**1 АНАЛИТИЧЕСКИЙ ОБЗОР СФЕРЫ РАЗРАБОТКИ ИГРОВЫХ ПРИЛОЖЕНИЙ**

**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 можно выделить следующие:

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

– игровые консоли (специализированные устройства для игр, *Xbox One*, *PlayStation 4*, *Nintendo Wii U*;

– 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*, *КРИ*, *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 сева 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 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 15* 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 *Farming Simulator*.



Рисунок 1.1 Кадр 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 прокачивать 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, наслаждаясь процессом 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, преграждение 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 над 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, связывая игровые ресурсы 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*). 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.

Оба подхода схематично показаны на рисунке 2.1.



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 [5, c.17].

**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 *ААА*-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 *С++*.

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

*Unreal* *Engin* 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.

На рисунке 1.4 представлен интерфейс *Unreal Engine*.

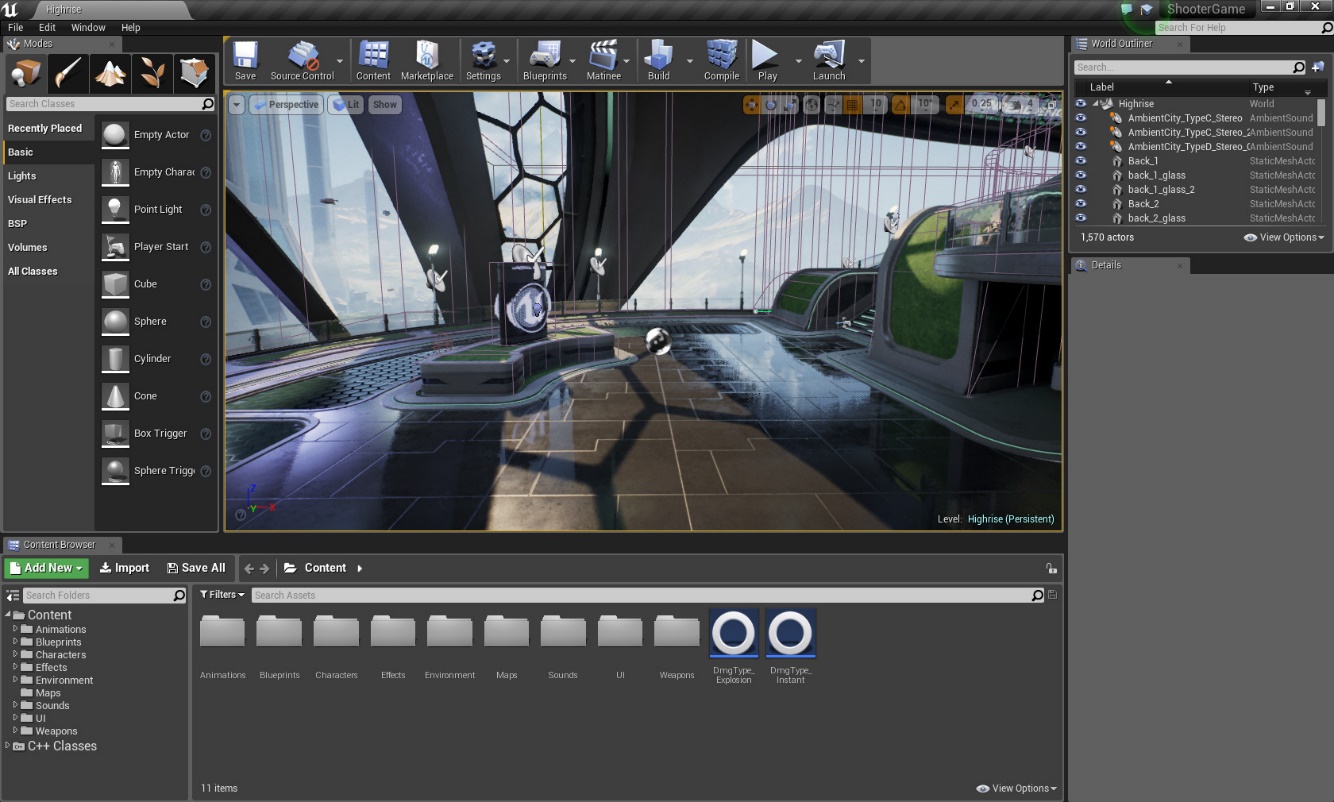


Рисунок 1.4 – интерфейс *Unreal Engine*

Для тех, кто далек от *кодинга*, 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.

