**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**«ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО»**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

по дисциплине «Программирование сетевых приложений»

**на тему: «СЕТЕВОЕ ИГРОВОЕ ПРИЛОЖЕНИЕ «БИТВА НА ВЕРТОЛЕТАХ» ДЛЯ ДВУХ ИГРОКОВ С ИСПОЛЬЗОВАНИЕМ СРЕДЫ РАЗРАБОТКИ *UNITY* И ПРОТОКОЛА *TCP*»**

Исполнитель: студент гр. ИТИ-41

Морозько И.В.

Руководитель: доцент

Курочка К.С.

Дата проверки: *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Дата допуска к защите: *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Дата защиты: *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Оценка работы: *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Подписи членов комиссии

по защите курсового проекта: *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Гомель 2023

**ЛИСТ ИСПРАВЛЕНИЙ**

Таблица 1.1– Исправление замечаний к пояснительной записке

|  |  |  |  |
| --- | --- | --- | --- |
| № пункта | Замечание | Исправления | Страницы |
| 1. |  |  |  |

**СОДЕРЖАНИЕ**

[Введение 4](#_Toc152200093)

[1 Игровые приложения и средства их разработки 5](#_Toc152200094)

[1.1 Обзор игр жанра аркада и симулятор 5](#_Toc152200095)

[1.2 Преимущества и недостатки *Unity* и *С*# 9](#_Toc152200096)

[1.3 Клиент-серверное взаимодействие в многопользовательских сетевых приложениях 11](#_Toc152200097)

[1.4 Применение протокола *TCP* в сетевых коммуникациях 13](#_Toc152200098)

[2 Основной этап разработки игрового приложения 16](#_Toc152200099)

[2.1 Компоненты игрового приложения 16](#_Toc152200100)

[2.2 Игровой движок *Unity* как основной компонент игрового приложения 18](#_Toc152200101)

[2.3 Сетевое взаимодействие в игровом приложении 18](#_Toc152200102)

[2.4 Программная реализация игрового приложения 19](#_Toc152200103)

[3 Верификация и апробация игрового приложения «битва на вертолётах» 25](#_Toc152200104)

[3.1 Принцип работы игрового приложения 25](#_Toc152200105)

[3.2 Результат верификации игрового приложения 26](#_Toc152200106)

[Заключение 29](#_Toc152200107)

[Список источников 30](#_Toc152200108)

[Приложение А Ихсодный код программы 31](#_Toc152200109)

[Приложение Б Руководство поользователя 77](#_Toc152200110)

[Приложение В Руководство программиста 79](#_Toc152200111)

[Приложение Г Руководство системного программиста 80](#_Toc152200112)

[Приложение Д Схема структуры приложения 81](#_Toc152200113)

ВВЕДЕНИЕ

В настоящий момент индустрия компьютерных игр развивается стремительными темпами, что играет одну из ключевых ролей в развитии всех компьютерных технологий. Для реализации крупных проектов необходимы технологии, способные обрабатывать большое количество информации.

Игры являются одним из самых популярных средств развлечений и времяпрепровождения. Разработка новых игровых приложений даёт жизнь новым идеям и альтернативным реализациям готовых решений, пересматривает устоявшиеся концепции, что способствует развитию игровой индустрии и компьютерных технологий. Результаты данной работы помогут выделению новых аспектов в разработке игровых приложений и игровой индустрии в целом, без которых развитие темы и поиск новых решений задач будет невозможен.

В реальном мире у людей мало шансов стать пилотом штурмового вертолёта и принять участие в воздушных сражениях без риска для жизни. Игровое приложение «Битва на вертолётах» позволит почувствовать себя пилотом боевого вертолёта и сразиться в воздухе с живым противником. Создаваемое игровое приложение послужит отличным досугом, поможет хорошо провести время с друзьями, улучшить реакцию, научиться тактике и стратегии, развить пространственное мышление. Приложение позволит учиться быстрее реагировать на неожиданные ситуации и принимать решения, а это обязательно пригодится в жизни, также развивает концентрацию и внимание.

