МИНИСТЕРСТВО ОБРАЗОВАНИЯ РУСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования
«Гомельский государственный технический университет имени П.О.Сухого»

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

ОТЧЕТ
по технологической практике

База практики: _____

Составил:
Студент гр. ИТИ-31 _____ Дубовцов И.Д.
(подпись, дата)

Руководитель практики
от предприятия: _____
(должность)

_____ Н.А.
(подпись, дата)

Руководитель практики
от университета: _____
(должность)

_____ Дорощенко И.В.
(подпись, дата)

Дата защиты: _____

Оценка работы: _____

Подписи членов комиссии:
_____

Гомель 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Технологическая практика является частью образовательного процесса подготовки специалистов, продолжением учебного процесса в производственных условиях и проводится на передовых предприятиях, в учреждениях, организациях различных отраслей. Технологическая практика направлена на закрепление полученных процессе обучения в вузе знаний, развитии и закреплении умений и приобретение навыков решения профессиональных задач в производственных условиях. Практика организуется с учетом будущей специальности, предрасположенности и заинтересованности студентов в определенной специфике деятельности.

Целями практики являются:

– изучение передовой технологии предприятий и направлений ее совершенствования, знакомство со структурой предприятия, уровнем его автоматизации, основными производственными процессами, системами и средствами их автоматизированной поддержки и управления и создание у обучаемого мотивационных ориентиров по отношению к будущей профессиональной деятельности;

– изучение и анализ деятельности предприятия, основных процессов, применяемых систем и средств автоматизации, методов разработки внедрения и использования программных продуктов и современных технологий в производственных условиях, анализ их обоснованности и эффективности использования, разработку предложений на улучшение;

– ознакомление с применяемыми на производстве современными программными разработками;

– изучение технической и программной документации применяемых информационных систем;

– анализ организации и охрана труда, обеспечения техники безопасности, пожарной и экологической безопасности на предприятии.

Задачи практики:

– ознакомление с технологией производства элементов, а также с методами сборки, наладки и контроля узлов и устройств, образующих средства технического обеспечения АСОИ;

– приобретение практических навыков работы с техническим оборудованием, измерительной контрольной аппаратурой;

– изучение назначения и структуры АСОИ, основных целей ее создания и перечня выполняемых функций.

В этом отчете будет описан ход выполнения поставленного индивидуального задания: автор опишет ход разработки игры в жанре экшион шутер от третьего лица. Автор придумает концепт-документ, в котором будут описаны все элементы игры, а также опишет процесс создания игрового приложения в жанре экшион шутер от третьего лица в среде разработки Unity.

1 ОБЩИЕ СВЕДЕНИЯ О ПРЕДПРИЯТИИ СП ОАО «СПАРТАК»

1.1 Сведения об организации

Кондитерская фабрика «Спартак» является одним из крупнейших производителей кондитерских изделий и полуфабрикатов собственного производства в Республике Беларусь, выпускающий около 350 наименований кондитерских изделий, включая изделия 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).

Сегодня успех современных компаний определяется качеством – начиная с производства заканчивая продукцией и предоставляемыми услугами. В современной рыночной экономике основным конкурентным преимуществом любого предприятия становиться качество производимой продукции. Качество - главный инструмент в достижении устойчивого успеха и благополучия любой организации.

«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 рецептур. Именно поэтому портфель наград фабрики полнее с каждым годом.

1.2 Деятельность предприятия

В рамках своей практической работы автор имел 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, чтобы обеспечить высокое качество и безопасность всех изделий.

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. Компания активно внедряет энергоэффективные технологии, осуществляет рациональное использование ресурсов и поддерживает различные благотворительные инициативы в своем регионе.

1.3 Должностные обязанности инженера программиста

В ходе практической работы автор ознакомился 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 включают:

1) Разработка программного обеспечения:

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

2) Тестирование и отладка:

– инженер программист проводит тестирование разработанных программных решений, выявляет и исправляет возможные ошибки и проблемы. Он осуществляет отладку программ, чтобы обеспечить их стабильную и надежную работу;

3) Управление базами данных:

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

4) Поддержка пользователей:

– инженер программист оказывает техническую поддержку пользователям программного обеспечения. Он отвечает на вопросы и решает проблемы, связанные с работой программ, обучает пользователей и предоставляет необходимую документацию;

5) Сотрудничество с другими отделами:

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

6) Исследования и развитие:

– инженер программист следит за новыми технологиями и тенденциями в области программирования и применяет их на практике. Он 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.

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.

2.1.1 Платформы.

Аппаратно-программные системы, позволяющие запускать интерактивные игровые приложения. Среди основных видов можно выделить:

– 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*).

2.1.2 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. Также современные игровые движки обеспечивают кроссплатформенность создаваемых продуктов. Из наиболее продвинутых движков можно выделить: *Unity 3D*, *Unreal Development Kit*, *CryENGINE* 3 *Free SDK*.

2.1.3 Разработка игр

Большое количество компаний и независимых команд занимаются созданием компьютерных игр. В разработке участвует специалисты разных профессий: программисты, гейм-дизайнеры, художники, *QA* специалисты и др.

К разработке крупных коммерческих игровых продуктов привлекаются большие профессиональные команды. И стоить подобные проекты в разработке могут 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, качественные и практически бесплатные движки, площадки по привлечению «народных» инвестиций (краудфандинг) и доступные каналы распространения.

2.1.4 Издание и оперирование игр.

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

2.1.5 Популяризация.

Специализированные средства массовой информации всегда являлись мощным каналом донесения информации до пользователей.

Сейчас наиболее эффективным и широко представленным направлением СМИ являются информационные сайты, посвященные игровой тематике. Игровые журналы, долгое время выступавшие главным источником информации об играх, в настоящее время, уступили свое место интернет ресурсам.

Специализированные выставки все еще остаются важным информационными площадками для игровой индустрии (*E3*, *GDC*, *Gamescom*, ИгроМир, КРИ, *DevGamm*). Прямое общение прессы и игроков с разработчиками, обмен опытом между участниками рынка, новые контакты – вот то, что предлагают конференции и выставки в концентрированной форме.

Еще один важный канал донесения полезной информации до игроков – это ТВ-передачи, идущие как в формате классического телевещания, так и на множестве каналов видео-контента.

2.1.6 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, прегражденние 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 которой 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 вы будете компоновать сцены будущей игры, связывая игровые ресурсы и код в интерактивные объекты. Он 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 1.1.

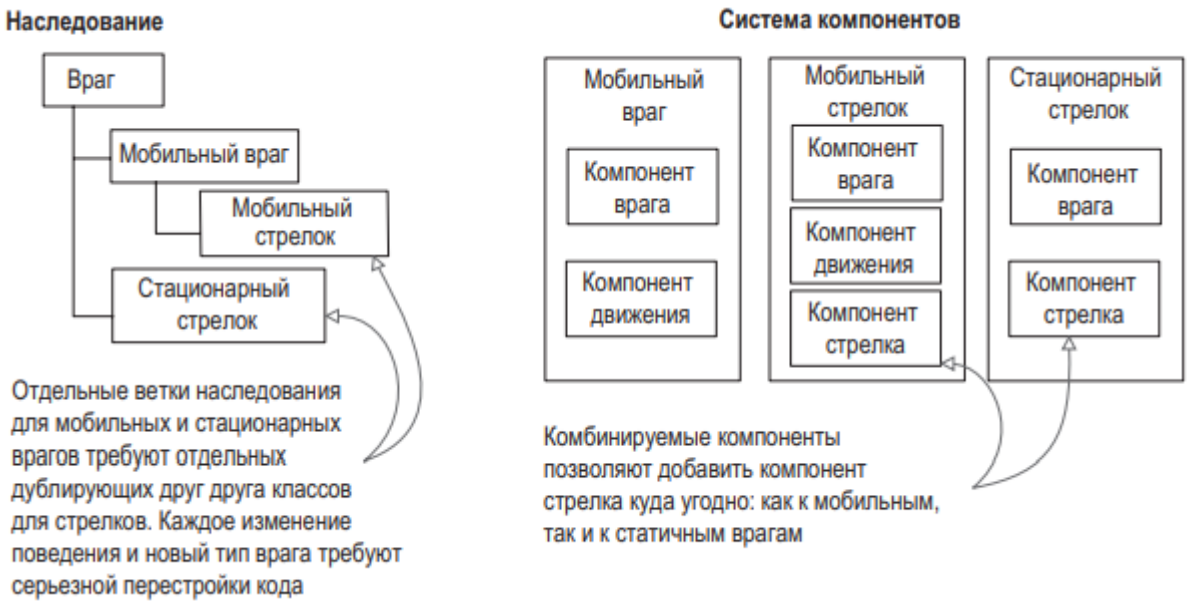

Рисунок 1.1 – Сравнение наследования с компонентной системой

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. Такая компоновка облегчает создание прототипов, потому что взять нужный набор компонентов куда быстрее и проще, чем перестраивать цепочку наследования при изменении каждого объекта. 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 [2, с.17].

2.3 Игровой движок *Unreal Engine*

Unreal Engine – 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 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.

«Анриал Энджин» остается популярным более 20 лет по нескольким причинам. Имеет ряд преимуществ:

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

2.3.1 Blueprint.

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.

2.3.2 Материалы.

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.

2.3.3 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.

2.3.4 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.

2.3.5 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*.

2.3.6 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*. Она позволяет задавать размеры частиц, траекторию и скорость их движения, цвет и масштабирование в течение всего срока их существования.

2.3.7 Искусственный интеллект.

В компьютерной игре существуют не только главные, но и второстепенные персонажи. Искусственный интеллект 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, по мнению автора, 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 движения управляются с помощью контроллеров анимации (*animation controllers*), а во втором по тому же принципу действуют анимационные *Blueprints*.

В обоих движках есть стейт-машины, определяющие переходы из одного состояния ассета в другое. В *UE4* система называется *Persona*, а в *Unity* – *Mecanim*. Также 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. По мнению автора, 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.

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.

2.5 Язык программирования *C#*

C# – это язык с *C*-подобным синтаксисом. Здесь он близок в этом отношении к *C++* и *Java* [3, с.21].

Будучи объектно-ориентированным языком, он много перенял у *Java* и *C++*. Как и *Java*, *C#* изначально предназначался для веб-разработки, и примерно 75% его синтаксических возможностей такие же, как у *Java*. *C#* также называют «очищенной версией *Java*». Ещё 10% *C#* позаимствовал из *C++* и 5% – из *Visual Basic*. Оставшиеся 10% *C#* – это реализация собственных идей разработчиков. Объектно-ориентированный подход позволяет строить с помощью *C#* крупные, но в то же время гибкие, масштабируемые и расширяемые приложения.

C# уже давно поддерживает много полезных функций:

– инкапсуляция;

– наследование;

– полиморфизм;

– перегрузка операторов;

– статическая типизация [4, с.43].

При этом он всё ещё активно развивается, и с каждой новой версией появляется всё больше интересного – например лямбды, динамическое связывание, асинхронные методы.

По сравнению с другими языками *C#* довольно молод, но в то же время он уже прошёл большой путь. Первая версия языка вышла вместе с релизом *Microsoft Visual Studio .NET* в феврале 2002 года. Текущей версией языка является версия *C# 8.0*, которая вышла в сентябре 2019 года вместе с релизом *.NET Core 3* [6].

У «шарпа» выделяют много преимуществ:

– поддержка подавляющего большинства продуктов *Microsoft*;

– бесплатность ряда инструментов для небольших компаний и некоторых индивидуальных разработчиков (*Visual Studio*, облако *Azure*, *Windows Server*, *Parallels Desktop* для *Mac Pro*);

– типы данных имеют фиксированный размер (32-битный *int* и 64-битный *long*), что повышает «мобильность» языка и упрощает программирование, так как вы всегда знаете точно, с чем вы имеете дело;

– автоматическая «сборка мусора», это значит, что нам в большинстве случаев не придётся заботиться об освобождении памяти. Вышеупомянутая общеязыковая среда *CLR* сама вызовет сборщик мусора и очистит память;

– большое количество «синтаксического «сахара» (специальных конструкций, разработанных для понимания и написания кода);

– низкий порог вхождения;

– с помощью *Xamarin* на *C#* можно писать программы и приложения для таких операционных систем, как *iOS*, *Android*, *MacOS* и *Linux* [5, с.38].

Но есть у *C#* и некоторые недостатки:

– приоритетная ориентированность на платформу *Windows*;

– язык бесплатен только для небольших фирм, индивидуальных программистов, стартапов и учащихся;

– инструментарий *C#* позволяет решать широкий круг задач, язык действительно очень мощный и универсальный.