*Blueprints* значительно проще для понимания и использования, чем *С++*, 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 мэши 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 *UE4* – в части *Details*. *Jayanam* также сравнивает возможности *Unity* *Prefabs* c *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 *UE4* 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 *UE4* 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 *UE4* 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 [7].

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 *UE4* встроен очень мощный редактор материалов, основанный, как и система *Blueprints*, на нодах.

Для программирования в *UE4* 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 *UE4* очень мощная и проработанная система визуального скриптования – *Blueprints*, с помощью которой можно достичь практически тех же результатов, что и *c C++*.

*Unity 5* поддерживает языки *C#* и *UnityScript*. *API* и его концепт очень похож на аналог из *UE4*. 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*. *UE4* также поддерживает спрайты в *Paper 2d*, но решения из *Unity* мощнее, кроме того, в последнем есть отдельный физический движок для *2d*-объектов.

В *UE4* 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 *UE4* – эффекты применяются только стеком или скриптами к камере.

*Sequencer* в *UE4* 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 выбрал игровой движок *Unity*, т.к. он обладает наибольшим инструментарием и информационной базой, которые в один момент являются удобными и простыми [6].

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:

– cоздание 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;

– громкости музыки и звуковых эффектов;

– 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 о 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 *1024x768* 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 обеспечивает мощные инструменты для разработки 2*D* игр, 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*»

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.

2.1 Определение основных функций

Для разработки архитектуры приложения важно выделить основные функции, которые будет реализовывать приложение. 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 конкретикой 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 расценивать за компоненты 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, которые являются своеобразными исходными данными, которые будут импортироваться и находится в рабочей области платформы.

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.

2.2 Структурные элементы

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

Основные структурные элементы, которые будут использоваться в проекте включают:

– 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 логическое составляющее.

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 любые 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 модуле.

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.

2.3 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 при этом сократив количество 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

Исходя из этого 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 конструктор 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 конструкторы, 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 из себя информацию о его местоположении, текущей анимации.

Сущность *Marker*, который представляет из себя 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 об предмете инвентаря.

Сущность *Inventory* представляет иp себя хранилище данных об 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 использует данные о вводе пользователя.

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

Сушность Chat представляет из себя данные о истории общения между игроком и NPC.

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

Сущность *Installer* представляет из себя входные данные для сущностей игры.

2.4 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 из себя главное меню, где пользователь с помощью *GUI* будет запускать игровую сцену. На рисунке 2.5 представлен каркас пользовательского интерфейса на начальной сцене.



Рисунок 2.5 – 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. Также 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 приложения.

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 2.6.



Рисунок 2.6 – 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 диегетические 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*).

Все элементы *HUD* по сути являются отдельными интерфейсами, каждый из которых существует для передачи игроку информации определенного типа и решения своей уникальной задачи. При этом *HUD*, полностью или частично, практически всегда находится непосредственно перед глазами игрока, так как все его элементы отображаются на переднем плане геймплея и/или в игровом мире во время игры.

В игровой сцене необходимо придумать каскад для следующих элементов интерфейса:

– текущее время суток;

– игровая валюта;

– активный инвентарь;

– вспомогательный инвентарь.

На рисунке 2.8 представлен каркас для игрового *HUD*.



Рисунок 2.8 – Каркас для игрового *HUD*

Такая организация *UI* элементов позволит корректно отображать необходимую информацию, при этом почти на заграждая обзор игрока на игровой мир.

**3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ**

**«*FARMER'S VALLEY*»**

**3.1 Взаимодействие игровых элементов.**

Приложения 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 составляющее игрового приложения и программное составляющие игрового приложения. Графическим составляющим являются спрайты, визуальные эффекты, настройки графики, анимации. Все эти элементы могут являться частями одного игрового объекта. Такие элементы инициализируются с помощью подобранного программного обеспечения, учитывая 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.

**3.1 Принцип работы модуля управления игровым миром и управления фермой.**

Модуль управления игровым миров 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 из себя 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, который содержит вложенные классы CropData, GroundData.

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.

Чтобы располагать игровые объекты на сетке, соблюдая некоторые условия, содержит методы, описанные в таблице 3.2.

