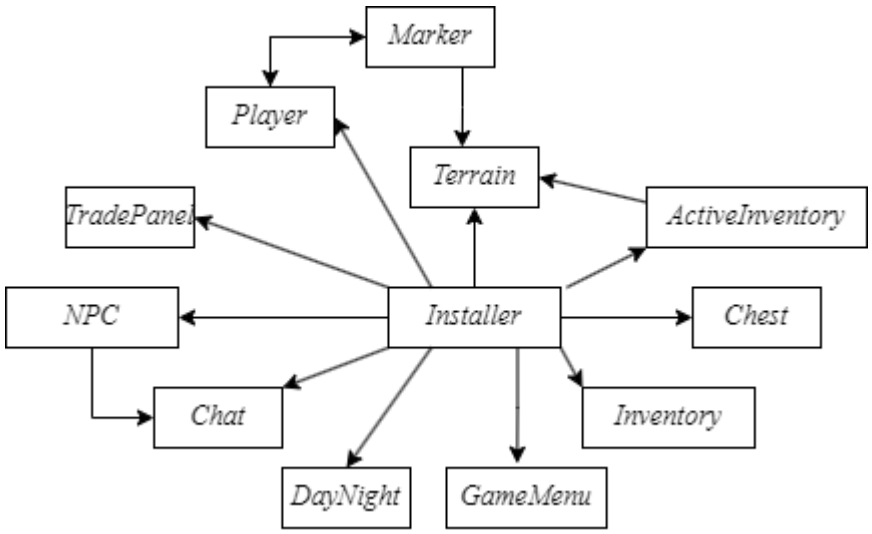

Рисунок 2.7 – Схема взаимодействия сущностей

Сущность *Player* представляет собой информацию о местоположении игрового персонажа, текущей анимации.

Сущность *Marker*, которая представляет собой цель взаимодействия, содержит информацию о своем местоположении по отношению к игроку.

Сущность «*Terrain*» содержит данные о том, где игрок установил семя, полил территорию и в какой стадии роста находится семя.

Сущность *InventoryItem* содержит данные о предмете инвентаря.

Сущность *Inventory* представляет собой хранилище данных о всех имеющихся предметах инвентаря.

Сущность *ActiveInventory* по своей сути является сущностью инвентаря, только использует данные о вводе пользователя, чтобы взаимодействовать с сущностью «территория».

Сущность «*Chest*» также по своей сути является сущностью инвентаря, только используется исключительно для хранения предметов.

Сущность *NPC* представляет собой неигрового персонажа, который обрабатывает данные о вводе пользователя для общения с ним в формате чата.

Сущность *Chat* представляет собой данные об истории общения между игроком и *NPC*.

Сущность «*DayNight*» представляет собой информацию о текущем времени суток.

Сущность *Installer* представляет собой входные данные для сущностей игры.

2.5 Архитектура пользовательского интерфейса

Отдельное внимание стоит уделить разработке *GUI* на каждой из сцен. Создание эффективных пользовательских интерфейсов (*UI*) для видеоигр может быть сложным и трудоемким процессом. Дизайн пользовательского интерфейса – невероятно важный компонент разработки игр, так как он определяет первое впечатление игрока от игры, а также его общий опыт. Первым шагом в создании любого пользовательского интерфейса должно быть исследование и уточнение.

После начального этапа исследования и уточнения следующим шагом является создание каркаса. Каркас – это процесс наброска различных элементов пользовательского интерфейса, таких как меню, кнопки и окна. Это включает в себя выбор макета, функциональности и внешнего вида интерфейса.

Начальная сцена будет представлять собой главное меню, где пользователь с помощью *GUI* будет запускать игровую сцену. На рисунке 2.8 представлен каркас пользовательского интерфейса на начальной сцене.

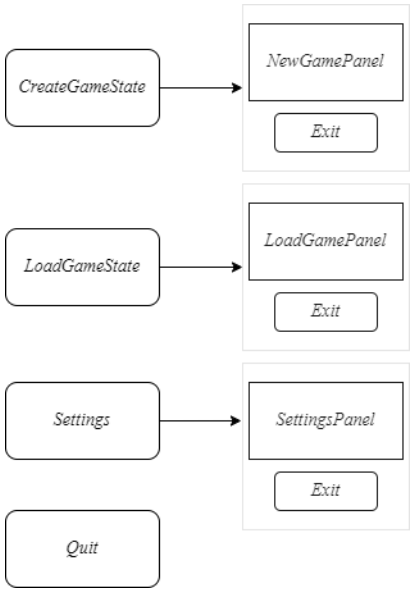

Рисунок 2.8 – Каркас пользовательского интерфейса на начальной сцене

Таким образом, начальная сцена должна содержать элементы пользовательского интерфейса для возможности запуска игровой сцены с

учетом выбранного игрового состояния. Кроме того, на начальной сцене должны присутствовать элементы пользовательского интерфейса для настройки игровой среды и закрытия приложения. В таком случае начальная сцена должна содержать следующие элементы:

– кнопка, которая отображает панель создания нового игрового состояния;

– панель создания нового игрового состояния;

– кнопка, которая отображает панель загрузки игрового состояния;

– панель загрузки игрового состояния;

– кнопка, которая отображает панель настроек игровой среды.

– панель настроек игровой среды;

– кнопка, с помощью которой можно закрыть игровое приложение.

Важно понимать, что пользовательский интерфейс не должен занимать весь экран, поэтому некоторые элементы *UI* не должны постоянно отображаться на экране.

В игровой сцене также должны присутствовать следующие элементы пользовательского интерфейса:

– кнопка, которая отвечает за продолжение игрового сеанса;

– кнопка, которая отображает панель настроек игровой среды;

– кнопка, которая отвечает за переход на начальную сцену.

Каркас пользовательского интерфейса на игровой сцене показан на рисунке 2.9.

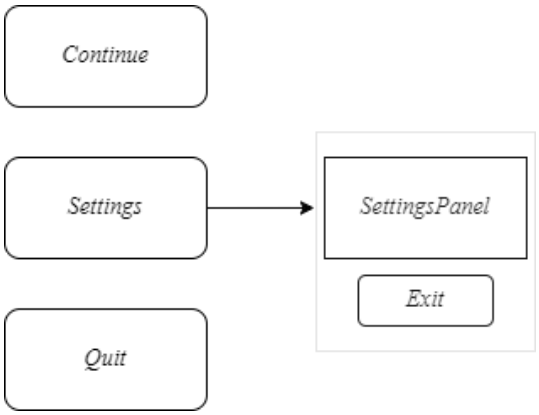

Рисунок 2.9 – Каркас пользовательского интерфейса на игровой сцене

После разработки каркасов управления приложением стоит перейти к разработке еще одного важного элемента пользовательского интерфейса, чтобы управлять игровым процессом на игровой сцене, который называется *HUD*.

HUD, если рассматривать его как цельный, самостоятельный интерфейс, может включать в себя множество различных элементов: полоски здоровья и стамины, указатели направления, метки в мире, различные элементы и мета-интерфейсы – все, что требуется для непосредственно игры в игру в

зависимости от ее жанра, геймплея, набора задач и игровых ситуаций, которые необходимо решить с помощью *HUD*(*heads-up display*).

В игровой сцене необходимо придумать каркас для следующих элементов интерфейса:

– текущее время суток;

– игровая валюта;

– активный инвентарь;

– вспомогательный инвентарь.

На рисунке 2.10 представлен каркас для игрового *HUD*.

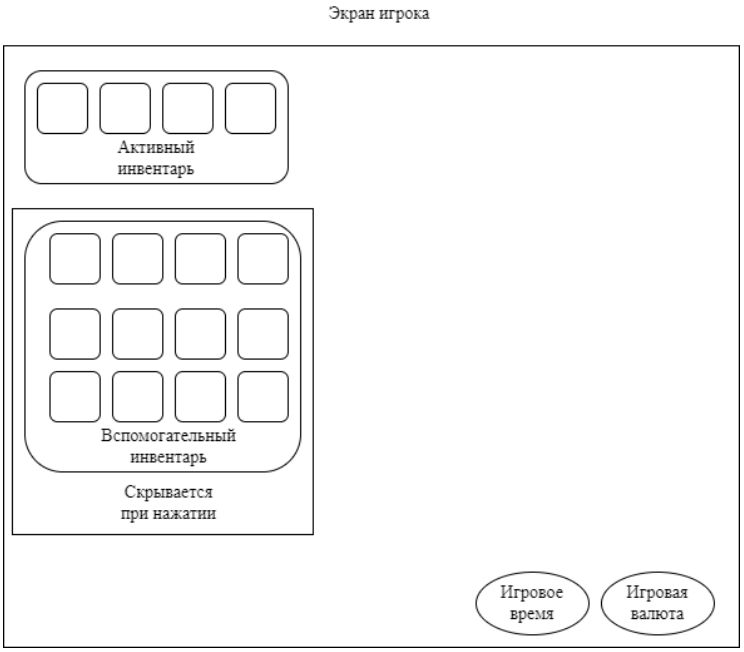

Рисунок 2.10 – Каркас для игрового *HUD*

Такая организация *UI* элементов позволит корректно отображать необходимую информацию, при этом почти без ограничения обзора игрока на игровой мир.

3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ «*FARMER'S VALLEY*»

3.1 Взаимодействие игровых элементов

Приложение реализовано средствами платформы *Unity*. Таким образом, работа модулей игрового приложения организует определенную структуру для работы с компонентами платформы. На рисунке 3.1 представлена работа модулей.

Рисунок 3.1 – Работа программных модулей

Использование программных компонентов платформы позволяет организовать логику взаимодействия всех игровых объектов. Под логикой взаимодействия всех игровых объектов будет подразумеваться графическое и программное составляющие игрового приложения. К графической составляющей относятся спрайты, визуальные эффекты, настройки графики, анимации. Все эти элементы могут являться частями одного игрового объекта. Такие элементы инициализируются с помощью подобранного программного обеспечения с учетом возможности платформы. Программное или логическое составляющее заключается в изменении свойств элементов игрового объекта в процессе обновления игрового цикла. Для изменения свойств игрового объекта в процессе обновления игрового цикла разрабатываются специальные классы и структуры данных, влияющие на организацию взаимодействия пользователя с игровым миром.

3.2 Принцип работы модуля управления игровым миром и управления фермой

Модуль управления игровым миром заключается в расстановке игровых объектов, формирующих игровую сцену и ее границы. Его работа тесно завязана с компонентом *Grid*. Компонент *Grid* является игровым объектом, представляющим сетку. Сетка характеризуется наличием ячеек. Ячейка представляет собой ее координаты, такие как координаты в мировом пространстве и координаты ее центра.

Суть компонента *Grid* заключается в том, чтобы устанавливать игровые объекты на игровое поле исключительно по сетке. По ней выстраиваются такие игровые объекты, как объект семени, объект обработанной земли и земли, политой лейкой, объекты декора.

Модуль управления фермой характеризуется классом *PlacementService* (Приложение А, с. 138), который содержит вложенные классы *CropData* (Приложение А, с. 139), *GroundData* (Приложение А, с. 139).

PlacementService – класс, который отвечает за установку предметов на сетке и следит за соблюдением необходимых условий для установки предмета на сетке. Реализует паттерн *Singleton*.

Все поля класса *PlacementService* показаны в таблице 3.1

Таблица 3.1 – Поля класса *PlacementService*

Название	Модификатор доступа	Тип данных
GroundTilemap	*public*	*Tilemap*
CropTilemap	*public*	*Tilemap*
WaterTilemap	*public*	*Tilemap*
WateredTile	*public*	*Tilemap*
TilleableTile	*public*	*Tilemap*
TilledTile	*public*	*Tilemap*
TillingEffectPrefab	*public*	*VisualEffect*
HarvestEffectPool	*private*	*Dictionary<Crop, List<VisualEffect>*
S_Instance	*private*	*PlacementService*
groundData	*Private*	*Dictionary<Vector3Int, GroundData>*
cropData	*private*	*Dictionary<Vector3Int, CropData>*

Поля класса *PlacementService* в таблице 3.1 описывают следующую логику:

– *GroundTilemap* – определяет карту расположения для спрайтов земли;

– *CropTile* – определяет карту расположения для спрайтов посева;

– *WaterTilemap* – определяет карту расположения для спрайтов политой земли;

– *WateredTile* – определяет спрайт политой земли;

– *TilleableTile* – определяет спрайт земли, который можно рыхлить;

– *TilledTile* – определяет спрайт взрыхленной земли;

– *TilingEffectPrefab* – определяет эффект взаимодействия;

– *HarvestEffectPool* – содержит эффекты взаимодействия, которые инициализируются заранее;

– *groundData* – содержит данные о состоянии объектов земли;

– *cropData* – содержит данные о состоянии объектов посева.

В таблице 3.2 описаны методы класса *PlacementService*, определяющие расположение игровых объектов на сетке при соблюдении некоторых условий.

Таблица 3.2 – Методы класса *PlacementService*

Название	Модификатор доступа	Тип возвращаемого значения	Параметры(тип данных)
Awake	*Private*	*Void*	–
Start	*Private*	*Void*	–
Update	*Private*	*Void*	–
IsTillable	*Public*	*Bool*	*Target(Vector3Int)*
IsPlantable	*Public*	*Bool*	*Target(Vector3Int)*
IsTilled	*Public*	*Bool*	*Target(Vector3Int)*
TillAt	*Public*	*void*	*Target(Vector3Int)*
PlantAt	*Public*	*void*	*Target(Vector3Int), cropToPlant(Crop)*
HarvestAt	*Public*	*Crop*	*Target(Vector3Int)*
WaterAt	*Public*	*Void*	*Target(Vector3Int)*
Save	*Private*	*Void*	–
Load	*Public*	*Void*	–
Instance	*Public*	*PlacementService*	–

Метода класса *PlacementService* в таблице 3.2 описывают следующую логику:

– *Awake* – инициализирует экземпляр класса;

– *Start* – инициализирует стартовый массив эффектов взаимодействия;

– *Update* – обновляет параметры процесса роста посева и мокрой земли;

– *IsTillable* – определяет возможность установки рыхлой земли;

– *IsPlantable* – определяет возможность установки объекта посева;

– *IsTilled* – определяет, стоит ли на определенной ячейке;

– *TillAt* – устанавливает объект рыхлой земли на ячейку;

– *PlantAt* – устанавливает объект посева на ячейку;

– *HarvestAt* – собирает объект посева и проигрывает определенный визуальный эффект на ячейке;

– *WaterAt* – устанавливает объект политой земли на ячейку;

– *Save* – сохраняет данные об объекты посева и позиции рыхлой земли;

– *Load* – устанавливает объекты посева и позиции рыхлой земли в соответствии с сохраненными данными;

– *Instance* – возвращает экземпляр класса.

CropData – класс, который содержит параметры процесса роста объекта посева. Все поля класса *CropData* показаны в таблице 3.3.

Таблица 3.3 – Поля класса *CropData*

Название	Модификатор доступа	Тип данных
GrowingCrop	*Public*	*Crop*
CurrentGrowthStage	*Public*	*Int*
GrowthRatio	*Public*	*float*
GrowthTimer	*Public*	*Float*
HarvestCount	*Public*	*Int*
DyingTimer	*Public*	*float*
HarvestDone	*Public*	*bool*

Поля класса *CropData* из таблицы 3.3 определяют следующую логику:

– *growingCrop* – является экземпляром объекта посева;

– *currentGrowthStage* – содержит текущую стадию роста объекта посева;

– *growthRatio* – коэффициент рост, чем больше этот параметр, тем быстрее увеличивается стадия роста;

– *growthTimer* – время роста;

– *harvestCount* – сколько раз можно собрать растительность;

– *dyingTimer* – время до удаления со сцены;

– *harvestDone* – если финальная стадия роста достигнута, то инициализируется значением *true*, если нет, *false*.

Класс *CropData* содержит методы позволяющие обновлять спрайт исходя из текущей стадии роста, а также методы, которые позволяют инициализировать данные о своем состоянии. Эти данные используются для создания объектов на сетке при загрузке сцены.

Сигнатура методов класса *CropData* описана в таблице 3.4.

Таблица 3.4 – Сигнатура методов класса *CropData*

Название	Модификатор доступа	Тип возвращаемого значения	Параметры(тип данных)
Harvest	*Public*	*Crop*	–
Save	*Public*	*Void*	*placedCropData(PlacedCropData)*
Load	*Public*	*Void*	*placedCropData (placedCropData), cropDataBase (cropDataBase)*

Методы, описанные в таблице 3.4, содержат следующую логику:

– *Harvest* – инкрементирует параметр текущей стадии роста, возвращает объект посева;

– *Save* – инициализирует структуру данных для сохранения своего состояния;

– *Load* – инициализирует свое состояние исходя из сохраненной структуры данных.

GroundData – класс, который содержит параметры состояния полива объекта земли. Его поля описаны в таблице 3.5.

Таблица 3.5 – Поля класса *GroundData*

Название	Модификатор доступа	Тип данных
WaterDuration	*Public*	*float*
WaterTimer	*Public*	*float*

Поля класса *GroundData* описывают следующую логику:

– *WaterDutation* – константный параметр, который содержит время нахождения спрайта мокрой земли на ячейке;

– *WaterTimer* – содержит текущее время нахождения объекта на сцене.

3.3 Принцип работы программных модулей управления инвентарем

Модуль управления инвентарем отвечает за управления предметами игрока и их взаимодействием с игровой сценой. Характеризуется классами:

– *InventoryBase* (Приложение А, с. 97);

– *InventoryCell* (Приложение А, с. 105);

– *PlayerInventory* (Приложение А, с. 109);

– *ActiveInventory* (Приложение А, с. 95);
– *InventoryItem* (Приложение А, с. 110);
– *ProductItem* (Приложение А, с. 104);
– *Crop* (Приложение А, с. 101);
– *HoeItem* (Приложение А, с. 102);
– *ItemContextData* (Приложение А, с. 108);
– *Pack* (Приложение А, с. 109).

InventoryItem – абстрактный класс, который содержит базовые параметры и методы для реализации логики предмета инвентаря. Поля класса описаны в таблице 3.6.

Таблица 3.6 – Поля класса *InventoryItem*

Название	Модификатор доступа	Тип данных
DisplayName	*Public*	*String*
Icon	*Public*	*Sprite*
UniqueName	*Public*	*String*
Color	*Public*	*Color*
MaxStackSize	*Public*	*Int*
Count	*Public*	*Int*
Consumble	*Public*	*Bool*
BuyPrice	*Public*	*Int*
IsCountTextActive	*Protected*	*bool*
VisualPrefab	*Public*	*GameObject*
UseSound	*Public*	*AudioClip[]*
PlayerAnimatorTriggerUse	*Public*	*string*

Поля класса *InventoryItem* из таблицы 3.6 описывают следующую логику:
– *DisplayName* – имя предмета, которое отображается на ячейке.
– *Icon* – картинка предмета;
– *UniqueName* – уникальное имя предмета;
– *Color* –цвет отображаемого имени объекта;
– *MaxStackSize* – максимальное количественное значение одного предмета;
– *Count* – текущее количество предмета;
– *Consumble* – содержит значение *true*, если это расходуемый предмет, *false*, если нет;
– *BuyPrice* – цена покупки предмета;
– *IsCountTextActive* – *true*, если нужно отображать текущее количество предмета в ячейке;

– *VisualPrefab* – игровой объект, который отображается в руке игрока при взаимодействии;

– *UseSound* – массив звуковых эффектов, которые проигрываются при взаимодействии с предметом инвентаря;

– *PlayerAnimatorTriggerUse* – название анимации, которая проигрывается при взаимодействии с предметом инвентаря.

Для реализации базовой логики взаимодействия игрока с предметом инвентаря класс *InventoryItem* содержит методы, сигнатура которых описана в таблице 3.7.

Таблица 3.7 – Сигнатура методов класса *InventoryItem*

Название	Модификатор доступа	Тип возвращаемого значения	Параметры (тип данных)
InitializeCopy	*Public*	*Void*	*inventoryItem (InventoryItem)*
ApplyCondition	*Public*	*Bool*	*Target(Vector3Int)*
Apply	*Public*	*Bool*	*Target(Vector3Int)*
RenderUI	*Public*	*Void*	*inventoryCell(InventoryCell)*
GetData	*Public*	*Inventory ItemData*	–
NeedTarget	*Public*	*Bool*	–
Clone	*Public*	*Object*	–

Методы класса *InventoryItem* содержат следующую логику:

– *InitializeCopy* – инициализирует другой экземпляр такого же типа своими значениями параметров;

– *ApplyCondition* – абстрактный метод, обязывающий наследников реализовать логику соблюдения условий для применения предмета;

– *Apply* – абстрактный метод, обязывающий наследников реализовать логику применения предмета;

– *RenderUI* – виртуальный метод, который содержит логику того, как отображать предмет в ячейке инвентаря;

– *NeedTarget* – виртуальный метод, который определяет, нужна ли цель для применения предмета;

– *Clone* – абстрактный метод, который обязует наследников реализовать интерфейс *IClonable*;

– *GetData* – виртуальный метод, который возвращает данные о предмете для сохранения или загрузки.