Основной целью данного курсового проекта является проектирование и создание игрового приложение «Битва на вертолетах» для игры по сети двух пользователей друг против друга, где каждый игрок будет управляет вертолетом, который может перемещаться вверх, вправо, влево, а также подвержен влиянию гравитации. Внизу вертолетов будет находится земля, столкновение с которой приведёт к проигрышу, кроме того, на игровой карте будут расположены платформы, столкновение с которыми уменьшает показатели брони вертолета. Вертолеты должны обладать «бронёй», при исчерпании которой вертолет погибает, а игрок проигрывает. Вертолет должен стрелять различными видами снарядов, траектория движения которых зависит от текущего положения вертолета и направления его вертикального движения. При этом можно будет собрать «призы», которые позволяют увеличить скорость полёта снаряда, его убойную силу и радиус поражения. Кроме того, с помощью «призов» можно пополнить запасы «брони». Также реализовать возможность сделать ускорение вертолёта (рывок), с помощью которого вертолёт сможет избегать летящие снаряды. Способность ускорения сделать восстанавливающейся со временем. Для реализации использовать среду разработки *Unity*. Для реализации сетевого взаимодействия необходимо использовать протокол *TCP*.

1. игровые приложения и СРЕДСТВА их РАЗРАБОТКИ
   1. Обзор игр жанра аркада и симулятор

Видеоигры-симуляторы представляют собой разнообразную категорию видеоигр, обычно предназначенных для точной имитации действий в реальном мире. Игра-симулятор пытается скопировать различные действия из реальной жизни в форме игры для различных целей, таких как обучение, анализ, прогнозирование или развлечение. Обычно в игре нет строго определенных целей, и игроку разрешается свободно управлять персонажем или окружением. Хорошо известными примерами являются военные игры, деловые игры и моделирование ролевых игр. Из трех основных типов стратегических упражнений, планирования и обучения: игр, симуляций и тематических исследований, можно рассмотреть ряд гибридов, включая имитационные игры, которые используются в качестве тематических исследований [1].

Аркада (англ. *arcade* *game*, *arcade* *genre*) – жанр компьютерных игр, характеризующийся коротким по времени, но интенсивным игровым процессом. В узком смысле аркадными называются игры для аркадных игровых автоматов. Игры, портированные с аркадных автоматов, также принято называть аркадными. Период с конца 1970-х до середины 1980-х был охарактеризован расцветом аркадных игр, и его часто называют золотым веком аркадных игр. Название «аркада» происходит от англ. *arcade* – пассаж, крытая галерея магазинов, где традиционно размещались аркадные игровые автоматы [2].

Классические аркады характеризуются следующими свойствами:

* Игра на одном экране. В классических аркадах весь игровой процесс сосредоточен на одном экране. Прежде всего это обусловлено исторически, что произошло из-за технических ограничений, но в то же время это значительно влияло на геймдизайн. Так, игроки в любой момент времени могли видеть весь игровой мир и принимать решения, исходя из полной информации о его состоянии. Множество игр жанра имело более одного экрана, и они сменяли друг друга как уровни.
* Бесконечная игра. Потенциально игроки могут играть в аркаду бесконечное время, и соответственно, не могут выиграть. Это влияло на то, что игроки делали вызов сами себе – насколько долго они смогут продержаться. Относительно геймдизайна в аркадах игрок никогда не выигрывал, и каждая игра заканчивалась поражением. В то же время, игры проектировались таким образом, чтобы со временем игроку становилось всё сложнее, и таким образом бесконечная игра предлагала бесконечную сложность. Данная ситуация изменилась с появлением рынка домашних компьютеров, когда издатели изменили своё желание, чтобы игроки проходили игру и после этого хотели купить новую.
* Множество жизней. Обычно, классическая аркада предлагает игроку несколько попыток (жизней). Такой подход позволяет новичкам получить большую возможность изучить игровые механики до того, как игра заканчивается. Если игрок лучше понимал игру, то возрастала вероятность того, что он вернётся к ней снова. Данная особенность также позволяла предоставлять игрокам за их успехи дополнительные жизни в качестве награды.
* Игровой счёт. Практически все классические аркады включают в себя игровой счёт, когда игрок получает очки за выполнение различных целей или задач. Например, в *Centipede* игроки получают 1 очко за уничтожение гриба, 10 за сегмент многоножки, 100 за её голову, и 1000 за скорпиона. Здесь имеется другой важный концепт аркад, унаследованный от игр пинбола: очки позволяют игроку понять, насколько хорошо он играл, несмотря на то, что выиграть невозможно. При этом типичное время игры среднего игрока составляет около двух минут, а у опытного до десятков минут. На основании данной особенности у аркад как правило есть таблица рекордов, где игрок может рядом со своим результатам ввести свои инициалы, и тем самым сравнивать себя с другими игроками, и далее «хвастаться» и соревноваться.
* Быстрое обучение, простой игровой процесс. Для классических аркад характерно то, что игрокам легко научиться геймплею, но становится практически невозможным стать мастером в игре из-за её сложности. Вместе с тем, если игрок погибает в аркаде, то это практически всегда происходит по его вине. В таких играх нет «специальных комбинаций клавиш», которые игрок должен выучить из документации для того, чтобы сделать что-то особенное. Очень мало игр расширяют концепт посредством очков здоровья, щитов или таблеток силы. Это связано с тем, что с коммерческой точки зрения аркадам было необходимо охватить как можно больший спектр игроков, то есть, фактически каждый человек в баре или магазине должен быть способным подойти и попробовать сыграть. В то же время, простой игровой процесс не подразумевает что он «плохой» или «ограниченный», – он может быть «элегантным» и «отполированным».
* Нет сюжета/истории. Классические аркады практически всегда избегали попыток рассказать какую-либо историю, и данная тенденция продолжается для современных аркад. Играм жанра всегда требовалось, чтобы игроки быстро поняли что происходит – это научная фантастика, война, спорт или что-то ещё. Множество аркад создавало свои уникальные сеттинги. Геймдизайнеры классических аркадных игр не чувствовали, что им нужно наполнять свои миры чем-то и отдельно объяснять игрокам почему они должны стрелять в те или иные цели различной формы.

Типичным представителем жанра аркадного симулятора является серия игр *Farming* *Simulator*, разработанных компанией *Giants* *Software*. Главной целью игрового процесса является ведение различной сельскохозяйственной деятельности, например, выращивание сельскохозяйственных культур, разведение животных, лесозаготовка, производство энергоносителей, а также, получение прибыли от этой деятельности. В игре нет конечной цели для достижения победы. Игрокам доступно большое количество различной сельскохозяйственной техники и оборудования, из которых более 90 % являются лицензированными моделями известных мировых брендов. Большое внимание уделяется реалистичности управления и высокой детализации техники и оборудования.



Рисунок 1.1 – *Farming* *Simulator* 19

Игрок начинает карьеру фермера, имея в своем распоряжении небольшой парк устаревшей техники, несколько небольших полей и небольшой стартовый капитал. Начальная техника позволяет заниматься только земледелием. Чтобы получать урожай, игрок должен обрабатывать поля. Обработка полей включает такие этапы, как: вспашка полей плугом, культивирование, посев, удобрение органическими или промышленными удобрениями, а также сбор урожая. В дальнейшем, помимо земледелия, игрок сможет заниматься животноводством и лесозаготовкой. Продавая урожай, игрок получает игровую валюту, на которую он может приобретать новую технику и оборудование, животных, новые поля и статические объекты. Статические объекты приносят регулярную прибыль, не зависящую от деятельности игрока. Урожай можно продавать сразу или накапливать в специальных бункерах, ожидая более выгодных цен. Игрок может брать кредит в банке. Любым видом деятельности можно заниматься разными способами, используя различную технику. В игре доступны дополнительные побочные задания, за выполнение которых игрок получает вознаграждение. Игра имеет высокий уровень реализма, но для удобства игрового процесса некоторые игровые условия сильно отличаются от реальных. Например, размеры полей, объём собираемого с них урожая, цены на продукцию, но главное отличие – в игровом времени. Все игровые процессы происходят быстро, к тому же, время можно ускорять. Созревание урожая занимает от одного до трех дней. В игре всегда лето, но в разные дни бывает различная погода, например, может пойти дождь или град. Также, в игре есть смена дня и ночи.

