

СОДЕРЖАНИЕ

| | |
|--|----|
| Введение | 4 |
| 1 Общие сведения о предприятии СП ОАО «Спартак» | 5 |
| 1.1 Сведения о предприятии | 5 |
| 1.2 Деятельность предприятия..... | 6 |
| 1.3 Должностные обязанности инженера-программиста | 7 |
| 1.4 Охрана труда и техника безопасности на рабочем месте | 8 |
| 2 Описание области разработки в игровой индустрии..... | 9 |
| 2.1 Характеристика предметной области..... | 9 |
| 2.2 Игровой движок <i>Unity</i> | 10 |
| 2.3 Игровой движок <i>Unreal Engine</i> | 13 |
| 2.4 Сравнение <i>Unity</i> и <i>Unreal Engine</i> | 15 |
| 3 Проектирование приложения « <i>SAVELOADINVENTORY</i> » | 17 |
| 3.1 Описание индивидуального задания..... | 17 |
| 3.2 Жанр симулятора фермы | 17 |
| 3.3 Архитектура приложения « <i>SAVELOADINVENTORY</i> » | 20 |
| 3.4 Программные классы компонентов..... | 22 |
| 3.5 Верификация реализованных программных компонентов | 26 |
| Заключение..... | 27 |
| Список использованных источников..... | 28 |
| Приложение А Код программы..... | 29 |

ВВЕДЕНИЕ

Преддипломная практика является важной частью образовательного процесса, направленная на подготовку квалифицированных специалистов, способных успешно применять теоретические знания в практической деятельности. Этот этап обучения нацелен на совмещение теории с практикой, предоставляя студентам возможность углубления, расширения и систематизации учебного материала.

Место практики: СП ОАО «Спартак».

Главная цель практики – использование и углубление теоретических знаний, полученных в процессе обучения, а также развитие навыков проектирования и конструирования информационных систем.

Основные задачи преддипломной практики:

- развитие и закрепление практических навыков анализа предметной области;
- приобретение опыта проектирования программных систем;
- использование языков и инструментальных средств моделирования при проектировании;
- создание программных систем с использованием современных сред разработки, поддерживающих командную работу и контроль версий;
- разработка документации к системе, включая техническое задание и инструкции пользователя и программиста.

В ходе преддипломной практики изучаются современные технологии и методы улучшения работы организации, осуществляется знакомство со структурой организации и её производственными процессами, а также системами автоматизации и управления. Этот опыт является важным шагом в профессиональной подготовке специалистов и их подготовке к защите дипломной работы.

Индивидуальным заданием, является разработка архитектуры, программных модулей для игры в жанре аркадного симулятора фермера, результатом которого является приложение «*SaveLoadInventory*». Этот проект не только позволит применить теоретические знания на практике, но и приобрести ценный опыт работы в индустрии разработки программного обеспечения.

1 ОБЩИЕ СВЕДЕНИЯ О ПРЕДПРИЯТИИ СП ОАО «СПАРТАК»

1.1 Сведения о предприятии

Кондитерская фабрика «Спартак» является одним из крупнейших в Республике Беларусь производителей кондитерских изделий и полуфабрикатов, выпускающий около 350 наименований кондитерских изделий, включая продукцию лечебно-профилактического действия.

Фабрика была создана 4 июня 1924 года и первоначально называлась «Просвет». Своё нынешнее название фабрика получила 8 ноября 1931 года. На фабрике сегодня существуют 4 основных цеха: бисквитный, карамельный, вафельный, конфетно-шоколадный, где в широком ассортименте производят вышеуказанные виды продукции. Основными видами продукции, выпускаемой фабрикой, являются: карамель, конфеты, шоколад и шоколадные изделия, печенье, вафельные изделия, торты и пирожные.

Новая стратегия развития фабрики предусматривает значительное изменение ассортимента продукции и повышение её качества. Этому способствует наличие сети цеховых и центральных лабораторий, оснащённых самым современным оборудованием, где осуществляется строгий входной контроль сырья, полуфабрикатов и готовой продукции.

Одним из основных секретов успеха кондитерской фабрики является то, что кондитерские изделия производятся исключительно из экологически чистого сырья высокого качества, поставляемого из стран ближнего и дальнего зарубежья, к тому же, на современном высокотехнологичном оборудовании, которым в достаточном объеме оснащено предприятие.

Продукция предприятия сертифицирована по международной системе стандартов:

- *ISO 9001* (система менеджмента качества);
- *ISO 14001* (система менеджмента окружающей среды);
- *HACCP* (анализ рисков и контроль критических точек).

Успех современных компаний определяется качеством – начиная с производства и заканчивая продукцией и предоставляемыми услугами. В современной рыночной экономике основным конкурентным преимуществом любого предприятия становится качество производимой продукции.

«Спартак» – марка самых популярных кондитерских изделий. Фирменная продукция предприятия обладает отменным вкусом благодаря постоянному поиску, разработке и освоению новейших технологий и оборудования, использованию высококачественного и экологически чистого сырья и оригинальных рецептов. Именно поэтому портфель наград фабрики становится полнее с каждым годом[1].

1.2 Деятельность предприятия

В рамках своей практической работы была возможность ближе познакомиться с деятельностью кондитерской фабрики, входящей в состав СП ОАО «Спартак». Компания СП ОАО «Спартак» является одним из ведущих производителей кондитерских изделий в нашей стране и на протяжении многих лет завоевывает доверие потребителей своими качественными и вкусными продуктами.

Кондитерская фабрика СП ОАО «Спартак» представляет собой современное производственное предприятие, оснащенное передовым оборудованием и технологическими решениями. Здесь каждый этап производства, начиная от подготовки ингредиентов и заканчивая упаковкой готовой продукции, тщательно контролируется для обеспечения высокого качества и безопасности всех изделий[1].

Основной ассортимент продукции, выпускаемой на кондитерской фабрике СП ОАО «Спартак», включает широкий спектр сладостей: шоколадные конфеты, печенье, торты, пирожные, вафли и другие десерты. Каждый продукт отличается оригинальным дизайном, безупречным вкусом и использованием только высококачественных ингредиентов.

Одной из особенностей деятельности кондитерской фабрики СП ОАО «Спартак» является постоянное внимание к инновациям и развитию новых вкусовых решений. Команда опытных кондитеров и продуктовых технологов постоянно работает над созданием новых продуктов, учитывая современные тенденции и предпочтения потребителей.

Компания также активно участвует в маркетинговых кампаниях и рекламных акциях, чтобы продвигать свою продукцию на рынке и поддерживать лояльность клиентов. Благодаря стратегическому партнерству с известными брендами и умелому позиционированию, кондитерская фабрика СП ОАО «Спартак» занимает прочные позиции на рынке и имеет широкую аудиторию поклонников своих продуктов.

Важным аспектом деятельности кондитерской фабрики является соблюдение стандартов качества и безопасности пищевых продуктов. Компания тесно сотрудничает с контролирующими органами и систематически проходит проверки, чтобы гарантировать соответствие своей продукции всем необходимым требованиям и стандартам.

В целях устойчивого развития и ответственного ведения бизнеса, кондитерская фабрика СП ОАО «Спартак» также уделяет внимание вопросам экологии и социальной ответственности.

1.3 Должностные обязанности инженера-программиста

В ходе преддипломной практики была возможность ознакомиться с должностными обязанностями инженера-программиста в кондитерской фабрике СП ОАО «Спартак». Инженер-программист играет важную роль в обеспечении эффективной работы и автоматизации процессов на предприятии.

Основные обязанности инженера-программиста распределены по следующим направлениям работы:

- разработка программного обеспечения: инженер-программист отвечает за разработку и поддержку программного обеспечения, используемого в производственных процессах и системах управления кондитерской фабрики, работает над созданием новых программ, модификацией и улучшением существующих систем;

- тестирование и отладка: инженер-программист проводит тестирование разработанных программных решений, выявляет и исправляет возможные ошибки и проблемы, осуществляет отладку программ для обеспечения их стабильной и надежной работы;

- управление базами данных: инженер-программист занимается управлением базами данных, включая их создание, модификацию, анализ и оптимизацию, обеспечивает сохранность и безопасность данных, а также разрабатывает методы резервного копирования и восстановления информации;

- поддержка пользователей: инженер-программист оказывает техническую поддержку пользователям программного обеспечения, отвечает на вопросы и решает проблемы, связанные с работой программ;

- сотрудничество с другими отделами: инженер-программист активно взаимодействует с сотрудниками других отделов, таких как производство, логистика и отдел качества, обсуждает их требования к программному обеспечению, консультирует и предлагает решения для оптимизации бизнес-процессов;

- исследования и развитие: инженер-программист следит за новыми технологиями и тенденциями в области программирования и применяет их на практике, исследует новые инструменты и методы разработки, участвует в проектах по внедрению инновационных решений.

Работа инженера-программиста в кондитерской фабрике СП ОАО "Спартак" требует высокого уровня компетенции в программировании, аналитических навыков и умения работать в команде. Эта должность играет важную роль в автоматизации производственных процессов и обеспечении эффективной работы компании.