InventoryCell – класс, который представляет собой ячейку, которая отображает предмет инвентаря. Сигнатура полей класса *InventoryCell* описана в таблице 3.8.

Таблица 3.8 – Сигнатура полей класса *InventoryCell*

Название	Модификатор доступа	Тип данных
IconElement	*Private*	*Image*
TextElement	*Private*	*TextMeshProUGUI*
CountTextElement	*Private*	*TextMeshProUGUI*
SelectIconElement	*Private*	*Image*
GlobalVisualContext	*Public*	*Transform*
OriginVisualContext	*Public*	*Transform*
BeginDragSiblingIndex	*Public*	*Int*
itemResourceDroper	*Private*	*ItemResourceDropper*
mouseCursor	*Private*	*MouseCursor*

Поля, описанные в таблице 3.8 содержат следующую логику:

– *IconElement* – содержит экземпляр компонента для управления работой изображения иконки предмета;

– *TextElement* – содержит экземпляр компонента для редактирования отображаемого текста названия предмета;

– *CountTextElement* – содержит экземпляр компонента для редактирования текста, который отображает текущее количество предмета;

– *SelectionElement* – содержит экземпляр для управления изображением, отображающим выбранный предмет;

– *GlobalVisualContext* – содержит информацию о компоненте высшего по иерархии объекта интерфейса;

– *OriginVisualContext* – содержит информацию о родительском компоненте объекта интерфейса по отношению к текущему;

– *BeginDragSiblingIndex* –содержит информацию о порядке предмета в инвентаре в момент его перемещения;

– *ItemResourceDropper* – содержит экземпляр, позволяющий выкидывать предмет из инвентаря;

– *MouseCursor* – содержит экземпляр, который позволяет менять иконку курсора мыши;

Класс *IncentoryCell* содержит методы, которые позволяют реализовать логику взаимодействия игрока с ячейкой инвентаря. Сигнатура методов класса *InventoryCell* показана в таблице 3.9.

Таблица 3.9 – Сигнатура методов класса *InventoryCell*

Название	Модификатор доступа	Тип возвращаемого значения	Параметры (тип данных)
Initialize	*Public*	*Void*	*Global VisualContext(Transform), Inventory Item(InventoryItem)*
RegisterEvents	*Public*	*Void*	*endDragEvent(Action), beginDrag Event(Action<InventoryCell>)*
Update	*Private*	*Void*	–
OnDrag	*Private*	*Void*	*eventData(PointerEventData)*
OnBeginDrag	*Private*	*Void*	*eventData(PointerEventData)*
OnEndDrag	*Private*	*Void*	*eventData(PointerEventData)*

Методы, описанные в таблице 3.9, организуют следующую логику:

– *Initialize* – инициализирует экземпляр *InventoryCell* предметом инвентаря и контекстом действия;

– *RegisterEvent* – регистрирует методы инвентаря на события ячейки;

– *Update* – отслеживает текущее количество предмета инвентаря, чтобы удалить его, если его количество равняется нулю;

– *OnDrag* – метод, реализующий интерфейс *IDragHandler*, который перемещает ячейку инвентаря за курсором пользователя;

– *OnBeginDrag* – метод, реализующий интерфейс *IBeginDragHandler*, который меняет контекст перемещения ячейки в начале перемещения ячейки курсором;

– *OnEndDrag* – метод, реализующий интерфейс *IEndDragHandler*, который отслеживает пересечения курсора игрока с другими инвентарями в конце перемещения ячейки, перемещает ячейку в контекст другого инвентаря, если есть пересечения с другим инвентарем;

InventoryBase – абстрактный класс, который представляет сущность хранилища предметов инвентаря, следит за расположением ячеек инвентаря. Содержит все необходимые методы и параметры для реализации логики инвентаря.

Сигнатура полей класса *InventoryBase* описана в таблице 3.10.

Таблица 3.10 – Сигнатура полей класса *InventoryBase*

Название	Модификатор доступа	Тип данных
1	2	3

Продолжение таблицы 3.10

1	2	3
Container	*Public*	*Transform*
TotalSize	*Protected*	*Int*
CurrentSize	*Protected*	*Int*
InventoryItems	*Protected*	*List<InventoryItem>*
InventoryCellFactory	*protected*	*IInventoryCellFactory*
ContextRect	*Private*	*RectTransform*
OnBeginDragEvent	*Protected*	*Action<InventoryCell>*
OnEndDragEvent	*Protected*	*OnendDtagEvent*
MouseCurosr	*Private*	*MouseCursor*

Поля класса *InventoryBase* представляют из себя следующую логику:

– *Container* – компонент, представляющий контекст видимости для ячеек, которые находятся в инвентаре;

– *TotalSize* – содержит информацию о максимальном количестве вмещаемых ячеек в инвентаре;

– *CurrentSize* – содержит информацию о текущем количестве предметов, которые находятся в инвентаре;

– *InventoryItems* – список предметов инвентаря;

– *IInventoryCellFactory* – содержит экземпляр, реализующий интерфейс *IInventoryCellFactory*;

– *ContextRect* – содержит компонент, который представляет собой позицию панели инвентаря в пространстве пользовательского интерфейса;

– *OnBeginDragEvent* – содержит ссылки на методы ячеек, которые вызываются в начале взаимодействия с ячейкой;

– *OnEndDragEvent* – содержит ссылки на методы ячеек, которые вызываются в конце взаимодействия с ячейкой;

– *MouseCurosr* – содержит экземпляр класса *MouseCursor* для управления иконкой курсора мыши.

Для реализации взаимодействия игрока с инвентарем класс *InventoryBase* содержит методы, сигнатура которых показана в таблице 3.11.

Таблица 3.11 – Сигнатура методов класса *InventoryBase*

Название	Модификатор доступа	Тип возвращаемого значения	Параметры (тип данных)
1	2	3	4
Initialize	*Public*	*Void*	*InventoryItems(bool)*
RegisterDragEvents	*Public*	*Void*	*inventoryCell(int)*

Продолжение таблицы 3.11

1	2	3	4
AddItem	*Public*	*Bool*	*InventoryItem (InventoryItem)*
RemoveItem	*Public*	*Void*	*itemIndex(int)*
CreateCellForItem	*Private*	*Void*	*InventoryItem (InventoryItem)*
IsFull	*Public*	*Bool*	–
OnBeginDragCell	*Protected*	*Void*	–
OnEndDrag	*Protected*	*Void*	–
OnDrag	*Protected*	*Void*	*eventData (PointerEventData)*
OverwriteInventory ItemsSequence	*Private*	*List<InventoryItem>*	–
GetItemsContextData	*Public*	*List <ItemContextData>*	–
OnBeginDrag	*Public*	*Void*	*eventData (PointerEventData)*
OnEndDrag	*Public*	*Void*	*eventData (PointerEventData)*

Методы, описанные в таблице 3.11, содержат следующую логику:

– *Initialize* – инициализирует инвентарь списком предметов;

– *RegisterDragEvents* – регистрирует методы ячейки на события в инвентаре;

– *AddItem* – создает ячейку инвентаря;

– *RemoveItem* – удаляет ячейку инвентаря по индексу;

– *CreateCellForItem* – создает объект ячейки инвентаря;

– *IsFull* – возвращает *true*, если инвентарь заполнен, *false*, если нет;

– *OnBeginDragCell* – создает пустую ячейку;

– *OnEndDrag* – удаляет пустую ячейку;

– *OnDrag* – реализует интерфейс *IDragHandler*;

– *OverwriteInventoryItemsSequence* – переписывает порядок ячеек;

– *GetItemsContextData* – возвращает список данных о каждой ячейке;

– *OnBeginDrag* – реализует интерфейс *IBeginDragHandler*, меняет иконку курсора;

– *OnEndDrag* – реализует интерфейс *IEndDtragHandler*, меняет иконку курсора.

3.3 Принцип работы модуля экономической и социальной системы

Модуль экономической системы содержит классы и структуры данных для реализации механик покупки и продажи продукции. Характеризуется следующими классами: *TradeService* (Приложение А, с. 122), *TradePanel* (Приложение А, с. 119), *TradeElement* (Приложение А, с. 119).

TradeService – класс, который содержит логику соблюдения условий продажи и покупки продукции. Сигнатура методов класса *TradeService* описана в таблице 3.12.

Таблица 3.12 – Сигнатура методов класса *TradeService*

Название	Модификатор доступа	Тип возвращаемого значения	Параметры (тип данных)
BuyCondition	*Public*	*Bool*	*Item (InventoryItem)*
Buy	*Public*	*Void*	*Item (InventoryItem), Amount(int)*
SellCondition	*Public*	*Bool*	*Item (InventoryItem)*
Sell	*Public*	*Void*	*context Data(ItemContextData)*

Методы класса *TradeService*, описанные в таблице 3.12, содержат следующую логику:

– *BuyCondition* – возвращает *true*, если есть возможность купить продукцию;

– *Buy* – добавляет в инвентарь игрока предмет, при этом тратит игровую валюту в соответствии со стоимостью покупки продукции;

– *SellCondition* – возвращает *true*, если продукцию можно продать;

– *Sell* – удаляет из инвентаря предмет для продажи и добавляет игровую валюту в размере стоимости продажи предмета.

TradePanel – класс, который представляет панель для осуществления действий пользователя для покупки и продажи продукции. Сигнатура методов класса *TradePanel* описана в таблице 3.13.

Таблица 3.13 – Сигнатура методов класса 3.13

Название	Модификатор доступа	Тип возвращаемого значения	Параметры (тип данных)
1	2	3	4
OnBuy	*Public*	*Void*	–

Продолжение таблицы 3.13

1	2	3	4
OnSell	*Public*	*Void*	–
OnExit	*Public*	*Void*	–
ClearElements	*Private*	*Void*	–

Методы, описанные в таблице 3.13, содержат следующую логику:

– *OnSell* – создает элементы взаимодействия для каждого предмета, который можно продать в инвентаре игрока;

– *OnBuy* – создает элементы взаимодействия на панели для покупки предмета;

– *OnExit* – скрывает панель покупки и продажи;

– *ClearElements* – очищает элементы взаимодействия с панели покупки и продажи.

TradeElement – класс, который представляет элемент взаимодействия на панели покупки и продажи предметов. Сигнатура полей класса *TradeElement* описана в таблице 3.14.

Таблица 3.14 – Сигнатура полей класса *TradeElement*

Название	Модификатор доступа	Тип данных
IconElement	*Public*	*Image*
IconName	*Public*	*TextMeshProUGUI*
ButtonText	*Public*	*TextMeshProUGUI*
ButtonElement	*Public*	*Button*

Поля, описанные в таблице 3.14, содержат следующую логику:

– *IconElement* – содержит компонент для управления отображением;

– *IconName* – содержит компонент для управления текстом;

– *ButtonText* – содержит компонент для управления текстом кнопки;

– *ButtonElement* – содержит компонент позволяющий управлять объектом кнопки.

Модуль социальной системы отвечает за взаимодействие пользователя с *NPC* в формате чата. Характеризуется классами: *ChatService* (Приложение А, с. 92), *ChatPanel* (Приложение А, с. 88), *MassagePanel* (Приложение А, с. 91). В этом модуле используются библиотека *OpenAI* и *SharpToken*. Библиотека *OpenAI* нужна, чтобы организовывать запросы к *API ChatGPT*. Библиотека *SharpTokern* применяется для оптимизации запросов путем подсчета количества токенов в текущем контексте.

ChatService – класс, который реализует запросы к *API ChatGPT* с помощью библиотеки *OpenAI*. Сигнатура методов класса *ChatService* описана в таблице 3.14.

Таблица 3.14 – Сигнатура методов класса *ChatService*

Название	Модификатор доступа	Тип возвращаемого значения	Параметры (тип данных)
AddUserInput	*Public*	*Void*	*Text(string)*
AddSystemInput	*Public*	*Void*	*Text(string)*
GetResponseAsync	*Public*	*String*	–
GetContextText	*Public*	*String*	–
ClearChat	*Public*	*Void*	–

Методы, описанные в таблице 3.14, содержат следующую логику:

– *AddUserInput* – добавляет текст сообщения игрока в роли «*user*»;

– *AddSystemInput* – добавляет текст сообщения для чат-бота в роли «*system*»;

– *GetResponseAsync* – асинхронный метод, который получает ответ от чат-бота;

– *ClearChat* – очищает историю сообщений.

MassagePanel – класс, который представляет из себя панель, содержащую сообщение, введенное игроком, или сообщение, полученное в качестве ответа чат-бота. Содержит поля, являющиеся компонентами объекта панели сообщения, которые позволяют управлять контентом панели. Реализует логику написания сообщения с помощью методов.

ChatPanel – класс, который представляет собой место расположения панелей сообщения. Содержит методы, реализующие логику создания панели сообщения в результате ввода пользователя.

3.4 Паттерны в приложении «*Farmer's Valley*»

При разработке игрового приложения «*Farmer's Valley*» использовались различные паттерны проектирования для обеспечения гибкости, расширяемости и поддерживаемости кода [5]. Основные паттерны, использованные в проекте, включают:

– *Singleton*;

– *Observer*;

– *FactoryMethod*;

– *State*;

– *ServiceLocator*.

«Одиночка» (*Singleton*, Синглтон) – порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

Класс *LoadedData* реализует этот паттерн проектирования, чтобы иметь возможность обратиться к сохраненным данным при смене сцены.

Паттерн «Наблюдатель» (*Observer*) представляет поведенческий шаблон проектирования, который использует отношение «один ко многим». В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

Этот паттерн использует класс *SettingsMenu*, чтобы реагировать на изменения настроек игровой среды в пользовательском интерфейсе.

Фабричный метод (*Factory Method*) – это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать, происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

В проекте есть множество классов, реализующих этот паттерн. Такие классы создают объекты на игровой сцене, а также элементы интерфейса.

Состояние (*State*) – шаблон проектирования, который позволяет объекту изменять свое поведение в зависимости от внутреннего состояния.

Этот паттерн реализует класс *ItemResource*, который реализует несколько состояний внутри себя с изменением его поведения.

Локатор служб (*Service Locator*) – это шаблон проектирования, используемый в разработке программного обеспечения для инкапсуляции процессов, связанных с получением какого-либо сервиса с сильным уровнем абстракции.

Такой паттерн реализует класс *FactoriesProvider*, который инициализирует разные реализации фабрик для создания игровых элементов [6].

4 ВАЛИДАЦИЯ И ВЕРИФИКАЦТЯ РЕЗУЛЬТАТОВ РАБОТЫ ИГРОВОГО ПРИЛОЖЕНИЯ «*FARMER'S VALLEY*»

4.1 Принципы работы пользовательского графического интерфейса и основные механики игрового приложения «*Farmer's Valley*»

После запуска игры открывается сцена с главным меню. На рисунке 4.1 представлен кадр после запуска игры.

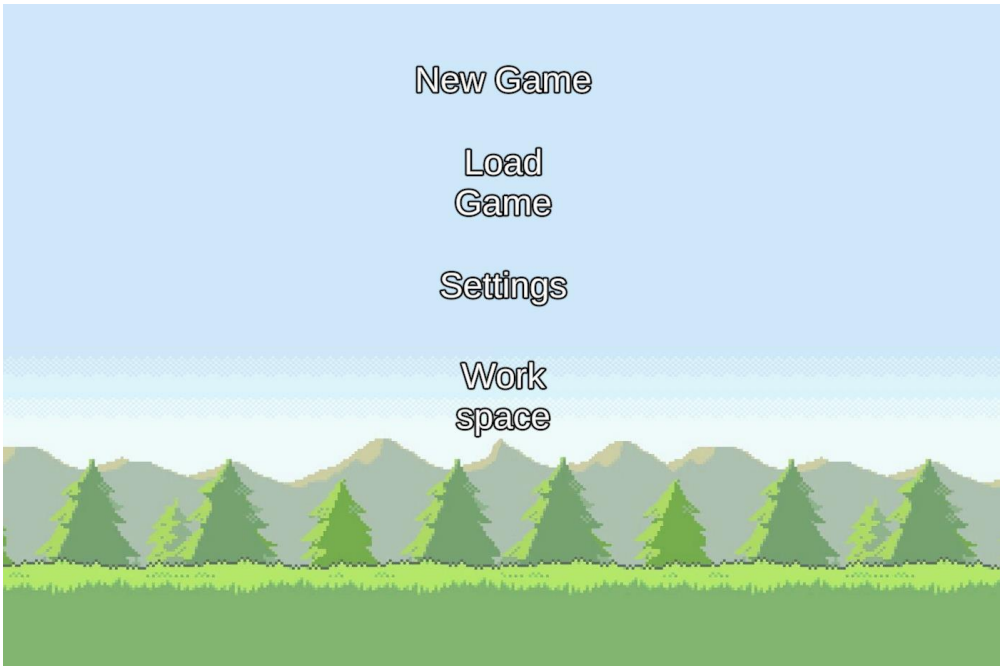

Рисунок 4.1 – Кадр после запуска игры

Сцена с главным меню является стартовой сценой с соответствующим графическим интерфейсом.

Пользователь может взаимодействовать со следующими элементами графического интерфейса:

– кнопка *NewGame*;

– кнопка *LoadGame*;

– кнопка *Settings*;

– кнопка *WorkSpace*.

При нажатии на кнопку *NewGame* перед пользователем откроется панель создания нового игрового состояния, где нужно ввести название нового игрового состояния. После ввода названия нового игрового состояния игрок может нажать на кнопку *Back*, чтобы вернуться к изначальному виду или же нажать на кнопку *Continue*, чтобы перейти к загрузке игровой сцены. Однако в случае нажатия на кнопку *Continue* пользователь не всегда перейдет к загрузке

игровой сцены. Чтобы перейти к загрузке сцены, должны быть выполнены некоторые условия:

– название нового игрового состояния не должно начинаться с цифры и содержать пробелы;

– названия игровых состояний не должны повторяться.

На рисунке 4.2 показана панель создания нового игрового состояния.

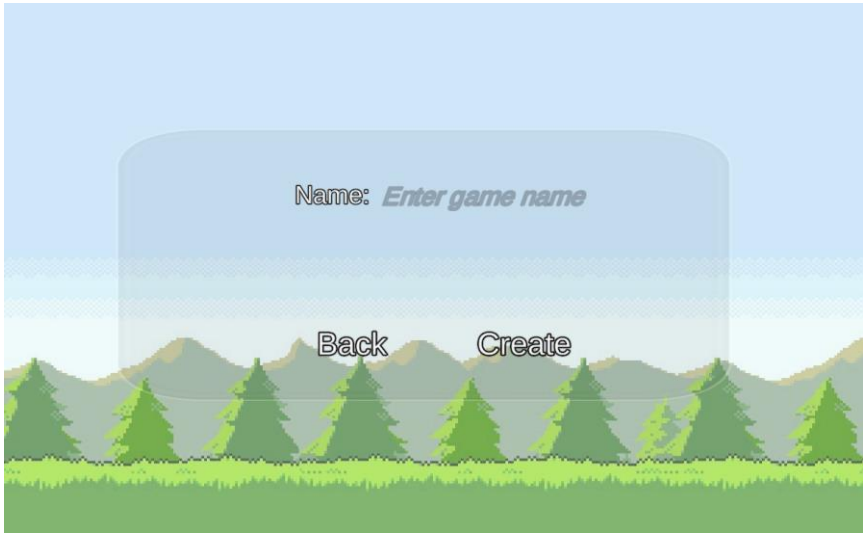

Рисунок 4.2 – Панель создания нового игрового состояния

При нажатии на кнопку *LoadGame* пользователю откроется панель загрузки игрового состояния. На рисунке 4.3 представлена панель загрузки игрового состояния.

Рисунок 4.3 – Панель загрузки игровых состояний

В панели загрузки игровых состояний отображаются все сохраненные игровые состояния, которые имеются на компьютере. Элемент игрового состояния представляет собой название игрового состояния, игровую валюту, которая осталась у игрока при сохранении игрового состояния, и иконку игрового персонажа. Содержит элементы интерфейса:

– кнопка *Load*;

– rкнопка *Delete*.