На нём часто разрабатывают:

– веб-приложения;

– игры;

– мобильные приложения для *Android* или *iOS*;

– программы под *Windows*.

Перечень возможностей разработки практически не имеет ограничений благодаря широчайшему набору инструментов и средств. Конечно, всё это можно реализовать при помощи других языков. Но некоторые из них узкоспециализированные, а в некоторых придётся использовать дополнительные инструменты сторонних разработчиков. В *C#* решить широкий круг задач возможно быстрее, проще и с меньшими затратами времени и ресурсов.

3 ХАРАКТЕРИСТИКА ЖАНРА «ШУТЕР ОТ ТРЕТЬЕГО ЛИЦА»

3.1 Особенности жанра

Шутер от третьего лица – жанр компьютерных игр, разновидность трехмерных шутеров, в которой управляемый игроком персонаж виден на экране, а геймплей в значительной части состоит из стрельбы. При этом виртуальная камера, как правило, находится позади персонажа, за спиной или плечом.

Он тесно связан с шутерами от первого лица, но с персонажем игрока, видимым на экране во время игры. В то время как *2D* -игры *shoot 'em up* также используют вид от третьего лица, жанр *TPS* отличается тем, что в игре представлен аватар игрока в качестве основного фокуса обзора камеры.

Границы между шутерами от третьего лица и от первого лица не всегда четкие. Например, многие шутеры от третьего лица позволяют игроку использовать ракурс от первого лица для задач, требующих точного прицеливания, в то время как другие просто позволяют игроку свободно переключаться между ракурсами от первого и третьего лица по желанию.

На рисунке 3.1 продемонстрирован пример вида от первого лица.

Рисунок 3.1 – Пример вида от первого лица

Многие игры в этом жанре, такие как серия *ARMA* и ее потомки (включая популярный шутер *PUBG* в стиле королевской битвы), позволяют игрокам

свободно переходить от первого к третьему лицу по своему желанию.

В *TPS* уровни – это замкнутые локации, где присутствуют противники, препятствия и полезные предметы. Такими являются предметы для возобновления состояния персонажа (аптечки, броня), для решения головоломок (подсказки) и для дальнейшего прохождения уровня (ключи от дверей). Так же шутеры от третьего лица имеют интерфейс *HUD*, где на экране отображается информация про игрока: оружие, боеприпасы, здоровье и броня и т.д. Пройти уровень можно либо воспользовавшись одним с нескольких альтернативных путей, либо только одним, строго обозначенным. Часто, чтобы пройти уровень нужно выполнять различные задания (уничтожить заданную цель, найти пропавшего персонажа и т.д.). В конце уровня игроку потребуется убить босса – особо сильного противника, которого нужно победить не столько превосходящей силой, но и знанием его слабых мест.

На рисунке 3.2 продемонстрирован пример вида от третьего лица.

Рисунок 3.2 – Пример вида от третьего лица

Герой носит при себе разнообразное оружие, часто при прохождении игры появляется возможность его усовершенствования. Противники тоже вооружены, но могут быть и безоружные враги такие как звери, чудовища или зомби. В шутере от третьего лица враги могут нападать как по одному, так и группами.

3.2 Анализ представителей

3.2.1 Warframe – кооперативная компьютерная онлайн игра в жанрах *Action/RPG* и шутера от третьего лица, разработанная и выпущенная канадской

студией *Digital Extremes* для платформ *Microsoft Windows*, *PlayStation 4* и *Xbox One*. Разработкой версии игры для *Nintendo Switch* занимается компания *Panic Button*.

Действие *Warframe* происходит в вымышленной научно-фантастической вселенной, в которой несколько сторон соперничают за власть над Солнечной системой. Игроки принимают на себя роли Тэнно – древних воинов, использующих в сражениях дистанционно управляемые тела «варфреймы» с разнообразным оружием и способностями.

Геймплей *Warframe* сочетает в себе бои со стрельбой и использованием холодного оружия, паркур и также включает в себя элементы компьютерных ролевых игр – игрок постоянно улучшает снаряжение своего персонажа.

На рисунке 3.3 продемонстрирован кадр из игры *Warframe*.

Рисунок 3.3 – Кадр из игры *Warframe*

Игра рассчитана на четырёх человек. Игроки используют варфрейм (с собственным набором умений и характеристик) и оснащены видами вооружения: основное оружие (такое как винтовки, дробовики, луки или снайперское вооружение), вторичное оружие (обычно пистолет, но также имеются пистолеты-пулемёты, метательное оружие), оружие ближнего боя (мечи, парные мечи, кинжалы, топоры, косы, глефы, перчатки), и паразон (специальный техно-кинжал, предназначенный для добивания врагов и взлома консолей). Игрок может прыгать, бегать, скользить и перекатываться, а также комбинировать методы для быстрого перемещения по уровню и избежания вражеского огня, а также для получения доступа к секретной области.

Warframe распространяется по модели *free-to-play* и поддерживается за счёт микроплатежей – игроки могут за реальные деньги приобрести в игровом магазине внутриигровые предметы.

Warframe получила смешанные отзывы на всех платформах, согласно сайту агрегирования обзоров *Metacritic*. Летом 2018 года, благодаря уязвимости в защите *Steam Web API*, стало известно, что точное количество пользователей сервиса *Steam*, которые играли в игру хотя бы один раз, составляет 16 332 217 человек [8].

3.2.2 Red Dead Redemption 2 – компьютерная игра в жанрах *action adventure* и шутера от третьего лица с открытым миром, разработанная *Rockstar Studios* и выпущенная *Rockstar Games* для консолей *PlayStation 4* и *Xbox One* 26 октября 2018 года и для персональных компьютеров под управлением *Windows* 5 ноября 2019 года. Является третьей игрой в серии *Red Dead* и приквелом к *Red Dead Redemption* 2010 года.

На рисунке 3.4 продемонстрирован кадр из игры *RDR2*.

Рисунок 3.4 – Кадр из игры *Red Dead Redemption 2*

Действие *Red Dead Redemption 2*, оформленной в духе вестерна, происходит на территории нескольких вымышленных штатов США на рубеже XIX – XX веков. Сюжет игры построен вокруг приключений банды Датча Ван дер Линде; под управлением игрока находится один из членов банды – Артур Морган, а после прохождения сюжетной линии до эпилога – другой член банды, Джон Марстон.

Игра предлагает игроку свободно путешествовать по обширному миру игры, самостоятельно находя интересные места и занятия – к числу таких

возможных занятий принадлежат перестрелки, ограбления, охота на диких животных, скачки на лошадях и нахождения коллекционных предметов (сигаретные карточки, ловля легендарных рыб или животных, кости динозавров, наскальные рисунки и т. д.). Особая система «чести», учитывающая как достойные, так и преступные поступки игрока, влияет как на сюжет игры, так и на игровой процесс.

После выхода *Red Dead Redemption 2*, как одна из самых ожидаемых и широко рекламируемых в это время игр, побила несколько рекордов продаж – лишь за две недели после их начала было продано свыше 17 миллионов копий игры, принеся компании *Rockstar* свыше 725 миллионов долларов прибыли. Она получила самые высокие оценки прессы – обозреватели удостоили самых высоких похвал сюжет, персонажей, предоставленную игроку свободу и чрезвычайное внимание разработчиков к деталям. По итогам 2018 года игра также собрала ряд престижных наград, в том числе «Выбор критиков» на церемонии *Golden Joystick Awards* и «Лучшее повествование» и «Лучший саундтрек» на церемонии *The Game Awards*. Также в 2018 году игра стала лауреатом 22-й ежегодной премии *D.I.C.E. Awards* – премии Академии интерактивных искусств и наук (*Academy of Interactive Arts & Sciences*), одержав победу в номинации «Выдающееся техническое достижение» [9].

3.2.3 Lost Planet: Extreme Condition – компьютерная игра в жанре шутер от третьего лица, разработанная и опубликованная *Capcom* для *Xbox 360, Microsoft Windows* и *PlayStation 3*. Игра была выпущена на *Xbox 360* в Японии в декабре 2006 года, в Северной Америке и *PAL* регионах в январе 2007 года, на ПК(портативных компьютерах) в Северной Америке и *PAL*-регионах в июне 2007 года, и на *PlayStation 3* во всем мире в феврале 2008 года.

Событие игры *Lost Planet* происходит в тысяча восьмидесятом году, когда условия для проживания на Земле для человека стали невозможны.

Игрок управляет главным персонажем игры с видом от третьего лица. Игрок может переключаться между видом от первого лица и видом от третьего лица в любой момент. Игрок может путешествовать пешком или ездить на различных механизированных костюмах, называемых бронекостюмами. Бронекостюм несёт тяжелое вооружение, такое как пулемёт Гатлинга и ракетные пусковые установки. Можно подобрать оружие, которое лежит на земле, и вести огонь из разного оружия одновременно. Передвигаясь пешком, игроки могут использовать крюк, чтобы добраться до труднодоступных мест, или подключить к бронекостюму и захватить его. Вождение бронекостюма и использование некоторых видов оружия требует тепловой энергии. Кроме того, холодная температура планеты постоянно уменьшает уровень тепловой энергии игрока. Игрок может пополнять свой уровень тепловой энергии, побеждая врагов или активируя посты передачи данных. Посты передачи данных также позволяют

игроку использовать их навигационные радары, чтобы увидеть приближающихся врагов.

На рисунке 3.5 продемонстрирован кадр из игры *LostPlanet*.

Рисунок 3.5 – Кадр из игры *Lost Planet*

Lost Planet получила смешанные отзывы на *PlayStation 3* и ПК, но более позитивный прием получила версия *Xbox 360*, которая была оригинальной ведущей платформой. К январю 2007 года по всему миру было продано более миллиона копий игры, что стало вторым миллионным тиражом *Capcom* для *Xbox 360*. По состоянию на март 2016 года, только на *Xbox 360* было продано более 1 600 000 копий игры, включая загружаемые копии. *IGN* присудил версии для *Xbox 360* награду «Выбор редактора», и она получила награду за лучшую игру для *Xbox 360* на Лейпцигской игровой конвенции [10].

3.2.4 Saints Row IV – мультиплатформенная компьютерная игра в жанре приключенческого боевика с элементами открытого мира, разработанная компанией *Volition*. Релиз состоялся 23 августа 2013 года на *PC, PlayStation 3* и *Xbox 360*.

Как и в предыдущих частях серии, игрок управляет лидером «Святых с 3-ей улицы», который становится президентом США. Действие игры происходит в вымышленном городе Стилпорт из *Saints Row: The Third*, преобразованном под ретрофутуристическую антиутопию с элементами чёрного юмора.

Перенесение места действия в виртуальное пространство дало авторам возможность расширить геймплейные элементы. Ложки гнутся силой мысли, люди прыгают чуть ли не до орбиты, а по улицам разгуливают терминаторы и унитазы с оружием. *Volition* сделала финт от концепции *GTA* в сторону игр про буйства супергероев в открытом городе. Чтобы лучше всего понять сущность

геймплея *Saints Row 4*, нужно представить себе третью часть, погрузить ее в вышеописанную атмосферу и добавьте возможности из *Prototype*.

На рисунке 3.6 продемонстрирован кадр из игры *Saints Row 4*.

Рисунок 3.6 –Кадр из игры *Saint Row 4*

Saints Row 4 ни на секунду не дает уверовать в то, что игрок разобрался во всех аспектах геймплея. Она постоянно подкидывает новых врагов, оружие, способности. Помимо основного процесса отстрела противников нам предложат множество мини-игр. Так, вскрытие магазинных замков оформлено как небольшой пазл, в котором мы должны при помощи имеющихся деталей проложить путь от одной точки до другой. На сюжетной тропинке также встретятся неожиданные решения: в одном квесте нам придется покататься на двухмерном танке и пройти текстовую адвенчуру; в следующем дадут поуправлять гигантским роботом; не обошлось и без своеобразной стелс-миссии, где придется отстреливать лампочки и прятаться в картонных коробках. И так на протяжении всей игры.

Saints Row IV в значительной степени получила положительные отзывы. Веб-сайты *GameRankings* и *Metacritic* дали *PC*-версии 88,53 % и 86/100, *PS3*-версии 80,27 % и 77/100, и *Xbox* 360-версии 84,16 % и 83/100, соответственно.

Летом 2018 года, благодаря уязвимости в защите *Steam Web API*, стало известно, что точное количество пользователей сервиса *Steam*, которые играли в игру хотя бы один раз, составляет 5 275 914 человек [7].