1.4 Охрана труда и техника безопасности на рабочем месте

Охрана труда представляет собой комплекс мер, направленных на обеспечение безопасности и здоровья работников в процессе их трудовой деятельности. Включает в себя юридические, социально-экономические, организационные, технические, психофизиологические, санитарно-гигиенические, лечебно-профилактические, реабилитационные и другие меры и средства.

Управление предприятия уделяет особое внимание улучшению условий труда, соблюдению гигиены и совершенствованию организации рабочего процесса. Деятельность в области охраны труда на предприятии строго регулируется действующим законодательством Республики Беларусь, соблюдением санитарных норм, гигиенических стандартов и указаний надзорных органов.

Оптимизация организации рабочих мест является ключевым фактором для обеспечения безопасных условий труда. Профессиональное рабочее место, адаптированное для инженера-программиста, должно быть организовано эффективно с учетом пространства, формы и размеров, обеспечивая удобное положение для работы и высокую производительность при минимальных физических и психических нагрузках.

Соответствие требованиям ГОСТ 12.2.032-78 «Система стандартов безопасности труда». Рабочее место при выполнении работ сидя. Общие эргономические требования является основным принципом при организации рабочего места программиста. Это включает оптимальное размещение оборудования, достаточное рабочее пространство для движений, и уровень акустического шума, не превышающий допустимых значений.

Эргономический дизайн рабочего места программиста способствует уменьшению утомления. Правильная организация размещения предметов, инструментов и документации на рабочем столе обеспечивает удобство использования. Помещения для работы программиста должны быть обеспечены естественным и искусственным освещением, соответствующим требованиям нормативов [3].

Искусственное освещение включает в себя применение люминесцентных ламп, обеспечивающих оптимальное освещение поверхности стола и экрана монитора. Рабочий стол должен быть регулируемым по высоте, позволяя поддерживать правильную рабочую позу. Тип и конструкция рабочего стула также играет важную роль в поддержании здоровья при работе за компьютером.

Соблюдение требований по освещению, размещению оборудования и организации рабочего места важно не только для удобства, но и для повышения производительности.

2 ОПИСАНИЕ ОБЛАСТИ РАЗРАБОТКИ В ИГРОВОЙ ИНДУСТРИИ

2.1 Характеристика 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 участвуют 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 *ММО*) 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 телевидения, 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.

2.2 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.

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 2.1 – 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, с.17].

2.3 Игровой движок *Unreal Engine*

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

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

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

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

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

- широкий функционал;
- визуальное программирование;
- бесплатная лицензия;
- возможность создать кросс-платформер;
- большая база пользователей.

Epic Games решила дать разработчикам больше, чем простой инструмент – в UE пользователи могут начать работу даже без узкоспециализированных знаний в области языков программирования. Для тех, кто далек от 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 C++, при этом их функции и возможности в большинстве случаев схожи. Однако иногда все же придется прибегнуть к кодингу: для произведения сложных математических расчетов, изменения исходного кода самого движка UE и ряда базовых классов проекта.

В игровом мире существуют объекты с уникальными оттенками, фактурами и физическими свойствами. В движке UE внешний вид зависит от настроек материалов. Цвет, прозрачность, блеск – задать можно практически любые параметры. При работе над игрой в UE материалы можно наносить на

любые объекты, вплоть до мелких частиц. Отметим, что речь идет не просто о настройке текстур: материалы открывают более широкие возможности. К примеру, можно создавать необычные визуальные эффекты, причем UE позволяет делать это прямо в процессе игры.

Пользовательский интерфейс. Игроку важно не только видеть действия своего персонажа и карту, на которой он находится, но и иметь текстовую информацию, а также сведения о количестве очков, пунктах здоровья, инвентаре и т. д. С этой целью разработчики тщательно продумывают пользовательский интерфейс (*User Interface, UI*). В движке *Unreal* для создания UI применяется *Unreal Motion Graphics (UMG)*. Он позволяет выстраивать интуитивно понятный UI, выводить на экран необходимую пользователю информацию, а также менять положение кнопок и текстовых меток.

Анимация. Персонаж любой современной игры подвижен и гибок, умеет бегать и прыгать. Все это возможно благодаря анимированию. В UE начинающие разработчики могут импортировать уже готовые 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.

2.4 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* с *Blueprints*.

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

Анимации, включённые для скелетных мешей, также можно импортировать. В *Unity* они прикрепляются к импортированному объекту, как клипы анимации (*animation clips*), а в *UE4* называются последовательностями анимации (*animation sequences*). В первом движке 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 с 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 выбран игровой движок *Unity*, т.к. он обладает наибольшим инструментарием и информационной базой, которые в один момент являются удобными и простыми [6].

3 ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ «*SAVELOADINVENTORY*»

3.1 Описание индивидуального задания

Необходимо разработать архитектуру и программные модули для игрового приложения в жанре аркадного симулятора фермера.

Для игр в жанре аркадного симулятора фермера основным и важным фактором геймплея является накопление собранных ресурсов и их сохранение для последующего продвижения. Эти игровые элементы называются механиками инвентаря и сохранения.

Для демонстрации решения разработано приложение «*SaveLoadInventory*», которое несет демонстрационный характер. В качестве типа визуализации полученного результата выбрана 2D графика. Для написания программных механик используется платформа *Unity*, язык программирования *C#*.

Механика инвентаря должна реализовывать следующие возможности:

- хранение предметов разных типов для различных видов применения;
- использование системы *drag and drop*;
- возможность сортировки;
- обеспечение предметов разными иконками;
- перемещение предметов в другой инвентарь;
- подбор предмета;
- выброс предмета.

Для реализации механики сохранения нужно разработать программные компоненты, которые будут сохранять и загружать данные в формате *JSON*, а также дополнительные компоненты, которые будут корректно инициализировать объекты сохранения.

3.2 Жанр симулятора фермы

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.

Одной из особенностей фермерских игр является то, что они могут предоставлять игрокам обширные возможности для творчества и экспериментов. Каждая ферма может стать уникальной благодаря выбору культур, животных, оборудования и строений. Это позволяет игрокам создавать и развивать ферму в соответствии со своими предпочтениями и стратегией.

Пример фермерской игры данного жанра представлен на рисунке 3.1, демонстрируя разнообразие игрового процесса и возможностей управления фермой.



Рисунок 3.1 – Кадр из игры «*Stardew Valley*»

Stardew Valley – компьютерная игра в жанре симулятора жизни фермера с элементами ролевой игры.

Stardew Valley представляет собой симулятор фермера, напоминающий игры серии *Harvest Moon*. В начале игрок создает игрового персонажа, выбирая имя, пол и внешность. По сюжету, персонаж – офисный работник – получает от умершего деда-фермера наследство: участок земли и дом в долине Стардью. Игрок может выбрать один из семи возможных вариантов фермы в соответствии со своими предпочтениями: например, на карте с рекой, протекающей рядом с фермой, у него появляется больше мест для ловли более разнообразной рыбы. Изначально ферма представляет собой заброшенный участок, поросший

сорняками, деревьями, а также забитый брёвнами и валунами – игроку предстоит расчистить площадку под посадки и сооружения.

Выращивание растений и животных на ферме приносит доход, который можно потратить на её дальнейшее развитие.

Игровой персонаж может взаимодействовать с неигровыми персонажами (*NPC*), населяющими соседний городок, общаться с ними, даже обзавестись мужем или женой и детьми

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 игрокам возможность создать идеальную ферму по своему вкусу.

3.3 Архитектура приложения «*SAVELOADINVENTORY*»

Представление надежной и эффективной архитектуры сетевого приложения является критическим компонентом в разработке современных многопользовательских игр и 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 декомпонировать – значит разбить абстрактную большую задачу на маленькие задачи, которые можно легко оценить.

Результат разбиения задачи на отдельные компоненты и, как следствие, полученная архитектура показан на рисунке 3.2.

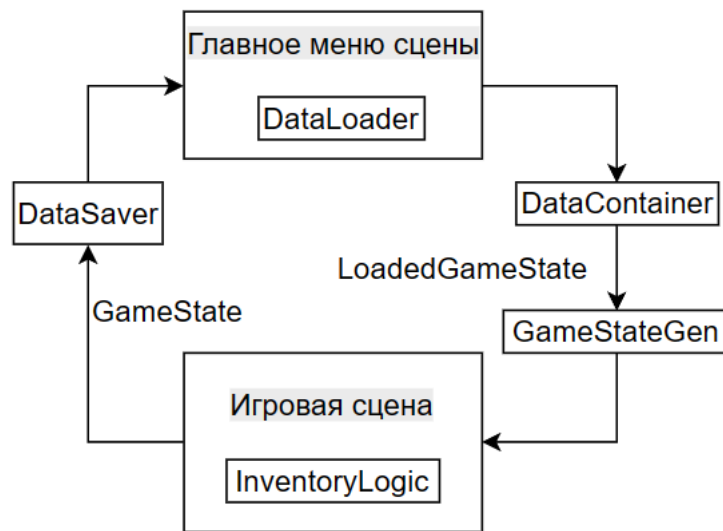


Рисунок 3.2 – Результат разбиения задачи на отдельные компоненты

Игровая логика будет разрабатываться на игровом движке *Unity*. Он будет отвечать за отображение графических элементов, также на нем будут разработаны механики и сценарии игры.