При нажатии на кнопку *Load* начнется загрузка игровой сцены с учетом сохраненных данных. При нажатии на кнопку *Delete*, сохраненное игровое состояние будет удалено с компьютера и в дальнейшем не будет появляться при открытии панели загрузки игрового состояния. При нажатии на кнопку *Back*, пользователь вернется к изначальному виду.

После нажатия на кнопку *Settings* пользователю откроется панель настроек игровой среды. На рисунке 4.4 представлена панель настройки игровой среды.

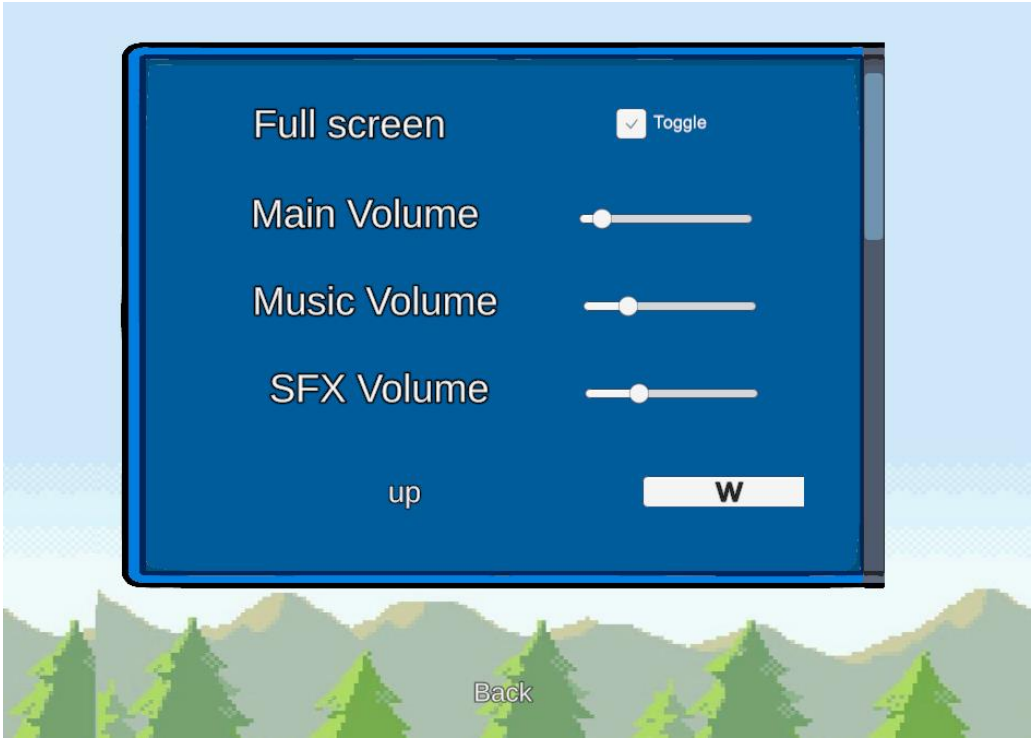

Рисунок 4.4 – Панель настройки игровой среды

На панели настройки игровой среды присутствуют следующие элементы графического интерфейса:

– *Screen Resolution*;

– *Full Screen*;

– *Main Volume*;

– *SFX Volume*;

– элементы смены клавиш взаимодействия с игровым миром.

При взаимодействии с элементом *Screen Resolution* пользователь может выбрать любое из представленных разрешений экрана. При взаимодействии с элементом *Full Screen* пользователь может поменять режим игрового приложения на оконный или полноэкранный. При взаимодействии с элементом *Main Volume* можно настроить громкость звука, которая влияет на остальные значения звука с выполнением функции микшера. При взаимодействии с элементом *Music Volume* можно влиять на громкость фоновой музыки напрямую. При взаимодействии с элементом *SFX Volume* можно изменить громкость звуковых эффектов во время ходьбы.

С помощью элемента смены клавиши взаимодействия можно поменять клавишу взаимодействия путем нажатия на кнопку выбранного действия и нажатия новой клавиши после вывода сообщения об ожидании ввода. Нажав на кнопку *Back*, пользователь вернется к исходному виду. Стандартное назначение клавиш в настройках показано в таблице 4.1.

Таблица 4.1 – Стандартное назначение клавиш в настройках клавиш взаимодействия

Клавиша	Действие
1	Выбор первой ячейки
2	Выбор второй ячейки
3	Выбор третьей ячейки
4	Выбор четвертой ячейки
W	Перемещение персонажа вверх
A	Перемещение персонажа вправо
S	Перемещение персонажа вниз
D	Перемещение персонажа влево
LMB	Взаимодействие
Tab	Скрыть/Открыть вспомогательный инвентарь

Клавиши, представленные в таблице 4.1, можно перенастроить как на главной сцене, так и на игровой. При этом после открытия панели настроек доступные для перенастройки клавиши взаимодействия будут называться в соответствии с выбранной раскладкой клавиатуры во время открытия панели нестроек.

При нажатии на клавишу *WorkSpace* пользователь закроет приложение.

После выбора клавиш, загружающих игровую сцену, пользователь увидит кадр, представленный на рисунке 4.5.

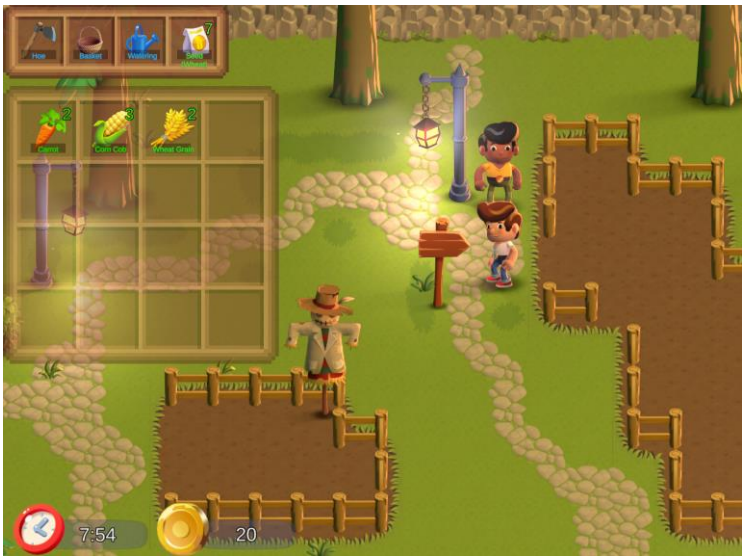

Рисунок 4.5 – Кадр из игровой сцены

Перед глазами игрока появляется игровой уровень. На нем виднеется игровой персонаж и *NPC*. Также можно увидеть земельные участки, огражденные забором. На рисунке 4.6 представлен результат использования предметов мотыга, лейка и мешок с семенами.

Рисунок 4.6 – Результат использования предметов мотыга, лейка, мешок с семенами

На таких участках игрок может взаимодействовать с предметами инвентаря для рыхления земли, полива земли и посадки семян.

Слева в верхнем углу экрана пользователь может увидеть активный инвентарь и вспомогательный. В них находятся предметы для взаимодействия. Синим цветом обозначены инструменты, такие как мотыга, лейка и корзинка, а зеленым цветом обозначены семена и предметы сбора. Чтобы использовать предметы вспомогательного инвентаря нужно перетащить их в активный инвентарь путем выбора определенного предмета из вспомогательного инвентаря и последующего зажатия левой кнопки мыши для перетаскивания предмета в активный инвентарь. Если конечный инвентарь, в который игрок пытался перенести определенный предмет, был заполнен, то выбранный предмет будет помещен в ближайшую ячейку инвентаря в зависимости от расположения курсора мыши.

Нажав на *NPC,* рядом появится панель, на которой находятся графические элементы интерфейса:

– *Talk*;

– *Trade*.

На рисунке 4.7 представлен результат нажатия игроком на *NPC*.

Рисунок 4.7 – Результат нажатия на *NPC*

При нажатии на кнопку *Trade* откроется панель покупки и продажи. С помощью элементов графического интерфейса на панели покупки и продажи игрок может продавать и покупать продукцию. После нажатия на кнопку *Trade* панель взаимодействия с *NPC* будет закрыта и откроется при следующем нажатии на *NPC*.

На панели покупки и продажи находятся следующие элементы графического интерфейса:

– кнопка *Buy*;

– кнопка *Sell;*

– кнопка закрытия панели торговли.

Пользователь путем нажатия на эти элементы интерфейса может переключаться между панелями покупки и продажи эелементов продукции. После нажатия на кнопку закрытия панель тарговли будет закрыта.

В случае нажатия на кнопку *Sell* перед пользователем появляются предметы инвентаря, которые он может продать, чтобы получить игровую валюту. На этих элементах отображается название продукции и сколько игрок получит игровой валюты за продажу текущего количества определенной продукции. Результат нажатия на кнопку *Sell* показан на рисунке 4.8.

Рисунок 4.8 – Результат нажатия на кнопку *Sell*

Перед пользователем появляются предметы инвентаря, которые он может продать, чтобы получить игровую валюту. Чтобы продать ту или иную продукцию пользователю нужно нажать на кнопку, которая помечена зеленым цветом. После чего будет начислена игровая валюта, а предмет продажи будет удален из инвентаря.

При нажатии на кнопку *Buy* перед пользователем появляются элементы взаимодействия, с помощью которых можно купить и добавить в инвентарь мешок семян. За одно нажатие игрок может приобрести одно семя. В случае, если игровой валюты не хватает, то кнопка покупки для пользователя будет не доступна.

Чтобы закрыть панель покупки и продажи, пользователю нужно нажать на кнопку, обозначенную крестиком, тогда панель закроется.

При нажатии на кнопку *Talk* открывается чат-панель, которая содержит элементы графического интерфейса для реализации общения с *NPC* в формате чата. Содержит следующие элементы:

– область для написания текста;

– кнопка *Enter*;
– кнопка *Reset*;
– кнопка закрытия чат-панели.
Чат-панель представлена на рисунке 4.9.

Рисунок 4.9 – Чат-панель

При нажатии на кнопку *Enter* отправляется сообщение из введенного текста в области ввода. Сообщение ограничено сотней символов.

Кнопка *Reset* будет доступна тогда, когда величина контекста общения с *NPC* будет превышена. После ее нажатия очистится история сообщений и будет доступна кнопка *Enter*.

В область ввода текста можно вводить текст любого типа. Писать можно на русском или английском языке, при этом получать ответ от *NPC* на русском или английском языке соответственно. Важен факт того, что на одинаковый текст запроса, можно получить разные ответы.

Пользовательский графический интерфейс является основным инструментом пользователя для взаимодействия с игровым миром.

При эксплуатации игры было проанализировано соответствие техническому заданию. Кроме того, была оценена целостность геймплея и удобство в управлении.

4.2 Виды тестирования игр

Перед публикацией финальной версии разработанное игровое приложение было несколько раз протестировано с помощью различных методов тестирования.

Существует перечень видов тестирования игровых приложений:

– функциональное тестирование – тестирование, целью которого является выявление отклонений от функциональных требований к игровому приложению, его фрагменту или отдельной функции [7]. Данное тестирование сводится к многократному запуску и прохождению игры, выявлению неполадок, багов либо несоответствий с заявленным функционалом и поиску методов или способов решения выявленных проблем и устранения неполадок. Данный тип тестирования применялся на протяжении всех этапов разработки игрового приложения и является основным;

– нагрузочное тестирование – тестирование игровых приложений, при котором воссоздаются ситуации, требующие большой вычислительной мощности и максимально нагружающие используемое устройство. Таким образом, тестировщик может проверить производительность системы в стрессовой ситуации или на потенциально слабых устройствах. Данный метод облегчает поиск потенциально ненадёжных фрагментов кода, которые требуют оптимизации. Для данного вида тестирования моделировались ситуации, которые потенциально могли перегружать возможности ипользуемого устройства. Например, в игровую сцену добавлялось большое количество предметов и объектов декора для более острых ощущений атмосферности от игрового процесса. Благодаря подобным тестам удалось убедиться в исправном функционировании игрового приложения при избытке объектов и графики в сцене;

– тестирование на совместимость – вид нефункционального тестирования, основной целью которого является проверка корректной работы продукта в определенном окружении. Проверке подвергалась возможность запуска игрового приложения на разных устройствах, имеющих различные технические характеристики.

4.3 Функциональное тестирование

Как было упомянуто выше, функциональное тестирование – это вид тестирования ПО, цель которого в проверке реализуемости функциональных требований, а именно способности ПО в определенных условиях корректно решать поставленные перед ним задачи. Функциональные требования определяют, что именно делает ПО, какие задачи оно решает. Тестирование

было проведено путем запуска игрового приложения при определенных тестировщиком условиях для наблюдения и оценки корректности работы различных аспектов игрового приложения. В процессе функционального тестирования производился анализ приложения для выявления несоответствий между существующими деталями функционирования игрового приложения и требованиями к их корректному функционированию.

Тест № 1. Цель: проверить запуск приложения.

Ожидаемый результат: игра должна запускаться на мобильных устройствах с ОС *Android* 4.0 и выше.

Вывод: ожидаемый и полученный результат совпали, цель теста достигнута.

Тест № 2. Цель: проверить работоспособность кнопок главного меню.

Ожидаемый результат: в зависимости от нажатой в главном меню кнопки происходит запуск игровой сцены, отображение окон настройки игровой среды создания нового игрового состояния, окна загрузок игровых состояний, или выход из приложения.

Вывод: ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 3. Цель: проверить соответствие отображения основного игрового интерфейса на устройстве с отображением в среде разработки.

Ожидаемый результат: на экране отображен интерфейс игры, отображаются: панель игрового интерфейса, содержащая панель активного инвентаря, панель вспомогательного инвентаря, панель с индикацией текущего времени суток и игровой валютой. Все элементы интерфейса выглядят уместно и не подвергнуты деформации.

Вывод: ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 4. Цель: проверить работоспособность игрового меню.

Ожидаемый результат: при нажатии кнопки паузы ввод клавиш, отвечающих за взаимодействия с игровой средой, блокируется, появляется панель с функциональными кнопками меню.

Вывод: ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 5. Цель: проверить работоспособность передвижения игрока по игровой сцене.

Ожидаемый результат: при нажатии на клавиши перемещения персонаж передвигается, при этом, если клавиши перемещения не нажаты персонаж поворачивается в сторону курсора мыши. Кроме того, персонаж игрока не должен проходить сквозь объекты сцены, блокирующие проход.

Вывод: персонаж игрока успешно перемещается по игровой сцене, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 6. Цель: проверить прямое взаимодействие игрока с активным инвентарем. При этом, в случае нажатия клавиш выбора предметов, должна появляться иконка текущего выбранного предмета.

Ожидаемый результат: при нажатии на один из предметов, текущий предмет передвигается за курсором, если клавиша перетаскивания отпущена, то предмет перемещается в ближайшую ячейку, также при нажатии клавиш выбора предметов, текущий выбранный предмет помечается иконкой выбранного предмета.

Вывод: игрок перемещает предметы инвентаря корректно, при нажатии клавиш выбора предмета текущий выбранный предмет помечается иконкой выбранного предмета, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 9. Цель: проверить корректность функционирования механизма посадки, сбора и полива.

Ожидаемый результат: при выборе предмета мотыга и нажатии на клавишу взаимодействия появляется объект рыхлой земли, при выборе предмета лейка и нажатии на объект рыхлой земли появляется объект политой земли, при выборе мешка с семенами и нажатии на объект рыхлой земли появляется объект семени, при выборе предмета корзинка и нажатии клавиши взаимодействия во время наведения на объект семени, в случае, если семя достигло последней стадии роста, объект семени уничтожается.

Вывод: при взаимодействии с землей с предметом мотыга, появляется объект рыхлой земли, при взаимодействии с объектом земли с выбранным предметом лейка, появляется объект политой земли, при нажатии на объект рыхлой земли с выбранным предметом мешок с семенем появляется объект семени, при нажатии на объект семени в последней стадии роста с выбранным предметом корзинка объект семени уничтожается. Таким образом, ожидаемый и достигнутый результат совпадают, цель достигнута.

Тест № 10. Цель: проверить функционирование открытия и закрытия панели купли и продажи продукции.

Ожидаемый результат: при нажатии на *NPC* появляется интерфейс с кнопкой *Trade*, нажав на которую появляется панель покупки и продажи, а после нажатия на кнопки закрытия панель покупки и продажи скрывается.

Вывод: панель покупки и продажи продукции корректно открывается и закрывается, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 11. Цель: проверить корректность покупки и продажи предметов в панели торговли.

Ожидаемый результат при нажатии на кнопку *Buy* в панели появляются предметы для покупки, при нажатии на клавишу покупки одного из предметов тратится игровая валюта, а предмет появляется в инвентаре, при нажатии кнопки *Sell* появляются предметы для продажи из инвентаря игрока, при нажатии на клавишу продажи одного из предметов определенный предмет инвентаря удаляется и начисляется игровая валюта:

Вывод: при покупке и продажи определенный предметов корректно начисляется и снимается игровая валюта, определенные предметы добавляются и удаляются из инвентаря, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 12. Цель: проверить функционал открытия и закрытия чат-панели.

Ожидаемый результат: при нажатии кнопкой взаимодействия на *NPC* появляется кнопка *Talk*, при нажатии на которую открывается чат-панель, при нажатии на кнопку закрытия чат-панели, панель закрывается.

Вывод: чат-панель корректно открывается и закрывается, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 13. Цель: проверить функционал чат-панели, функциональность отправки и получения сообщений.

Ожидаемый результат: игрок вводит сообщение в области ввода текста, после чего нажимает на кнопку *Enter* и сообщение появляется в панели справа, во время получения ответа кнопка *Enter* блокируется, после чего снова активируется, при достижении максимально доступного размера контекста активируется кнопка *Reset*, при нажатии на которую история сообщений удаляется, после чего кнопка *Enter* вновь оказывается доступной для взаимодействия.

Вывод: ввод и вывод сообщений отображается корректно, при превышении количества контекста доступна кнопка *Reset*, при нажатии на которую история сообщения очищается, после чего кнопка *Enter* снова доступна для отправки сообщения, ожидаемый и полученный результат совпадают, цель теста достигнута.

4.3 Юзабилити-тестирование

Юзабилити-тестирование – это исследование, выполняемое с целью определения степени удобства и понятности приложения для его потенциальных пользователей, другими словами, это метод оценки удобства продукта в использовании, основанный на привлечении пользователей в качестве тестировщиков, испытателей и суммировании полученных от них выводов. Юзабилити-тестирование является распространенным подходом при тестировании приложений.

Метод исследования «Мысли вслух». «Мысли вслух» – один из самых популярных методов юзабилити-тестирования был выбран в качестве метода исследования. Метод «Мысли вслух» позволяет понять, как пользователь подходит к интерфейсу и какими соображениями он руководствуется, используя функционал приложения.

План проведения тестирования:

– планирование – разработка заданий и набор участников тестирования;

– проведение тестирования;

– анализ полученных данных.

Методика проведения тестирования. Участник тестирования получает список заданий целиком. Участник должен ознакомиться со списком, после чего приступить к самостоятельному выполнению, проговаривая свои мысли, чувства, мнения, ощущения, возникающие в процессе взаимодействия с игрой.

Задания для респондентов продемонстрированы в таблице 4.1.

Таблица 4.1 – Протокол заданий

№ задания	Отправная точка	Задание
1	2	3
1	Сцена меню	Ознакомиться с главным меню
2	Сцена меню	Запустить игровую сцену
3	Игровая сцена	Переместить персонажа
4	Игровая сцена	Открыть и закрыть вкладки инвентаря
5	Игровая сцена	Использовать предмет мотыга
6	Игровая сцена	Использовать предмет лейка
7	Игровая сцена	Использовать предмет корзинка
8	Игровая сцена	Посадить семя
9	Игровая сцена	Поговорить с *NPC*
10	Игровая сцена	Продать предмет
11	Игровая сцена	Купить предмет
12	Игровая сцена	Засадить один из участков
13	Игровая сцена	Изменить настройки звука и разрешения
14	Игровая сцена	Поменять управление взаимодействиями

Было принято решение отбирать для тестирования людей различного возраста, увлечений и игрового опыта для разнообразия прохождения тестовых заданий. Все респонденты успешно справились с поставленными задачами, отметив удобство графического интерфейса, гармоничность оформления и интересный геймплей.

5 ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ИГРОВОГО ПРИЛОЖЕНИЯ «*FARMER'S VALLEY*»

5.1 Технико-экономическое обоснование целесообразности разработки программного продукта и оценка его конкурентоспособности

Игровое приложение предназначено для удовлетворения досуга пользователя и время провождения. В ходе использования пользователь погружается в процессе игры в фермерскую деятельность.