4. РАЗРАБОТКА ИГРЫ «*DEFENDER ATTACK*»

4.1 Концепт игры

4.1.1 Жанр и аудитория.

Игра «*Defender Attack*» относится к жанру игр шутер от третьего лица, разрабатывается для версии *PC*, но также могут появится версии для *Android* и *iOS*.

Игра ориентирована на достаточно широкую аудиторию, не содержит ограничительного контента, кроме смерти главного героя или его противников. Минимальный возраст игрока –14 лет. Дополнительную привлекательность игра несет для заядлых любителей приятной стрельбы и не дорогих комплектующих *PC*. Понравится людям, которые хотят проверить свои навыки стрельбы и ведения динамического боя, любой ценой выполнять поставленные задачи и при этом правильно расходовать ресурсы. Понравится геймерам, которым нравится наблюдать за своим персонажем, за окружающей территорией.

Игра не использует торговые марки или другую собственность, подлежащую лицензированию.

4.1.2 Ключевые особенности игры (*USP*):

– противники ближнего боя разного типа;
– противники дальнего боя разного типа;
– лутинг;
– атмосферный сундтрэк;
– приятная стрельба из оружия разного типа;
– уровни, задания, миссии;
– возможность смены плеча;
– переключение между видом от третьего и первого лица.

4.1.3 Описание игры.

Основная задача игрока – пройти все уровни при этом на каждом из них выполнить поставленные задачи и цели. В ходе выполнения задач игроку будут мешать противники дальнего и ближнего боя. Во время сражения для уничтожения противника геймеру придется уничтожить противников, чтобы облегчить поставленную задачу. У каждого юнита, как и у самого игрока, есть полоса здоровья. Для уничтожения противника нужно наносить урон при этом полосу здоровья нужно опустить до нуля. Для того, чтобы наносить урон, игроку предоставляется огнестрельное оружие или возможность атаковать в ближнем бою. Во время стрельбы расходуются патроны. Когда количество патронов в обойме достигает нуля игроку нужно перезарядится. Время перезарядки может обнулится, если игрок начнет двигаться или атаковать в ближнем бою. Общее

количество здоровья и патронов можно пополнить. На карте игрок может найти ящики: зеленые пополняют здоровье, желтые – патроны. Такие ящики также выпадают из поверженных противников. По ходу сражения игрока будет сопроводить атмосферный саундтрэк.

4.1.4 Сравнение и особенности позиционирования.

Игра «*Defender Attack*» имеет, с одной стороны, некоторые оригинальные решения в жанре, но в то же время концепция игры использует следующие лучшие свойства выбранных образцов:

– «*Warframe*» – возможность игрока к ускоренному бегу;

– «*Lost Planet*» – переключение между видом от первого и от третьего лица;

– «*Ghost Rakon: BreakPoint*» – собирательство и луттинг;

– «*Red Dead Redemption 2*» – атмосферный саундтрэк.

4.1.5 Платформа

Разработка игры планируется, в основном, для *PC Windows*. В дальнейшем, возможно, для *Android* и *iOS*.

Требуемая и минимальная настройка конфигурации портативного компьютера представлена в таблице 3.1.

Таблица 4.1 – Требуемая и минимальная конфигурация компьютера

Требования	Минимальные	Рекомендуемые
Операционная система	*Windows* 10	
Процессор	*Pentium 4 HT*	*Intel Core 2 DUO*
ОЗУ	512*MB*	1*GB*
Видео карта	*GeForce 6600*	*GeForce 8600*
Звуковая карта	Совместимая с *DirectX*	
Версия *DirectX*	9.0	10
Периферийные устройства	Клавиатура, Мышь	

4.2 Классы и (или) скрипты игры «*DefenderAttack*»

В этом подразделе будут описаны классы и скрипты игрового приложения. Под скриптами будут подразумеваться классы, которые наследуются от *MonoBehaviour*.

Рассмотрим каждый из них.

OrbitCamera – скрипт, отвечающий за такое положение камеры вокруг объекта, которое сопоставимо с видом от третьего лица. Считывает изменение положения мыши.

ChangingTheShoulder – скрипт, отвечающий за смену плеча (левое, правое) по нажатию определенной клавиши. Имеет метод *ChangeShoulder*, который изменяет текущий статус плеча.

EyesScript – скрипт, реализующий логику глаз вражеского юнита.

VisabilityScript – скрипт, отвечающий за зону видимости вражеского юнита. Имеет методы *SetFarFindPlayer* и *SetCloseFindPlayer*, где первый активирует состояние юнита на большой дистанции, а второй на ближней.

EnemyReaction – абстрактный класс, который задает конкретный функционал реакции вражеского юнита на окружающую среду. Задает методы:

– *Reaction* – здесь задается то, как будет действовать юнит, в зависимости от текущего статуса.

– *OnStart* – инициализатор.

Также имеет метод *SetStatus*, который меняет текущий статус вражеского юнита.

AFKReaction – класс, который наследуется от *EnemyReaction*. Отвечает за действия вражеского юнита в состоянии спокойствия.

PatrolingReaction – класс, который наследуется от *EnemyReaction*. Отвечает за действия вражеского юнита в состоянии патрулирования.

RunReaction – класс, который наследуется от *EnemyReaction*. Отвечает за действия вражеского юнита в состоянии преследования. Имеет метод Sprint, который изменяет скорость юнита.

WalkingReaction – класс, который наследуется от *EnemyReaction*. Отвечает за действия вражеского юнита в состоянии прогулки (перемещение в определенную точку). Имеет метод *RotateToDefault*, который поворачивает юнита в исходное положение, если его состояние, по умолчанию, спокойствие.

EnemyDynamic – скрипт, который динамически оперирует состояниями юнита в зависимости от видимости игрока.

EnemyAgent – скрипт, который упрощает управление юнитом как ИИ.

MutantAnimation – скрипт, который оперирует анимациями мутанта в зависимости от текущего состояния юнита.

MutantHealth – скрипт, который отвечает за состояние здоровья юнита. Имеет метод *Died*, который меняет статус юнита на «мертв», *DropSuply*, который «рандомно» выкидывает ресурсы когда статус юнита «мертв».

ShootSensability – скрипт, который отвечает за количество времени, на протяжении которого юнит реагирует на выстрелы игрока. Имеет метод *SetHitTime*, который задает время (во время попадания по юниту).

Swipe – скрипт, который реагирует на соприкосновение острой конечности юнита с игроком. Имеет метод *OnTriggerEnter*, который наносит урон игроку при соприкосновении.

DropSuply – класс, который отвечает за создание объектов: ящиков здоровья, патронов. Содержит метод *AmmoDrop*, который создает объект «ящик с патронами», *MedDrop*, который создает объект «ящик со здоровьем».

SuplyScript – абстрактный класс, содержащий логику для ящиков снаряжения. Содержит метод *OnInteract*, который вызывается при сборе ящика.

AmmoScrpt – скрипт, который представляет ящик с патронами. При вызове метода *OnInteract* добавляет определенное количество патронов игроку.

MedScript – скрипт, который представляет ящик со здоровьем. При вызове метода *OnInteract* добавляет определенное количество здоровья игроку.

AmmoScript – скрипт, представляющий ящик с патронами. Содержит метод.

EnemyUI – скрипт, который в зависимости от состояния здоровья юнита, заполняет полосу здоровья над ним.

PlayerControlManager – скрипт, который считывает ввод клавиш *W, A, S, D*. Имеет методы *GetVerticalInput* и *GetHorizontalInput*. Где первый – возвращает вектор изменения нажатия клавиш *A, D*, второй – возвращает вектор изменения нажатия клавиш *W, S*.

PlayerControlScript – скрипт, который отвечает за передвижение игрока в зависимости от поворота камеры.

PlayerControl – скрипт, который упрощает управление вводом.

PlayerAttackModeControlScript – скрипт, отвечающий за передвижение игрока в режиме стрельбы.

LegHitScript – скрипт, который отвечает за удар в ближнем бою по нажатию определенной клавиши.

LegAttack – скрипт, который отвечает за нанесение урона при ударе в ближнем бою, содержит метод *OnTriggerEnter*, который при соприкосновении с вражеским юнитом наносит определенный урон.

SprintScript – скрипт, который отвечает за ускорение персонажа игрока при нажатой определенной клавише.

AnimationControl – скрипт, который отвечает за анимацию персонажа игрока.

ReloadScript – скрипт, который отвечает за перезарядку персонажа игрока при нажатой определенной клавише.

ShootScript – скрипт, который реализует логику стрельбы при зажатой левой клавиши мыши.

GunControlPosition – скрипт, контролирующий положение оружия в зависимости от расположения рук.

Weapon – абстрактный класс, содержащий логику вывода информации о патронах оружия и его создания, инициализации.

G36A2Gun – класс, который наследуется от *Weapon*. Представляет оружие *G36A2*.

Clip – класс представляющий обойму.

ShootGun – абстрактный класс, представляющий логику оружия. Содержит метод *Reload*, который перезаряжает обойму, Shoot, который отнимает один патрон в обойме.

G36A2 – класс, унаследованный от *ShootGun*. Представляет оружие.

Unit – абстрактный класс, представляющий юнита. Имеет методы *AddHealth*, *DealDamage*. Первый – добавляет здоровье, второй – уменьшает здоровье.

LevelManager – скрипт, содержащий количество противников на карте.

MainMenuManager – отвечает за запуск уровней и выход из игры.

Managers – скрипт, содержащий ссылки на менеджеры.

PlayerManager – скрипт. Менеджер игрока.

UIManager – скрипт. Менеджер игрового меню.

Sphere – класс, представляющий энергетическую сферу.

4.3 Верификация игрового приложения

При запуске игры открывается главное меню. В главном меню можно увидеть персонажа, в роли которого будет отыгрывать игрок, а также услышит звуковое сопровождение. Здесь же можно нажать на кнопку «*Play*» или «*WorkSpace*». При нажатии на кнопку «*Play*» игрок начнет игру, «*WorkSpace*» игрок выйдет на рабочий стол.

На рисунке 3.1 продемонстрировано главное меню.

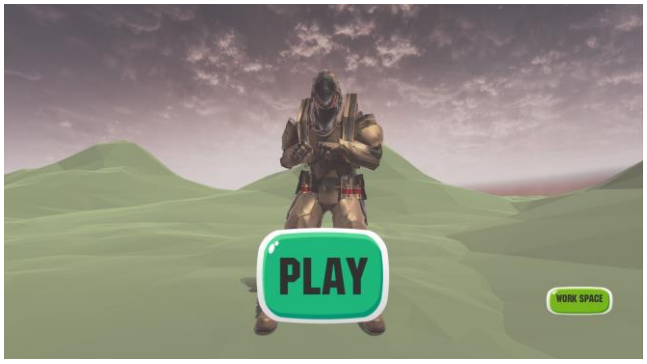

Рисунок 4.1 – Главное меню

Вначале игры игрок увидит своего персонажа на карте.

На уровне можно зайти в меню паузу.

Нажав на клавишу *Esc* игрок откроет меню-паузу – игра остановится. В меню-паузе есть кнопки:

– *Continue*;

– *Restart*;

– *Options*;

– *Quit*.

На рисунке 4.2 продемонстрировано меню-пауза.

Рисунок 4.2 – Меню-пауза

Выбрав кнопку *Continue* – игра продолжиться, *Restart* – уровень перезагрузится, *Quit* – вернет в главное меню. Также, нажав на кнопку Options игрок попадет в меню настроек, где есть возможность настроить чувствительность мыши по вертикали и горизонтали, громкость звука. Воспользоваться меню – паузой можно по ходу прохождения уровня (в любой момент).

На рисунке 4.3 продемонстрировано меню настроек.

Рисунок 4.3 – Меню настроек

После нажатия на кнопку *Exit* игрок вернется в меню-паузу.

Следить за состоянием здоровья и количеством патронов игроку помогает *HUD*.

Полоса здоровья показана на рисунке 4.4.

Рисунок 4.4 – Полоса здоровья

На рисунке 4.5 продемонстрирован текст, отображающий общее количество патронов, количество патронов в обойме.

Рисунок 4.5 – Отображение количества патронов

На протяжении миссии игрок должен правильно распоряжаться своими ресурсами, а также стараться не промахиваться, стреляя по противникам.

На рисунке 4.6 продемонстрирован игрок с видом от третьего лица.

Рисунок 4.6 – Персонаж игрока

Противники патрулируют территорию охраняя определенные точки на карте. При замечании игрока начинают атаковать.