В результате разбиения задачи с помощью декомпозиции были выделены следующие компоненты:

- главное меню сцены;
- игровая сцена;
- *DataLoader*;
- *DataSaver*;
- *GameState*;
- *DataContainer*;
- *GameStateGen*;
- *InventoryLogic*.

Компонент «главное меню сцены» является сценой, где игрок может выбрать следующие действия:

- *New game*;
- *Load game*;
- *Settings*;
- *Work space*.

Выбирая действие «*New game*», игрок создаст новое состояние игры или установит состояние игры по умолчанию, после чего загрузиться в игровую сцену.

Выбирая действие «*Load game*», игрок сможет выбрать одно из своих сохраненных состояний игры или продолжить сохраненный сеанс.

При выборе «*Settings*» пользователь сможет переназначить клавиши управления, а при выборе «*Work space*» закроет приложение.

Компонент «игровая сцена» является сценой, где реализуется геймплей игры.

Компонент *DataLoader* отвечает за загрузку игрового состояния и будет использоваться в сцене главного меню.

Компонент *DataSaver* отвечает за сохранение игрового состояния и будет использоваться после выхода из игровой сцены.

Компонент *GameState* является структурой данных, описывающий игровое состояние.

Компонент *DataContainer* отвечает за перенос данных между сценой главного меню и игрового меню.

Компонент *GameStateGen* будет реализовывать сохраненное состояние игры, получая данные из контейнера.

Компонент *InventoryLogic* отображает логику инвентаря в игровой сцене.

Таким образом, с помощью декомпозиции удалось разбить задачу на подзадачи, с помощью чего была разработана архитектура приложения и выделены основные программные компоненты, требующие реализации.

3.4 Программные классы компонентов

В этом подразделе описана скриптовая реализация игровой механики инвентаря с использованием инструментария *Unity* на языке *Unity Sharp*. В результате разработки игровой механики, позволяющей игроку иметь инвентарь, были реализованы скрипты, отвечающие за взаимодействие игрока с предметами, а также взаимодействие предметов с игровым интерфейсом, являющимся основой механики.

Класс *InputService* – сервис обработки ввода пользовательских действий. Данный класс содержит следующие методы:

- *GetMovement()*: возвращает вектор движения, полученный из ввода;
- *IsChosenCell()*: определяет, была ли выбрана ячейка инвентаря в текущем кадре, и возвращает индекс выбранной ячейки;
- *IsLBK()*: проверяет, была ли нажата левая кнопка мыши в текущем кадре;
- *IsRBK()*: проверяет, была ли нажата правая кнопка мыши в текущем кадре;
- *IsOpenCloseMenu()*: проверяет, было ли открыто/закрыто игровое меню в текущем кадре;
- *Dispose()*: освобождает ресурсы, отключает обработчики ввода;

Класс *GameMenu* – управление игровым меню. Данный класс содержит следующие методы:

- *OnContinue()*: скрывает меню, если оно отображается;
- *OnExit()*: вызывает событие выхода из игры, сохраняет текущее состояние игры и загружает главное меню.

Класс также имеет следующие поля:

- *GameObject menuPanel*: панель игрового меню;
- *InputService inputService*: сервис обработки ввода.

Класс *ItemSourceSO* – объект-контейнер параметров предметов. Данный класс содержит следующие поля:

Класс *GameInstaller* – установщик зависимостей игры. Данный класс содержит метод *InstallBindings()*, который выполняет следующие привязки:

- *InputService*: привязка как единственного экземпляра сервиса ввода;
- *GameMenu*: привязка к компоненту *GameMenu* в иерархии сцены как единственного экземпляра.

Класс *Chest* – представляет сундук как объект размещения. Данный класс реализует интерфейсы *IOccupyingOneCell*, *ISaveLoadPlacementItem*, *IInteractable*. Данный класс содержит следующие методы и поля:

- метод *Start()* – инициализирует аниматор, хэши для анимаций, список *inventoryItems*, панель инвентаря *chestStorage*, устанавливает флаг *openCloseFlag* в *true*;
- метод *OnMouseDown()* – обработчик нажатия мыши: открывает или закрывает сундук, анимирует это действие и активирует или деактивирует панель инвентаря;
- метод *Initialize()* – инициализирует *inventoryItems* переданным списком инвентарных предметов;
- метод *GetOccupyingCell()* – возвращает позицию размещения сундука;
- метод *GetData()* – возвращает данные о сундуке в виде объекта *ChestData*, который содержит позицию и список предметов в инвентаре.

Класс *InventoryBase* – абстрактный базовый класс для инвентарей. Реализует интерфейс *IDragHandler*. Методы данного класса:

- *Initialize()*: инициализирует инвентарь переданным списком предметов;
- *RegisterDragEvents()*: регистрирует события перетаскивания для ячейки инвентаря;
- *AddItem()*: добавляет предмет в инвентарь и создает для него ячейку;
- *IsFull()*: возвращает *true*, если инвентарь заполнен;
- *OnBeginDragCell()*: виртуальный метод вызывается при начале перетаскивания ячейки;
- *OnEndDrag()*: виртуальный метод вызывается при окончании перетаскивания;
- *OnDrag()*: виртуальный метод для обработки события перетаскивания;

- *OverwriteInventoryItemsSequence()*: возвращает список предметов в инвентаре после перезаписи последовательности ячеек;
- *GetItems()*: возвращает список предметов в инвентаре;
- *SaveInventory()*: виртуальный метод для сохранения инвентаря (реализация в дочерних классах).

Класс *ItemResource* представляет ресурсы, которые игрок может подобрать в игре. Методы данного класса:

- *Construct()*: конструктор класса, инициализирует необходимые поля;
- *Start()*: метод запускается при старте объекта, устанавливает гравитацию и иконку предмета, инициализирует стейт-машины;
- *Update()*: метод вызывается каждым кадром для выполнения действий стейт-машины;
- *FixedUpdate()*: метод вызывается каждым фиксированным кадром для выполнения действий стейт-машины;
- *Initialize()*: метод инициализирует предмет в инвентаре, устанавливает его иконку;
- *InitializeStateMachine()*: метод инициализирует стейт-машины предмета;
- *PickupResource()*: метод пытается подобрать ресурс игроком, возвращает *true*, если удалось;
- *GetDistanceToPlayer()*: метод возвращает расстояние до игрока;
- *GetPlayerDirectionVector()*: метод возвращает вектор направления до игрока.

Класс *InventoryItem* представляет элемент инвентаря, который может быть отображен в *UI*. Методы данного класса:

- *RenderUI()*: метод для отображения элемента в *UI* инвентаря. Устанавливает иконку и текст предмета в ячейке инвентаря;
- *GetItemData()*: метод для получения данных элемента инвентаря. Создает и возвращает объект *InventoryItemData*, содержащий имя предмета (*SoName*);

Класс *InventoryItem* является основой для создания конкретных типов предметов, и может быть наследован для добавления дополнительной функциональности или свойств.

Класс *InventoryCell* представляет ячейку инвентаря, которая отображает один элемент инвентаря в интерфейсе пользователя. Методы данного класса:

- *Construct()*: метод для внедрения зависимости от *ItemResourceDroper*;
- *Initialize()*: метод для инициализации ячейки инвентаря, устанавливает глобальный контекст визуализации, оригинальный контекст визуализации и предмет инвентаря;
- *RegisterEvents()*: метод для регистрации событий начала и окончания перетаскивания ячейки;
- *OnDrag()*: метод вызывается при перетаскивании ячейки, устанавливает позицию ячейки по позиции мыши;

– *OnBeginDrag()*: метод вызывается при начале перетаскивания ячейки, устанавливает начальный индекс ячейки, вызывает событие начала перетаскивания и устанавливает родительский контейнер в глобальный контекст визуализации;

– *OnEndDrag()*: метод вызывается при окончании перетаскивания ячейки, определяет, пересекается ли ячейка с другим инвентарем, и осуществляет соответствующие действия: если пересечение есть, то либо возвращается на место, либо переносится в другой инвентарь; если нет пересечения, то уничтожается.

Класс *OnEndDrag()* отвечает за отображение элемента инвентаря в интерфейсе и управление его перетаскиванием и событиями связанными с этим действием.

Класс *Movement* отвечает за управление движением игрока. Методы данного класса:

– *Start()*: метод вызывается при старте объекта, инициализирует компоненты и устанавливает позицию игрока, если она была загружена;

– *Animate()*: метод для управления анимацией движения, устанавливает параметры анимации для горизонтального и вертикального движения;

– *OnExitTheGame()*: метод вызывается при выходе из игры, сохраняет позицию игрока в состояние игровых данных;

– *SetLoadedPosition()*: метод для установки загруженной позиции игрока.