Существуют другие крупные проекты, которые целенаправленно ориентированы на ту же целевую аудиторию. Эти проекты являются аналогом или базовым эталоном для разработанного программного обеспечения, но они имеют большие недостатки, выраженные низкой частотой обновлений, стоимостью, отсутствием возможности играть на различных операционных системах. Базовым эталоном счиатется программный продукт с названием «*Stardew Valley*». Исходя из анализа существующих проектов можно сделать вывод о том, что разработка продукта целесообразна.

Техническая прогрессивность разрабатываемого программного продукта определяется коэффициентом эквивалентности ($K_{эк}$). Расчет данного коэффициента осуществляется путем сравнения технического уровня разрабатываемого программного продукта по отношению к эталонному уровню программного продукта данного направления с использованием формулы (Д.1).

Результат расчета коэффициента эквивалентности приведен в таблице 5.1. Полученное значение коэффициента эквивалентности больше единицы, следовательно, разрабатываемый программный продукт является технически прогрессивным.

Таблица 5.1 – Расчет коэффициентов эквивалентности

Наименование параметра	Вес параметра, β	Значения параметра			$\frac{P_б}{P_э}$	$\frac{P_н}{P_э}$	$\beta\frac{P_б}{P_э}$	$\beta\frac{P_н}{P_э}$
		$P_б$	$P_н$	$P_э$				
Объем памяти	0.3	10	8	7	1,43	1,1	0,43	0,33
Время обработки данных	0,4	0,8	0,5	0,2	4	2,5	1,6	1
Отказы	0,6	2	1	1	2	1	1,2	0,6
Итого							3,23	1.93
Коэффициент эквивалентности							3,23/1,93 =1,673	

Далее рассчитывается коэффициент изменения функциональных возможностей ($K_{ф.в}$) нового программного обеспечения по формуле (Д.3). Расчет коэффициента изменения функциональных возможностей нового программного продукта приведен в таблице 5.2.

Таблица 5.2 – Расчет коэффициента изменения функциональных возможностей

Наименование показателя	Балльная оценка базового ПП	Балльная оценка нового ПП
1	2	3
Объем памяти	4	4
Быстродействие	3	4
Удобство интерфейса	2	5
Степень утомляемости	3	2
Производительность труда	2	4
Итого	17	23
Коэффициент функциональных возможностей	23/17 =1,35	

Новый программный продукт превосходит по своим функциональным возможностям базовый в 1,35 раза.

Конкурентоспособность нового программного продукта по отношению к базовому можно оценить с помощью интегрального коэффициента конкурентоспособности, учитывающего все ранее рассчитанные показатели. Для расчета конкурентоспособности нового программного продукта по отношению к базовому была использована соответствующая формула (Д.4).

Коэффициент цены потребления рассчитывается как отношение договорной цены нового программного продукта к договорной цене базового (таблица 5.3).

Новый программный продукт превосходит по своим функциональным возможностям базовый в 1,35 раза.

Конкурентоспособность нового программного продукта по отношению к базовому можно оценить с помощью интегрального коэффициента конкурентоспособности, учитывающего все ранее рассчитанные показатели. Для расчета конкурентоспособности нового программного продукта по отношению к базовому была использована соответствующая формула (Д.4).

Коэффициент цены потребления рассчитывается как отношение договорной цены нового программного продукта к договорной цене базового (таблица 5.3).

Таблица 5.3 – Расчет уровня конкурентоспособности нового ПП

Коэффициенты	Значение
1	2
Коэффициент эквивалентности	1,673
Коэффициент изменения функциональных возможностей	1,35
Коэффициент соответствия нормативам	1
Коэффициент цены потребления	0,90
Интегральный коэффициент конкурентоспособности	$(1{,}673 \cdot 1{,}35 \cdot 1)/0{,}90 = 2{,}5$

Интегральный коэффициент конкурентоспособности ($K_{\text{и}}$) больше единицы, это значит, что новый программный продукт является более конкурентоспособным, чем базовый.

5.2 Оценка трудоемкости работ по созданию программного обеспечения

Общий объем программного обеспечения ($V_{\text{о}}$) определяется исходя из количества и объема функций, реализуемых программой, по каталогу функций программного обеспечения по формуле (Д.5).

Уточненный объем программного обеспечения ($V_{\text{у}}$) определяется по формуле (Д.6).

Результаты произведённых вычислений объема функций ПО представлены в таблице 5.4.

Таблица 5.4 – Перечень и объем функций ПО

Код функций	Наименование (содержание) функций	Объем функции строк исходного кода	
		по каталогу ($V_{\text{о}}$)	уточненный ($V_{\text{у}}$)
1	2	3	4
101	Организация ввода информации	150	300
102	Контроль, предварительная обработка и ввод информации	688	800

Продолжение таблицы 5.4

1	2	3	4
107	Организация ввода-вывода информации в интерактивном режиме	320	500
109	Управление вводом-выводом	2400	665
305	Формирование файла	2460	70
303	Обработка файлов	1100	620
405	Система настройки ПО	370	100
501	Монитор ПО (управление работой компонентов)	1340	800
506	Обработка ошибочных сбойных ситуаций	1720	400
507	Обеспечение интерфейса между компонентами	1820	260
702	Расчетные задачи (расчет режимов обработки)	1330	300
706	Предварительная обработка, печать	470	350
707	Графический вывод результатов	590	720
709	Изменение состояния ресурсов в интерактивном режиме	630	480
Итого		15388	6365

ПО относится ко третьей категории сложности.

На основании принятого к расчету (уточненного) объема ($V_{\text{у}}$) и категории сложности ПО определяется нормативная трудоемкость ПО ($T_{\text{н}}$) выполняемых работ, представлена в таблице 5.5.

Таблица 5.5 – Нормативная трудоемкость на разработку ПО ($T_{\text{н}}$)

Уточнённый объем, $V_{\text{у}}$	3-я категория сложности ПО	Номер нормы
6365	263	53

Дополнительные затраты труда, связанные с повышением сложности разрабатываемого ПО, учитываются посредством коэффициента повышения сложности ПО ($К_с$). $К_с$ рассчитывается по формуле (Д.7):

$$К_с = 1 + 0,06 = 1,06.$$

Влияние фактора новизны на трудоемкость учитывается путем умножения нормативной трудоемкости на соответствующий коэффициент, учитывающий новизну ПО ($К_н$).

Разработанная программа обладает категорией новизны Б, а значение $К_н = 0,72$.

Степень использования в разрабатываемом ПО стандартных модулей определяется их удельным весом в общем объеме ПО.

В данном программном комплексе используется до 45% стандартных модулей, что соответствует значению коэффициента $К_т = 0,65$.

Программный модуль разработан с помощью объектно-ориентированных технологий, что соответствует коэффициенту, учитывающему средства разработки ПО, $К_{у.р} = 0,55$. Значения коэффициентов удельных весов трудоемкости стадий разработки ПО в общей трудоемкости ПО определяются с учетом установленной категории новизны ПО и приведены в таблице 5.6.

Таблица 5.6 – 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				
	$К_{т.з}$	$К_{э.п}$	$К_{т.п}$	$К_{р.п}$	$К_{в.н}$
Б	0,10	0,20	0,30	0,30	0,10

Нормативная трудоемкость ПО ($Т_н$) выполняемых работ по стадиям разработки корректируется с учетом коэффициентов: коэффициента повышения сложности ПО ($К_с$), коэффициента новизны ПО ($К_н$), коэффициента степени использования стандартных модулей ($К_т$), коэффициент средств разработки ПО ($К_{у.р}$). Данные коэффициенты определяются для стадии ТЗ по формуле (Д.8), для стадии ЭП по формуле (Д.9), для стадии ТП по формуле (Д.10), для стадии РП по формуле (Д.11), для стадии ВН по формуле (Д.12). Коэффициенты $К_н$, $К_с$ и $К_{у.р}$ вводятся на всех стадиях разработки, а коэффициент $К_т$ вводится только на стадии РП.

Для уменьшения общей трудоёмкости разработки введем коэффициент используемости разработанного ПО, который равен 0,1.

$$T_{у.т.з} = 263 \cdot 0,1 \cdot 1,06 \cdot 0,72 \cdot 0,55 = 11 \text{ чел.-дн.},$$

$$T_{у.э.п} = 263 \cdot 0,2 \cdot 1,06 \cdot 0,72 \cdot 0,55 = 22 \text{ чел.-дн.},$$

$$T_{у.т.п} = 263 \cdot 0,3 \cdot 1,06 \cdot 0,72 \cdot 0,55 = 33 \text{ чел.-дн.},$$

$$T_{у.р.п} = 263 \cdot 0,3 \cdot 1,06 \cdot 0,72 \cdot 0,65 \cdot 0,55 = 21 \text{ чел.-дн.},$$

$$T_{у.в.н} = 263 \cdot 0,1 \cdot 1,06 \cdot 0,72 \cdot 0,55 = 11 \text{ чел.-дн.}$$

Общая трудоемкость разработки программного обеспечения ($T_о$) определяется суммированием нормативной (скорректированной) трудоемкости программного обеспечения на всех стадиях разработки программного обеспечения по формуле (Д.13):

$$T_о = 11 + 22 + 33 + 21 + 11 = 98 \text{ чел.-дн.}$$

Параметры расчетов по определению нормативной и скорректированной трудоемкости программного обеспечения на всех стадиях разработки и общей трудоемкости разработки программного обеспечения представлены в таблице 5.7.

Таблица 5.7 – Расчет общей трудоемкости разработки ПО

Показатели	Стадии разработки					Итого
	ТЗ	ЭП	ТП	РП	ВН	
1	2	3	4	5	6	7
Общий объем ПО ($V_о$), кол-во строк *LOC*	–	–	–	–	–	15388
Общий уточненный объем ПО ($V_у$), кол-во строк *LOC*	–	–	–	–	–	6365
Категория сложности разрабатываемого ПО	–	–	–	–	–	3
Нормативная трудоемкость разработки ПО ($T_н$), чел.-дн.	–	–	–	–	–	263
Коэффициент повышения сложности ПО ($K_с$)	1,06	1,06	1,06	1,06	1,06	–
Коэффициент, учитывающий новизну ПО ($K_н$)	0,72	0, 72	0, 72	0, 72	0, 72	–

Продолжение таблицы 5.7

1	2	3	4	5	6	7
Коэффициент, учитывающий степень использования стандартных модулей ($К_{т}$)	–	–	–	0,65	–	–
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 ($К_{т.з}$,$К_{э.п}$,$К_{т.п}$,$К_{р.п.}$,$К_{в.н}$)	0,10	0,20	0,30	0,30	0,10	1,0
Распределение скорректированной (с учетом $К_{с}$,$К_{н}$, $К_{т}$,$К_{у.р}$) трудоемкости ПО по стадиям, чел.-дн.	11	22	33	21	11	–
Общая трудоемкость разработки ПО ($Т_{о}$), чел.-дн.	–	–	–	–	–	98

Таким образом, были вычислены параметры расчетов по определению нормативной и скорректированной трудоемкости разработки программного обеспечения на всех стадиях разработки, а также общая трудоемкость разработки программного обеспечения.

5.3 Расчёт затрат на разработку программного продукта

Суммарные затраты на разработку программного обеспечения ($З_{р}$) определяются по формуле (Д.14). Параметры расчета производственных затрат на разработку программного обеспечения приведены в таблице 5.8.

Таблица 5.8 – Параметры расчета производственных затрат на разработку ПО

Параметр	Единица измерения	Значение
1	2	3
Базовая тарифная ставка	руб.	250
Доплата за стаж	% (руб.)	19,5 (93,57)
Премия	%	5
Доплата по контракту	%	0,5 от оклада
Разряд разработчика	–	12

Продолжение таблицы 5.8

1	2	3
Тарифный коэффициент	–	2,84
Коэффициент $К_{ув}$	–	1,5
Норматив отчислений на доп. зарплату разработчиков ($Н_{доп}$)	%	20
Численность обслуживающего персонала	чел.	0
Средняя годовая ставка арендных платежей ($С_{ар}$) (по результатам мониторинга предложений по аренде помещений)	руб./м^2	13,69
Площадь помещения (S)	м^2	12
Количество ПЭВМ ($Q_{эвм}$)	шт.	1
Затраты на приобретение единицы ПЭВМ	руб.	2000
Стоимость одного кВт-часа электроэнергии ($С_{эл}$)	руб.	0,337
Коэффициент потерь рабочего времени ($К_{пот}$)	–	0,2
Затраты на технологию ($З_{тех}$)	руб.	–
Норматив общепроизводственных затрат ($Н_{доп}$)	%	10

Расходы на оплату труда разработчиков с отчислениями ($З_{тр}$) определяются по формуле (Д.15).

Основная заработная плата разработчиков рассчитывается по формуле (Д.16).

Средняя часовая тарифная ставка определяется по формуле (Д.17).

Часовая тарифная ставка определяется путем деления месячной тарифной ставки на установленный при восьмичасовом рабочем дне фонд рабочего времени 168 ч ($F_{мес}$), формула (Д.18).

$$С_{.ч} = \frac{250 \cdot 2,84}{168} = 4,2 \text{ руб.},$$

$$С_{ср.ч} = \frac{4,2 \cdot 1}{1} = 4,2 \text{ руб.},$$

$$ЗП_{осн} = 4,2 \cdot 98 \cdot 1,5 = 617,4 \text{ руб.}$$

Дополнительная заработная плата рассчитывается по формуле (Д.19):

$$ЗП_{доп}\ 617,4 \cdot 20 \div 100\% = 123,5 \text{ руб.}$$

Отчисления от основной и дополнительной заработной платы рассчитываются по формуле (Д.20):

$$ОТЧ_{с.н} = \frac{(617,4 + 123,5) \cdot 36\%}{100\%} = 251 \text{ руб.},$$

$$З_{тр} = 617,4 + 123,5 + 251 = 992 \text{ руб.}$$

Годовые затраты на аренду помещения определяются по формуле (Д.27):

$$З_{ар} = 13,69 \cdot 12 = 164,28 \text{ руб.}$$

Сумма годовых амортизационных отчислений ($З_{ам}$) определяется по формуле (Д.28):

$$З_{ам} = 2000 \cdot (1 + 0,12) \cdot 1 \cdot 0,125 = 280 \text{ руб.}$$

Стоимость электроэнергии, потребляемой за год, определяется по формуле (Д.29).

Действительный годовой фонд времени работы ПЭВМ ($F_{эвм}$) рассчитывается по формуле (Д.30):

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

$$З_{эвм} = \frac{0,44 \cdot 1619 \cdot 0,337 \cdot 0,9}{1} = 216,1 \text{ руб.}$$

Затраты на материалы ($З_{эвм}$), необходимые для обеспечения нормальной работы ПЭВМ, составляют около 1% от балансовой стоимости ЭВМ, и определяются по формуле (Д.31):

$$З_{в.м} = 2000 \cdot (1 + 0,12) \cdot 0,01 = 22,4 \text{ руб.}$$

Затраты на текущий и профилактический ремонт ($З_{т.р}$) принимаются равными 5% от балансовой стоимости ЭВМ и вычисляются по формуле (Д.32):

$$З_{т.р} = 2000 \cdot (1 + 0,12) \cdot 0,08 = 179,2 \text{ руб.}$$

Прочие затраты, связанные с эксплуатацией ЭВМ ($З_{пр}$), состоят из амортизационных отчислений на здания, стоимости услуг сторонних организаций и составляют 5 % от балансовой стоимости и определяются по формуле (Д.33):

$$З_{п.р} = 2000 \cdot (1 + 0{,}12) \cdot 0{,}05 = 112 \text{ руб.}$$

Для расчета машинного времени ЭВМ ($t_{эвм}$ в часах), необходимого для разработки и отладки проекта специалистом, следует использовать формулу (Д.34):

$$t_{эвм} = 33 \cdot 8 \cdot 1 = 264 \text{ ч.},$$

Затраты машинного времени ($З_{м.в}$) определяются по формуле (Д.21):

$$З_{м.в} = 0{,}5 \cdot 1 \cdot 264 = 132 \text{ руб.}$$

Стоимость машино-часа определяется по формуле (Д.22):

$$С_{ч} = \frac{280 + 216{,}1 + 22{,}4 + 179{,}2 + 112}{1619} = 0{,}5 \text{ руб./ч.}$$

Расчет затрат на изготовление эталонного экземпляра ($З_{эт}$) осуществляется по формуле (Д.35):

$$З_{эт} = (992 + 132) \cdot 0{,}05 = 56{,}2 \text{ руб.}$$

Так же рассчитываются затраты на материалы ($З_{мат}$) по формуле (Д.36):

$$З_{мат} = 2000 \cdot (1 + 0{,}12) \cdot 0{,}01 = 22{,}4 \text{ руб.}$$

Общепроизводственные затраты ($З_{общ.пр}$) рассчитываются по формуле (Д.37):

$$З_{общ.пр} = \frac{617{,}4 \cdot 10}{100} = 61{,}74 \text{ руб.}$$

И наконец, непроизводственные затраты ($З_{непр}$) рассчитываются по формуле (Д.38):

$$З_{непр} = \frac{617{,}4 \cdot 5}{100} = 30{,}85 \text{ руб.}$$

Итого, получаем суммарные затраты на разработку:

$$З_{р} = 992 + 56{,}2 + 132 + 22{,}4 + 61{,}74 + 30{,}85 = 1295{,}2 \text{ руб.}$$

5.4 Расчёт договорной цены и частных экономических эффектов от производства и использования программного продукта

Игровое приложение доступно на онлайн сервисе *itch.io*, где пользователи могут приобрести копии игры за определенную плату. Стоит выделить, что в этом онлайн сервисе не взымается плата за распространение программного продукта.

После просмотра и анализа аналогов программного продукта можно сделать вывод, что цена за приобритение копии продукта варируется от 24 руб. до 68 руб., значит, для сохранения конкурентноспособности цена одной копии, разработанного программного продукта, составит 38 руб.

Таким образом, прибыль для первого года может расчитываться по формуле (Д.43):

$$Э = (100 \cdot 38) \cdot (100 - 20) \div 100 = 3040 \text{ руб.}$$

Сумму выплаченных налогов можно расчитать по формуле (Д.44):

$$N_{TAX} = 3040 \cdot 20 \div (100 - 20) = 750 \text{ руб.}$$

При оценивании экономической эффективности проекта целесообразно рассчитать следующие итоговые показатели, характеризующие экономическую эффективность проекта. Рентабельность затрат или инвестиций на новую информационную технологию и программный продукт в первый год эксплуатации игрового приложения может быть вычеслено по формуле (Д.45):

$$Р = 3040 \cdot 100 \div 1295,2 = 234\%.$$

Срок окупаемости служит для определения степени рисков реализации проекта и ликвидности инвестиций. Срок окупаемости затрат (инвестиций) расчитывается по формуле (Д.46):

$$Т_{пр} = 1295{,}2 \div 3040 = 0{,}42 \text{ года.}$$

Поскольку срок окупаемости составляет менее одного календарного года, проведение динамической оценки (расчета динамических показателей эффективности) становится излишним. Следовательно, результаты анализа подтверждают, что реализация проекта обоснована и экономически целесообразна. Это подтверждается следующими факторами: срок окупаемости менее года и значительный годовой экономический эффект, при котором чистая

прибыль начинает накапливаться уже после первого года реализации. Технико-экономические показатели проекта определены в таблице 5.9.

Таблица 5.9 – Технико-экономические показатели проекта

Наименование показателя	Единица измерения	Проектный вариант
1	2	3
Интегральный коэффициент конкурентоспособности	–	2,5
Коэффициент эквивалентности	–	1,673
Коэффициент изменения функциональных возможностей	–	1,35
Коэффициент соответствия нормативам	–	1
Коэффициент цены потребления	–	0,90
Общая трудоемкость разработки ПО	чел.- дн	98
Суммарные затраты на разработку ПО ($З_р$)	руб.	1295,2
Затраты на оплату труда разработчиков	руб.	992
Затраты машинного времени	руб.	132
Затраты на изготовление эталонного экземпляра	руб.	56,2
Затраты на технологию	руб.	–
Затраты на материалы	руб.	22,4
Общепроизводственные затраты	руб.	61,74
Непроизводственные (коммерческие) затраты	руб.	30,85
Число снимаемых копий ПП	шт.	100
Стоимость копии продукта	руб.	38
Рентабельность затрат	%	234
Простой строк окупаемости проекта	лет	0,42

6 ОХРАНА ТРУДА И ТЕХНИКА БЕЗОПАСНОСТИ

6.1 Требования к производственному освещению

Производственное освещение играет существенную роль в обеспечении безопасных, комфортных и производительных условий труда. Неправильное освещение может привести к утомлению глаз, снижению производительности, травмам и профессиональным заболеваниям. В этой работе будут рассмотрены ключевые требования к производственному освещению.