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 визуальные 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 исходя из сохраненных данных;

– 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;

– 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(Placed CropData)* |
| *Load* | *Public* | *Void* | *placedCropData (placedCropData), cropDataBase (cropDataBase)* |

Методы, описанные в таблице 3.4 сдержат следующую логику:

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

– Save – инициализирует структуру данных для сохранения своего состояния;

– Load – инициализирует свое состояния 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:

– InventoryCell;

– PlayerInventory;

– ActiveInventory;

– ChestInventory;

– InventoryStorage;

– InventoryItem;

– ProductItem;

– SeedBagItem;

– WateringItem;

– Crop;

– HoeItem;

– WateringItem;

– ItemContextData;

– Pack.

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 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 об 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 |
| 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 метода. Сигнатура методов класса InventoryBase показана в таблице 3.11.

Таблица 3.11 – Сигнатура методов класса InventoryBase

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Модификатор доступа | Тип возвращаемого значения | Параметры(тип данных) |
| Initialize | Public | Void | InventoryItems(bool) |
| RegisterDragEvents | Public | Void | inventoryCell(int) |

Таблица 3.11 – Сигнатура методов класса InventoryBase

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Модификатор доступа | Тип возвращаемого значения | Параметры(тип данных) |
| 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 ячейеу 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 списко 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, TradePanel, TradeElement.

TradeService – класс, который содержит логику для 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 | contextData(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) |
| OnBuy | Public | Void | – |
| 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, ChatPanel, MassagePanel. В этом модуле используются библиотека OpenAI и SharpToken. Библиотека OpenAI нужна, чтобы организовывать запросы к API ChatGPT. Библиотека SharpTokern применяется для оптимизации запросов путем подсчитывания 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) |

Таблица 3.14 – Сигнатура методов класса ChatService

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Модификатор доступа | Тип возвращаемого значения | Параметры(тип данных) |
| 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, которая содержит сообщение, введенное игроком или сообщения 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. 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 изменяя его поведение.

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 раздает реализации фабрик.

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 запуская 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 названия 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;

– название игрового состояния не должно 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 из себя 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 удалиться с компьютера и не появиться 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 присутствую следующие эельементы графического интерфейса:

– Screen Resolution;

– Full Screen;

– Main Volume;

– SFX Volime;

– 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, который влияет на остальные значения звука. 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 и остальных.

С помощью элементом смены клавиши взаимодействия можно поменять клавишу взаимодействия путям нажатия на кнопку выбранного действия и нажатия новой клавиши после. Нажав на кнопку Back пользователь вернется к исходному виду.

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, ограниченные забором. 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.

На рисунке 4.6 представлен результат использования предмета мотыга, лейка и мешок с семенами.



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

Нажав на Npc рядом появится панель, на которой находятся графические элементы интерфейса, такие как кнопка Talk и Trade. На рисунке 4.7 представлен результат нажатия на 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.

На панели покупки и продажи находятся следующие элементы графического интерфейса: кнопка Buy и кнопка Sell. 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. На этих элементах отображается название продукции и сколько игрок получит игровой валюты за продажу текущего количества определенной продукции. 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.

Пользовательский графический интерфейс является основным инструментом пользователя п

При эксплуатации игры было проанализировано соответствиетехническому заданию. Так же была оценена целостность геймплея и удобство в управлении.

**4.1 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, которые требуют большой вычислительной мощности и максимально нагружают устройство, на котором запущено игровое приложение. 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 мобильного устройства. Например, в игровую сцену добавлялось большое количество предметов экипировки и противников, что моделировало ситуацию перегрузки сцены графикой, если игрок решил бы пробежаться по локациям и собрать за собой большое количество преследующих его противников для более острых ощущений от игрового процесса. Благодаря подобным тестам ужалось 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 мобильных устройствах, имеющих различные технически характеристики.

**4.2 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 соответствие отображения игрового интерфейса на устройстве с отображением в среде разработки.

Ожидаемый результат: на экране отображен интерфейс игры, отображаются: панель игрового интерфейса, содержащая шкалу здоровья, шкалу опыта, кнопки вызова панелей инвентаря, характеристик и квестов, а также кнопка, вызывающая меню паузы. Все элементы интерфейса выглядят уместно и не подвергнуты деформации.

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

Тест № 3. Цель: проверить работоспособность кнопок главного меню.

Ожидаемый результат: в зависимости от нажатой в главном меню кнопки происходит запуск игровой сцены, отображение окна регулировки звука или выход из приложения.

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

Тест № 4. Цель: проверить соответствие отображения игрового интерфейса на устройстве с отображением в среде разработки.