Класс *Movement* отвечает за обработку ввода от игрока, управление анимацией и физическим движением игрока, а также за сохранение его позиции при выходе из игры.

Класс *GameDataState* представляет состояние игровых данных, включая информацию о игроке, объектах размещения и инвентаре. Методы данного класса:

– *UpdateActivePackInventory()*: обновляет активный инвентарь данными из списка объектов;

– *AddItemData()*: добавляет данные объекта размещения в список;

– *RemoveItemData()*: удаляет данные объекта размещения из списка.

Класс *GameDataState* предназначен для хранения и управления данными игрового состояния, включая информацию об игроке, объектах размещения и инвентаре. Он позволяет обновлять и изменять эти данные в соответствии с игровым процессом.

Класс *GameDataSaveLoader* отвечает за сохранение и загрузку игровых данных в хранилище, такое как *PlayerPrefs*. Он также содержит внутренний класс *ItemPlacementDataConverter*, который помогает конвертировать данные размещения в *JSON* и обратно. Методы данного класса:

– *SaveGameState()*: сохраняет состояние игровых данных в формате *JSON* в *PlayerPrefs*;

3.5 Верификация реализованных программных компонентов

При запуске приложения игрок увидит главное игровое меню, где ему можно закрыть приложение путем нажатия кнопки «*Work space*». Создать новый игровой сеанс игрок может выбором кнопки «*New game*», а загрузить игровой сеанс – выбором кнопки «*Load game*».

На рисунке 3.3 представлено игровое меню.

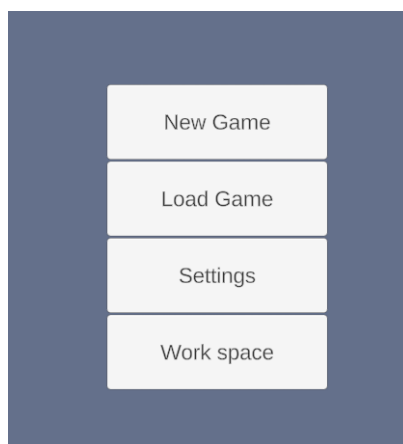


Рисунок 3.3 – Меню

После загрузки игрового сеанса игроку виден слева-вверху его активный инвентарь, предметы которого можно использовать. Немного ниже активного инвентаря находится инвентарь-рюкзак. Его емкость 16 ячеек и его, в отличие от активного инвентаря, можно скрыть. На рисунке 3.4 представлен вид из игровой сцены.



Рисунок 3.4 – Игровая сцена

Инвентарь сундука можно перемещать по экрану и перекладывать туда предметы. Кроме того, предмет можно выкинуть и подобрать. Чтобы подобрать предмет, нужно подойти к нему на близкую дистанцию.

ЗАКЛЮЧЕНИЕ

Прохождение преддипломной практики является важным элементом учебного процесса по подготовке специалиста в области программирования. Во время её прохождения будущий программист применяет полученные в процессе обучения знания, умения и навыки на практике.

Практика способствует учиться самостоятельно решать определенные задачи, возникающие в ходе работы программиста. Это важный навык, который пригодится в профессиональной жизни. Основными задачами преддипломной практики являются:

- получение практического опыта работы в качестве программиста;
- улучшение качества профессиональной подготовки;
- закрепление полученных знаний по общим и специальным дисциплинам.

В ходе разработки приложения решены следующие задачи:

- произведен аналитический обзор доступных программных средств разработки, позволяющих разработать игровое приложение;
- разработан пользовательский графический интерфейс, реализующий взаимодействие пользователя с элементами инвентаря;
- разработана сцена, содержащая игровые объекты;
- разработана общая структура и механики, позволяющие осуществлять процесс взаимодействия с разработанными программными компонентами;
- использован принцип декомпозиции для ускорения и оптимизации разработки;
- произведена верификация тестирование конечного программного продукта;
- разработан программный код приложения.

Приложение представляет из себя уровень, где пользователь может продемонстрировать работу механики инвентаря и сохранения игры. Для этого предоставляется возможность взаимодействия с элементами инвентаря и игровым меню для сохранения и загрузки игровых данных.

Отличительной особенностью в рамках разработанных программных компонентов для игр жанра фермерского симулятора является наличие механики взаимодействия с инвентарем других игровых объектов, таких как сундук. А также настраиваемое положение панелей инвентаря.

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

Кроме того, приложение является масштабируемым, что является результатом разработанной архитектуры. Таким образом, можно впоследствии внедрять новую функциональность.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Официальный сайт СП ОАО «Спартак» [Электронный ресурс]. Режим доступа: <https://it-light.by/news?start=16> – Дата доступа: 20.03.2024.
2. Юридический словарь [Электронный ресурс]. Режим доступа: <http://multilang.pravo.by/ru/Term/Index/525?langName=ru&ch=Bce&size=25&page=4&type=3> – Дата доступа: 16.03.2022.
3. Система стандартов безопасности труда. Рабочее место при выполнении работ сидя. Общие эргономические требования: ГОСТ 12.2.032-78. – Введ. 01.01.1979 – Москва: ИПК Издательство стандартов, 2001. – 27с.
4. Хокинг, Дж. Unity в действии. Мультиплатформенная разработка на C#/Дж.Хокинг. – Питер, 2019. – 352 с.
5. Основы разработки 2D-игр [Электронный ресурс]. Режим доступа: <https://proglib.io/p/sozdaem-2d-igru-na-unity-instrukciya-dlya-novichka-2020-09-01> – Дата доступа: 04.07.2022.
6. Руководство: 2D или 3D проекты [Электронный ресурс]. Режим доступа: <https://docs.unity3d.com/ru/530/Manual/2Dor3D.html> – Дата доступа: 04.07.2022.
7. Unity, платформа разработки в реальном времени [Электронный ресурс]. Режим доступа: <https://unity.com/ru> – Дата доступа: 04.07.2022.

ПРИЛОЖЕНИЕ А

(обязательное)

Код программы

Листинг GameEvents.cs:

```
public class GameEvents
{
    public static Action OnExitTheGameEvent;

    public static void InvokeExitTheGameEvent()
    {
        OnExitTheGameEvent?.Invoke();
    }
}
```

Листинг GameMenu.cs:

```
public class GameMenu : MonoBehaviour
{
    [SerializeField] GameObject _menuPanel;
    InputService _inputService;
    GameState _gameDataState;
    GameDataSaveLoader _gameDataSaveLoader;

    [Inject]
    public void Construct(InputService inputService,
        GameState gameDataState,
        GameDataSaveLoader gameDataSaveLoader)
    {
        _inputService = inputService;
        _gameDataState = gameDataState;
        _gameDataSaveLoader = gameDataSaveLoader;
    }

    private void Start()
    {
        if (_menuPanel.activeSelf == true)
        {
            _menuPanel.SetActive(false);
        }
    }

    public void OnContinue()
    {
        if (_menuPanel.activeSelf)
        {
            _menuPanel.SetActive(false);
        }
    }
}
```

```

private void Update()
{
    if (_inputService.IsOpenCloseMenu())
    {
        if (_menuPanel.activeSelf == false)
        {
            _menuPanel.SetActive(true);
            return;
        }
        if (_menuPanel.activeSelf == true)
        {
            _menuPanel.SetActive(false);
            return;
        }
    }
}

public void OnExit()
{
    GameEvents.InvokeExitTheGameEvent();

    _gameDataSaveLoader.SaveGameState(_gameDataState);
    SceneManager.LoadScene(GameConfiguration.MainMenuSceneName);
}
}

```

```

[CreateAssetMenu(fileName = "InventoryInfo", menuName = "SO/InventoryInfo")]
public class InventoryInfo : ScriptableObject
{
    public int TotalSize;
}

```

Листинг ItemSourceSO.cs:

```

[CreateAssetMenu(fileName = "ItemSource", menuName = "SO/ItemSource")]
public class ItemSourceSO : ScriptableObject
{
    [Header("Скорость следования за игроком")]
    public float FollowSpeed;
    [Header("Дистанция, на которой предмет начинает следовать за игроком")]
    public float FollowDistance;
    [Header("Скорость, с которой вылетает ресурс из инвентаря")]
    public float PushSpeed;
    public float GravityScale;
    [Header("Дистанция, на которую перемещается ресурс, который выпал с игрока")]
    public float PushDistance;
    [Header("Дистанция, от игрока к ресурсу, при которой ресурс подбирается")]
    public float PickupDistance;
    [Header("Дистанция между конечной точкой броска и текущей, чтобы перейти в состояние <На земле>")]
    public float DistanceToChangeGroundState;
}

```

Листинг InputService.cs:

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

    public InputService()
    {
        _inputActions = new InputActions();
        _inputActions.Enable();
    }

    public Vector2 GetMovement()
    {
        return _inputActions.PlayerMap.Movement.ReadValue<Vector2>();
    }

    public bool IsChosenCell(out int index)
    {
        index = 0;
        bool wasPerformed = false;
        if(_inputActions.InventoryMap.ChooseFirstCell.WasPerformedThisFrame())
        {
            index = c_firstItemCellIndex;
            wasPerformed = true;
        }
        if (_inputActions.InventoryMap.ChooseSecondCell.WasPerformedThisFrame())
        {
            index = c_secondItemCellIndex;
            wasPerformed = true;
        }
        if (_inputActions.InventoryMap.ChooseThirdCell.WasPerformedThisFrame())
        {
            index = c_thirdItemCellIndex;
            wasPerformed = true;
        }
        if (_inputActions.InventoryMap.ChooseFourthCell.WasPerformedThisFrame())
        {
            index = c_fourthItemCellIndex;
            wasPerformed = true;
        }
        return wasPerformed;
    }

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

    public bool IsRBK()
```

```

    {
        return _inputActions.PlayerMap.RBK.WasPerformedThisFrame();
    }
    public bool IsOpenCloseMenu()
    {
        return _inputActions.PlayerMap.OpenCloseGameMenu
            .WasPerformedThisFrame();
    }
    public void Dispose()
    {
        _inputActions.Disable();
        _inputActions.Dispose();
    }
}

```

Листинг GameInstaller.cs:

```

public class GameInstaller : MonoInstaller
{
    public override void InstallBindings()
    {
        Container.Bind<InputService>().AsSingle();
        Container.Bind<GameMenu>()
            .FromComponentInHierarchy()
            .AsSingle();
    }
}

```

Листинг GameStateInstaller.cs:

```

using System;
using System.Collections.Generic;
using Zenject;
using UnityEngine;
using Scripts.PlacementCode;
using Scripts.SaveLoader;
using Scripts.MainMenuCode;

namespace Scripts.Installers
{
    public class GameStateInstaller : MonoInstaller
    {
        [Header("Дефолтные объекты на уровне")]
        [SerializeField] private List<PlacementItem> _placementItems;
        GameDataState _gameDataState;

        public override void InstallBindings()
        {
            if (LoadedData.IsDefault)
            {

```

```

        if (LoadedData.Instance() == null)
        {
            _gameDataState = new GameDataState("Editor mode");
        }
        else
        {
            _gameDataState = LoadedData.Instance().GameDataState;
        }
    }
    else
    {
        _gameDataState = LoadedData.Instance().GameDataState;
        foreach (var placementItem in _placementItems)
        {
            Destroy(placementItem.gameObject);
        }
    }
    Container.BindInstance(_gameDataState).AsSingle();
    Container.Bind<GameDataSaveLoader>().AsSingle();
}
}
}

```

Листинг InventoryInstaller.cs:

```

using Scripts.InventoryCode;
using Scripts.InventoryCode.ItemResources;
using Scripts.SO.Inventory;
using System;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;
using Zenject;

namespace Scripts.Installers
{
    public class InventoryInstaller : MonoInstaller
    {
        [Header("Canvas")]
        [SerializeField] Transform DragParent;
        [Header("Cell prefab")]
        [SerializeField] private InventoryCell CellTemplate;
        [Header("Empty cell prefab")]
        [SerializeField] private GameObject EmptyCellTemplate;
        [Space]
        [Header("Active inventory")]
        [SerializeField] private InventoryInfo _activeInventoryInfo;
        [Space]
        [Space]
        [Header("Backpack inventory")]
        [SerializeField] private InventoryInfo _storageInventoryInfo;
        [Space]
    }
}

```

```

[Header("ItemResourceSO")]
[SerializeField] private ItemSourceSO ItemSourceSO;
[Header("Item resource prefab")]
[SerializeField] private ItemResource ItemResourcePrefab;
public override void InstallBindings()
{
    BindInventories();
    BindCellTemplate();
    BindGlobalVisualContext();
    BindFactories();
    BindItemResourceLogic();
    BindCells();
}
void BindGlobalVisualContext()
{
    Container.BindInstance(DragParent)
        .WithId("DragParent")
        .AsTransient();
}
void BindCellTemplate()
{
    Container.BindInstance(CellTemplate)
        .WithId("CellTemplate")
        .AsTransient();
    Container.BindInstance(EmptyCellTemplate)
        .WithId("EmptyCellTemplate")
        .AsTransient();
}
void BindInventories()
{
    Container.Bind<InventoryStorage>().FromComponentInHierarchy().AsTransient();
    Container.BindInstance(_storageInventoryInfo)
        .WithId("InventoryStorageInfo")
        .AsTransient();
    Container.Bind<ActiveInventory>().AsSingle();
    Container.BindInstance(_activeInventoryInfo)
        .WithId("ActiveInventoryInfo")
        .AsTransient();

    Container.Bind<PlayerInventory>()
        .FromComponentInHierarchy()
        .AsSingle();
}
void BindFactories()
{
    Container.Bind<IInventoryCellFactory>()
        .To<StorageCellFactory>()
        .WithArguments(CellTemplate, EmptyCellTemplate)
        .WhenInjectedInto<InventoryStorage>();
}
void BindItemResourceLogic()
{

```



```

        Container.Bind<ItemResourceDroper>().AsSingle();
        Container.BindInstance(ItemSourceSO).AsTransient();
        Container.BindInstance(ItemResourcePrefab).AsTransient();
        Container.Bind<ItemResource>().AsTransient();
        Container.Bind<IItemResourceFactory>()
            .To<ItemResourceFactory>()
            .WithArguments(Container, ItemResourcePrefab)
            .WhenInjectedInto<ItemResourceDroper>();
    }
    void BindCells()
    {
        Container.Bind<InventoryCell>()
            .FromComponentInHierarchy()
            .AsTransient();
    }
}
}

```

Листинг ItemsDataInstaller.cs:

```

using System;
using System.Collections.Generic;
using Zenject;
using UnityEngine;
using Scripts.InventoryCode;
using Scripts.PlacementCode;
using UnityEngine.Tilemaps;
using Scripts.InteractableObjects;
using Scripts.SaveLoader;
using Assets.Scripts.Inventory.Items;

namespace Scripts.Installers
{
    public class ItemsDataInstaller : MonoInstaller
    {
        [Header("Все предметы инвентаря в игре")]
        [SerializeField] List<InventoryItem> _inventoryItemAssetList;

        [SerializeField] private ChestInventory InventoryStorageTemplate;

        [SerializeField] private Tilemap _gameElementsMap;
        [SerializeField] private Chest _chestTemplate;
        public override void InstallBindings()
        {
            BindInventoryItemsDictionary();
            //BindItemsData();
            BindPlacement();
            BindObjectsLogic();
        }
        void BindInventoryItemsDictionary()
        {
            Dictionary<string, IInventoryItem> InventoryItemsDictionary

```

```

        = new Dictionary<string, IInventoryItem>();

        foreach (var item in _inventoryItemAssetList)
        {
            InventoryItemsDictionary[item.Name] = item;
        }

        Container.BindInstance(InventoryItemsDictionary).AsSingle();
    }
    void BindObjectsLogic()
    {
        Container.Bind<ChestInventory>().FromInstance(InventoryStorageTemplate)
            .AsTransient();
        Container.Bind<IInventoryPanelFactory>()
            .To<InventoryChestPanelFactory>()
            .WhenInjectedInto<Chest>();

        IChestFactory chestFactory = new ChestFactory(Container, _chestTemplate);
        Container.BindInstance(chestFactory).AsTransient();

        Container.Bind<Chest>().FromComponentInHierarchy()
            .AsTransient();
    }
    void BindItemsData()
    {
        Container.Bind<IInventoryItem>()
            .To<InventoryItem>();
        Container.Bind<IQuantitativeInventoryItem>()
            .To<QuantitativeInventoryItem>();
        Container.Bind<IProductionInventoryItem<RuleTile>>()
            .To<IHoeInventoryItem>();
        Container.Bind<IHoeInventoryItem>()
            .To<HoeInventoryItem>();
        Container.Bind<IBagInventoryItem>()
            .To<BagInventoryItem>();
    }
    void BindPlacement()
    {
        ItemPlacementMap itemPlacementMap =
            new ItemPlacementMap(_gameElementsMap);
        Container.BindInstance(itemPlacementMap).AsTransient();
        Container.Bind<PlacementItem>().To<Chest>().AsTransient();
    }
}
}
}

```

Листинг MainMenuInstaller.cs:

```

using System;
using System.Collections.Generic;
using Zenject;

```

```

using UnityEngine;
using Scripts.MainMenuCode;
using Scripts.MainMenuScripts;

namespace Scripts.Installers
{
    class MainMenuInstaller : MonoInstaller
    {
        [Header("Главное меню")]
        [SerializeField] private GameObject _menuObject;
        [Header("Меню загрузки")]
        [SerializeField] private GameStatePanel _statePanelTemplate;
        [SerializeField] private GameObject _loadMenuPanel;
        [SerializeField] private Transform _content;
        [Header("Меню создания")]
        [SerializeField] private NewGamePanel _newGamePanel;
        public override void InstallBindings()
        {
            BindMainMenu();
            BindLoadMenu();
            BindNewGameMenu();
        }
        void BindMainMenu()
        {
            Container.BindInstance(_menuObject)
                .WithId("MainMenuObject")
                .AsTransient();
            Container.Bind<MainMenu>()
                .FromComponentInHierarchy()
                .AsSingle();
        }
        void BindLoadMenu()
        {
            Container.BindInstance(_statePanelTemplate)
                .WithId("StatePanelTemplate").AsTransient();
            Container.BindInstance(_loadMenuPanel).WithId("LoadMenuPanel")
                .AsTransient();
            Container.BindInstance(_content).WithId("LoadMenuContent")
                .AsTransient();
            Container.Bind<IGameStatePanelFactory>()
                .To<StatePanelFactory>()
                .WhenInjectedInto<LoadMenu>();
            Container.Bind<LoadMenu>().AsSingle();
        }
        void BindNewGameMenu()
        {
            Container.BindInstance(_newGamePanel)
                .WithId("NewGamePanel")
                .AsTransient();
            Container.Bind<NewGameMenu>().AsSingle();
        }
        void BindSatePanel()
    }
}

```

```

        {
            Container.Bind<GameStatePanel>().AsTransient();
        }
    }
}

```

Листинг PlayerInstaller.cs:

```

using Scripts.SO;
using Scripts.SO.Player;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Zenject;

namespace Scripts.Installers
{
    public class PlayerInstaller : MonoInstaller
    {
        [SerializeField] PlayerSO PlayerSO;
        [SerializeField] Transform _playerTransform;
        public override void InstallBindings()
        {
            Container.BindInstance(_playerTransform)
                .WithId("PlayerTransform")
                .AsTransient();
            Container.Bind<Movement>().FromComponentInHierarchy()
                .AsSingle();
            Container.BindInstance(PlayerSO).AsTransient();
        }
    }
}

```

Листинг ChestFactory.cs:

```

using Scripts.InventoryCode;
using System;
using System.Collections.Generic;
using UnityEngine;
using Zenject;
namespace Scripts.InteractableObjects
{
    public class ChestFactory : IChestFactory
    {
        Chest _chestTemplate;
        DiContainer _container;
        public ChestFactory()
        {
        }

        public ChestFactory(DiContainer diContainer,

```

```

        Chest chestTemplate)
    {
        _chestTemplate = chestTemplate;
        _container = diContainer;
    }
    public Chest Create(List<IInventoryItem> inventoryItems)
    {
        var chest
            = _container.InstantiatePrefabForComponent<Chest>(_chestTemplate);
        chest.Initialize(inventoryItems);
        return chest;
    }
}

```

Листинг ChestFromResourceFactory.cs:

```

using Scripts.InventoryCode;
using System;
using System.Collections.Generic;
using UnityEngine;
using Zenject;

namespace Scripts.InteractableObjects
{
    public class ChestFromResourceFactory : IChestFactory
    {
        const string c_resourcePath = @"/Prefabs/Chest/";
        const string c_prefabName = @"ChestBottom.prefab";
        DiContainer _container;
        public ChestFromResourceFactory(DiContainer diContainer)
        {
            _container = diContainer;
        }

        public Chest Create(List<IInventoryItem> inventoryItems)
        {
            var chest =
                _container.InstantiatePrefabForComponent<Chest>
                (Resources.Load(Application.dataPath +
                    c_resourcePath +
                    c_prefabName));
            return chest;
        }
    }
}

```

Листинг PlacementItem.cs:

```



using Assets.Scripts.Placement;
using Scripts.InventoryCode;

```

```

using Scripts.PlacementCode;
using Scripts.SaveLoader;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Zenject;

namespace Scripts.InteractableObjects
{
    [RequireComponent(typeof(Animator))]

    public class Chest : PlacementItem, IOccupyingOneCell, ISaveLoadPlacementItem, IInteractable
    {
        Animator _animator;
        int _aniIsCloseCode;
        int _aniIsOpenCode;
        /// <summary>
        /// True , false 
        /// </summary>
        bool _openCloseFlag;
        List<IInventoryItem> _inventoryItems;
        IInventoryPanelFactory _inventoryStoragePanelFactory;
        InventoryBase _chestStorage;

        [Inject]
        public void ConstructChest(
            IInventoryPanelFactory inventoryStoragePanelFactory)
        {
            _inventoryStoragePanelFactory = inventoryStoragePanelFactory;
        }

        protected override void Start()
        {
            _animator = GetComponent<Animator>();
            _aniIsCloseCode = Animator.StringToHash("IsClose");
            _aniIsOpenCode = Animator.StringToHash("IsOpen");
            if(_inventoryItems == null)
            {
                _inventoryItems = new List<IInventoryItem>();
            }
            _chestStorage = _inventoryStoragePanelFactory.Create(_inventoryItems);
            _chestStorage.gameObject.SetActive(false);
            _openCloseFlag = true;

            base.Start();
        }

        void Update()
        {
        }
    }
}

```

```

private void OnMouseDown()
{
    if (_openCloseFlag == true)
    {
        _animator.SetBool(_aniIsOpenCode, _openCloseFlag);
        _animator.SetBool(_aniIsCloseCode, false);
        _chestStorage.gameObject.SetActive(true);
    }
    else
    {
        _animator.SetBool(_aniIsCloseCode, true);
        _animator.SetBool(_aniIsOpenCode, _openCloseFlag);
        _inventoryItems = _chestStorage.GetItems();
        _chestStorage.gameObject.SetActive(false);
    }
    _openCloseFlag = !_openCloseFlag;
}
public void Initialize(List<IInventoryItem> inventoryItems)
{
    _inventoryItems = inventoryItems;
}
public Vector2Int GetOccupyingCell()
{
    return PlacePosition;
}

public override PlacementItemData GetData()
{
    ChestData chestData = new ChestData();
    var data = base.GetData();
    chestData.SetPosition(data.GetPosition());
    chestData.UpdateItems(_chestStorage.GetItems());
    return chestData;
}
}
}

```

Листинг InventoryBase.cs:

```

using Scripts.SaveLoader;
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.EventSystems;
using Zenject;

namespace Scripts.InventoryCode
{
    public abstract class InventoryBase : MonoBehaviour, IDragHandler
    {
        [SerializeField] protected Transform ContainerField;
        public Transform Container

```

```

{
    get { return ContainerField; }
    private set { ContainerField = value; }
}
protected int TotalSize;
protected int CurrentSize => Container.childCount;
protected List<IInventoryItem> InventoryItems;
protected IInventoryCellFactory _inventoryCellFactory;
private Transform _globalVisualContext;
private RectTransform _contextRect;

protected Action<InventoryCell> OnBeginDragEvent;
protected Action OnEndDragEvent;

private GameObject _tmpEmptyCell;
protected GameState _gameDataState;

[Inject]
public void Construct([Inject(Id = "DragParent")] Transform dragParent,
    GameState gameDataState)
{
    _globalVisualContext = dragParent;
    _gameDataState = gameDataState;
}

public void Initialize(List<IInventoryItem> inventoryItems)
{
    InventoryItems = new List<IInventoryItem>(TotalSize);
    InventoryItems.InsertRange(0, inventoryItems);
    foreach (var item in inventoryItems)
    {
        AddItem(item);
    }
}

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

public virtual void AddItem(IInventoryItem inventoryItem)
{
    InventoryItems.Add(inventoryItem);
    InventoryCell newCell = _inventoryCellFactory.Create(Container);
    newCell.Initialize(_globalVisualContext, inventoryItem);
    RegisterDragEvents(newCell);
}

/// <summary>
/// Возвращает true, если инвентарь заполнен
/// </summary>
/// <returns></returns>
public bool IsFull()

```



```

{
    if (CurrentSize >= TotalSize)
    {
        return true;
    }
    else
    {
        return false;
    }
}
private void Awake()
{
    _contextRect = gameObject.GetComponent<RectTransform>();
}
protected virtual void Start()
{
}
protected virtual void OnEnable()
{
    DragExtension.RegisterInventoryRectTransform(
        _contextRect);
    OnBeginDragEvent += OnBeginDragCell;
    OnEndDragEvent += OnEndDrag;
}
protected virtual void OnDisable()
{
    DragExtension.UnregisterInventoryRectTransform(_contextRect);
    OnBeginDragEvent -= OnBeginDragCell;
    OnEndDragEvent -= OnEndDrag;
}
protected virtual void OnBeginDragCell(InventoryCell inventoryCell)
{
    _tmpEmptyCell = _inventoryCellFactory.CreateEmpty(inventoryCell);
}
protected virtual void OnEndDrag()
{
    if (_tmpEmptyCell != null)
        Destroy(_tmpEmptyCell);
    InventoryItems = OverwriteInventoryItemsSequence();
}

public virtual void OnDrag(PointerEventData eventData)
{
}

List<IInventoryItem> OverwriteInventoryItemsSequence()
{
    List<IInventoryItem> items = new List<IInventoryItem>();
    for (int i = 0; i < Container.childCount; i++)
    {

```