Ещё одним примером игр жанра аркадного симулятора является *Prison* *Architect* – компьютерная игра, разработанная британской компанией *Introversion* *Software*.

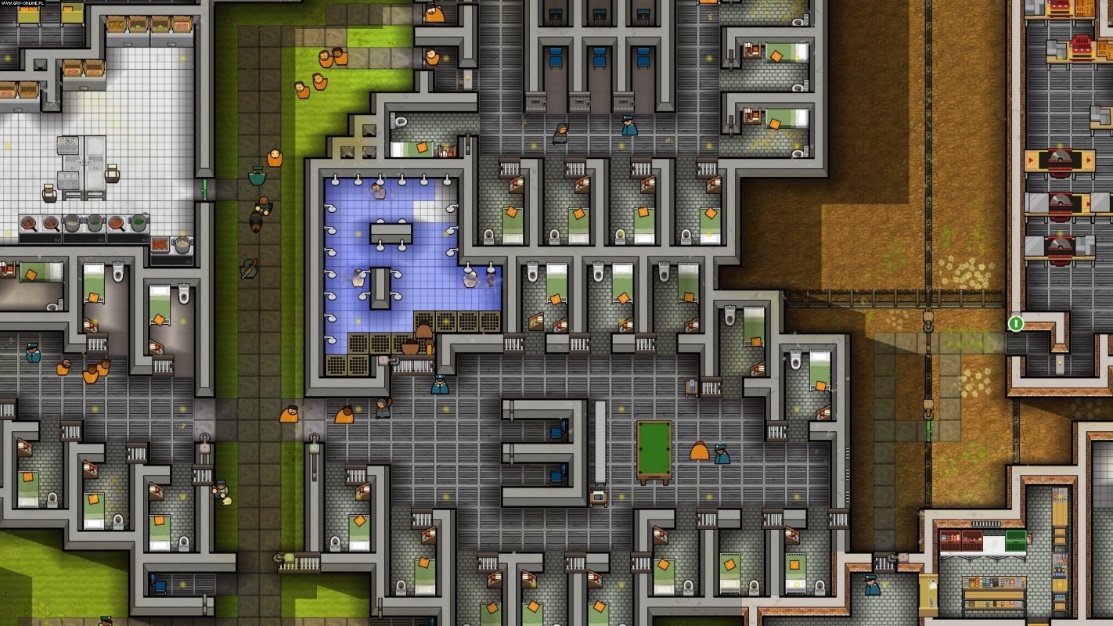


Рисунок 1.2 – *Prison* *Architect*

Игра представляет собой экономический симулятор, где игрок выступает в роли управляющего тюрьмы, в полномочия которого входит: постройка инфраструктуры, наём персонала, определение внутреннего распорядка и политики. В *Prison* *Architect* присутствует кампания в которой совмещено обучение, сюжет и песочница. Также в игре имеется возможность играть за заключённого и сбегать из тюрем, построенных вами и другими игроками.

Стоит отметить *Viscera* *Cleanup* *Detail* – это научно-фантастическая видеоигра в жанре симулятора, в которой игрокам предстоит убирать кровавые последствия успешно отраженного инопланетного вторжения. Игра была разработана и издана южноафриканской инди-студией-разработчиком *RuneStorm* и выпущена 4 апреля 2014 года через *Steam* *Early* *Access*. Полный релиз стал доступен 23 октября 2015 года.