На рисунке 4.7 продемонстрирован противник ближнего боя.

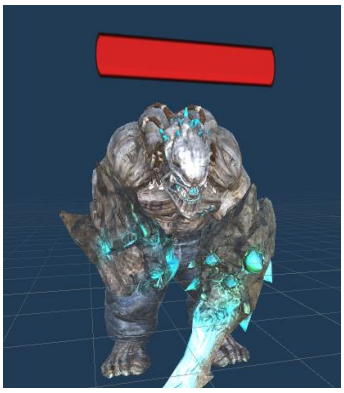

Рисунок 4.7 – Противник ближнего боя.

Для сражения с противниками игроку полезно собирать ящики, позволяющие восполнить здоровье и патроны.

На рисунке 3.8 показан ящик со здоровьем.

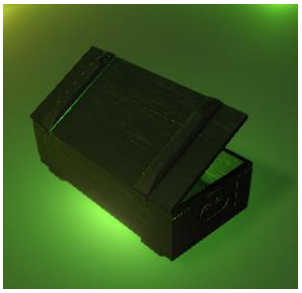

Рисунок 4.8 – Ящик со здоровьем

На рисунке 4.9 показан ящик с патронами.

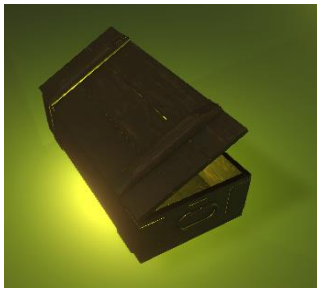

Рисунок 4.9 – Ящик с патронами

Ящики с патронами и здоровьем помогают игроку сражаться с противниками, упрощают путь к цели.

Чтобы собирать такие ящики придется соприкоснуться с ними.

ЗАКЛЮЧЕНИЕ

В ходе практики были достигнуты поставленные цели, включающие изучение передовой технологии предприятий и их совершенствования, знакомство со структурой предприятия, уровнем автоматизации, производственными процессами и системами их автоматизации, а также создание мотивационных ориентиров для будущей профессиональной деятельности.

В рамках анализа деятельности предприятия были изучены основные процессы, системы и средства автоматизации, а также методы разработки и использования программных продуктов и современных технологий в производственных условиях. Был проведен анализ обоснованности и эффективности использования данных технологий, что позволило сформулировать предложения по их улучшению. Также были ознакомлены с современными программными разработками и изучены техническая и программная документация применяемых информационных систем. В рамках практики был проведен анализ организации труда и обеспечения безопасности на предприятии, включая техническую безопасность, охрану труда, пожарную и экологическую безопасность.

Задачи практики успешно выполнены, что позволило ознакомиться с технологией производства элементов, методами сборки, наладки и контроля узлов и устройств, а также приобрести практические навыки работы с техническим оборудованием и измерительной контрольной аппаратурой. Было изучено назначение и структура автоматизированной системы обработки информации (АСОИ), а также основные цели ее создания и функции, которые она выполняет.

Процесс выполнения поставленных задач на предприятии позволил глубже понять современные технологии и процессы в сфере автоматизации предприятий. Полученные знания и практические навыки будут полезны в будущей профессиональной деятельности и способствуют развитию в области автоматизации и информационных систем.

При выполнении индивидуальной задачи был проведён аналитический обзор научной литературы и интернет-источников в области компьютерной графики и информационных технологий. Отобраны максимально удачные способы решения поставленной задачи, использованы новые идеи для реализации различных геймплей-механик. Отобрано программное обеспечение для разработки графических ресурсов, материалов. Реализованы механики, которые характеризуют игры в жанре шутеров от третьего лица.

СПИСОК ИПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *GamesBeat* [Электронный ресурс]. – 2023. – Режим доступа: *https://venturebeat.com/2019/03/24/the-truth-about-hypercasual-games* – Дата доступа: 20.06.2023;

2. *Hocking, J. Unity In Action* : *Multiplatform Game Development in C# / J. Hocking* . – Питер, Москва, Екатеринбург, Вороне, Нижний Новгород, Ростов-на-Дону, Самара, Минск: 2-ое международное издание, 2019.-344;

3. Приемы объектно-ориентированного программирования. Паттерны проектирования / Э. Гамма – СПб.: Питер, 2015.-368; 5. *CLR via C#*. Программирование на платформе *Microsoft .NET Framework 4*;

4. На языке *C#.* / Дж. Рихтер. – СПб: Питер, 2019.-896;

5. Приемы объектно-ориентированного программирования. Паттерны проектирования / Э. Гамма – СПб.: Питер, 2015.-111;

6. *Microsoft Docs* [Электронный ресурс]. – 2023. – Режим доступа: *https://docs.microsoft.com/ru-ru/nuget/what-is-nuget* – Дата доступа: 20.06.2023;

7. *Wikipedia* [Электронный ресурс]. – 2023. Режим доступа: *https:// ru.wikipedia.org/wiki/Saints_Row_IV* – Дата доступа: 20.06.2023;

8. *Wikipedia* [Электронный ресурс]. – 2023. Режим доступа: *https:// ru.wikipedia.org/wiki/Warframe* – Дата доступа: 21.06.2023;

9. *Wikipedia* [Электронный ресурс]. – 2023. Режим доступа: *https:// ru.wikipedia.org/wiki/Red_Dead_Redemption_2* – Дата доступа: 12.12.2022;

10. *Wikipedia* [Электронный ресурс]. – 2023. Режим доступа: *https:// en.wikipedia.org/wiki/Lost_Planet* – Дата доступа: 22.06.2023.

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг программы

OrbitCamera.cs:

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

public class OrbitCamera : MonoBehaviour
{
   Camera _camera;
   [Header("")]
   [SerializeField] private Transform target;
   [Header("")]
   public float sensetivityX;
   [Header("")]
   public float sensetivityY;
   [Header("")]
   public float maxRotateX;
   [Header("")]
   public float minRotateX;
   [Header("")]
   public float TranslateSpeed;
   private float _rotY;
   private float _rotX;
   private float _rotationX;
   private float _rotationY;

   private Vector3 _offset;
   // Start is called before the first frame update
   void Start()
   {
      _camera = GetComponent<Camera>();
      float startRotateX = 0;
      startRotateX = Mathf.Clamp(startRotateX, minRotateX - 5, maxRotateX - 5);
      transform.rotation = Quaternion.Euler(startRotateX, transform.localEulerAngles.y,
transform.localEulerAngles.z);
      _offset = target.position - transform.position;
      Cursor.lockState = CursorLockMode.Locked;
      Cursor.visible = false;
      Managers.CanvasManager.OXSens.maxValue = sensetivityX + 200;
      Managers.CanvasManager.OYSens.maxValue = sensetivityY + 200;
      Managers.CanvasManager.OXSens.minValue = 100;
      Managers.CanvasManager.OYSens.minValue = 100;
   }
   private void LateUpdate()
   {
      _rotX += Input.GetAxis("Mouse X") * /*sensetivityX*/ Managers.CanvasManager.OXSens.value *
Time.deltaTime;
      _rotY += Input.GetAxis("Mouse Y") * /*sensetivityY*/ Managers.CanvasManager.OYSens.value *
Time.deltaTime;
      _rotationY = _rotX;
```

```
        _rotationX = -_rotY;
        _rotationX = Mathf.Clamp(_rotationX, minRotateX, maxRotateX);
        Quaternion rotation = Quaternion.Euler(_rotationX, _rotationY, 0);

        transform.position = Vector3.Slerp(transform.position, target.position - rotation * _offset,
Time.deltaTime * TranslateSpeed);
        transform.LookAt(target.position);
    }

    void OnGUI()
    {
        int size = 12;
        float posX = _camera.pixelWidth / 2 - size / 4;
        float posY = _camera.pixelHeight / 2 - size / 2;
        GUI.Label(new Rect(posX, posY, size, size), "*");
    }
}
```

ChangingTheShoulder.cs:

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

public class ChangingTheShoulder : MonoBehaviour
{
    private enum Shoulder
    {
        Right,
        Left,
    }

    Shoulder _currentShoulder;
    Vector3 pos;
    Vector3 opositeXVector;

    void Start()
    {
        pos = transform.localPosition;

        opositeXVector = new Vector3(-transform.localPosition.x,
            transform.localPosition.y, transform.localPosition.z);

        _currentShoulder = Shoulder.Right;
    }

    void Update()
    {
        if(Input.GetKeyDown(KeyCode.X))
        {

            if (_currentShoulder == Shoulder.Right)
            {
                transform.localPosition = opositeXVector;
            }
            else
```

```
            {
                transform.localPosition = pos;
            }
            _currentShoulder = ChangeShoulder(_currentShoulder);
        }


    }

    private Shoulder ChangeShoulder(Shoulder shoulder)
    {
        return (shoulder == Shoulder.Left) ? Shoulder.Right : Shoulder.Left;
    }
}
```

EyesScript.cs

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

public class EyesScript : MonoBehaviour
{
    private Transform TargetToLook;
    public float Distance;
    public Collider HitCollider { get; private set; }
    public LayerMask LayerMask;
    int mask;
    // Start is called before the first frame update
    void Start()
    {
        TargetToLook = Managers.PlayerManager.PublicTarget;
        mask = 1 << LayerMask.NameToLayer("Default");
    }

    // Update is called once per frame
    void Update()
    {
        Vector3 direction = TargetToLook.position - transform.position;
        Ray ray = new Ray(transform.position, direction);
        RaycastHit hit;
        if (Physics.Raycast(ray, out hit, Distance, LayerMask.value, QueryTriggerInteraction.Ignore))
        {
            HitCollider = hit.collider;
        }
    }
}
```

VisabilityScript.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class VisabilityScript : MonoBehaviour
```

```
{
    /*[SerializeField]*/
    private float _visabilityDistance;
    [SerializeField] private float VisabilityCloseDistance;
    [SerializeField] private EyesScript _eyesScript;
    [SerializeField] private float VisabilityZone;
    public Vector3 Direction { get; private set; }
    public bool IsFarFindPlayer { get; private set; }
    public bool IsCloseFindPlayer { get; private set; }
    public float dist;
    // Start is called before the first frame update
    void Start()
    {
        _visabilityDistance = _eyesScript.Distance;
    }

    // Update is called once per frame
    void Update()
    {
        float distance = Vector3.Distance(transform.position,
Managers.PlayerManager.PlayerTransform.position);
        dist = distance;
        Direction = Managers.PlayerManager.PlayerTransform.position - transform.position;
        if (distance < _visabilityDistance && distance > VisabilityCloseDistance)
        {
            if (_eyesScript.HitCollider?.gameObject.GetComponent<CharacterController>() != null)
            {

                if (Vector3.Dot(transform.forward, Direction.normalized) > VisabilityZone)
                {
                    IsFarFindPlayer = true;
                }
                else IsFarFindPlayer = false;
            }
            else IsFarFindPlayer = false;

        }
        else IsFarFindPlayer = false;
        if (distance < VisabilityCloseDistance)
        {
            IsCloseFindPlayer = true;
            IsFarFindPlayer = false;
        }
        else IsCloseFindPlayer = false;
        if (IsFarFindPlayer)
        {
            Debug.DrawRay(transform.position, Direction, Color.magenta);
        }
        else Debug.DrawRay(transform.position, Direction, Color.white);
    }
    public void SetFarFindPlayer()
    {
        IsFarFindPlayer = true;
    }
    public void SetCloseFindPlayer()
    {
        IsCloseFindPlayer = true;
```

```
    }
}
```

EnemyReaction.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;
using CodeLibrary.Enums;

    [RequireComponent(typeof(EnemyAgent))]
    public abstract class EnemyReaction : MonoBehaviour
    {
        private EnemyStatus _currentStatus;
        public EnemyAgent NavAgent { get; protected set; }
        public abstract EnemyStatus Status { get; }
        public void Start()
        {
            NavAgent = GetComponent<EnemyAgent>();
            OnStart();
        }
        public void LateUpdate()
        {
            if(NavAgent.CurrentStatus == Status)
            {
                Reaction();
            }
        NavAgent.LastStatus = NavAgent.CurrentStatus;
        }

        public void SetStatus(EnemyStatus enemyStatus) { _currentStatus = enemyStatus; }

        public abstract void Reaction();
        public abstract void OnStart();
    }
```

AFKReaction .cs:

```
using CodeLibrary.Enums;
using System;
using UnityEngine;

public class AFKReaction : EnemyReaction
{
    public override EnemyStatus Status => EnemyStatus.AFK;
    public Vector3 AFK { get; private set; }
    public override void OnStart()
    {
        AFK = transform.position;
    }

    public override void Reaction()
    {
```

```
        NavAgent.Agent.SetDestination(AFK);
    }
}
```

PatrolingReaction .cs:

```
using CodeLibrary.Enums;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;


    public class PatrolingReaction : EnemyReaction
    {
        private int _currentGoal;

        public float DistanceToChangeGoal;
        public float PatrolingSpeed;

        public override EnemyStatus Status => EnemyStatus.Patroling;

        public override void OnStart()
        {
            _currentGoal = 0;

        }

        public override void Reaction()
        {
            NavAgent.Agent.speed = PatrolingSpeed;
            NavAgent.Agent.destination = NavAgent.TranformedGoals[_currentGoal];

            if(NavAgent.Agent.remainingDistance < DistanceToChangeGoal)
            {
                _currentGoal++;
                if (_currentGoal == NavAgent.Goals.Count)
                    _currentGoal = 0;
            }
        }
    }
```

RunReaction.cs:

```
using CodeLibrary.Enums;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

public class RunReaction : EnemyReaction
{
```

```
    private float currentSpeed;
    public float DefaultSpeed { get; private set; }
    [SerializeField] private float SprintSpeed;

    public override EnemyStatus Status => EnemyStatus.Runing;

    public override void OnStart()
    {
        DefaultSpeed = NavAgent.Agent.speed;
        currentSpeed = DefaultSpeed;
    }

    public override void Reaction()
    {
        NavAgent.Agent.destination = Managers.PlayerManager.PlayerTransform.position;
        NavAgent.Agent.speed = currentSpeed;
        StartCoroutine(Sprint());
    }

    private IEnumerator<WaitForSeconds> Sprint()
    {
        currentSpeed = SprintSpeed;
        yield return new WaitForSeconds(0.5f);
        NavAgent.Agent.speed = DefaultSpeed;
    }
}
```

AttackReaction.cs:

```
using CodeLibrary.Enums;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

[RequireComponent(typeof(EnemyAgent))]
[RequireComponent(typeof(VisabilityScript))]
public class AttackReaction : EnemyReaction
{
    VisabilityScript _visabilityScript;
    public float Damage;
    public float DamageDelayTime;
    public float _attackTime;
    [SerializeField] public bool IsAttack; /*{ get; private set; }*/
    public override EnemyStatus Status => EnemyStatus.Attacking;

    public override void OnStart()
    {
        _visabilityScript = GetComponent<VisabilityScript>();
    }

    public override void Reaction()
    {
        IsAttack = false;
```

```
        _attackTime += Time.deltaTime;

        Quaternion lookRot = Quaternion.LookRotation(_visabilityScript.Direction);
        transform.rotation = Quaternion.Lerp(transform.rotation, lookRot, Time.deltaTime * 5f);
        NavAgent.Agent.speed = 0f;
        if(_attackTime >= DamageDelayTime)
        {
            _attackTime = 0;
            IsAttack = true;
        }
    }
}
```

WalkingReaction.cs:

```
using CodeLibrary.Enums;
using System;
using System.Collections.Generic;
using System.Collections;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

public class WalkingReaction : EnemyReaction
{
    public override EnemyStatus Status => EnemyStatus.Walking;
    private Quaternion StartRotation;
    public float RotateToDefaultSpeed;

    public override void OnStart()
    {
        StartRotation = transform.rotation;
        //StartRotation = new Vector3(transform.rotation.x, transform.rotation.y, transform.rotation.z);

    }

    public override void Reaction()
    {
        if(NavAgent.Agent.remainingDistance < 0.5f)
        {
            StartCoroutine(RotateToDefault());
        }

    }
    IEnumerator RotateToDefault()
    {
        yield return new WaitForSeconds(3f);
        transform.rotation = Quaternion.Lerp(transform.rotation, StartRotation, Time.deltaTime *
RotateToDefaultSpeed);
    }
}
```

EnemyDynamic.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;
using CodeLibrary.Enums;
[RequireComponent(typeof(AttackReaction))]
[RequireComponent(typeof(RunReaction))]
[RequireComponent(typeof(ShootSensability))]
[RequireComponent(typeof(EnemyAgent))]
[RequireComponent(typeof(AFKReaction))]
public class EnemyDynamic : MonoBehaviour
{
   EnemyAgent _enemyAgent;
   VisabilityScript _visabilityScript;
   AFKReaction _afkReaction;
   RunReaction _runReaction;
   ShootSensability _shootSensability;
   AttackReaction _attackReaction;
   public void Start()
   {
      _enemyAgent = GetComponent<EnemyAgent>();
      _visabilityScript = GetComponent<VisabilityScript>();
      _afkReaction = GetComponent<AFKReaction>();
      _runReaction = GetComponent<RunReaction>();
      _shootSensability = GetComponent<ShootSensability>();
      _attackReaction = GetComponent<AttackReaction>();
   }
   private void Update()
   {
      _shootSensability.Update();
      if (_enemyAgent.CurrentStatus != EnemyStatus.Died && !Managers.PlayerManager.IsDead)
      {
         switch (_visabilityScript.IsFarFindPlayer)
         {
            case true:
               {
                  _enemyAgent.SetStatus(EnemyStatus.Runing);
                  break;
               }
            case false:
               {
                  switch (_visabilityScript.IsCloseFindPlayer)
                  {
                     case true:
                        {
                           _enemyAgent.SetStatus(EnemyStatus.Attacking);
                           break;
                        }
                     case false:
                        {
                           if (_enemyAgent.Agent.remainingDistance > .3f && _enemyAgent.DefltStatus ==
EnemyStatus.AFK)
                           {
```

```
                        _enemyAgent.SetStatus(EnemyStatus.Walking);
                        _enemyAgent.Agent.SetDestination(_afkReaction.AFK);
                    }
                    else
                    {
                        _enemyAgent.SetStatus(_enemyAgent.DefltStatus);
                    }

                    break;
                }
            }
            break;

        }

    }
    if(_enemyAgent.CurrentStatus != EnemyStatus.Attacking)
    {
        _attackReaction._attackTime = _attackReaction.DamageDelayTime;
    }
  }
  if(Managers.PlayerManager.IsDead)
  {
    _enemyAgent.SetStatus(EnemyStatus.Dance);
  }


 }
}
```

EnemyAgent.cs:

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CodeLibrary.CodeUnit;
using UnityEngine;
using UnityEngine.AI;
using CodeLibrary.Enums;

[RequireComponent(typeof(NavMeshAgent))]
public class EnemyAgent : MonoBehaviour
{
   [SerializeField] public EnemyStatus CurrentStatus { get; private set; }
   [SerializeField] private EnemyStatus DefaultStatus;
   public EnemyStatus LastStatus;
   public EnemyStatus CurSt;
   public EnemyStatus DefltStatus
   {
      get { return DefaultStatus; }
      set { DefaultStatus = value; }
   }
```

```
    [SerializeField] public List<Transform> Goals;
    public List<Vector3> TranformedGoals { get; private set; }

    public NavMeshAgent Agent { get; private set; }
    public Unit Unit { get; private set; }

    public void Awake()
    {
        Agent = GetComponent<NavMeshAgent>();
    }

    private void Start()
    {
        CurrentStatus = DefaultStatus;
        TranformedGoals = new List<Vector3>();
        TransformGoals();
    }
    void TransformGoals()
    {
        foreach (Transform transform in Goals)
        {
            TranformedGoals.Add(transform.TransformPoint(transform.position));
        }
    }
    private void Update()
    {
        CurSt = CurrentStatus;
    }
    public void SetStatus(EnemyStatus enemyStatus) { CurrentStatus = enemyStatus; }
}
```

MutantAnimation.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using CodeLibrary.Enums;
[RequireComponent(typeof(Animator))]
[RequireComponent(typeof(EnemyAgent))]
[RequireComponent(typeof(MutantHealth))]
[RequireComponent(typeof(MutantAnimation))]

public class MutantAnimation : MonoBehaviour
{

    EnemyAgent _agent;
    MutantHealth _mutantHealth;
    AttackReaction _attackReaction;
    EnemyStatus Current;
    Animator _animator;
    public EnemyStatus EnemyStatus;
    // Start is called before the first frame update
    void Start()
    {
        _agent = GetComponent<EnemyAgent>();
        _animator = GetComponent<Animator>();
```

```
        _mutantHealth = GetComponent<MutantHealth>();
        _attackReaction = GetComponent<AttackReaction>();
    }
    float deadTime;
    // Update is called once per frame
    void Update()
    {
        //Current = _agent.CurrentStatus;
        EnemyStatus = _agent.CurrentStatus;
        if (_agent.CurrentStatus == EnemyStatus.Died && deadTime + _mutantHealth.DyingTime <
Time.time)
        {
            deadTime = Time.time;
            _animator.SetTrigger("DyingTrigger");
        }
        else
        {
            _animator.SetBool("IsDying", false);
        }
        if (_agent.CurrentStatus == EnemyStatus.Runing)
        {
            _animator.SetBool("IsRuning", true);
        }
        else
        {
            _animator.SetBool("IsRuning", false);
        }

        if (_agent.CurrentStatus == EnemyStatus.Patroling || _agent.CurrentStatus == EnemyStatus.Walking)
        {
            _animator.SetBool("IsPatroling", true);
        }
        else
        {
            _animator.SetBool("IsPatroling", false);
        }
        if (_agent.CurrentStatus == EnemyStatus.Attacking)
        {

            if(_attackReaction.IsAttack)
            {
                _animator.SetTrigger("AttackTrigger");
            }

        }
        if(_agent.CurrentStatus == EnemyStatus.Dance)
        {
            _animator.SetTrigger("DanceTrigger");
        }
    }
}
```

MutantHealth.cs:

```
using System.Collections;
```

```
using System.Collections.Generic;
using UnityEngine;
using CodeLibrary.CodeUnit;
using CodeLibrary.Enums;

[RequireComponent(typeof(DropSuply))]
[RequireComponent(typeof(EnemyAgent))]
public class MutantHealth : MonoBehaviour
{
    DropSuply _dropSuply;
    EnemyAgent _agent;
    public Unit Unit { get; private set; }
    public float HelthCount;
    public float DyingTime;
    public bool IsDead;
    public float DropUpOffset;
    float deadTime;
    public float CurHealth;
    // Start is called before the first frame update
    void Start()
    {
        Managers.LevelManager.AddtionMutantCount();
        Unit = new Unit(HelthCount, "Mutant");
        _agent = GetComponent<EnemyAgent>();
        _dropSuply = GetComponent<DropSuply>();
        IsDead = false;
    }

    // Update is called once per frame
    public void Update()
    {
        CurHealth = Unit.Health;
        if(Unit.Health <= 0 && !IsDead)
        {
            Managers.LevelManager.SubstractMutantCount();
            Died();
        }

    }

    public void Died()
    {
        _agent.SetStatus(EnemyStatus.Died);
        _agent.Agent.speed = 0;

        StartCoroutine(StartDied(DyingTime));
    }
    IEnumerator StartDied(float time)
    {
        yield return new WaitForSeconds(time);

        DropSuply();

        Destroy(gameObject);
    }

    void DropSuply()
```

```
    {
        Vector3 vector = new Vector3(0, DropUpOffset, 0);
        int dropType = Random.Range(1, 4);
        switch (dropType)
        {
            case 1:
                {
                    float medORammo = Random.Range(1, 3);
                    if (medORammo == 1)
                    {
                        _dropSuply.AmmoDrop(transform.position, vector);
                    }
                    else
                    {
                        _dropSuply.MedDrop(transform.position, vector);
                    }
                    break;
                }
            case 2:
                {
                    float medORammo = Random.Range(1, 3);
                    if (medORammo == 1)
                    {
                        _dropSuply.AmmoDrop(transform.position, vector);
                        _dropSuply.AmmoDrop(transform.position, vector);
                    }
                    else
                    {
                        _dropSuply.MedDrop(transform.position, vector);
                        _dropSuply.AmmoDrop(transform.position, vector);
                    }
                    break;
                }
            case 3:
                {
                    _dropSuply.AmmoDrop(transform.position, vector);
                    break;
                }

        }
    }


}
```