```

        var cellObject = Container.GetChild(i);
        InventoryCell inventoryCell;
        if (cellObject.TryGetComponent(out inventoryCell))
        {
            items.Add(inventoryCell.InventoryItem);
        }
    }
    return items;
}
public List<IInventoryItem> GetItems()
{
    return InventoryItems;
}

protected virtual void SaveInventory()
{
}
}
}

```

Листинг ItemResource.cs:

```

using System;
using System.Collections.Generic;
using UnityEngine;
using Zenject;

namespace Scripts.InventoryCode.ItemResources
{
    [RequireComponent(typeof(Rigidbody2D))]
    public class ItemResource : MonoBehaviour
    {
        [SerializeField] SpriteRenderer ItemVisualRenderer;
        ItemSourceSO _itemSourceSO;
        ItemResourceStateMachine _stateMachine;
        Transform _playerTransform;
        Sprite _icon;
        public PlayerInventory PlayerInventory { get; private set; }
        public IInventoryItem InventoryItem { get; private set; }
        public Rigidbody2D Rigidbody { get; private set; }
        public SpriteRenderer SpriteRenderer
        {
            get { return ItemVisualRenderer; }
        }

        public PushState PushState { get; private set; }
        public FollowState FollowState { get; private set; }
        public OnGroundState OnGroundState { get; private set; }

        [Inject]
        public void Construct(ItemSourceSO itemSourceSO,

```

```

[Inject(Id = "PlayerTransform")] Transform playerTransform,
PlayerInventory playerInventory)
{
    PlayerInventory = playerInventory;
    _itemSourceSO = itemSourceSO;
    _playerTransform = playerTransform;
}

private void Start()
{
    Rigidbody = GetComponent<Rigidbody2D>();
    Rigidbody.gravityScale = _itemSourceSO.GravityScale;
    SpriteRenderer.sprite = _icon;
    InitializeStateMachine();
}

private void Update()
{
    _stateMachine.Perform();
}

private void FixedUpdate()
{
    _stateMachine.FixedPerform();
}

public void Initialize(IInventoryItem inventoryItem)
{
    InventoryItem = inventoryItem;
    _icon = inventoryItem.Icon;
}

void InitializeStateMachine()
{
    _stateMachine = new ItemResourceStateMachine(this);

    PushState = new PushState(_stateMachine, this, _itemSourceSO);
    FollowState = new FollowState(_stateMachine, this, _itemSourceSO);
    OnGroundState = new OnGroundState(_stateMachine, this, _itemSourceSO);
    _stateMachine.Initialize(PushState);
}

public bool PickupResource()
{
    return PlayerInventory.TryPickupResource(this);
}

public float GetDistanceToPlayer()
{
    return Vector2.Distance(_playerTransform.position,
        Rigidbody.position);
}

public Vector2 GetPlayerDirectionVector()
{
    Vector2 playerPos = _playerTransform.position;
    Vector2 itemPos = transform.position;
    Vector2 direction = playerPos - itemPos;
    return direction.normalized;
}

```

```

    }
}
}

```

Листинг InventoryItem.cs:

```

using Scripts.InventoryCode;
using Scripts.SaveLoader;
using System;
using System.Collections.Generic;
using UnityEngine;

namespace Scripts.InventoryCode
{
    [CreateAssetMenu(fileName = "InventoryItem",
        menuName = "SO/InventoryItems/InventoryItem")]
    public class InventoryItem : ScriptableObject, IInventoryItem
    {
        public string Name => _name;

        public Sprite Icon => _icon;

        [SerializeField] private string _name;
        [SerializeField] private Sprite _icon;

        public virtual void RenderUI(InventoryCell inventoryCell)
        {
            inventoryCell.Icon.sprite = _icon;
            inventoryCell.Text.text = _name;
        }

        public virtual InventoryItemData GetItemData()
        {
            InventoryItemData inventoryItemData
                = new InventoryItemData() { SoName = _name };
            return inventoryItemData;
        }
    }
}

```

Листинг InventoryCell.cs:

```

using Scripts.InventoryCode.ItemResources;
using System;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;
using Zenject;

```

```

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

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

        public Image Icon => IconElement;
        public TextMeshProUGUI Text => TextElement;

        public int BeginDragSiblingIndex { get; private set; }

        ItemResourceDroper _itemResourceDroper;

        [Inject]
        public void Construct(ItemResourceDroper itemResourceDroper)
        {
            _itemResourceDroper = itemResourceDroper;
        }

        private void OnDisable()
        {
            _endDragEvent = null;
        }

        private void Start()
        {
        }

        }

        public void Initialize(Transform globalVisualContext,
            IInventoryItem inventoryItem)
        {
            _globalVisualContext = globalVisualContext;
            OriginVisualContext = transform.parent;
            InventoryItem = inventoryItem;
        }

        public void RegisterEvents(Action endDragEvent, Action<InventoryCell> beginDragEvent)
        {
            _endDragEvent = endDragEvent;
            _beginDragEvent = beginDragEvent;
        }

        private void Update()
        {
            InventoryItem?.RenderUI(this);
        }
    }
}

```

```

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

public void OnEndDrag(PointerEventData eventData)
{
    InventoryBase inventory;
    if (DragExtension.CheckMouseIntersectionWithContainers(eventData,
        out inventory))// если есть пересечение с другим инвентарем
    {
        // если это тот же инвентарь
        if (OriginVisualContext == inventory.Container)
        {
            DragExtension.PlaceInTheNearestCellLocal(OriginVisualContext,
                this, BeginDragSiblingIndex);
        }
        else//другой инвентарь
        {
            if (inventory.IsFull())// если полон, то не перекладывать
            {
                DragExtension.PlaceInTheNearestCellLocal(OriginVisualContext,
                    this, BeginDragSiblingIndex);
            }
            else// переложить и переподписать ячейку на события другого инвентаря
            {
                DragExtension.PlaceInTheNearestCellGlobal(inventory.Container, this);
                OriginVisualContext = inventory.Container;
                _endDragEvent?.Invoke();
                inventory.RegisterDragEvents(this);
                _endDragEvent?.Invoke();
                return;
            }
        }
    }
    else// нет пересечений с другим инвентарем
    {
        _endDragEvent?.Invoke();
        Destroy(this.gameObject);
        _itemResourceDroper.DropByPlayer(InventoryItem);
    }
    _endDragEvent?.Invoke();
}
}
}

```

Листинг InventoryCell.cs:

```
using Scripts.InventoryCode.ItemResources;
using System;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;
using Zenject;

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

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

        public Image Icon => IconElement;
        public TextMeshProUGUI Text => TextElement;

        public int BeginDragSiblingIndex { get; private set; }

        ItemResourceDroper _itemResourceDroper;

        [Inject]
        public void Construct(ItemResourceDroper itemResourceDroper)
        {
            _itemResourceDroper = itemResourceDroper;
        }

        private void OnDisable()
        {
            _endDragEvent = null;
        }

        private void Start()
        {
        }

        public void Initialize(Transform globalVisualContext,
            InventoryItem inventoryItem)
        {
            _globalVisualContext = globalVisualContext;
            OriginVisualContext = transform.parent;
            InventoryItem = inventoryItem;
        }
    }
}
```

```

    }
    public void RegisterEvents(Action endDragEvent, Action<InventoryCell> beginDragEvent)
    {
        _endDragEvent = endDragEvent;
        _beginDragEvent = beginDragEvent;
    }
    private void Update()
    {
        InventoryItem?.RenderUI(this);
    }

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

    public void OnEndDrag(PointerEventData eventData)
    {
        InventoryBase inventory;
        if (DragExtension.CheckMouseIntersectionWithContainers(eventData,
            out inventory))// если есть пересечение с другим инвентарем
        {
            // если это тот же инвентарь
            if (OriginVisualContext == inventory.Container)
            {
                DragExtension.PlaceInTheNearestCellLocal(OriginVisualContext,
                    this, BeginDragSiblingIndex);
            }
            else//другой инвентарь
            {
                if (inventory.IsFull())// если полон, то не перекладывать
                {
                    DragExtension.PlaceInTheNearestCellLocal(OriginVisualContext,
                        this, BeginDragSiblingIndex);
                }
                else// переложить и переподписать ячейку на события другого инвентаря
                {
                    DragExtension.PlaceInTheNearestCellGlobal(inventory.Container, this);
                    OriginVisualContext = inventory.Container;
                    _endDragEvent?.Invoke();
                    inventory.RegisterDragEvents(this);
                    _endDragEvent?.Invoke();
                    return;
                }
            }
        }
    }
}

```



```

        else// нет пересечений с другим инвентарем
        {
            _endDragEvent?.Invoke();
            Destroy(this.gameObject);
            _itemResourceDroper.DropByPlayer(InventoryItem);
        }
        _endDragEvent?.Invoke();
    }
}
}

```

Листинг ChestData.cs:

```

using Scripts.InventoryCode;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace Scripts.SaveLoader
{
    public class ChestData : PlacementItemData
    {
        public List<InventoryItemData> Items;
        public ChestData()
        {
            Items = new List<InventoryItemData>();
            ItemTypeName = nameof(ChestData);
        }

        public void UpdateItems(List<InventoryItem> inventoryItems)
        {
            foreach (var item in inventoryItems)
            {
                Items.Add(item.GetItemData());
            }
        }
    }
}

```

Листинг GameDataSaveLoader.cs:

```

using System;
using System.Collections.Generic;
using UnityEngine;
using Newtonsoft.Json;
using Assets.Scripts;
using Newtonsoft.Json.Linq;

namespace Scripts.SaveLoader
{

```

```

public class GameDataSaveLoader
{

    public GameDataSaveLoader() { }

    public void SaveGameState(GameDataState gameDataState)
    {
        string json = JsonConvert.SerializeObject(gameDataState, Formatting.Indented);
        PlayerPrefs.SetString(gameDataState.GameDataStateName,
            json);
    }

    public GameDataState LoadGameState(string keyName)
    {
        string json = PlayerPrefs.GetString(keyName);
        GameDataState gameDataState =
            JsonConvert.DeserializeObject<GameDataState>(json,
                new JsonSerializerSettings
                {
                    Converters = new List<JsonConverter> { new ItemPlacementDataConverter() }
                });
        return gameDataState;
    }

    public void SaveWorldNamesJson(List<string> worldNames)
    {
        string json = JsonConvert.SerializeObject(worldNames);
        PlayerPrefs.SetString(GameConfiguration.SaveLevelNamesKeyName, json);
    }

    public List<string> LoadWorldNamesJson()
    {
        string json =
            PlayerPrefs.GetString(GameConfiguration.SaveLevelNamesKeyName);
        var names =
            JsonConvert.DeserializeObject<List<string>>(json);
        return names;
    }

    public class ItemPlacementDataConverter : JsonConverter
    {
        public override bool CanConvert(Type objectType)
        {
            return objectType == typeof(PlacementItemData);
        }

        public override object ReadJson(JsonReader reader, Type objectType, object existingValue,
            JsonSerializer serializer)
        {
            JObject item = JObject.Load(reader);
            var typeName = item["ItemTypeName"].ToString();
            switch (typeName)
            {
                case nameof(ChestData):

```

```

        return item.ToObject<ChestData>(serializer);
    default:
        return item.ToObject<PlacementItemData>(serializer);
    }
}

public override void WriteJson(JsonWriter writer, object value, JsonSerializer serializer)
{
    throw new NotImplementedException();
}
}
}
}

```

Листинг GameState.cs:

```

using Scripts.InventoryCode;
using Scripts.PlacementCode;
using System;
using System.Collections.Generic;
using System.Numerics;

namespace Scripts.SaveLoader
{
    public class GameState
    {
        public string GameStateName;
        public PlayerData PlayerData;
        public List<PlacementItemData> PlacementObjectsDataList;
        public List<InventoryItemData> ActivePackInventory;
        public List<InventoryItemData> BackPackInventory;

        public GameState(string gameStateName)
        {
            ActivePackInventory = new List<InventoryItemData>();
            BackPackInventory = new List<InventoryItemData>();
            GameStateName = gameStateName;
            PlacementObjectsDataList = new List<PlacementItemData>();
        }

        public void UpdateActivePackInventory(List<IInventoryItem> inventoryItems)
        {
            List<InventoryItemData> itemDataList =
                new List<InventoryItemData>();
            foreach (var item in inventoryItems)
            {
                itemDataList.Add(item.GetItemData());
            }
            ActivePackInventory = itemDataList;
        }

        public void UpdateBackPackInventory(List<IInventoryItem> inventoryItems)
        {
            List<InventoryItemData> itemDataList =

```

```

        new List<InventoryItemData>();
        foreach (var item in inventoryItems)
        {
            itemDataList.Add(item.GetItemData());
        }
        BackPackInventory = itemDataList;
    }
    public void AddItemData(PlacementItemData itemData)
    {
        PlacementObjectsDataList.Add(itemData);
    }
    public void RemoveItemData(PlacementItemData itemData)
    {
        PlacementObjectsDataList.Remove(itemData);
    }
}
}

```

Листинг GameStateGenerato.cs:

```

using Scripts.InteractableObjects;
using Scripts.InventoryCode;
using Scripts.MainMenuCode;
using Scripts.PlacementCode;
using Scripts.SaveLoader;
using System;
using System.Collections.Generic;
using UnityEngine;
using Zenject;

namespace AScripts.SaveLoader
{
    public class GameStateGenerator : MonoBehaviour
    {
        [Header("Back Pack")]
        [SerializeField] List<InventoryItem> _backPackStartItemKit;
        [SerializeField] InventoryBase _backPackInventory;
        [Space]
        [Space]
        [Header("Active Pack")]
        [SerializeField] List<InventoryItem> _activeStartItemKit;
        [SerializeField] InventoryBase _activeInventory;
        [Space]
        [Space]
        [Header("Seller Pack")]
        [SerializeField] List<InventoryItem> _sellerStartItemKit;
        [SerializeField] InventoryBase _sellerInventory;

        GameState _gameDataState;
        Dictionary<string, InventoryItem> _inventoryItemsDictionary;
        IChestFactory _chestFactory;
        ItemPlacementMap _placementMap;
    }
}

```

```

Movement _movement;

[Inject]
public void Construct(GameDataState gameDataState,
    Dictionary<string, IInventoryItem> inventoryItemsDictionary,
    ItemPlacementMap itemPlacementMap,
    IChestFactory chestFactory,
    Movement movement)
{
    _gameDataState = gameDataState;
    _inventoryItemsDictionary = inventoryItemsDictionary;
    _chestFactory = chestFactory;
    _placementMap = itemPlacementMap;
    _movement = movement;
}

private void Start()
{
    LoadPlayerInventories();
    LoadPlacementItems();
    LoadPlayerData();
    //LoadChestData();
}

void LoadPlayerData()
{
    PlayerData playerData = _gameDataState.PlayerData;
    if(LoadedData.IsDefault == false)
    {
        _movement.SetLoadedPosition(playerData.GetPosition());
    }
}

void LoadPlacementItems()
{
    List<PlacementItemData> placementItems =
        _gameDataState.PlacementObjectsDataList;

    foreach (PlacementItemData itemData in placementItems)
    {
        if(itemData is ChestData)
        {
            var data = itemData as ChestData;
            List<IInventoryItem> inventoryItems = ProcessInventoryItemsData(data.Items);
            Chest chest = _chestFactory.Create(inventoryItems);
            Vector2Int pos = itemData.GetPosition();
            Vector3Int pos3 = new Vector3Int(pos.x, pos.y, 0);
            _placementMap.PlaceObjectOnCell(chest.gameObject, pos3);
            _placementMap.AddPosition(pos);
        }
    }
}

void LoadPlayerInventories()
{
    if (LoadedData.IsDefault)

```

```

    {
        List<IInventoryItem> activeItems = new List<IInventoryItem>(_activeStartItemKit);
        List<IInventoryItem> backItems = new List<IInventoryItem>(_backPackStartItemKit);
        _backPackInventory.Initialize(backItems);
        _activeInventory.Initialize(activeItems);
    }
    else
    {
        List<IInventoryItem> backPackInventoryItems =
            ProcessInventoryItemsData(_gameDataState.BackPackInventory);
        _backPackInventory.Initialize(backPackInventoryItems);

        List<IInventoryItem> activePackInventoryItems =
            ProcessInventoryItemsData(_gameDataState.ActivePackInventory);
        _activeInventory.Initialize(activePackInventoryItems);
    }
}

private IInventoryItem ProcessItemData(InventoryItemData itemData)
{
    string itemName = itemData.SoName;
    IInventoryItem inventoryItem =
        _inventoryItemsDictionary[itemName];
    switch (itemData)
    {
        case BagInventoryItemData data:
        {
            IBagInventoryItem bag = inventoryItem as IBagInventoryItem;
            bag.Count = data.Count;
            return bag;
        }
        case ChestInventoryItemData data:
        {
            IChestInventoryItem chest = inventoryItem as IChestInventoryItem;
            List<IInventoryItem> inventoryItems
                = ProcessInventoryItemsData(data.ItemsList);
            chest.Items = inventoryItems;
            return chest;
        }
        case QuantitativeItemData data:
        {
            IQuantitativeInventoryItem quantitativeInventoryItem
                = inventoryItem as IQuantitativeInventoryItem;
            quantitativeInventoryItem.Count = data.Count;
            return quantitativeInventoryItem;
        }
        default:
        {
            return inventoryItem;
        }
    }
}

```

```
private List<IInventoryItem> ProcessInventoryItemsData(  
    List<InventoryItemData> inventoryItemsData)  
{  
    List<IInventoryItem> inventoryItems  
        = new List<IInventoryItem>();  
    foreach (var itemData in inventoryItemsData)  
    {  
        inventoryItems.Add(ProcessItemData(itemData));  
    }  
    return inventoryItems;  
}  
}
```