Ожидаемый результат: на экране отображен интерфейс игры, отображаются: панель игрового интерфейса, содержащая шкалу здоровья, шкалу опыта, кнопки вызова панелей инвентаря, характеристик и квестов, а также кнопка, вызывающая меню паузы. Все элементы интерфейса выглядят уместно и не подвергнуты деформации.

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

Тест № 5. Цель: проверить работоспособность меню паузы.

Ожидаемый результат: при нажатии кнопки паузы время в игре останавливается, появляется панель функциональными с кнопками меню.

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

Тест № 6. Цель: проверить работоспособность передвижения игрока по игровой сцене.

Ожидаемый результат: при нажатии на точку на земле игроком персонаж должен переместиться в указанное место кратчайшим путём, при этом персонаж игрока должен быть повёрнут лицом в сторону движения. Также 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 № 7. Цель: проверить корректность работоспособности атаки и получения урона персонажем.

Ожидаемый результат: персонаж игрока при нажатии на противника должен наносить ему урон, а также терять очки здоровья при ответных атаках.

Вывод: игрок атакует противника, наносит урон и теряет очки здоровья при ответной атаке, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 8. Цель: проверить корректность функционирования механизма смерти и возрождения персонажа.

Ожидаемый результат: когда здоровье персонажа становится равным нулю, он должен быть перемещён в точку возрождения.

Вывод: персонаж умирает и возрождается корректно, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 9. Цель: проверить корректность функционирования механизма поднятия предметов, их появления в инвентаре и удаления из него.

Ожидаемый результат: после двойного нажатия на лежащий на земле предмет его трёхмерное представление в сцене должно исчезнуть, а в инвентаре при наличии свободного слота появиться иконка соответствующего предмета. Так же при нажатии крестика в углу заполненного слота иконка должна исчезнуть, очистив слот, а в игровой сцене появиться трёхмерное представление предмета.

Вывод: предметы корректно добавляются в инвентарь и удаляются из него, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 10. Цель: проверить корректность начисления очков опыта персонажу игрока.

Ожидаемый результат: очки опыта должны начисляться персонажу игрока в следующих случаях:

* при убийстве противника. Очки начисляются в соответствии с типом противника;
* при завершении квеста. Очки начисляются в соответствии с прописанной в квесте наградой

Вывод: на протяжении игровой сессии корректно начислялись очки опыта при убийстве противников и выполнении квестов, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 11. Цель: проверить возможность достижения нового уровня и присвоение очков характеристик.

Ожидаемый результат: при получении достаточного количества очков опыта происходит повышение уровня персонажа игрока, ему присваивается очко для повышения характеристик

Вывод: при получении достаточного количества опыта повышается уровень персонажа, что позволяет повысить одну из его характеристик с помощью очков повышения, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 12. Цель: проверить корректность улучшения характеристик персонажа игрока при использовании очков повышения. Проверить работу кнопок «+» и «-» в рамках имеющихся очков повышения характеристик и подтвердить изменения с помощью соответствующей кнопки.

Ожидаемый результат: при наличии очков повышения характеристик рядом с численным значением на панели характеристик персонажа появляется кнопка «+», нажатие на которую прибавляет единицу к выбранной характеристике, после чего появляется кнопка «-», позволяющая вернуть распределённое очко. Любые изменения численного показателя характеристик возможны только при наличии очков повышения характеристик и до нажатия кнопки подтверждения распределения.

Вывод: при наличии очков повышения характеристик и нажатии соответствующих кнопок происходит изменение численного значения повышаемой характеристики персонажа игрока, которое вступает в силу и не может быть изменено после нажатия кнопки подтверждения распределения, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 13. Цель: проверить правильное воспроизведение звуковых эффектов.

Ожидаемый результат: звуковые эффекты воспроизводятся в соответствии с прописанным сценарием.

Вывод: звуковые эффекты воспроизводятся корректно и в необходимый момент игровой сессии, ожидаемый и полученный результат совпадают, цель теста достигнута.

Тест № 14. Цель: проверить корректное обновление графического интерфейса на различных панелях.