ShootSensability.cs:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

public class ShootSensability : MonoBehaviour
```

```
{
   private float _hitTime;
   private float _currentVisibleTime;
   public bool IsShootSensability;
   public float VisibleTime;
   VisabilityScript _visabilityScript;
  // public float TimeV;
   public void Start()
   {
      _currentVisibleTime = -1;
      _visabilityScript = GetComponent<VisabilityScript>();
      StartCoroutine(SetVisible());
   }
   IEnumerator SetVisible()
   {
      yield return new WaitForSeconds(VisibleTime);
      _currentVisibleTime = VisibleTime;
   }
   public void Update()
   {
      IsShootSensability = false;
      //TimeV = _currentVisibleTime;
      if(_hitTime + _currentVisibleTime > Time.time && !_visabilityScript.IsCloseFindPlayer)
      {
         IsShootSensability = true;
         _visabilityScript.SetFarFindPlayer();
      }
   }

   public void SetHitTime(float time)
   {
      _hitTime = time;
   }
}
```

Swipe.cs:

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

public class Swipe : MonoBehaviour
{
   public float SwipeDamage;
   [SerializeField] AttackReaction _attackReaction;
   private void OnTriggerEnter(Collider other)
   {
      if(other.gameObject.GetComponent<PlayerControlManager>() != null)
      {
            Managers.PlayerManager.player.DealDamage(SwipeDamage);
      }
   }
}
```

DropSuply.cs:

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

public class DropSuply : MonoBehaviour
{
   [SerializeField] GameObject AmmoPrefab;
   [SerializeField] GameObject MedPrefab;
   public void AmmoDrop(Vector3 pos)
   {
      Instantiate(AmmoPrefab, pos,
         Quaternion.Euler(transform.localEulerAngles.x, transform.localEulerAngles.y,
transform.localEulerAngles.z));
   }
   public void AmmoDrop(Vector3 pos, Vector3 offset)
   {
      GameObject ammoObj = Instantiate(AmmoPrefab, pos + offset, Random.rotation);
      ammoObj.GetComponent<Rigidbody>().AddForce(transform.up * 60f);
   }

   public void MedDrop(Vector3 pos, Vector3 offset)
   {
      GameObject medObj = Instantiate(MedPrefab, pos + offset, Random.rotation);

      medObj.GetComponent<Rigidbody>().AddForce(transform.up * 60f);
   }


}
```

SuplyScript.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

public abstract class SuplyScript : MonoBehaviour
{
   public float DistanceToInteract;
   public float dist;
   public void Update()
   {
      float distance = Vector3.Distance(Managers.PlayerManager.PlayerTransform.position,
transform.position);
      dist = distance;
      if (distance <= DistanceToInteract)
      {
         OnInteract();

         Destroy(gameObject);
```

```
        }
        OnUpdate();
    }
    public abstract void OnUpdate();
    public abstract void OnInteract();
}
```

AmmoScrpt.cs:

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

public class AmmoScript : SuplyScript
{
    public int BulletsToAdd;

    public override void OnInteract()
    {
        Managers.PlayerManager.UpdateAmmo(BulletsToAdd);
    }
}
```

MedScript.cs:

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

public class MedScript : SuplyScript
{
    public float HealthToAdd;
    public override void OnInteract()
    {
        Managers.PlayerManager.UpdateHealth(HealthToAdd);
    }
}
```

AmmoScript.cs :

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

public class AmmoScript : SuplyScript
{
    public int BulletsToAdd;

    public override void OnInteract()
    {
        Managers.PlayerManager.UpdateAmmo(BulletsToAdd);
    }
}
```

EnemyUI .cs:

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

public class EnemyUI : MonoBehaviour
{
    [SerializeField] MutantHealth MutantHealth;
    [SerializeField] Slider Slider;
    // Start is called before the first frame update
    void Start()
    {
        Slider.maxValue = MutantHealth.HelthCount;
        Slider.minValue = 0.5f;
    }

    // Update is called once per frame
    void Update()
    {
        MutantHealth.Update();
        Slider.value = MutantHealth.CurHealth;
        if(Slider.value <= Slider.minValue)
        {
            Slider.gameObject.SetActive(false);
        }
    }
}
```

PlayerControlManager.cs:

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

public class PlayerControlManager : MonoBehaviour
{
    private const float c_defaultPlayerSpeed = 2f;
    private float _horInput;
    private float _verInput;
    public float gravity = -9.8f;

    public float CurrentPlayerSpeed { get; set; }

    public Vector3 Movement
    {
        get
        {
            return _movement;
        }
    }
    private Vector3 _movement;
    // Start is called before the first frame update
    void Start()
```

```
    {
        CurrentPlayerSpeed = c_defaultPlayerSpeed;
    }

    // Update is called once per frame
    void Update()
    {
        _movement = Vector3.zero;
        _horInput = Input.GetAxis("Horizontal");
        _verInput = Input.GetAxis("Vertical");


        _movement.z = _verInput * CurrentPlayerSpeed;
        _movement.x = _horInput * CurrentPlayerSpeed;

        //_movement.y = gravity;

        _movement = Vector3.ClampMagnitude(_movement, CurrentPlayerSpeed);
        _movement.y = gravity;
    }

    public float GetHorizontalInput()
    {
        return _horInput;
    }
    public float GetVerticalInput()
    {
        return _verInput;
    }

}
```

PlayerControlScript.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
[RequireComponent(typeof(PlayerControlScript))]
[RequireComponent(typeof(LegHitScript))]
public class PlayerControlScript : PlayerControl
{
    PlayerAttackModeControlScript _playerAttackModeControlScript;
    LegHitScript _legHitScript;
    [SerializeField] ReloadScript _reloadScript;

    [Header("")]
    public float RotSpeed;

    public override void OnStart()
    {
        _playerAttackModeControlScript = GetComponent<PlayerAttackModeControlScript>();
        _legHitScript = GetComponent<LegHitScript>();
    }

    public override void OnUpdate()
    {
```

```
        if (_playerAttackModeControlScript.BoolAttackMode == false
            && _legHitScript.IsLegHit == false && _reloadScript.IsReload == false)
        {
            MoveWithOrbitCamera();
            ReplacePlayer();
        }
    }


    private void MoveWithOrbitCamera()
    {
        Quaternion tmp = CameraTransform.rotation;
        CameraTransform.eulerAngles = new Vector3(0, CameraTransform.eulerAngles.y, 0);
        Movement = CameraTransform.TransformDirection(Movement);
        CameraTransform.rotation = tmp;

        if ((HorizontalInput != 0 && VerticalInput > 0) || (HorizontalInput == 0 && VerticalInput > 0) ||
(HorizontalInput != 0 && VerticalInput == 0))
        {
            MoveDirectionRotation(Movement);
        }

        if (VerticalInput < 0)
        {
            OpositeMoveRotation(Movement);
        }
    }

    private void MoveDirectionRotation(Vector3 movement)
    {
        Vector3 movementXZ = new Vector3(movement.x, 0, movement.z);
        Quaternion direction = Quaternion.LookRotation(movementXZ);
        transform.rotation = Quaternion.Lerp(transform.rotation, direction, RotSpeed * Time.deltaTime);
    }

    private void OpositeMoveRotation(Vector3 movement)
    {
        Vector3 currentMovement = movement;
        Vector3 opMovement = new Vector3(-currentMovement.x, 0, -currentMovement.z);
        Quaternion direction = Quaternion.LookRotation(opMovement);
        transform.rotation = Quaternion.Lerp(transform.rotation, direction, RotSpeed * Time.deltaTime);
    }
}
```

PlayerControl.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
[RequireComponent(typeof(CharacterController))]
[RequireComponent(typeof(PlayerControlManager))]
public abstract class PlayerControl : MonoBehaviour
{
    [SerializeField] protected Transform CameraTransform;
    // Start is called before the first frame update
    protected CharacterController CharacterController;
```

```
    private PlayerControlManager _controlManager;
    protected Vector3 Movement;
    protected float Gravity;
    public float VerticalInput { get; private set; }
    public float HorizontalInput { get; private set; }

    public abstract void OnStart();
    public abstract void OnUpdate();

    public void Start()
    {
        CharacterController = GetComponent<CharacterController>();
        _controlManager = GetComponent<PlayerControlManager>();
        OnStart();
    }

    public void Update()
    {
        if(PlayerManager.CurHelth > 0)
        {
            Movement = _controlManager.Movement;
            VerticalInput = _controlManager.GetVerticalInput();
            HorizontalInput = _controlManager.GetHorizontalInput();
            Gravity = _controlManager.gravity;
            OnUpdate();
        }
        ;

    }
    public void SetZeroMovement()
    {
        Movement = Vector3.zero;
    }
    public void SetSpeed(float speed)
    {
        _controlManager.CurrentPlayerSpeed = speed;
    }
    protected void ReplacePlayer()
    {
        Movement *= Time.deltaTime;
        CharacterController.Move(Movement);

    }

}
```

PlayerAttackModeControlScript.cs:

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

public class PlayerAttackModeControlScript : PlayerControl
{
    LegHitScript _legHitScript;
    [SerializeField] ReloadScript _reloadScript;
```

```
[Header("")]
public float delayToStartMode;
private float _startClickTime;
[Header("")]
public float RotSpped;
private bool _attackMode;
public int IntAttackMode
{
   get
   {
      if (_attackMode)
         return 1;
      else return -1;
   }
}

public bool BoolAttackMode
{
   get
   {
      return _attackMode;
   }
   set
   {
      _attackMode = value;
   }
}

public override void OnStart()
{
   _legHitScript = GetComponent<LegHitScript>();
}

public override void OnUpdate()
{
   if(_legHitScript.IsLegHit == false && _reloadScript.IsReload == false)
   {
      if (Input.GetMouseButtonDown(0))
      {
         _startClickTime = Time.time;
      }
      if (_startClickTime + delayToStartMode <= Time.time)
      {
         if (Input.GetMouseButton(0))
         {
            _attackMode = true;
            Movement = new Vector3(Movement.x, 0, Movement.z);
            Movement = CameraTransform.TransformDirection(Movement);
            Movement = new Vector3(Movement.x, Gravity, Movement.z);
            MoveInAttackMode();
            ReplacePlayer();
         }
         else _attackMode = false;
      }
   }
}
private void MoveInAttackMode()
```

```
    {
        Vector3 eulerY = new Vector3(0, CameraTransform.eulerAngles.y, 0);
        transform.rotation = Quaternion.Lerp(transform.rotation, Quaternion.Euler(eulerY), Time.deltaTime *
RotSpped);
    }
}
```

LegHitScript.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
[RequireComponent(typeof(PlayerControlScript))]
public class LegHitScript : MonoBehaviour
{
    PlayerControlScript _playerControlScript;
    PlayerAttackModeControlScript _playerAttackModeControlScript;
    [SerializeField] ReloadScript ReloadScript;

    [SerializeField] GameObject GunObject;
    private bool _isLegHitClick;

    [Header("")]
    public float LegHitTime;

    private float _legHitClickTime;
    public bool IsLegHit { get; private set; }
    public bool IsLegHitClick
    {
        get
        {
            return _isLegHitClick;
        }
    }

    void Start()
    {
        _playerControlScript = GetComponent<PlayerControlScript>();
        _playerAttackModeControlScript = GetComponent<PlayerAttackModeControlScript>();
    }

    void Update()
    {

        _isLegHitClick = false;

        if (Input.GetKeyDown(KeyCode.F) && IsLegHit == false)
        {
            _isLegHitClick = true;
            _legHitClickTime = Time.time;
            StartCoroutine(GunInLegHitTime(LegHitTime));

        }

        if (_legHitClickTime + LegHitTime > Time.time)
        {
```

```
            IsLegHit = true;
            ReloadScript.IsStopReload = true;
        }
        else
        {
            ReloadScript.IsStopReload = false;
            IsLegHit = false;
        }
    }

    IEnumerator StayWhileLegHit(float legHitTime)
    {
        _playerControlScript.SetZeroMovement();
        _playerAttackModeControlScript.SetZeroMovement();
        yield return new WaitForSeconds(legHitTime);
        _isLegHitClick = false;

    }

    IEnumerator GunInLegHitTime(float time)
    {
        GunObject.SetActive(false);
        yield return new WaitForSeconds(time);
        GunObject.SetActive(true);
    }
}
```

LegAttack.cs:

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

public class LegAttack : MonoBehaviour
{
    public float LegDamage;
    private void OnTriggerEnter(Collider other)
    {
        MutantHealth mutantHealth = null;

        if ((mutantHealth = other.gameObject.GetComponent<MutantHealth>()) != null)
        {
            mutantHealth.Unit.DealDamage(LegDamage);
            Debug.Log("");
        }
    }

}
```

SprintScript.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
[RequireComponent(typeof(PlayerControlScript))]
```