К производственному освещению предъявляются следующие нормативные требования:

1) освещенность рабочих мест должна соответствовать установленным нормативам, нормы освещенности устанавливаются в зависимости от категории зрительных работ, системы освещения, вида работ и других факторов;

2) равномерность освещения: освещение должно быть равномерным и без резких теней, недопустимо наличие слепящей блескости от источников света и отраженных от блестящих поверхностей предметов.

3) спектральный состав света: спектральный состав искусственного света должен быть близок к дневному;

4) размещение и мощность светильников: светильники должны быть правильно размещены и иметь соответствующую мощность.

Производственное освещение можно классифицировать на несколько видов:

– общее освещение: создает общий световой фон в производственном помещении;

– местное освещение: освещает отдельные рабочие места;

– аварийное освещение: обеспечивает освещение в случае отключения рабочего освещения;

– комбинированное освещение: сочетает в себе общее и местное освещение.

При выборе системы освещения необходимо учитывать следующие факторы:

– категорию зрительных работ;

– вид работ;

– размеры и конфигурацию производственного помещения;

– наличие источников естественного света;

– требования к цветопередаче;

– экономичность системы освещения;

Системы производственного освещения должны регулярно контролироваться и обслуживаться. Контроль включает в себя измерение

освещенности, проверку работоспособности светильников и электропроводки. Обслуживание включает чистку светильников, замену ламп и другие работы.

На рисунке 6.1 представлен наглядный визуальный пример освещения на производстве.

Рисунок 6.1 – Наглядный визуальный пример освещения на производстве

Производственное освещение влияет на здоровье и работоспособность. Рациональное производственное освещение оказывает положительное влияние на здоровье и работоспособность работников. Недостаточная освещенность может привести к:

– утомлению глаз;

– снижению производительности труда;

– повышению риска травматизма;

– развитию профессиональных заболеваний;

– слепящая блескость может вызвать

– утомление глаз.

– снижение остроты зрения.

– развитие близорукости.

Неправильный спектральный состав искусственного света может привести к:

– нарушению цветовосприятия;

– снижению работоспособности;

– повышению утомляемости;

– примеры из практики

Учитывая приведенные факторы, в качестве примера, можно описать системы освещения на рабочем месте, соответствующее всем нормативным требованиям.

В металлообрабатывающем цехе, где работники выполняют точные задачи, такие как фрезерование, токарная обработка и шлифование, система освещения состоит из общего люминесцентного освещения и локальных светодиодных светильников на каждом рабочем месте. Уровень освещенности соответствует стандартам, освещение равномерное и без бликов, спектральный состав света близок к дневному.

Также, в качестве примера, можно провести анализ причин несоответствия системы освещения на рабочем месте нормативным требованиям и предложения по ее исправлению.

При осмотре столярного цеха выявлено, что уровень освещенности на некоторых рабочих местах не соответствует стандартам. Также наблюдалась слепящая блескость от окон и блестящих поверхностей станков. Для устранения этих нарушений было предложено установить дополнительные локальные светильники на проблемных рабочих местах и использовать жалюзи на окнах.

В ходе изучения требований к производственному освещению была изучена информация о методах и расчетах освещенности, которая может быть полезна для проектирования и обслуживания систем производственного освещения.

Методы измерения освещенности:

– люксметр: ручной прибор, измеряющий освещенность в люксах (лк);

– фотометр: более точный прибор, который может измерять освещенность в люксах, свечах на квадратную фут (*fc*) или других единицах.

Формулы для расчета освещенности:

– закон обратных квадратов;

– метод по точкам.

Закон обратных квадратов звучит так: освещенность в точке обратно пропорциональна квадрату расстояния от источника света до точки.

Метод по точкам заключается в расчете освещенности в каждой точке системы освещения с учетом вклада всех источников света.

Программное обеспечение для расчета освещенности:

1) *DIALux* – профессиональное программное обеспечение для проектирования освещения, которое можно использовать для расчета освещенности, моделирования световых сцен и создания отчетов об освещении.

2) *Relux* – программное обеспечение для проектирования освещения с функциями, похожими на *DIALux*.

Производственное освещение является важнейшим фактором, влияющим на здоровье, производительность и безопасность работников. Соблюдение нормативных требований, правильный выбор системы освещения и регулярный контроль, и обслуживание систем освещения являются ключом к созданию безопасных, комфортных условий.

7 ЭНЕРГОСБЕРЕЖЕНИЕ И РЕСУРСОСБЕРЕЖЕНИЕ ПРИ ЭКСПЛУАТАЦИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

7.1 Энергосбережение и ресурсосбережение при внедрении и эксплуатации программного обеспечения

Энерго- и ресурсосбережение представляют собой ключевые направления деятельности, направленные на рациональное использование ресурсов и минимизацию затрат на их потребление. В условиях ограниченности природных ресурсов и роста их стоимости, данные направления становятся все более актуальными.

Энергосбережение определяется как комплекс мер, направленных на эффективное использование топливно-энергетических ресурсов и вовлечение возобновляемых источников энергии. Актуальность энергосбережения обусловлена растущими ценами на традиционные энергоресурсы и истощением их мировых запасов.

Стоит рассмотреть понятие и задачи энергосбережения. Ресурсосбережение включает организационные, экономические, технические, научные и информационные меры, направленные на рациональное использование ресурсов на всех стадиях жизненного цикла изделий [8]. Задачи ресурсосбережения включают:

– сбережение топлива и энергии.

– рациональное использование материальных ресурсов.

– максимальное сохранение природных ресурсов.

– сохранение равновесия между развитием производств и потреблением вторичных материальных ресурсов.

– совершенствование систем управления качеством продукции и услуг.

– экономически эффективное и безопасное использование вторичных ресурсов.

Экономия электроэнергии – важный аспект, влияющий как на производственные процессы, так и на бытовую сферу. Эффективное энергопотребление можно достигнуть через различные меры:

– оптимизация использования компьютеров и периферийных устройств. Использование ПК преимущественно в дневное время позволяет снизить расходы на электроэнергию, учитывая более высокие тарифы в ночное время. Также обновление оборудования и использование устройств с меньшим энергопотреблением помогает сократить затраты.

– уменьшение времени работы неиспользуемых устройств. Выключение компьютеров и периферийных устройств при длительном простое снижает энергопотребление;

– энергосберегающие технологии в ПО.

Применение оптимизированных алгоритмов и структур данных уменьшает нагрузку на вычислительные ресурсы и снижает потребление электроэнергии.

Дальше следует рассмотреть принципы ресурсосбережения при разработке ПО [9].

Для эффективного ресурсосбережения в процессе разработки программного обеспечения необходимо:

– снижение объемов используемых ресурсов. Оптимизация текстур, уменьшение количества полигонов в трехмерных моделях, исключение дублирования идентичных ресурсов;

– оптимизация графического интерфейса. Использование выдержанной, неконтрастной цветовой гаммы и плавных анимаций уменьшает нагрузку на глаза пользователей и снижает энергопотребление.

Рассмотрим пример экономии электроэнергии при использовании мобильных версий приложений. Среднее энергопотребление настольного ПК составляет около 350 Вт·ч, тогда как смартфон или планшет потребляют около 15 Вт·ч. Использование мобильных устройств может значительно снизить энергопотребление.

Формула для расчета экономии электроэнергии:

$$Э = (Т_{пк} - Т_{м}) \cdot Р \cdot С_{эл} \cdot К_{и}, \quad (7.1)$$

где $Т_{пк}$ – среднее энергопотребление персонального настолького компьютера или иного подобного устройства, кВт·ч; $Т_{м}$ – среднее энергопотребление мобильного устройства, кВт·ч; Р – время работы используемого устройства, ч; $С_{эл}$ – стоимость одного кВт·ч электроэнергии, руб; $К_{и}$ – коэффициент использования.

В среднем электропотребление при стандартной загрузке среднестатистического персонального компьютера составляет 350 Вт·ч. Энергопотребление смартфона или планшета составляет 15 Вт·ч. По состоянию на 2 мая 2024 года стоимость 1 кВт·ч. электроэнергии в Республике Беларусь для физических лиц равен 0,2537 рубля [12]. Коэффициент использования принят за 0,6. Из выше сказанного, стоимость сэкономленной электроэнергии за один месяц при игре 4 часа каждый день составит:

$$Э = (0{,}35 – 0{,}015) \cdot 120 \cdot 0{,}2537 \cdot 0{,}6 = 6{,}12 \text{ руб.}$$

Экономию электроэнергии за нужный период времени можно рассчитать по формуле:

$$Э_{пер} = Э \cdot n, \quad (7.2)$$

где Э – сэкономленная электроэнергия за месяц, руб.; n – период использования автоматизированной системы, месяцев.

$$Э_{пер} = 6{,}12 \cdot 12 = 73{,}44 \text{ руб.}$$

Данный расчет показывает, что при использовании кроссплатформенности разработанного игрового приложения можно сэкономить на электроэнергии до 6,12 рублей в месяц, а за год 73,44 рублей.

Энерго- и ресурсосбережение при разработке и эксплуатации программного обеспечения являются важными аспектами повышения экономической и экологической эффективности. Оптимизация кода, использование энергосберегающего оборудования и рациональное управление ресурсами позволяют снизить затраты и улучшить устойчивость информационных систем. Эти меры способствуют не только экономии ресурсов, но и сохранению окружающей среды [10].

ЗАКЛЮЧЕНИЕ

Результатом дипломной работы является игровое *2D* приложение «*Farmer's Valley*», представляющего собой аркадный фермерский симулятор в *top-down* проекции на платформе *Unity*. В процессе выполнения работы были достигнуты следующие результаты и решены поставленные задачи:

– проведен анализ современных методов разработки игровых приложений и выбраны наиболее подходящие средства и технологии для создания приложения;

– проведен аналитический обзор игр в жанре аркадного фермерского симулятора;

– разработан графический интерфейс пользователя, реализующий взаимодействие пользователя с программой;

– разработана эффективная и гибкая архитектура приложения, определены его компоненты и взаимодействие между ними, что обеспечило удобство разработки и возможность дальнейшего расширения приложения;

– проведено тестирование приложения, которое подтвердило его стабильность, надежность и эффективность, обнаруженные ошибки были исправлены, а функционал приложения проверен на соответствие заявленным требованиям;

Функционал игрового приложения реализован средствами платформы *Unity*.

Разработанное игровое приложение «*Farmer's Valley*» представляет собой продукт, объединяющий в себе передовые технологии и увлекательный игровой процесс. Его отличительные особенности включают высококачественную графику, интересный игровой процесс и возможность социального взаимодействия с *NPC*.

Проведенное тестирование показало, что игровое приложение выполняет свои функции, игровые механики функционируют должным образом. Кроме того, система является масштабируемой.

Как рекомендация по использованию результатов дипломной работы, предлагается дальнейшее развитие и улучшение приложения путем добавления новых функций, механик и контента.

В ходе выполнения дипломной работы были опубликованы две работы (Приложение Ж).

Список использованных источников

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

2. Гибсон, Б. Д. Unity и C#. Геймдев от идеи до реализации / Б. Д. Гибсон – СПб. : Питер, 2019. – 228 с.

3. Куксон, А. А. Разработка игр на Unity / А. А. Куксон – М. : Бомбора, 2019. – 298 с.

4. Арам Куксон, Д. Д. Разработка игр на UnrealEngine 4. 4-е изд. / А. Л. Куксон. – СПб. : Питер, 2017. – 917 с.

5. Фримен, Э. К. Паттерны проектирования / Э. К. Фримен – СПб. : Питер, 2011. – 176 с.

6. Feature driven development. – Электронные данные. – 2017. – Режим доступа: https://intellect.icu/feature-driven-development-5180. – Дата доступа: 16.05.2022.

7. Различные виды тестирования ПО. – Электронные данные. – Режим доступа: https://www.atlassian.com/ru/continuous-delivery/software-testing/types-of-software-testing. – Дата доступа: 16.05.2022.

8. Андрижиевский, А. А. Энергосбережение и энергетический менеджмент : учеб. пособие для студ. / А. А. Андрижиевский, В. И. Володин. – 2-е изд., испр. – Мн. : Вышэйшая школа, 2017. – 294 с.

9. Ляшенко, Д. Д. Ресурсосбережение: инновации в энергосбережении / Д. Д. Ляшенко, В. И. Мартынович // Международное научное обозрение проблем и перспектив современной науки и образования Сборник статей по материалам XXX Международной научно-практической конференции, 2018. – С.32.

10. Организация энергосбережения (энергоменеджмент). Решения ЗСМК-НКМКНТМК-ЕВРАЗ : учебное пособие / под ред. В. В. Кондратьева – М.: ИНФРАМ, 2017. – 108 с.

ПРИЛОЖЕНИЕ А
(обязательное)

Программный код приложения

Листинг класса ChatPanel:

```
using PimDeWitte.UnityMainThreadDispatcher;
using Scripts.SO.Chat;
using System;
using System.Threading.Tasks;
using TMPro;
using UnityEngine;
using UnityEngine.UI;
using Zenject;
using Assets.Scripts.GameSO.Chat;

namespace Scripts.ChatAssistant
{
   public class ChatPanel : MonoBehaviour
   {
      [SerializeField] private string Name;
      Transform _container;
      MassagePanelsFactories _massagePanelsFactories;

      ChatService _chatService;
      ChatContextOptimizer _chatContextOptimizer;

      [SerializeField] private TMP_InputField _inputField;
      [SerializeField] private Button _enterButton;
      [SerializeField] private Button _resetButton;
      [SerializeField] private GameObject _chatPanel;

      InputService _inputService;
      ChatSODataBase _chatSOData;


      [Inject]
      public void Construct(MassagePanelsFactories massagePanelsFactories,
         ChatSODataBase chatSODataBase,
         InputService inputService)
      {
         _chatSOData = chatSODataBase;
         _massagePanelsFactories = massagePanelsFactories;
         _inputService = inputService;
      }

      private void Start()
      {
         ChatSO chatSO = _chatSOData.GetItemByName(Name);

         if(chatSO == null)
         {
            Debug.LogError($"Cannot get chat SO Object by name {Name}");
```

```
            Destroy(gameObject);
        }
        else
        {
            _chatService = new ChatService(chatSO);

            _chatContextOptimizer = new ChatContextOptimizer(chatSO.EncodingModelID,
                chatSO.ResetContextSize);

            if(_chatContextOptimizer.TryInitialize() == false)
            {
                Debug.LogError($"Cannot initialize chat context optimizer by name {Name}");
                Destroy(gameObject);
            }
        }
        _container = gameObject.transform;
        _resetButton.interactable = false;
    }

    public async Task CreateMassagePanelAsync(IMassagePanelFactory massagePanelFactory, string text)
    {

        MassagePanel massagePanel = massagePanelFactory.CreateMassagePanel(_container);

        ResizeMassagePanel(massagePanel, text);

        await massagePanel.WriteTextAsync(text);
    }
    public void CreateMassagePanel(IMassagePanelFactory massagePanelFactory, string text)
    {

        MassagePanel massagePanel = massagePanelFactory.CreateMassagePanel(_container);

        ResizeMassagePanel(massagePanel, text);

        massagePanel.WriteText(text);
    }

    void ResizeMassagePanel(MassagePanel massagePanel, string text)
    {
        massagePanel.TextField.text = text;

        massagePanel.TextField.ForceMeshUpdate();

        var size = massagePanel.TextField.GetPreferredValues();

        float panelHeight = size.y;

        massagePanel.Size = new Vector2(massagePanel.Size.x, panelHeight + massagePanel.Size.y);
    }

    public async void OnEnterText()
    {
        if (_inputField.text.Length > 0)
        {
            IMassagePanelFactory playerPanelFactory =
_massagePanelsFactories.GetByType(MassagePanelType.Player);
```

```
            CreateMassagePanel(playerPanelFactory, _inputField.text);

            _enterButton.interactable = false;

            _chatService.AddUserInput(_inputField.text);

            _inputField.text = String.Empty;
            await ConstructResponse();
        }

    }
    public void OnReset()
    {
        _chatService.ClearChat();
        ClearMassagePanels();
        _enterButton.interactable = true;
        _resetButton.interactable = false;
    }

    async Task ConstructResponse()
    {
        await Task.Run(() =>
        {
            UnityMainThreadDispatcher.Instance().Enqueue(async () =>
            {
                string text = "";
                IMassagePanelFactory assistantPanelFactory =
_massagePanelsFactories.GetByType(MassagePanelType.Assistant);
                try
                {
                    text = await _chatService.GetResponseAsync();
                }
                catch (Exception ex)
                {
                    text += ex.Message;
                }
                finally
                {
                    await CreateMassagePanelAsync(assistantPanelFactory, text)
                        .ContinueWith((t) =>
                        {
                            UnityMainThreadDispatcher.Instance().Enqueue(() =>
                            {
                                _enterButton.interactable = true;
                                Debug.Log("End cunstruct response!");

                                if(_chatContextOptimizer.IsNeedReset(_chatService.GetContextText()))
                                {
                                    _enterButton.interactable = false;
                                    _resetButton.interactable = true;
                                }
                            });
                        });
                }
            });
        });
```

```
}
```

Листинг класса MassagePanel:

```
using PimDeWitte.UnityMainThreadDispatcher;
using Scripts.SO.Chat;
using System;
using System.Threading.Tasks;
using TMPro;
using UnityEngine;
using Zenject;

namespace Scripts.ChatAssistant
{
    public class MassagePanel : MonoBehaviour
    {
        [SerializeField] private RectTransform _rectTransform;
        [SerializeField] private RectTransform _childRect;
        [SerializeField] private RectTransform _textRect;
        [SerializeField] private TextMeshProUGUI _textField;
        public TextMeshProUGUI TextField => _textField;
        public float FontSize => TextField.fontSize;
        public Vector2 Size
        {
            get
            {
                return _rectTransform.rect.size;
            }
            set
            {
                _rectTransform.sizeDelta = new Vector2(value.x, value.y);
                _childRect.sizeDelta = new Vector2(_childRect.rect.size.x, value.y);
                _textRect.sizeDelta = new Vector2(_textRect.rect.size.x, value.y);
            }
        }

        MassageSO _massageSO;

        [Inject]
        public void Construct(MassageSO massageSO)
        {
            _massageSO = massageSO;
        }

        public async Task WriteTextAsync(string text)
        {
            await Task.Run(() =>
            {
                UnityMainThreadDispatcher.Instance().Enqueue(async () =>
                {
                    _textField.text = String.Empty;
                    foreach (var ch in text)
                    {
                        _textField.text += ch;
                        await Task.Delay(_massageSO.WriteSpeed);
```

```
                }
            });
        });
    }

    public void WriteText(string text)
    {
        _textField.text = String.Empty;
        foreach (var ch in text)
        {
            _textField.text += ch;
        }
    }
  }
}
```

Листинг класса ChatService:

```
using Scripts.SO.Chat;
using OpenAI_API;
using OpenAI_API.Chat;
using System.Threading.Tasks;
using System.Text;

namespace Scripts.ChatAssistant
{

  public class ChatService
  {
    ChatSO _chatSO;
    OpenAIAPI _api;
    APIAuthentication _authentication;

    Conversation _chat;

    StringBuilder _massagesTextBuilder;

    public ChatService(
      ChatSO chatSO)
    {
      _chatSO = chatSO;
      _authentication = new APIAuthentication(_chatSO.APIKey);

      _api = new OpenAIAPI(_authentication);
      _api.ApiUrlFormat = _chatSO.Url;
      _chat = _api.Chat.CreateConversation();
      _chat.Model.ModelID = _chatSO.ModelId;
      _chat.RequestParameters.Temperature = _chatSO.Temperature;
      _chat.RequestParameters.MaxTokens = _chatSO.MaxTokens;

      _massagesTextBuilder = new StringBuilder();

      AddSystemInput(_chatSO.SystemMassage);
    }
```

```
        public void AddUserInput(string text)
        {
            _chat.AppendUserInput(text);
        }

        void AddSystemInput(string text)
        {
            _chat.AppendSystemMessage(text);
        }

        public async Task<string> GetResponseAsync()
        {
            return await _chat.GetResponseFromChatbotAsync();
        }
        public string GetContextText()
        {
            _massagesTextBuilder.Clear();

            foreach (var massage in _chat.Messages)
            {
                _massagesTextBuilder.Append(massage.TextContent);
            }

            return _massagesTextBuilder.ToString();
        }
        public void ClearChat()
        {
            var massages = _chat.Messages;
            for (int i = 0; i < massages.Count; i++)
            {
                if (massages[i].Role.ToString() == "user" ||
                    massages[i].Role.ToString() == "assistant")
                {
                    massages.Remove(massages[i]);
                }
            }
        }
    }
}
```