Ожидаемый результат: детали графического интерфейса должны изменяться в случае:

* панель инвентаря – при сборе, экипировке или удалении предмета из инвентаря;
* панель характеристик персонажа – при экипировке снаряжения, которое повышает характеристики, при повышении уровня, при распределении очков повышения характеристик;
* панель квестов – при принятии квеста, в процессе прогресса квеста, после завершения квеста;
* шкала здоровья – при получении урона персонажем игрока, при повышении соответствующей характеристики при восстановлении здоровья;
* шкала опыта – при получении очков опыта путём победы над противниками и выполнения квестов;

Вывод: все элементы графического интерфейса, а именно панель инвентаря, панель характеристик персонажа, панель квестов, шкала здоровья, шкала опыта корректно обновляется после определённых действий и в соответствующих обстоятельствах, ожидаемый и полученный результат совпадают, цель 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.

***4.3.1*** 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.

***4.3.2*** План проведения тестирования:

* планирование – разработка заданий и набор участников тестирования;
* проведение тестирования;
* анализ полученных данных.

***4.3.3*** 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.

***4.3.4*** Протокол заданий. 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, характеристик и квестов |
| 5 | Игровая сцена | Поднять предмет |
| 6 | Игровая сцена | Экипировать предмет |
| 7 | Игровая сцена | Уничтожить противника |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 8 | Игровая сцена | Повысить любую характеристику после получения нового уровня |
| 9 | Игровая сцена | Получить урон от противника, умереть, воскреснуть и зафиксировать изменение характеристик с позицией возрождения |
| 10 | Игровая сцена | Взять квест и проверить его корректное отображение во вкладке квестов |

***4.3.5*** Респонденты.

Было принято решение отбирать для тестирования людей различного возраста, увлечений и игрового опыта

Все респонденты успешно справились с задачами, отметив удобство интерфейса. Гармоничность оформления и интересный, разнообразный геймплей.

4. 4 Итоги тестирования

В результате тестирования было установлено соответствие разработанного приложения и исходных данных, указанных на этапе проектирования.

Список устройств, используемых в тестировании, и их технические характеристики указаны в таблице 4.2.

Таблица 4.2 – Устройства, используемые в тестировании

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Устройство | ОС | *CPU* | *GPU* | Объем оперативной памяти |
| 1 | 2 | *3* | *4* | 5 |
| *Xiaomi Redmi Note* 8 | *Android* 9.0 *Pie* | *Qualcomm* *Snapdragon* 665 | *Adreno* 610 | 4 Гб |
| *Xiaomi* *Mi* 4*s* | *Android* 5.1 *Lollipop* | *Qualcomm* *Snapdragon* 808 *MSM*8992 | *Adreno* 418 | 3 Гб |
| *Samsung* *Galaxy* *A*20*e* | *Android* 9.0 *Pie* | *Samsung* *Exynos* 7 *Octa* 788 | *Samsung* *Exynos* 7 *Octa* 788 | 3 ГБ |
| *Xiaomi* *Redmi* *Note* 7 | *Android* 9.0 *Pie* | *Qualcomm* *Snapdragon* 660 *MSM*8956 | *Adreno* 512 | 4 ГБ |

Продолжение таблицы 4.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| *Xiaomi Redmi* 9 | *Android* 10.0 | *Mediatek Helio* *G*80 | *Mali*-*G*52 *MC*2 | 4 Гб |
| *Xiaomi Redmi* 7 | *Android* 9.0 *Pie* | *Qualcomm* *Snapdragon* 632 | *Adreno* 506 | 3 Гб |
| *Samsung Galaxy* *A*51 | *Android* 10.0 | *Samsung* *Exynos* 9611 | *Mali*-*G*72 *MP*3 | 6 Гб |

Несмотря на то, что список устройств, незначителен – стоит отметить широкое разнообразие конфигураций их технических характеристик. Использование лишь такого малого количества компьютеров позволило найти, идентифицировать и решить некоторые ошибки:

− отсутствие адаптивности интерфейса под различные разрешения экрана;

− отсутствие адаптивности анимаций к различной величине *FPS*;

− проблема отображения текстур на некоторых устройствах.

Что касается скорости работы, то все устройства позволили пользоваться приложением с приемлемым *FPS*. На всех устройствах этот показатель не падал ниже 30 кадров, что достаточно для комфортной игры. Следовательно, разработанное мобильное игровое приложение способно запускаться на большей части современных мобильных устройств на базе ОС *Android*. Минимальные системные требования довольно низкие.

За время проведения тестов, возникали ошибки в основном касающиеся визуальной составляющей.

Из всего вышеупомянутого можно сказать, что качество и стабильность итогового приложения находится на достаточно высоком уровне. Данное тестирование показывает, что на текущем этапе реализации проекта, игра соответствует большинству функциональных требований выделенных на этапе проектирования приложения.