```
[RequireComponent(typeof(PlayerAttackModeControlScript))]
public class SprintScript : MonoBehaviour
{
    PlayerControlScript _playerControlScript;
    PlayerAttackModeControlScript _playerAttackModeControlScript;
    [SerializeField] ReloadScript ReloadScript;
    [Header("")]
    public float MovementSpeed;
    [Header("")]
    public float SprintMovementSpeed;
    private float _currentSpeed;

    private bool _isRuning;

    public bool IsRuning
    {
        get
        {
            return _isRuning;
        }
    }
    // Start is called before the first frame update
    public bool Stop;
    void Start()
    {
        _playerControlScript = GetComponent<PlayerControlScript>();
        _playerAttackModeControlScript = GetComponent<PlayerAttackModeControlScript>();
    }

    // Update is called once per frame
    void Update()
    {
        Stop = ReloadScript.IsStopReload;
        if (Input.GetKey(KeyCode.LeftShift) && _playerControlScript.VerticalInput > 0 &&
!_playerAttackModeControlScript.BoolAttackMode /*&& !ReloadScript.IsReload*/)
        {
            _playerAttackModeControlScript.BoolAttackMode = false;
            _isRuning = true;
            _playerControlScript.SetSpeed(SprintMovementSpeed);
        }
        else
        {
            _isRuning = false;
            _playerControlScript.SetSpeed(MovementSpeed);
        }
        if(_isRuning)
        {
            ReloadScript.IsStopReload = true;
        }
        else ReloadScript.IsStopReload = false;
    }
}
```

AnimationControl.cs:

```
using System.Collections;
```

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

public class AnimationControl : PlayerControl
{
    Animator _animator;
    SprintScript _sprintScript;
    LegHitScript _legHitScript;
    PlayerAttackModeControlScript _playerAttackModeControlScript;

    [SerializeField] private ReloadScript _reloadScript;

    public override void OnStart()
    {
        _animator = GetComponent<Animator>();
        _sprintScript = GetComponent<SprintScript>();
        _legHitScript = GetComponent<LegHitScript>();
        _playerAttackModeControlScript = GetComponent<PlayerAttackModeControlScript>();
    }

    public override void OnUpdate()
    {
        Sprint();
        Move();
        Idle();
        LegHit();
    }

    private void Sprint()
    {
        if (_sprintScript.IsRuning)
        {
            _animator.SetBool("IsSpeed", true);
        }
        else
        {
            _animator.SetBool("IsSpeed", false);
        }
    }

    private void Move()
    {
        if (VerticalInput == 0 && !_playerAttackModeControlScript.BoolAttackMode)
        {
            if (HorizontalInput < 0)
            {
                _animator.SetFloat("FORWARD-BACK", -HorizontalInput);
                _animator.SetFloat("LEFT-RIGHT", 0);
            }
            else
            {
                _animator.SetFloat("FORWARD-BACK", HorizontalInput);
                _animator.SetFloat("LEFT-RIGHT", 0);
            }

        }
        else
```

```
        {
            _animator.SetFloat("FORWARD-BACK", VerticalInput);
            _animator.SetFloat("LEFT-RIGHT", HorizontalInput);
        }
    }

    private void Idle()
    {
        if (VerticalInput == 0 && HorizontalInput == 0 &&
_playerAttackModeControlScript.BoolAttackMode && !_reloadScript.IsReload)
        {
            _animator.SetBool("isAim", true);
        }
        else
        {
            _animator.SetBool("isAim", false);
        }

        if (_reloadScript.IsReloadClick && !_sprintScript.IsRuning)
        {
            _animator.SetTrigger("IsReloadingIdle");
        }


    }

    private void LegHit()
    {
        if (_legHitScript.IsLegHitClick)
        {
            _animator.SetTrigger("LegHitTrigger");
        }
    }

    public void Dying()
    {
        _animator.SetTrigger("DyingTrigger");
    }
}
```

ReloadScript.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
[RequireComponent(typeof(Weapon))]
public class ReloadScript : MonoBehaviour
{
    [SerializeField] KeyCode Key;
    [SerializeField] LegHitScript LegHitScript;
    [SerializeField] SprintScript SprintScript;
    [SerializeField] PlayerAttackModeControlScript AttackMode;
    private float tmpTime;
    public float ReloadTime;
    private float _currentReloadTime;
    private float _clickTime;
```

```
    public bool IsStopReload { get; set; }
    public bool IsReloadClick { get; private set; }
    public bool IsReload { get; private set; }

    public float time;
    public bool Click;
    public bool Is;
    public bool Stop;
    // Start is called before the first frame update
    Weapon weapon;
    void Start()
    {
        weapon = GetComponent<Weapon>();
        StartCoroutine(StartReload());
    }
    IEnumerator StartReload()
    {
        tmpTime = ReloadTime;
        ReloadTime = -1;
        yield return new WaitForSeconds(tmpTime);
        ReloadTime = tmpTime;
    }
    // Update is called once per frame
    void Update()
    {
        Stop = IsStopReload;
        time = _currentReloadTime;
        Click = IsReloadClick;
        Is = IsReload;
        IsReloadClick = false;

        if ((Input.GetKey(Key) || (Input.GetMouseButtonDown(0) && weapon.gun.CurrentBulletsInClip ==
0)) && IsReload == false)
        {
            if(weapon.gun.CanReloading())
            {
                _clickTime = Time.time;
                IsReloadClick = true;
                //weapon.gun.Reload();
            }
        }

        if(_clickTime + ReloadTime > Time.time && (LegHitScript.IsLegHit == false &&
SprintScript.IsRuning == false)/*IsStopReload == false*/)
        {
            IsReload = true;
        }
        else
        {
            IsReload = false;
        }

        if (IsReload)
        {
            _currentReloadTime += Time.deltaTime;
        }
        else _currentReloadTime = 0;
```

```
        if(_currentReloadTime >= ReloadTime)
        {
            weapon.gun.Reload();
        }


    }
}
```

ShootScript .cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Audio;
using CodeLibrary.Enums;

[RequireComponent(typeof(AudioSource))]
public class ShootScript : MonoBehaviour
{
    [Header("")]
    [SerializeField] AimScript aim;
    [SerializeField] Camera _camera;
    Weapon weapon;
    public float ShootDistance;
    public bool IsFire { get; private set; }
    [Header("")]
    public float delayBullets;

    private float _clickTime;

    [Header("
    [SerializeField] PlayerAttackModeControlScript _playerAttackModeControlScript;
    [Header("")]
    [SerializeField] SprintScript _sprintScript;

    public ParticleSystem shootParticle;
    public GameObject FlashLight;
    public Light lightComponent { get; private set; }

    ReloadScript _reloadScript;

    AudioSource audioSource;

    void Start()
    {
        weapon = GetComponent<Weapon>();
        _reloadScript = GetComponent<ReloadScript>();
        lightComponent = FlashLight.gameObject.GetComponent<Light>();
        lightComponent.enabled = false;
        audioSource = GetComponent<AudioSource>();
    }

    void Update()
```

```
    {
        if (Input.GetMouseButton(0) && _clickTime + delayBullets <= Time.time
        && !_sprintScript.IsRuning && _playerAttackModeControlScript.BoolAttackMode &&
_reloadScript.IsReload == false
        && weapon.gun.CurrentBulletsInClip > 0) //
        {
            _clickTime = Time.time;
            IsFire = true;
            weapon.gun.Shoot();
            lightComponent.enabled = true;
            shootParticle.Emit(3);

            RaycastHit raycastHit;

            Vector3 point = new Vector3(
            _camera.pixelWidth / 2, _camera.pixelHeight / 2, 0);
            audioSource.Play();
            audioSource.pitch = Random.Range(0.98f, 1.05f);
            Ray ray = _camera.ScreenPointToRay(point);

            if (Physics.Raycast(ray, out raycastHit, ShootDistance)) //
            {
                Collider HitCollider = raycastHit.collider;

                MutantHealth mutantHealth = null;

                ShootSensability shootSensability = null;

                if ((mutantHealth = HitCollider.gameObject.GetComponent<MutantHealth>()) != null)
                {
                    float damage = (weapon.gun.Damage / raycastHit.distance) * (ShootDistance / 2);
                    mutantHealth.Unit.DealDamage(damage);

                    shootSensability = HitCollider.gameObject.GetComponent<ShootSensability>();

                    shootSensability.SetHitTime(Time.time);
                    Debug.Log("");
                }
            }
        }
        else
        {
            IsFire = false;
            lightComponent.enabled = false;
        }
    }


}
```

GunControlPosition.cs:

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

```
public class GunControlPosition : MonoBehaviour
{
    [Header("")]
    [SerializeField] Transform RightHandMiddle1;
    [Header("Index1)")]
    [SerializeField] Transform Ladon;

    void Update()
    {
        transform.position = RightHandMiddle1.position;
        Vector3 relativePos = Ladon.position - RightHandMiddle1.position;
        Vector3 pos = new Vector3(0, relativePos.y, relativePos.z);
        Quaternion rotation = Quaternion.LookRotation(relativePos);
        transform.rotation = rotation;
    }
}
```

Weapon.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using CodeLibrary.CodeWeapon.Guns;
using CodeLibrary.CodeWeapon;
using CodeLibrary.Interfaces;

public abstract class Weapon : MonoBehaviour
{
    [Header("")]
    [SerializeField] protected Text CountText;
    [Header("")]
    [SerializeField] protected Text ClipText;
    public ShootGun gun { get; protected set; }

    private void Update()
    {
        CountText.text = $"{gun.CurrentBulletsCount}";

        ClipText.text = $"{gun.CurrentBulletsInClip}";
        OnUpdate();
    }

    public abstract void OnUpdate();

}
```

G36A2Gun.cs:

```
using CodeLibrary.Interfaces;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using CodeLibrary.CodeWeapon.Guns;
```

```
public class G36A2Gun : Weapon
{
   private const int c_clipsCount = 8;
   private const int c_bulletInClip = 30;

   void Start()
   {
      gun = new G36A2(c_clipsCount, c_bulletInClip);
   }


}
```

Clip.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeLibrary.CodeWeapon
{
   // класс, который является объектом обоймы
   public class Clip
   {
      private int BulletsCount { get;}
      public int CurrentBullets { get; set; }
      public Clip(int bulletsCount)
      {
         BulletsCount = bulletsCount;
         CurrentBullets = BulletsCount;
      }

      public bool IsFull()
      {
         return CurrentBullets == BulletsCount;
      }

      public bool IsZero()
      {
         return CurrentBullets <= 0;
      }

      public int BulletsRemains()
      {
         return BulletsCount - CurrentBullets;
      }
   }
}
```

ShootGun.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;
using CodeLibrary.Interfaces;

namespace CodeLibrary.CodeWeapon
{
   //абстрактный класс, который является шаблоном для всех стреляющих оружий в игре
   public abstract class ShootGun : IGun
   {
      public abstract int Damage { get; }
      public int BulletsCount { get; private set; }
      public int BulletsCountInClip { get; private set; }
      protected Clip Clip { get; set; }
      public int CurrentBulletsInClip
      {
         get
         {
            return Clip.CurrentBullets;
         }
         set { }
      }
      public int CurrentBulletsCount { get; set; }

      public ShootGun(int clipsCount, int bulletsInClip)
      {
         BulletsCount = clipsCount * bulletsInClip;
         BulletsCountInClip = bulletsInClip;
         Clip = new Clip(BulletsCountInClip);
         CurrentBulletsCount = BulletsCount - BulletsCountInClip;
      }

      protected virtual void Init()
      {

      }

      public void AddBullets(int bullets)
      {
         CurrentBulletsCount = (CurrentBulletsCount + bullets > BulletsCount) ? BulletsCount :
CurrentBulletsCount + bullets;
      }
      public void Reload()
      {

         if (!Clip.IsFull() && CurrentBulletsCount != 0)
         {

            int remains = Clip.BulletsRemains();
            if (remains > CurrentBulletsCount)
            {
               Clip.CurrentBullets += CurrentBulletsCount;
               CurrentBulletsCount = 0;
            }
            else
            {
               CurrentBulletsCount -= remains;
               Clip.CurrentBullets += remains;
```