Листинг класса InputService:

```
using System;
using UnityEngine;
using UnityEngine.InputSystem;
public class InputService : IDisposable
{
    const int c_firstItemCellIndex = 0;
    const int c_secondItemCellIndex = 1;
    const int c_thirdItemCellIndex = 2;
    const int c_fourthItemCellIndex = 3;
    InputActions _inputActions;

    public InputService()
    {
```

```
    _inputActions = new InputActions();
    _inputActions.Enable();
    InputActionAsset inputActionsAsset = _inputActions.asset;
}

public Vector2 GetMovement()
{
    return _inputActions.PlayerMap.Movement.ReadValue<Vector2>();
}
public bool IsChosenCell(out int index)
{
    index = 0;
    bool wasPerformed = false;
    if(_inputActions.InventoryMap.Choosefirstcell.WasPerformedThisFrame())
    {
        index = c_firstItemCellIndex;
        wasPerformed = true;
    }
    if (_inputActions.InventoryMap.Choosesecondcell.WasPerformedThisFrame())
    {
        index = c_secondItemCellIndex;
        wasPerformed = true;
    }
    if (_inputActions.InventoryMap.Choosethirdcell.WasPerformedThisFrame())
    {
        index = c_thirdItemCellIndex;
        wasPerformed = true;
    }
    if (_inputActions.InventoryMap.Choosefourthcell.WasPerformedThisFrame())
    {
        index = c_fourthItemCellIndex;
        wasPerformed = true;
    }
    return wasPerformed;
}

public bool IsLBK()
{
    return _inputActions.PlayerMap.Interact.WasPerformedThisFrame();
}

public bool IsOpenCloseMenu()
{
    return _inputActions.MenuActions.OpenClosegamemenu
        .WasPerformedThisFrame();
}
public void LockGamePlayControls()
{
    _inputActions.PlayerMap.Disable();
}
public void UnlockGamePlayControls()
{
    _inputActions.PlayerMap.Enable();
}
public void LockMenuControls()
{
    _inputActions.MenuActions.Disable();
```

```
    }
    public void UnlockMenuControls()
    {
        _inputActions.MenuActions.Enable();
    }
    public bool IsOpenCloseBackPack()
    {
        return _inputActions.InventoryMap.OpenClosebackpack.WasPerformedThisFrame();
    }
    public void Dispose()
    {
        _inputActions.Disable();
        _inputActions.Dispose();
    }
    public InputActionAsset GetInputActionAsset()
    {
        return _inputActions.asset;
    }
}
```

Листинг класса ActiveInventory:

```
using Scripts.SO.Inventory;
using Zenject;
using Scripts.PlacementCode;
using Scripts.PlayerCode;
using Scripts.Sounds;
using Random = UnityEngine.Random;

namespace Scripts.InventoryCode
{
    public class ActiveInventory : InventoryStorage
    {
        InputService _inputService;
        int _chosenIndex;
        Player _player;
        SoundService _soundService;
        public InventoryItem ChosenItem { get; private set; }
        MarkerController _markerController;

        [Inject]
        public void ConstructActive(InputService inputService,
            [Inject(Id = "ActiveInventoryInfo")] InventoryInfo inventoryInfo,
            MarkerController markerController,
            Player player,
            SoundService soundService,
            IInventoryCellFactory inventoryCellFactory)
        {
            _markerController = markerController;
            _soundService = soundService;
            _inputService = inputService;
            _player = player;
            base.ConstructStorage(inventoryInfo, inventoryCellFactory);
        }
```

```
public override void ConstructStorage([Inject(Id = "ActiveInventoryInfo")] InventoryInfo inventoryInfo,
    IInventoryCellFactory inventoryCellFactory)
{
}
protected override void Start()
{
    _chosenIndex = -1;
    _markerController.InteractMarker.Hide();
}
public override void Update()
{
    if (_inputService.IsChosenCell(out _chosenIndex) &&
        _chosenIndex < CurrentSize)
    {
        SelectCellByIndex(_chosenIndex);

    }

    ApplyItem();
}
void ApplyItem()
{
    if (ChosenItem != null &&
        ChosenItem.ApplyCondition(_markerController.CurrentTarget))
    {
        if(ChosenItem is ProductItem)
        {
            _markerController.InteractMarker.Hide();
        }
        else
        {
            _markerController.InteractMarker.Activate();
        }

        if(_inputService.IsLBK())
        {
            bool used = ChosenItem.Apply(_markerController.CurrentTarget);
            if(used)
            {
                _player.UseItemVisual(ChosenItem);

                if(ChosenItem.Consumable)
                {
                    ChosenItem.Count--;
                }

                if(ChosenItem.UseSound != null &&
                    ChosenItem.UseSound.Length > 0)
                {
                    _soundService.PlaySFXAt(_player.transform.position,
                        ChosenItem.UseSound[Random.Range(0, ChosenItem.UseSound.Length)], false);
                }
            }
        }
    }
    else
    {
```

```
            _markerController.InteractMarker.Hide();
        }
    }

    protected override void OnEndDrag()
    {
        base.OnEndDrag();

        SelectFirstCell();
    }
    void SelectCellByIndex(int index)
    {
        ChosenItem = InventoryItems[index];
        ChosenItem.IsSelected = true;
        DeactivateOther(ChosenItem);
    }
    void SelectFirstCell()
    {
        if (InventoryItems.Count > 0)
        {

            ChosenItem = InventoryItems[0];
            ChosenItem.IsSelected = true;
            DeactivateOther(ChosenItem);
        }
    }
    void DeactivateOther(InventoryItem selectedItem)
    {
        foreach (var item in InventoryItems)
        {
            if (selectedItem.Equals(item))
                continue;
            item.IsSelected = false;
        }
    }
    public override void OnDragInto(InventoryCell inventoryCell)
    {
    }

  }
}
```

Листинг класса InventoryBase:

```
using Scripts.Inventory;
using Scripts.MouseHandle;
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.EventSystems;
using Zenject;


namespace Scripts.InventoryCode
{
```

```
public abstract class InventoryBase : MonoBehaviour, IDragHandler, IBeginDragHandler, IEndDragHandler
{
    [SerializeField] protected Transform ContainerField;
    public Transform Container
    {
        get { return ContainerField; }
        private set { ContainerField = value; }
    }
    protected int TotalSize;
    protected int CurrentSize => InventoryItems.Count;
    protected List<InventoryItem> InventoryItems;
    protected IInventoryCellFactory _inventoryCellFactory;
    private Transform _globalVisualContext;
    private RectTransform _contextRect;


    protected Action<InventoryCell> OnBeginDragEvent;
    protected Action OnEndDragEvent;

    private GameObject _tmpEmptyCell;
    MouseCursor _mouseCursor;

    [Inject]
    public void Construct([Inject(Id = "DragParent")] Transform dragParent,
      MouseCursor mouseCursor)
    {
        _globalVisualContext = dragParent;
        _mouseCursor = mouseCursor;
    }
    private void Awake()
    {
        _contextRect = gameObject.GetComponent<RectTransform>();

    }
    protected virtual void OnEnable()
    {
        DragExtension.RegisterInventoryRectTransform
            (_contextRect);

    }
    protected virtual void OnDisable()
    {
        DragExtension.UnregisterInventoryRectTransform(_contextRect);
    }
    private void OnDestroy()
    {
        OnBeginDragEvent -= OnBeginDragCell;
        OnEndDragEvent -= OnEndDrag;
    }
    protected virtual void Start()
    {
    }
    public void Initialize(List<InventoryItem> inventoryItems)
    {
        OnBeginDragEvent += OnBeginDragCell;
        OnEndDragEvent += OnEndDrag;
        InventoryItems = new List<InventoryItem>(TotalSize);
```

```
    InventoryItems.InsertRange(0, inventoryItems);
    foreach (var item in inventoryItems)
    {
        CreateCellForItem(item);
    }
    foreach (var item in inventoryItems)
    {
        item.IsSelected = false;
    }
}

public void RegisterDragEvents(InventoryCell inventoryCell)
{
    inventoryCell.RegisterEvents(OnEndDragEvent, OnBeginDragEvent);
}

public bool AddItem(InventoryItem newItem)
{
    InventoryItem newItemCopy = (InventoryItem)newItem.Clone();
    int remainingToFit = newItemCopy.Count;

    //first we check if there is already that item in the inventory
    for (int i = 0; i < InventoryItems.Count; ++i)
    {
        if (InventoryItems[i].UniqueName == newItemCopy.UniqueName
            && InventoryItems[i].Count < newItemCopy.MaxStackSize)
        {
            int fit = Mathf.Min(newItemCopy.MaxStackSize
                - InventoryItems[i].Count,
                remainingToFit);
            InventoryItems[i].Count += fit;
            remainingToFit -= fit;

            if (remainingToFit == 0)
                return true;
        }
    }

    newItemCopy.Count = remainingToFit;
    CreateCellForItem(newItemCopy);
    InventoryItems = OverwriteInventoryItemsSequence();


    return remainingToFit == 0;
}
public void RemoveItem(int itemIndex)
{
    for (int i = 0; i < Container.childCount; i++)
    {
        if(i == itemIndex)
        {
            Transform cell = Container.GetChild(i);
            Destroy(cell.gameObject);
        }
    }
}
void CreateCellForItem(InventoryItem inventoryItem)
```

```
{
    InventoryCell newCell = _inventoryCellFactory.Create(Container);
    newCell.Initialize(_globalVisualContext, inventoryItem);
    RegisterDragEvents(newCell);
}
/// <summary>
/// Возвращает false, если инвентарь заполнен
/// </summary>
/// <returns></returns>
public bool IsFull()
{
    if (CurrentSize >= TotalSize)
    {
        return true;
    }
    else
    {
        return false;
    }
}

protected virtual void OnBeginDragCell(InventoryCell inventoryCell)
{
    _tmpEmptyCell = _inventoryCellFactory.CreateEmpty(inventoryCell);

}
protected virtual void OnEndDrag()
{
    if (_tmpEmptyCell != null)
        Destroy(_tmpEmptyCell);
    InventoryItems = OverwriteInventoryItemsSequence();
}

public virtual void OnDrag(PointerEventData eventData)
{

}

List<InventoryItem> OverwriteInventoryItemsSequence()
{
    List<InventoryItem> items = new List<InventoryItem>(TotalSize);
    for (int i = 0; i < Container.childCount; i++)
    {
        var cellObject = Container.GetChild(i);
        InventoryCell inventoryCell;
        if (cellObject.TryGetComponent(out inventoryCell))
        {
            items.Add(inventoryCell.InventoryItem);
        }
    }
    return items;
}
public List<InventoryItem> GetItems()
{
    return InventoryItems;
}
public List<ItemContextData> GetItemsContextData()
```

```
        {
            List<ItemContextData> itemContextDatas = new List<ItemContextData>();

            for (int i = 0; i < Container.childCount; i++)
            {
                var name = Container.name;
                var index = i;
                if(Container.GetChild(index).TryGetComponent(out InventoryCell cell))
                {
                    ItemContextData itemContextData = new ItemContextData(
                        cell.InventoryItem, index, name);

                    itemContextDatas.Add(itemContextData);
                }
            }
            return itemContextDatas;
        }


        public virtual void OnDragInto(InventoryCell inventoryCell)
        {

        }

        public void OnBeginDrag(PointerEventData eventData)
        {
            _mouseCursor.ChangeCursor(CursorType.Drag);
        }

        public void OnEndDrag(PointerEventData eventData)
        {
            _mouseCursor.ChangeCursor(CursorType.Default);
        }
    }
}
```

Листинг класса Crop:

```
using UnityEngine;
using UnityEngine.Tilemaps;
using UnityEngine.VFX;


namespace Scripts.InventoryCode
{
    [CreateAssetMenu(fileName = "Crop", menuName = "SO/InventoryItems/Crop")]
    public class Crop : ScriptableObject,
        IDataBaseItem
    {
        public string UniqueName => _name;
        [SerializeField] private string _name;

        public TileBase[] GrowthStagesTiles;

        public ProductItem Produce;
```

```
        public float GrowthTime = 1.0f;
        public int NumberOfHarvest = 1;
        public int StageAfterHarvest = 1;
        public int ProductPerHarvest = 1;
        public float DryDeathTimer = 30.0f;
        public VisualEffect PickEffect;

        public int GetGrowthStage(float growRatio)
        {
            return Mathf.FloorToInt(growRatio * (GrowthStagesTiles.Length - 1));
        }
    }
}
```

Листинг класса HoeItem:

```
using Scripts.PlacementCode;
using UnityEngine;

namespace Scripts.InventoryCode
{
    [CreateAssetMenu(fileName = "Hoe", menuName = "SO/InventoryItems/Hoe")]
    public class HoeItem : InventoryItem
    {
        public override bool Apply(Vector3Int target)
        {
            PlacementService.Instance().TillAt(target);
            return true;
        }

        public override bool ApplyCondition(Vector3Int target)
        {

            return PlacementService.Instance() != null &&
                PlacementService.Instance().IsTillable(target);
        }

        public override object Clone()
        {
            HoeItem hoeItem =
                CreateInstance<HoeItem>();
            hoeItem.InitializeCopy(this);
            return hoeItem;
        }
        public override InventoryItemData GetData()
        {
            HoeItemData hoeItemData = new();
            hoeItemData.Init(base.GetData());
            return hoeItemData;
        }
    }
}
```

Листинг класса InventoryItem:

```
using System;
using UnityEngine;


namespace Scripts.InventoryCode
{
   public abstract class InventoryItem : ScriptableObject, IDataBaseItem, ICloneable
   {
      public bool IsSelected { get; set; }

      public string DisplayName = "";

      public Sprite Icon => _icon;

      public Color Color => _color;

      public string UniqueName => _name;

      public int MaxStackSize = 10;
      public int Count
      {
         get
         {
            return _startCount;
         }
         set
         {
            _startCount = value;

            if (_startCount < 0)
               _startCount = 0;
         }
      }

      [SerializeField] private int _startCount;
      public bool Consumable = true;
      public int BuyPrice = -1;

      [SerializeField] private string _name;

      [SerializeField] private Sprite _icon;

      [ColorUsage(true)]
      [SerializeField] private Color _color;

      [SerializeField] protected bool IsCountTextActive;

      [Tooltip("Prefab that will be instantiated in the player hand when this is equipped")]
      public GameObject VisualPrefab;

      [Tooltip("Sound triggered when using the item")]
      public AudioClip[] UseSound;

      public string PlayerAnimatorTriggerUse = "GenericToolSwing";
```

```
        public virtual void InitializeCopy(InventoryItem inventoryItem)
        {
            DisplayName = inventoryItem.DisplayName;
            _icon = inventoryItem.Icon;
            _color = inventoryItem.Color;
            _name = inventoryItem.UniqueName;
            _startCount = inventoryItem.Count;
            MaxStackSize = inventoryItem.MaxStackSize;
            BuyPrice = inventoryItem.BuyPrice;
            Consumable = inventoryItem.Consumable;
            IsCountTextActive = inventoryItem.IsCountTextActive;
            VisualPrefab = inventoryItem.VisualPrefab;
            UseSound = inventoryItem.UseSound;
            PlayerAnimatorTriggerUse = inventoryItem.PlayerAnimatorTriggerUse;
        }
        public abstract bool ApplyCondition(Vector3Int target);

        public abstract bool Apply(Vector3Int target);

        public virtual void RenderUI(InventoryCell inventoryCell)
        {
            inventoryCell.Icon.sprite = _icon;
            inventoryCell.NameDisplayText.text = DisplayName;
            inventoryCell.NameDisplayText.color = _color;
            inventoryCell.CountText.color = _color;
            inventoryCell.CountText.text = Count.ToString();
            inventoryCell.CountText.gameObject.SetActive(IsCountTextActive);
            inventoryCell.SelectIcon.gameObject.SetActive(IsSelected);
        }


        public virtual bool NeedTarget()
        {
            return true;
        }

        public abstract object Clone();
        public virtual InventoryItemData GetData()
        {
            InventoryItemData inventoryItemData = new InventoryItemData()
            {
                SoName = UniqueName,
                Amount = Count
            };
            return inventoryItemData;
        }

    }
}
```

Листинг класса ProductItem:

```
using UnityEngine;
```

```
namespace Scripts.InventoryCode
{
   [CreateAssetMenu(fileName = "Product", menuName = "SO/InventoryItems/Product")]
   public class ProductItem : InventoryItem
   {
      public int SellPrice = 1;

      public override void InitializeCopy(InventoryItem inventoryItem)
      {
         base.InitializeCopy(inventoryItem);
         if(inventoryItem is ProductItem product)
         {
            SellPrice = product.SellPrice;
         }
      }
      public override bool Apply(Vector3Int target)
      {
         return true;
      }

      public override void RenderUI(InventoryCell inventoryCell)
      {
         base.RenderUI(inventoryCell);
      }

      public override bool ApplyCondition(Vector3Int target)
      {
         return Count > 0;
      }

      public override bool NeedTarget()
      {
         return false;
      }

      public override object Clone()
      {
         ProductItem productItem =
            CreateInstance<ProductItem>();
         productItem.InitializeCopy(this);
         return productItem;
      }
      public override InventoryItemData GetData()
      {
         ProductItemData productItemData = new();
         productItemData.Init(base.GetData());
         return productItemData;
      }
   }
}
```

Листинг класса InventoryCell:

```
using PimDeWitte.UnityMainThreadDispatcher;
using Scripts.InventoryCode.ItemResources;
using Scripts.MouseHandle;
using System;
using System.Threading.Tasks;
using TMPro;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;
using Zenject;


namespace Scripts.InventoryCode
{
    public class InventoryCell : MonoBehaviour, IDragHandler, IBeginDragHandler, IEndDragHandler
    {
        [SerializeField] Image IconElement;
        [SerializeField] TextMeshProUGUI TextElement;
        [SerializeField] TextMeshProUGUI CountTextElement;
        [SerializeField] Image SelectIconElement;
        Action _endDragEvent;
        Action<InventoryCell> _beginDragEvent;
        public InventoryItem InventoryItem { get; private set; }

        public Transform GlobalVisualContext { get; private set; }
        public Transform OriginVisualContext { get; private set; }

        public Image Icon => IconElement;
        public TextMeshProUGUI NameDisplayText => TextElement;
        public TextMeshProUGUI CountText => CountTextElement;
        public Image SelectIcon => SelectIconElement;

        public int BeginDragSiblingIndex { get; private set; }

        ItemResourceDroper _itemResourceDroper;
        MouseCursor _mouseCursor;

        [Inject]
        public void Construct(ItemResourceDroper itemResourceDroper,
            MouseCursor mouseCursor)
        {
            _itemResourceDroper = itemResourceDroper;
            _mouseCursor = mouseCursor;
        }


        private void OnDisable()
        {

        }
        private void Start()
        {
        }
        public void Initialize(Transform globalVisualContext,
            InventoryItem inventoryItem)
        {
            GlobalVisualContext = globalVisualContext;
```

```
    OriginVisualContext = transform.parent;
    InventoryItem = inventoryItem;

}
public void RegisterEvents(Action endDragEvent, Action<InventoryCell> beginDragEvent)
{
    _endDragEvent = endDragEvent;
    _beginDragEvent = beginDragEvent;
}
private async void Update()
{
    InventoryItem?.RenderUI(this);
    if(InventoryItem.Consumable &&
        InventoryItem.Count < 1)
    {
        Destroy(gameObject);
        await InvokeEndDragOnItemOver();
    }
}

async Task InvokeEndDragOnItemOver()
{
    await Task.Delay(100);
    UnityMainThreadDispatcher.Instance().Enqueue(() =>
    {
        Debug.Log("InvokeEndDragOnItemOver");
        _endDragEvent?.Invoke();
    });
}

public void OnDrag(PointerEventData eventData)
{
    transform.position = Input.mousePosition;
}
public void OnBeginDrag(PointerEventData eventData)
{
    BeginDragSiblingIndex = transform.GetSiblingIndex();
    _beginDragEvent?.Invoke(this);
    transform.SetParent(GlobalVisualContext);
    _mouseCursor.ChangeCursor(CursorType.Drag);
}

public async void OnEndDrag(PointerEventData eventData)
{
    _mouseCursor.ChangeCursor(CursorType.Default);
    InventoryBase inventory;
    if (DragExtension.CheckMouseIntersectionWithContainers(eventData,
        out inventory))// если есть пересечение с инвентарем
    {
        // если это тот же инвентарь
        if (OriginVisualContext == inventory.Container)
        {
            await DragExtension.PlaceInTheNearestCellLocal(OriginVisualContext,
            this, BeginDragSiblingIndex).ContinueWith((i) =>
            {
                UnityMainThreadDispatcher.Instance().Enqueue(() =>
                {
```

```
                    _endDragEvent?.Invoke();

                });
            });
        }
        else//другой инвентарь
        {
            if (inventory.IsFull())// если полон, то не перекладывать
            {
                await DragExtension.PlaceInTheNearestCellLocal(OriginVisualContext,
                this, BeginDragSiblingIndex).ContinueWith((i) =>
                {
                    UnityMainThreadDispatcher.Instance().Enqueue(() =>
                    {
                        _endDragEvent?.Invoke();

                    });
                });
            }
            else// переложить и переподписать ячейку на события другого инвентаря
            {
                DragExtension.PlaceInTheNearestCellGlobal(inventory.Container, this);
                OriginVisualContext = inventory.Container;
                _endDragEvent?.Invoke();
                inventory.RegisterDragEvents(this);
                _endDragEvent?.Invoke();
                inventory.OnDragInto(this);
                return;
            }
        }
    }
    else// нет пересечений с другим инвентарем
    {
        _endDragEvent?.Invoke();
        Destroy(this.gameObject);
        _itemResourceDroper.DropByPlayer(InventoryItem);
    }
    _endDragEvent?.Invoke();
  }
 }
}
```