```
            }
        }

    }

    public bool CanReloading()
    {
        bool canReload = false;
        if (!Clip.IsFull() && CurrentBulletsCount != 0)
        {
            canReload = true;
        }
        return canReload;
    }

    public void Shoot()
    {
        if (!Clip.IsZero())
        {
            Clip.CurrentBullets--;
        }
    }
  }
}
```

G36A2.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CodeLibrary.CodeWeapon;

namespace CodeLibrary.CodeWeapon.Guns
{
  public class G36A2 : ShootGun
  {
    public G36A2(int clipsCount, int bulletsInClip) : base(clipsCount, bulletsInClip)
    {
    }

    public override int Damage { get => 3;}

  }
}
```

Unit.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeLibrary.CodeUnit
```

```
{
   public class Unit
   {
      public float HealthCount { get; private set; }
      public float Health { get; private set; }
      public string Name { get; private set; }

      public  Unit(float health, string name)
      {
         HealthCount = health;
         Health = health;
         Name = name;
      }
      public void AddHealth(float health)
      {
         float currentHealth = Health + health;
         Health = (currentHealth > HealthCount) ? Health = HealthCount : currentHealth;
      }

      public void DealDamage(float damage)
      {
         float currentHealth = Health - damage;
         Health = (currentHealth < 0) ? 0 : currentHealth;
      }
   }
}
```

Player.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CodeLibrary.CodeItem;
using CodeLibrary.Enums;

namespace CodeLibrary.CodeUnit
{
   public class Player : Unit
   {
      private const int c_sphereToWin = 3;
      public int SphereCount { get { return Spheres.Count; } private set { SphereCount = value; } }

      public List<Sphere> Spheres { get; private set; }

      public Player(float health, string name) : base(health, name)
      {
         Spheres = new List<Sphere>();
      }
      public bool IsAll()
      {
         if (SphereCount == c_sphereToWin)
         {
            return true;
         }
```

```
            else return false;
        }
        public void AddSphere(SphereColor color)
        {
            Spheres.Add(new Sphere() { SphereColor = color });
        }
    }
}
```

Gun.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeLibrary.Interfaces
{
    //шаблон функционала стреляющих оружий
    public interface IGun
    {
        void Reload();
        void Shoot();
    }
}
```

IItemInteraction.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeLibrary.Interfaces
{
    public interface IItemInteraction
    {
      float DistanceToInteract { get; }
      void TriggerOnPlayerInteraction();
    }
}
```

Enums.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeLibrary.Enums
{
    public enum SphereColor
```

```
    {
        Red,
        Green,
        Blue
    }

    public enum KeyInputType
    {
        Up,
        Down,
        Always
    }

    public enum EnemyStatus
    {
        Patroling,
        AFK,
        Attacking,
        Runing,
        Default,
        OnShoot,
        Walking,
        Died,
        Dance
    }
}
```

InteractionDelay.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeLibrary.Delay
{
    public class InteractionDelay
    {
        private float _currentProjectTime;
        private float _interactTime;
        private float _delayTime;
        public InteractionDelay(float delayTime)
        {
            _delayTime = delayTime;
        }
        public void UpdateInteractTime(float interactTime)
        {
            _interactTime = interactTime;
        }
        public void UpdateProjectCurrentTime(float currentTime)
        {
            _currentProjectTime = currentTime;
        }

        public bool CanInteract()
```

```
        {
            return (_currentProjectTime >= _interactTime + _delayTime) ? true : false;
        }
    }
}
```

ObjectsInteract.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using CodeLibrary.Delay;
using CodeLibrary.Interfaces;

public class ObjectsInteract : MonoBehaviour
{
    [Header("")]
    [SerializeField] private KeyCode SelectKey;
    [Header("")]
    [SerializeField] private float InteractionDelayTime;

    InteractionDelay _interactionDelay;
    Camera _camera;

    private float _cameraCenterXPosition;

    private float _cameraCenterYPosition;
    [Header("")]
    public float Radius;

    private Vector3 _playerForward;

    private Ray _throwingRay;

    private RaycastHit _rayCastHit;

    private GameObject _hitObject;

    private IItemInteraction _itemsInteractionComponent;
    // Start is called before the first frame update
    void Start()
    {
        _interactionDelay = new InteractionDelay(InteractionDelayTime);
        _camera = GetComponent<Camera>();
    }

    // Update is called once per frame
    void Update()
    {
        _interactionDelay.UpdateProjectCurrentTime(Time.time);

        if (Input.GetKey(SelectKey))
        {
            if (_interactionDelay.CanInteract())
            {
                CatchItemObject();
```

```
            }
        }
    }

    private void CatchItemObject()
    {
        _cameraCenterXPosition = _camera.pixelWidth / 2;

        _cameraCenterYPosition = _camera.pixelHeight / 2;

        Vector3 origin = new Vector3(_cameraCenterXPosition, _cameraCenterYPosition, 5f);

        _throwingRay = _camera.ScreenPointToRay(origin);

        if (Physics.SphereCast(_throwingRay, Radius, out _rayCastHit))
        {
            _hitObject = _rayCastHit.transform.gameObject;

            if ((_itemsInteractionComponent = _hitObject.GetComponent<IItemInteraction>()) != null)
            {
                if (_rayCastHit.distance <= _itemsInteractionComponent.DistanceToInteract)
                {
                    _itemsInteractionComponent.TriggerOnPlayerInteraction();

                    _interactionDelay.UpdateInteractTime(Time.time);
                }
            }
        }
    }
}
```

SphereOnInteraction.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using CodeLibrary.Interfaces;
using CodeLibrary.Enums;

public class SphereOnInteraction : MonoBehaviour, IItemInteraction
{
    public float DistanceToInteract { get { return DistanceFromInteract; } }
    public float DistanceFromInteract;

    public SphereColor sphereColor;

    public void TriggerOnPlayerInteraction()
    {
        if (!Managers.PlayerManager.player.IsAll())
        {
            switch (sphereColor)
            {
                case SphereColor.Green:
                    {
                        Managers.CanvasManager.GreenSphere.SetActive(true);
                        break;
```

```
                }
            case SphereColor.Red:
                {
                    Managers.CanvasManager.RedSphere.SetActive(true);
                    break;
                }
            case SphereColor.Blue:
                {
                    Managers.CanvasManager.BlueSphere.SetActive(true);
                    break;
                }
        }
    }
    Managers.PlayerManager.player.AddSphere(sphereColor);
    gameObject.SetActive(false);
  }
}
```

LevelManager.cs:

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

public class LevelManager : MonoBehaviour
{
  public int MutantCount { get; private set; }
  public bool AllMutantDied { get; private set; }
  // Start is called before the first frame update
  void Start()
  {
    AllMutantDied = false;
  }

  // Update is called once per frame
  void Update()
  {
    if(MutantCount == 0)
    {
      AllMutantDied = true;
    }
  }

  public void SubstractMutantCount()
  {
    MutantCount--;
    if(MutantCount < 0)
    {
      MutantCount = 0;
    }
  }

  public void AddtionMutantCount()
  {
    MutantCount++;
  }
```

```
}
```

MainMenuManager.cs:

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

public class MainMenuManager : MonoBehaviour
{
   // Start is called before the first frame update
   void Start()
   {
      Time.timeScale = 1f;
   }


   public void StartGame(int sceneIndex)
   {
      SceneManager.LoadScene(sceneIndex);
   }
   public void GoToWorkSpace()
   {
      Application.Quit();
   }
}
```

Managers.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

public class Managers : MonoBehaviour
{
   public static PlayerManager PlayerManager { get; private set; }
   public static UIManager CanvasManager { get; private set; }
   public static LevelManager LevelManager { get; private set; }
   public void Awake()
   {
      PlayerManager = GetComponent<PlayerManager>();
      CanvasManager = GetComponent<UIManager>();
      LevelManager = GetComponent<LevelManager>();
   }
   public void Start()
   {

   }
}
```

PlayerManager.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using CodeLibrary.CodeUnit;
using UnityEngine.Audio;
using UnityEngine.UI;
public class PlayerManager : MonoBehaviour
{
    public Transform PlayerTransform;
    public Transform PublicTarget;
    public Player player;
    public G36A2Gun weapon;
    public float DyingTime;
    public float HelthCount;
    public AudioSource backGroundSource;
    [SerializeField] AnimationControl _animationControl;
    bool _dyingFrame;
    public bool IsDead { get { return _dyingFrame; } }
    public bool IsAllSphere { get; private set; }
    public static float CurHelth { get; private set; }

    [SerializeField] Slider volumeSlider;
    // Start is called before the first frame update
    void Start()
    {
        player = new Player(HelthCount, "Maria");
        _dyingFrame = false;
        IsAllSphere = false;
    }

    // Update is called once per frame
    public void Update()
    {
        backGroundSource.volume = volumeSlider.value;
        if(!backGroundSource.isPlaying)
        {
            backGroundSource.Play();
        }
        CurHelth = player.Health;
        if(player.IsAll())
        {
            IsAllSphere = true;
        }
        if(player.Health <= 0 && !_dyingFrame)
        {
            _dyingFrame = true;
            _animationControl.Dying();
            StartCoroutine(Dying());
        }
    }
    public void UpdateHealth(float health)
    {
        player.AddHealth(health);
    }
    IEnumerator Dying()
    {
        yield return new WaitForSeconds(DyingTime);
```

```
        _animationControl.gameObject.SetActive(false);
        //Time.timeScale = 0f;
    }
    public void UpdateAmmo(int bullets)
    {
        weapon.gun.AddBullets(bullets);
    }
}
```

UIManager.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;
using CodeLibrary.Enums;
using CodeLibrary.CodeItem;

public class UIManager : MonoBehaviour
{
    public KeyCode MenuKey;
    LevelManager LevelManager;
    public GameObject RedSphere;
    public GameObject BlueSphere;
    public GameObject GreenSphere;
    public GameObject GamePanel;
    public GameObject MenuPanel;
    public GameObject OptionPanel;
    public GameObject WinPanel;
    PlayerManager playerManager;
    bool GameInPause;

    [SerializeField] Slider HealthSlider;
    public Slider OXSens;
    public Slider OYSens;
    // Start is called before the first frame update
    void Start()
    {
        playerManager = GetComponent<PlayerManager>();
        LevelManager = GetComponent<LevelManager>();
        HealthSlider.maxValue = playerManager.HelthCount;
        HealthSlider.minValue = 20;
        MenuPanel.SetActive(false);
        OptionPanel.SetActive(false);
        WinPanel.SetActive(false);
        GameInPause = false;
    }

    // Update is called once per frame
    void Update()
    {
        playerManager.Update();
        HealthSlider.value = PlayerManager.CurHelth;
        if (HealthSlider.value <= HealthSlider.minValue)
        {
```

```
            HealthSlider.gameObject.SetActive(false);
        }
        else
        {
            HealthSlider.gameObject.SetActive(true);
        }

        if (Input.GetKeyDown(MenuKey))
        {
            if (GameInPause) ContinueGame();
            else StopGame();
        }

        if(playerManager.IsAllSphere && LevelManager.AllMutantDied)
        {
            WinGame();
        }
    }

    public void StopGame()
    {
        UnLockCursor();
        GameInPause = true;
        Time.timeScale = 0f;
        GamePanel.SetActive(false);
        MenuPanel.SetActive(true);
    }

    public void ContinueGame()
    {

        GameInPause = false;
        Time.timeScale = 1f;
        MenuPanel.SetActive(false);
        GamePanel.SetActive(true);
        LockCursor();
    }

    public void RestartGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
        LockCursor();
        Time.timeScale = 1f;
    }
    public void ExitOptions()
    {
        OptionPanel.SetActive(false);
        StopGame();
    }
    public void GameOptions()
    {
        MenuPanel.SetActive(false);
        OptionPanel.SetActive(true);
    }
    public void QuitTheGame(int sceneIndex)
    {
        Time.timeScale = 1f;
```

```
        SceneManager.LoadScene(sceneIndex);

    }
    public void WinGame()
    {
        UnLockCursor();
        WinPanel.SetActive(true);
        Time.timeScale = 0f;
    }

    void LockCursor()
    {
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }
    void UnLockCursor()
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
    }
}
```

Sphere.cs:

```
using System;

namespace CodeLibrary.CodeItem
{
    public class Sphere
    {
        public SphereColor SphereColor { get; set; }
    }
}
```

FPS.cs:

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

public class FPS : MonoBehaviour
{
    [Header("FPS ")]
    [SerializeField] private int _value = 60;

    private void OnValidate()
    {
        Application.targetFrameRate = _value;
    }
}
```

OutputFps.cs

```
using UnityEngine;
using System.Collections;

public class OutputFps : MonoBehaviour
{

    void OnGUI()
    {
        float fps = 1.0f / Time.deltaTime;
        GUILayout.Label("FPS = " + fps);
    }
}
```