Листинг класса ItemContextData:

```
using Scripts.InventoryCode;

namespace Scripts.Inventory
{
  public class ItemContextData
  {
    public InventoryItem Item { get; private set; }
    public int Index { get; private set; }
    public string ContextName { get; private set; }
    public ItemContextData(InventoryItem inventoryItem,
      int index, string Name)
```

```
        {
            Item = inventoryItem;
            Index = index;
            ContextName = Name;
        }
    }
}
```

Листинг класса Pack:

```
using UnityEngine;
using Zenject;

namespace Scripts.InventoryCode
{
    public class Pack : MonoBehaviour
    {
        [SerializeField] GameObject InventoryObject;

        InputService _inputService;

        [Inject]
        public void Construct(InputService inputService)
        {
            _inputService = inputService;
        }


        public void Update()
        {
            if (_inputService.IsOpenCloseBackPack())
            {
                if (InventoryObject.activeSelf)
                {
                    InventoryObject.SetActive(false);
                }
                else
                {
                    InventoryObject.SetActive(true);
                }
            }
        }
    }
}
```

Листинг класса PlayerInventory:

```
using Scripts.FarmGameEvents;
using Scripts.Inventory;
using Scripts.InventoryCode.ItemResources;
using Scripts.SaveLoader;
using System.Collections.Generic;
using UnityEngine;
using Zenject;
```

```
namespace Scripts.InventoryCode
{
    public class PlayerInventory : MonoBehaviour
    {
        static PlayerInventory s_instance;
        [SerializeField] InventoryBase _activePackInventory;
        [SerializeField] InventoryBase _backPackInventory;
        [SerializeField] private CanvasGroup _playerGroup;
        GameDataState _gameDataState;

        private void Awake()
        {
            if (s_instance == null)
            {
                s_instance = this;
            }
        }

        [Inject]
        public void Construct(GameDataState gameDataState)
        {
            _gameDataState = gameDataState;
        }

        private void OnEnable()
        {
            GameEvents.OnTradePanelOpenClose += OnTradePanelAction;
            GameEvents.OnExitTheGameEvent += OnExitTheGame;
        }
        private void OnDisable()
        {
            GameEvents.OnTradePanelOpenClose -= OnTradePanelAction;
            GameEvents.OnExitTheGameEvent -= OnExitTheGame;
        }

        public bool TryAddItem(InventoryItem inventoryItem)
        {
            if (_backPackInventory.IsFull() == false)
            {
                if (_backPackInventory.AddItem(inventoryItem))
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
            else if (_activePackInventory.IsFull() == false)
            {
                if (_activePackInventory.AddItem(inventoryItem))
                {
                    return true;
                }
                else
                {
```

```
            return false;
        }
    }
    return false;
}
public bool TryPickupResource(ItemResource itemResource)
{
    if (_backPackInventory.IsFull() == false)
    {
        _backPackInventory.AddItem(itemResource.InventoryItem);
        return true;
    }
    else if (_activePackInventory.IsFull() == false)
    {
        _activePackInventory.AddItem(itemResource.InventoryItem);
        return true;
    }
    return false;
}
public bool IsFull()
{
    if (!_backPackInventory.IsFull() ||
        !_activePackInventory.IsFull())
    {
        return false;
    }
    else
        return true;
}
public static PlayerInventory Instance()
{
    return s_instance;
}

public List<InventoryItem> GetAllItems()
{
    List<InventoryItem> inventoryItems = new List<InventoryItem>();

    inventoryItems.AddRange(_activePackInventory.GetItems());
    inventoryItems.AddRange(_backPackInventory.GetItems());

    return inventoryItems;
}

public List<ItemContextData> GetAllItemsContextData()
{
    List<ItemContextData> itemContextDatas = new List<ItemContextData>();

    itemContextDatas.AddRange(_activePackInventory.GetItemsContextData());
    itemContextDatas.AddRange(_backPackInventory.GetItemsContextData());

    return itemContextDatas;
}

public void RemoveItem(ItemContextData itemContextData)
{
    if(itemContextData.ContextName == _activePackInventory.Container.name)
```

```
            {
                _activePackInventory.RemoveItem(itemContextData.Index);
            }
            if(itemContextData.ContextName == _backPackInventory.Container.name)
            {
                _backPackInventory.RemoveItem(itemContextData.Index);
            }
        }
        void OnExitTheGame()
        {
            var activePackItems = _activePackInventory.GetItems();
            var backPackItems = _backPackInventory.GetItems();
            _gameDataState.UpdateActivePackInventory(activePackItems);
            _gameDataState.UpdateBackPackInventory(backPackItems);
        }
        void OnTradePanelAction(bool isIgnore)
        {
            _playerGroup.blocksRaycasts = isIgnore;
        }

    }
}
```

Листинг класса PlacementService:

```
using Scripts.InventoryCode;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Tilemaps;
using UnityEngine.VFX;
using Zenject;
using Scripts.SaveLoader;
using Crop = Scripts.InventoryCode.Crop;
using Scripts.FarmGameEvents;


namespace Scripts.PlacementCode
{
    public class PlacementService : MonoBehaviour
    {
        static PlacementService s_instance;
        Grid _grid;
        CropDataBase _cropDatabase;
        GameDataState _gameDataState;
        [Inject]
        public void Construct(Grid grid,
            CropDataBase cropDataBase,
            GameDataState gameDataState
            )
        {
            _grid = grid;
            _gameDataState = gameDataState;
            _cropDatabase = cropDataBase;
        }
```

```
public Tilemap GroundTilemap;
public Tilemap CropTilemap;

[Header("Watering")]
public Tilemap WaterTilemap;
public TileBase WateredTile;

[Header("Tilling")]
public TileBase TilleableTile;
public TileBase TilledTile;
public VisualEffect TillingEffectPrefab;

private Dictionary<Crop, List<VisualEffect>> harvestEffectPool = new();
private List<VisualEffect> _tillingEffectPool = new();

public class GroundData
{
   public const float WaterDuration = 60 * 1.0f;

   public float WaterTimer;
}
public class CropData
{


   public Crop GrowingCrop = null;
   public int CurrentGrowthStage = 0;

   public float GrowthRatio = 0.0f;
   public float GrowthTimer = 0.0f;

   public int HarvestCount = 0;

   public float DyingTimer { get; set; }
   public bool HarvestDone => HarvestCount == GrowingCrop.NumberOfHarvest;


   public Crop Harvest()
   {
      var crop = GrowingCrop;

      HarvestCount += 1;

      CurrentGrowthStage = GrowingCrop.StageAfterHarvest;
      GrowthRatio = CurrentGrowthStage / (float)GrowingCrop.GrowthStagesTiles.Length;
      GrowthTimer = GrowingCrop.GrowthTime * GrowthRatio;

      return crop;
   }

   public void Save(PlacedCropData placedCropData)
   {
      placedCropData.Stage = CurrentGrowthStage;
      placedCropData.CropId = GrowingCrop.UniqueName;
      placedCropData.DyingTimer = DyingTimer;
      placedCropData.GrowthRatio = GrowthRatio;
      placedCropData.GrowthTimer = GrowthTimer;
```

```
        placedCropData.HarvestCount = HarvestCount;
    }

    public void Load(PlacedCropData placedCropData,
        CropDataBase cropDataBase)
    {
        CurrentGrowthStage = placedCropData.Stage;
        GrowingCrop = cropDataBase.GetItemByName(placedCropData.CropId);
        DyingTimer = placedCropData.DyingTimer;
        GrowthRatio = placedCropData.GrowthRatio;
        GrowthTimer = placedCropData.GrowthTimer;
        HarvestCount = placedCropData.HarvestCount;
    }
}


private Dictionary<Vector3Int, GroundData> _groundData = new();
private Dictionary<Vector3Int, CropData> _cropData = new();

private void Awake()
{
    if(s_instance == null)
    {
        s_instance = this;
    }
}
private void OnEnable()
{
    GameEvents.OnExitTheGameEvent += OnExitTheGame;
}
private void OnDisable()
{
    GameEvents.OnExitTheGameEvent -= OnExitTheGame;
}

private void Start()
{
    for (int i = 0; i < 4; ++i)
    {
        var effect = Instantiate(TillingEffectPrefab);
        effect.gameObject.SetActive(true);
        effect.Stop();
        _tillingEffectPool.Add(effect);
    }
}

private void Update()
{
    foreach (var (cell, groundData) in _groundData)
    {
        if (groundData.WaterTimer > 0.0f)
        {
            groundData.WaterTimer -= Time.deltaTime;

            if (groundData.WaterTimer <= 0.0f)
            {
                WaterTilemap.SetTile(cell, null);
```

```
            //GroundTilemap.SetColor(cell, Color.white);
        }
    }

    if (_cropData.TryGetValue(cell, out var cropData))
    {
        if (groundData.WaterTimer <= 0.0f)
        {
            cropData.DyingTimer += Time.deltaTime;
            if (cropData.DyingTimer > cropData.GrowingCrop.DryDeathTimer)
            {
                _cropData.Remove(cell);
                UpdateCropVisual(cell);
            }
        }
        else
        {
            cropData.DyingTimer = 0.0f;
            cropData.GrowthTimer = Mathf.Clamp(cropData.GrowthTimer + Time.deltaTime, 0.0f,
                cropData.GrowingCrop.GrowthTime);
            cropData.GrowthRatio = cropData.GrowthTimer / cropData.GrowingCrop.GrowthTime;
            int growthStage = cropData.GrowingCrop.GetGrowthStage(cropData.GrowthRatio);

            if (growthStage != cropData.CurrentGrowthStage)
            {
                cropData.CurrentGrowthStage = growthStage;
                UpdateCropVisual(cell);
            }
        }
    }
  }
}

public bool IsTillable(Vector3Int target)
{
  return GroundTilemap.GetTile(target) == TilleableTile;
}

public bool IsPlantable(Vector3Int target)
{
  return IsTilled(target) && ! _cropData.ContainsKey(target);
}

public bool IsTilled(Vector3Int target)
{
  return _groundData.ContainsKey(target);
}

public void TillAt(Vector3Int target)
{
  if (IsTilled(target))
    return;

  GroundTilemap.SetTile(target, TilledTile);
  _groundData.Add(target, new GroundData());

  var inst = _tillingEffectPool[0];
```

```
    _tillingEffectPool.RemoveAt(0);
    _tillingEffectPool.Add(inst);

    inst.gameObject.transform.position = _grid.GetCellCenterWorld(target);

    inst.Stop();
    inst.Play();
}

public void PlantAt(Vector3Int target, Crop cropToPlant)
{
    var cropData = new CropData();

    cropData.GrowingCrop = cropToPlant;
    cropData.GrowthTimer = 0.0f;
    cropData.CurrentGrowthStage = 0;

    _cropData.Add(target, cropData);

    UpdateCropVisual(target);

    if (!harvestEffectPool.ContainsKey(cropToPlant))
    {
        InitHarvestEffect(cropToPlant);
    }
}
public Crop HarvestAt(Vector3Int target)
{
    _cropData.TryGetValue(target, out var data);

    if (data == null || !Mathf.Approximately(data.GrowthRatio, 1.0f)) return null;

    var produce = data.Harvest();

    if (data.HarvestDone)
    {
        _cropData.Remove(target);
    }

    UpdateCropVisual(target);

    var effect = harvestEffectPool[data.GrowingCrop][0];
    effect.transform.position = _grid.GetCellCenterWorld(target);
    harvestEffectPool[data.GrowingCrop].RemoveAt(0);
    harvestEffectPool[data.GrowingCrop].Add(effect);
    effect.Play();

    return produce;
}

public void WaterAt(Vector3Int target)
{
    var groundData = _groundData[target];

    groundData.WaterTimer = GroundData.WaterDuration;

    WaterTilemap.SetTile(target, WateredTile);
```

```
    //GroundTilemap.SetColor(target, WateredTiledColorTint);
}

void UpdateCropVisual(Vector3Int target)
{
    if (!_cropData.TryGetValue(target, out var data))
    {
        CropTilemap.SetTile(target, null);
    }
    else
    {
        CropTilemap.SetTile(target, data.GrowingCrop.GrowthStagesTiles[data.CurrentGrowthStage]);
    }
}
public CropData GetCropDataAt(Vector3Int target)
{
    _cropData.TryGetValue(target, out var data);
    return data;
}
public void InitHarvestEffect(Crop crop)
{
    harvestEffectPool[crop] = new List<VisualEffect>();
    for (int i = 0; i < 4; ++i)
    {
        var inst = Instantiate(crop.PickEffect);
        inst.Stop();
        harvestEffectPool[crop].Add(inst);
    }
}

void OnExitTheGame()
{
    Save();
}
void Save()
{
    PlacementData placementData = new PlacementData();

    foreach (var cropData in _cropData)
    {
        TilePlacementData tilePlacementData = new TilePlacementData();
        tilePlacementData.SetPosition(cropData.Key);

        placementData.CropsTilePlacementDatas.Add(tilePlacementData);

        PlacedCropData placedCropData = new PlacedCropData();
        cropData.Value.Save(placedCropData);

        placementData.PlacedCropDatas.Add(placedCropData);
    }

    foreach (var ground in _groundData)
    {
        TilePlacementData tilePlacementData = new TilePlacementData();
        tilePlacementData.SetPosition(ground.Key);

        placementData.GroundTilePlacementDatas.Add(tilePlacementData);
```

```
            placementData.GroupDatas.Add(ground.Value);
        }

        _gameDataState.UpdatePlacementData(placementData);

    }
    public void Load()
    {
        PlacementData placementData = _gameDataState.PlacementData;

        _groundData = new Dictionary<Vector3Int, GroundData>();
        for (int i = 0; i < placementData.GroupDatas.Count; ++i)
        {
            var pos = placementData.GroundTilePlacementDatas[i].GetPosition();
            _groundData.Add(pos, placementData.GroupDatas[i]);

            GroundTilemap.SetTile(pos, TilledTile);

            WaterTilemap.SetTile(pos, placementData.GroupDatas[i].WaterTimer > 0.0f ? WateredTile : null);
            //GroundTilemap.SetColor(data.GroundDataPositions[i], data.GroundDatas[i].WaterTimer > 0.0f ?
WateredTiledColorTint : Color.white);
        }

        //clear all existing effect as we will reload new one
        foreach (var pool in harvestEffectPool)
        {
            if (pool.Value != null)
            {
                foreach (var effect in pool.Value)
                {
                    Destroy(effect.gameObject);
                }
            }
        }


        _cropData = new Dictionary<Vector3Int, CropData>();
        for (int i = 0; i < placementData.PlacedCropDatas.Count; ++i)
        {
            CropData newData = new CropData();
            newData.Load(placementData.PlacedCropDatas[i], _cropDatabase);

            var pos = placementData.CropsTilePlacementDatas[i].GetPosition();
            _cropData.Add(pos, newData);

            UpdateCropVisual(pos);

            if (!harvestEffectPool.ContainsKey(newData.GrowingCrop))
            {
                InitHarvestEffect(newData.GrowingCrop);
            }
        }
    }

    public static PlacementService Instance()
    {
```

```
            return s_instance;
        }
    }
}
```

Листинг класса TradeElement:

```
using Scripts.InventoryCode;
using TMPro;
using UnityEngine;
using UnityEngine.UI;
using Zenject;


namespace Scripts.SellBuy
{
    public abstract class TradeElement : MonoBehaviour
    {
        public Image IconElement => _iconElementField;
        public TextMeshProUGUI IconName => _iconNameField;
        public TextMeshProUGUI ButtonText => _buttonTextField;

        protected Button ButtonElement => _buttonElementField;

        [SerializeField] private Image _iconElementField;
        [SerializeField] private TextMeshProUGUI _iconNameField;
        [SerializeField] private TextMeshProUGUI _buttonTextField;
        [SerializeField] private Button _buttonElementField;

        protected TradeService TradeService;

        protected InventoryItem Item;

        [Inject]
        public void Construct(TradeService tradeService)
        {
            TradeService = tradeService;
        }
        public void Init(InventoryItem item)
        {
            Item = item;
        }
        public virtual void OnButtonAction()
        {
            if(Trade())
                Destroy(gameObject);
        }


        public abstract bool Trade();
    }
}
```

Листинг класса TradePanel:

```
using Scripts.InventoryCode;
using System.Collections.Generic;
using System.Text;
using UnityEngine;
using Zenject;
using Scripts.InteractableObjects;
using Scripts.Inventory;
using Scripts.FarmGameEvents;
using UnityEngine.EventSystems;
using Scripts.MouseHandle;

namespace Scripts.SellBuy
{
    public class TradePanel : MonoBehaviour, IDragHandler, IBeginDragHandler, IEndDragHandler
    {
        [SerializeField] Transform ContentArea;

        TradeService _tradeService;
        PlayerInventory _playerInventory;
        StringBuilder _buttonTextBuilder = new StringBuilder();
        FactoriesProvider _factoriesProvider;
        ITradeElementFactory _sellElementFactory;
        ITradeElementFactory _buyElementFactory;
        BuyItemsDatabase _buyItemsDatabase;
        MouseCursor _mouseCursor;
        InputService _inputService;

        [Inject]
        public void Construct(TradeService tradeService,
            PlayerInventory playerInventory,
            FactoriesProvider factoriesProvider,
            BuyItemsDatabase buyItemsDatabase,
            MouseCursor mouseCursor,
            InputService inputService)
        {
            _inputService = inputService;
            _mouseCursor = mouseCursor;
            _tradeService = tradeService;
            _playerInventory = playerInventory;
            _factoriesProvider = factoriesProvider;
            _buyItemsDatabase = buyItemsDatabase;
        }
        private void OnEnable()
        {
            GameEvents.OnSellItemEvent += OnSellItem;
        }
        private void OnDisable()
        {
            GameEvents.OnSellItemEvent -= OnSellItem;
        }
        private void Start()
        {
            _sellElementFactory =
                (SellTradeElementFactory)_factoriesProvider.GetFactory<SellTradeElementFactory>();
            _buyElementFactory =
```

```
        (BuyTradeElementFactory)_factoriesProvider.GetFactory<BuyTradeElementFactory>();
    gameObject.SetActive(false);
}

public void OnBuy()
{
    ClearElements();
    List<InventoryItem> inventoryItems =
        _playerInventory.GetAllItems();
    foreach (var item in _buyItemsDatabase.Items)
    {

        var UIElement = _buyElementFactory.Create(ContentArea);
        var itemClone = item.Clone() as InventoryItem;
        UIElement.Init(itemClone);
        UIElement.IconElement.sprite = item.Icon;
        UIElement.IconName.text = item.DisplayName;

        int amount = _tradeService.ItemAmount(item);
        _buttonTextBuilder.Append($"Spend {item.BuyPrice} " +
            $"for {itemClone.Count}");

        UIElement.ButtonText.text = _buttonTextBuilder.ToString();

        _buttonTextBuilder.Clear();
    }
}
public void OnSell()
{
    ClearElements();

    var itemsContext = _playerInventory.GetAllItemsContextData();

    //DebugItemContextData(itemsContext);

    foreach (ItemContextData contextData in itemsContext)
    {
        if (_tradeService.SellCondition(contextData.Item))
        {
            var UIElement = _sellElementFactory.Create(ContentArea);

            if( UIElement is SellTradeElement tradeElement)
            {
                tradeElement.InitializeItemContext(contextData);
                tradeElement.Init(contextData.Item.Clone() as InventoryItem);
            }


            UIElement.IconElement.sprite = contextData.Item.Icon;
            UIElement.IconName.text = contextData.Item.DisplayName;

            Debug.Log("Ставлю :" +  contextData.Item.DisplayName);

            _buttonTextBuilder.Append($"Take {contextData.Item.Count * contextData.Item.BuyPrice} " +
                $"for {contextData.Item.Count}");

            UIElement.ButtonText.text = _buttonTextBuilder.ToString();
```

```
                _buttonTextBuilder.Clear();
            }
        }
    }
    void DebugItemContextData(List<ItemContextData> itemContextDatas)
    {
        StringBuilder stringBuilder = new StringBuilder();

        foreach (var item in itemContextDatas)
        {
            stringBuilder.Append("Get " + item.Item.DisplayName + " ");
        }
        Debug.Log(stringBuilder.ToString());
    }
    public void OnExit()
    {
        ClearElements();
        gameObject.SetActive(false);
        GameEvents.InvokeTradePanelActionEvent(true);
        _inputService.UnlockGamePlayControls();
    }
    void ClearElements()
    {
        for (int i = 0; i < ContentArea.childCount; i++)
        {
            Destroy(ContentArea.GetChild(i).gameObject);
        }
    }

    void OnSellItem()
    {
        OnSell();
    }

    public void OnDrag(PointerEventData eventData)
    {
        Vector3 mousePos = Input.mousePosition;

        gameObject.transform.position = mousePos;
    }

    public void OnBeginDrag(PointerEventData eventData)
    {
        _mouseCursor.ChangeCursor(CursorType.Drag);
    }

    public void OnEndDrag(PointerEventData eventData)
    {
        _mouseCursor.ChangeCursor(CursorType.Default);
    }
  }
}
```

Листинг класса TradeService:

```
using Scripts.InventoryCode;
using Scripts.PlayerCode;
using Zenject;
using UnityEngine;
using Scripts.Inventory;

namespace Scripts.SellBuy
{
   public class TradeService
   {
      PlayerInventory _playerInventory;
      PlayerMoney _playerMoney;
      [Inject]
      public void Construct(PlayerInventory playerInventory,
         PlayerMoney playerMoney)
      {
         _playerInventory = playerInventory;
         _playerMoney = playerMoney;
      }
      public bool BuyCondition(InventoryItem item)
      {
         if(_playerInventory.IsFull() && item.BuyPrice > 0)
         {
            return false;
         }
         if(item.Consumable)
         {
            return item.BuyPrice <= _playerMoney.Money;
         }
         else
         {
            return false;
         }

      }

      public void Buy(InventoryItem item, int amount)
      {
         for (int i = 0; i < amount; i++)
         {
            InventoryItem inventoryItem =
               (InventoryItem)item.Clone();
            _playerInventory.TryAddItem(inventoryItem);
         }
         _playerMoney.Money -= item.BuyPrice;
      }
      public bool SellCondition(InventoryItem item)
      {
         return item is ProductItem && item.BuyPrice > 0;

      }
      public void Sell(ItemContextData contextData)
      {
         var item = contextData.Item;
         Debug.Log("Удаляю :" + item.DisplayName);
```

```
            _playerInventory.RemoveItem(contextData);

            _playerMoney.Money += item.BuyPrice * item.Count;
        }
        public int ItemAmount(InventoryItem item)
        {
            return Mathf.RoundToInt(_playerMoney.Money / item.BuyPrice);
        }

    }
}
```

ПРИЛОЖЕНИЕ Б
(обязательное)

Руководство системного программиста

1. Общие сведения о программе.

Разработанное игровое приложение предназначено для игры между двумя игроками на одном компьютере или на разных устройствах по сети. Разработанное игровое приложение предназначено для развития внимания и реакции, а также для развлечения. Кроме того, приложение развивает концентрацию и внимание и значительно улучшает память, позволяя запоминать всё большие объёмы информации.

Для корректной работы приложения необходима следующая конфигурация технических средств аппаратного обеспечения:

– центральный процессор *Intel Core* 2 *Duo c* тактовой частотой 2.30 МГц или более;

– наличие клавиатуры, мыши и монитора *SVGA* с разрешением не менее 1640 на 750 пикселей;

– операционная система *Windows* 7 и выше;

– 100 Мб оперативной памяти;

– скорость интернет-соединения не ниже 100 килобит в секунду.

2. Структура программы.

Игровое приложение логически можно разбить на несколько составляющих: игровой движок, содержащий средства работы с графикой, непосредственно логика игровых объектов и игрового процесса, проект сетевого взаимодействия и графический интерфейс пользователя.

3. Настройка программы.

Приложение необходимо запускать от имени администратора. Для запуска решения необходима среда разработки *Visual Studio* с установленными фреймворком *.NET* и *Unity* 2021.3.16f1.

4. Дополнительные возможности.

Приложение является узконаправленным и не имеет дополнительных возможностей.

ПРИЛОЖЕНИЕ В
(обязательное)

Руководство программиста

1) Назначения и условия применения программы.

Игровое приложение является платформой для однопользовательской игры на одном устройстве. Основное применение – получение досуга. Доступен лишь один режим – игра на двоих по сети.

Программа показала свою максимальную производительность при взаимодействии с:

– операционной системой *Windows* 7;

– компьютерными средствами: клавиатура, мышь;

– устройствами вывода: монитор.

2) Характеристика программы.

Сразу же после запуска программы загружается главное меню. В главном меню необходимо нажать соответствующую клавишу, сразу после этого начинается игровой процесс, который длиться до тех пор, пока не будет нажата клавиша перехода на главную сцену и клавиша закрытия приложения после.

3) Обращение к программе.

Для старта работы приложения необходимо запустить файл с расширением *Farmer.exe*.

4) Входные и выходные данные.

В качестве средств разработки для приложения используется язык программирования *C#*, игровой движок *Unity* 2021.3.16f1.

5) Сообщение программисту.

При возникновении непредвиденных ошибок или остановке работы приложения рекомендуется провести перезапуск всего приложения.

ПРИЛОЖЕНИЕ Г
(обязательное)

Руководство пользователя

1) Введение.

Игровое приложение «*Farmer's Valley*» является аркадной платформой для однопользовательской игры на одном устройстве. Основное применение – получение досуга.

Приложение имеет два этапа работы: главное меню, непосредственный игровой процесс.

2) Назначение и условия применения.

Игровое приложение «*Farmer's Valley*» является аркадной платформой для многопользовательской игры по сети.

Программа показала свою максимальную производительность при взаимодействии с:

– операционной системой *Windows* 7;

– компьютерными средствами: клавиатура, мышь;

– устройствами вывода: компьютерный монитор.

3) Подготовка к работе.

Для старта работы приложения необходимо запустить файл с расширением *Farmer.exe*.

4) Описание операций.

Основные манипуляции со стороны игрока:

– старт игры;

– выбор игрового состояния;

– перемещение игрового персонажа по игровому полю влево и вправо;

– перемещение игрового персонажа по игровому полю вверх;

– взаимодействие с интерфейсом;

– окончание игры.

5) Непредвиденные ситуации.

Программа грамотно верифицирована, но в ситуации возникновения ошибки рекомендуется полный перезапуск игрового приложения.

6) Рекомендации.

Данная игра не требует высокого склада ума и хорошей интуиции, каждый желающий игрок может насладиться этой замечательной и увлекательной игрой.

ПРИЛОЖЕНИЕ Д
(справочное)

Формулы расчета экономической эффективности

Для расчёта экономической эффективности разработанного программного продукта, используются следующие формулы:

$$К_{эк} = \frac{К_{т.н}}{К_{т.б}}, \quad (Д.1)$$

где $К_{т.н}$, $К_{т.б}$ – коэффициенты технического уровня нового и базисного программного продукта, которые можно рассчитать по формуле (Д.2):

$$К_m = \sum_{i=1}^{n} \beta \frac{P_i}{P_э}, \quad (Д.2)$$

где β – коэффициенты весомости i-го технического параметра;

n – число параметров;

$Р_i$ – численное значение i-го технического параметра сравниваемого программного продукта;

$Р_э$ – численное значение i-го технического параметра эталона.

$$К_{ф.в} = \frac{К_{ф.в.н}}{К_{ф.в.б}}, \quad (Д.3)$$

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

$$К_и = \frac{К_{эк} \cdot К_{ф.в} \cdot К_н}{К_ц}, \quad (Д.4)$$

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

$К_ц$ – коэффициент цены потребления.

$$V_o = \sum_{i=1}^{n} V_i, \quad (Д.5)$$

где V_i – объем отдельной функции ПО;

n – общее число функций.

$$V_y = \sum_{i=1}^{n} V_{yi}, \qquad (Д.6)$$

где V_{yi} – уточненный объем отдельной функции ПО в строках исходного кода.

$$К_с = 1 + \sum_{i=1}^{n} К_i, \qquad (Д.7)$$

где $К_i$ – коэффициент, соответствующий степени повышения сложности;

n – количество учитываемых характеристик.

$$Т_{у.т.з} = Т_н \cdot К_{т.з} \cdot К_с \cdot К_н \cdot К_{у.р}, \qquad (Д.8)$$

$$Т_{у.э.п} = Т_н \cdot К_{э.п} \cdot К_с \cdot К_н \cdot К_{у.р}, \qquad (Д.9)$$

$$Т_{у.т.п} = Т_н \cdot К_{т.п} \cdot К_с \cdot К_н \cdot К_{у.р}, \qquad (Д.10)$$

$$Т_{у.р.п} = Т_н \cdot К_{р.п} \cdot К_с \cdot К_н \cdot К_т \cdot К_{у.р}, \qquad (Д.11)$$

$$Т_{у.в.н} = Т_н \cdot К_{в.н} \cdot К_с \cdot К_н \cdot К_{у.р}, \qquad (Д.12)$$

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.

$$Т_о = \sum_{i=1}^{n} Т_{yi}, \qquad (Д.13)$$

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

n – количество стадий разработки.

$$З_р = З_{тр} + З_{эт} + З_{тех} + З_{м.в} + З_{мат} + З_{общ.пр} + З_{непр}, \qquad (Д.14)$$

$$З_{тр} = ЗП_{осн} + ЗП_{доп} + ОТЧ_{зп}, \qquad (Д.15)$$

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.

$$ЗП_{осн}=С_{ср.час}\cdot Т_{о}\cdot К_{ув}, \quad (Д.16)$$

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

$Т_{о}$ – общая трудоемкость разработки, чел.-час;

$К_{ув}$– коэффициент доплаты стимулирующего характера, $К_{ув}$= 1,8.

$$С_{ср.час}=\frac{\sum_i С_{чi}\cdot n_i}{\sum_i n_i}, \quad (Д.17)$$

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

n_i – количество разработчиков *i*-й категории.

$$С_{ч}=\frac{С_{м1}\cdot Т_{к1}}{F_{мес}}, \quad (Д.18)$$

где $С_{м1}$ – тарифная ставка 1-го разряда;

$Т_{к1}$ – тарифный коэффициент.

$$ЗП_{доп}=\frac{ЗП_{осн}\cdot Н_{доп}}{100}, \quad (Д.19)$$

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

$$ОТЧ_{с.н}=\frac{(ЗП_{осн}+ЗП_{доп})\cdot Н_{з.п}}{100}, \quad (Д.20)$$

где $Н_{з.п}$– процент отчислений на социальные нужды и обязательное страхование от суммы основной и дополнительной заработной платы ($Н_{з.п}$ = 36%).

$$З_{м.в}=С_{ч}\cdot К_{т}\cdot t_{эвм}, \quad (Д.21)$$

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

$К_{т}$ – коэффициент мультипрограммности, показывающий распределение времени работы ЭВМ в зависимости от количества пользователей ЭВМ, $К_{т}$= 1;

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

$$С_ч = \frac{ЗП_{об} + З_{ар} + З_{ам} + З_{э.п} + З_{в.м} + З_{т.р} + З_{пр}}{F_{эвм}}, \quad (Д.22)$$

где $ЗП_{об}$ – затраты на заработную плату обслуживающего персонала с учетом всех отчислений, руб./год;

$З_{ар}$ – стоимость аренды помещения под размещение вычислительной техники, руб./год;

$З_{ам}$ – 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.

$$ЗП_{об} = \frac{ЗП_{осн.об} + ЗП_{доп.об} + ОТЧ_{эп.об}}{100}, \quad (Д.23)$$

$$ЗП_{осн.об} = 12 \cdot \sum_i (С_{м.об}i \cdot n_i), \quad (Д.24)$$

$$ЗП_{доп.об} = \frac{ЗП_{осн.об} \cdot Н_{доп}}{100}, \quad (Д.25)$$

$$ОТЧ_{зп.об} = \frac{(ЗП_{осн.об} + ЗП_{доп.об}) \cdot Н_{зп}}{100}, \quad (Д.26)$$

где $ЗП_{осн.об}$ – основная заработная плата обслуживающего персонала, руб.;

$ЗП_{доп.об}$ – дополнительная заработная плата обслуживающего персонала, руб.;

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

$Q_{эвм}$ – количество обслуживаемых ПЭВМ, шт.;

$С_{м.об}i$ – месячная тарифная ставка i-го работника, руб.;

n – численность обслуживающего персонала, чел.;

$Н_{доп}$ – процент дополнительной заработной платы обслуживающего персонала от основной;

$Н_{зп}$ – процент отчислений на социальные нужды и обязательное страхование от суммы основной и дополнительной заработной платы.

$$З_{ар}= \frac{С_{ар}\cdot S}{Q_{эвм}}, \qquad (Д.27)$$

где $С_{ар}$ – средняя годовая ставка арендных платежей, руб./м^2;

S – площадь помещения, м^2.

$$З_{ам}= \frac{\sum_i З_{пр i}(1 + К_{доп})\, m_i \cdot Н_{ам i}}{100}, \qquad (Д.28)$$

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

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

$З_{пр i}(1+К_{доп})$ – балансовая стоимость ЭВМ, руб;

$Н_{ам i}$ – норма амортизации, %.

$$З_{эвм}= \frac{М_{сум}\cdot F_{эвм}\cdot С_{эл}\cdot A}{100}, \qquad (Д.29)$$

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

$М_{сум}$ = 0,44 кВт;

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

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

$$F_{эвм}= \left(Д_г - Д_{вых} - Д_{пр}\right)\cdot F_{см}\cdot К_{см}\cdot(1 - К_{пот}), \qquad (Д.30)$$

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

$Д_{вых}$, $Д_{пр}$ – число выходных и праздничных дней в году, $Д_{вых}+Д_{пр}$ = 121 дней;

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

$К_{см}$ – коэффициент сменности, $К_{см}$= 1;

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

$$З_{в.м}= \sum_i З_{пр i}\left(1 + К_{доп}\right)m_i \cdot К_{м.з}, \qquad (Д.31)$$

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

$К_{доп}$ – коэффициент, характеризующий дополнительные 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).

$$З_{т.р}=\sum_i З_{прi}\left(1 + К_{доп}\right)m_i \cdot К_{т.р}, \quad (Д.32)$$

где $К_{т.р}$ – коэффициент, характеризующий затраты на текущий и профилактический ремонт, $К_{т.р}$ = 0,08.

$$З_{пр}=\sum_i З_{прi}\left(1 + Д_{доп}\right)m_i \cdot К_{пр}, \quad (Д.33)$$

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

$$t_{эвм} = \left(t_{р.п} + t_{вн}\right) \cdot F_{см} \cdot К_{см}, \quad (Д.34)$$

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

$t_{вн}$ – срок реализации стадии «Ввод в действие» (ВП);

$t_{р.п} + t_{вн}$ = 33;

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

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

$$З_{эт}=\left(З_{т.р}+З_{тех}+З_{м.в}\right)К_{эт}, \quad (Д.35)$$

где $К_{эт}$– коэффициент, учитывающий размер затрат на изготовление эталонного экземпляра, $К_{эт}$= 0,05.

$$З_{мат}=\sum_i Ц_i N_i(1+К_{т.з}) - Ц_{0i}N_{0i}, \quad (Д.36)$$

где $Ц_i$ – цена i-го наименования материала полуфабриката, комплектующего, руб.;

N_i – потребность в i-м материале, полуфабрикате, комплектующем, натур. ед.;

$К_{т.з}$ – коэффициент, учитывающий сложившийся процент транспортно-заготовительных расходов в зависимости от способа доставки товаров, $К_{т.з}$= 0,1;

$Ц_{0i}$ – цена возвратных отходов i-го наименования материала, руб.;

N_{0i} – количество возвратных отходов i-го наименования, натур. ед.;

n – количество наименований материалов, полуфабрикатов, и т.д.

$$З_{общ.пр}= \frac{ЗП_{осн}\cdot Н_{доп}}{100}, \quad (Д.37)$$

где $Н_{доп}$ – норматив общепроизводственных затрат.

$$З_{непр}= \frac{ЗП_{осн}\cdot Н_{непр}}{100}, \quad (Д.38)$$

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

$$Ц_{отп} = З_р + П_р, \quad (Д.39)$$

$$П_р= \frac{З_р\cdot У_р}{100}, \quad (Д.40)$$

где $З_р$ – себестоимость ПО, руб.;

$П_р$ – прибыль от реализации программного продукта, руб.;

$У_р$ – уровень рентабельности программного продукта, % ($У_р$= 30 %).

$$Ц_{отп}= З_р+ П_р+ Р_{ндс}, \quad (Д.41)$$

$$Р_{ндс}= \frac{(З_р + П_р)\cdot Н_{ндс}}{100}, \quad (Д.42)$$

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

$$Э = (N \cdot С_к)\cdot(100 – Т_{ах}), \quad (Д.43)$$

где N – ожидаемое количество релизованных копий в год, N = 100;

$С_к$ – стоимость одной копии, $С_к = 38$;

$Т_{ах}$ – подоходный налог, $Т_{ах} = 20\%$ для ИП.

$$N_{TAX} = Э \cdot Т_{ах} \div (100 – Т_{ах}). \quad (Д.44)$$

$$Р = Э \cdot 100 \div З_р. \quad (Д.45)$$

$$Т = З_р \div Э. \quad (Д.46)$$

ПРИЛОЖЕНИЕ Ё
(обязательное)

Результат опытной эксплуатации

Приложение оперирует при следующей минимальной конфигурации аппаратного обеспечения:

– процессор с архитектурой *x86-64* и тактовой частотой не менее 2.0 ГГц;

– клавиатура и мышь, обеспечивающие стандартное взаимодействие с приложением;

– монитор с разрешением 1280 на 720 пикселей или выше;

– операционная система *Windows 7/8/10*, *macOS 10.12* и выше или *Linux*;

– 4 Гб оперативной памяти;

– графический процессор с поддержкой *DirectX 11* или *OpenGL 4.2*, с 2 Гб видеопамяти или более.

Опытная эксплуатация проводилась в течение двух часов на персональных компьютерах различной конфигурации, используемых разными пользователями. В ходе тестирования осуществлялась проверка следующих аспектов:

– запуск приложения;

– полное прохождение игры пользователем;

– взаимодействие с инвентарем;

– корректное отображение интерфейса;

– корректная работа торговли;

– загрузка и сохранение игрового состояния;

– закрытие приложения.

Необходимости в доработке или устранении проблем не выявлено, так как в процессе эксплуатации не было выявлено никаких ошибок или недочетов.

ПРИЛОЖЕНИЕ Ж
(рекомендуемое)

Список опубликованных работ

1. Дубовцов, И.Д., Кравченко, О.А. Искусственный интеллект в играх жанра RPG / И. Д. Дубовцов, О. А. Кравченко // Новые математические методы и компьютерные технологии в проектировании, производстве и научных исследованиях : научное издание, материалы XXVII Республиканской научной конференции студентов и аспирантов (Гомель, 18 –20 марта 2024 года). – Гомель: ГГУ им. Франциско Скорины, 2024. – С. 192 – 193;

2. Дубовцов, И. Д., научный руководитель. Кравченко, О.А. Применение генеративного ИИ для игр в жанре RPG / И. Д. Дубовцов, О. А. Кравченко // E.R.A – Современная наука: электроника, робототехника, автоматизация [Электронный ресурс] : материалы I Междунар. науч.-техн. конф, студентов, аспирантов и молодых ученых, Гомель, 29 фев. 2024 г. / Гомел. гос. техн. ун-т им. П. О. Сухого [и др.] ; под общ. ред. А. А. Бойко. – Гомель : ГГТУ им. П. О. Сухого, 2024. – С. 231 – 232.