Рисунок 1.3 – *Viscera* *Cleanup* *Detail*

В игре *Viscera* *Cleanup* *Detail* игрокам отводится роль "уборщиков космических станций", которым поручено очистить и отремонтировать помещения, ставшие ареной кровопролитных сражений во время инопланетного вторжения или другой катастрофы. Задачи включают сбор и утилизацию мусора, в том числе расчлененных тел инопланетян и людей, стреляных гильз и битого стекла, пополнение запасов настенных аптечек, заделку пулевых отверстий в стенах, очистку полов, стен и потолков от брызг крови и копоти, а также второстепенные бонусные задания. К ним относятся укладка предметов, таких как ящики и бочки, в специально отведенном месте для складывания и составление отчетов о катастрофах и смертях, произошедших на соответствующем уровне.

* 1. Преимущества и недостатки *Unity* и *С*#
     1. *Unity* – межплатформенная среда разработки компьютерных игр, разработанная американской компанией *Unity* *Technologies*. *Unity* позволяет создавать приложения, работающие на более чем 25 различных платформах, включающих персональные компьютеры, игровые консоли, мобильные устройства, интернет-приложения и другие. Выпуск *Unity* состоялся в 2005 году и с того времени идёт постоянное развитие [3].

Как правило, игровой движок предоставляет множество функциональных возможностей, позволяющих их задействовать в различных играх, в которые входят моделирование физических сред, карты нормалей, динамические тени и многое другое. В отличие от многих игровых движков, у *Unity* имеется два основных преимущества: наличие визуальной среды разработки и межплатформенная поддержка. Первый фактор включает не только инструментарий визуального моделирования, но и интегрированную среду, цепочку сборки, что направлено на повышение производительности разработчиков, в частности, этапов создания прототипов и тестирования. Под межплатформенной поддержкой предоставляется не только места развёртывания (установка на персональном компьютере, на мобильном устройстве, консоли и т. д.), но и наличие инструментария разработки (интегрированная среда может использоваться под *Windows* и *Mac* *OS*).

Третьим преимуществом называется модульная система компонентов *Unity*, с помощью которой происходит конструирование игровых объектов, когда последние представляют собой комбинируемые пакеты функциональных элементов. В отличие от механизмов наследования, объекты в *Unity* создаются посредством объединения функциональных блоков, а не помещения в узлы дерева наследования. Такой подход облегчает создание прототипов, что актуально при разработке игр.

В качестве недостатков приводятся ограничение визуального редактора при работе с многокомпонентными схемами, когда в сложных сценах визуальная работа затрудняется. Вторым недостатком называется отсутствие поддержки *Unity* ссылок на внешние библиотеки, работу с которыми программистам приходится настраивать самостоятельно, и это также затрудняет командную работу. Ещё один недостаток связан с использованием шаблонов экземпляров. С одной стороны, эта концепция *Unity* предлагает гибкий подход визуального редактирования объектов, но с другой стороны, редактирование таких шаблонов является сложным. Также, *WebGL*-версия движка, в силу специфики своей архитектуры (трансляция кода из *C*# в *С*++ и далее в *JavaScript*), имеет ряд нерешённых проблем с производительностью, потреблением памяти и работоспособностью на мобильных устройствах.

* + 1. *C*# – современный объектно-ориентированный и типобезопасный язык программирования. *C*# позволяет разработчикам создавать разные типы безопасных и надежных приложений, выполняющихся в .*NET*. *C*# относится к широко известному семейству языков *C*, и покажется хорошо знакомым любому, кто работал с *C*, *C*++, *Java* или *JavaScript*.

*C*# предоставляет языковые конструкции для прямой поддержки этой концепции работы. Это делает *C*# подходящим для создания и использования программных компонентов. С момента своего создания *C*# был обогащен функциями для поддержки новых рабочих нагрузок и передовыми методами разработки современного программного обеспечения.

Возможности языка *C*# для создания надежных и стабильных приложений:

* сборка мусора автоматически освобождает память, занятую недоступными, неиспользуемыми объектами;
* типы, допускающие значение *NULL*, обеспечивают защиту от переменных, которые не относятся к выделенным объектам;
* обработка исключений предоставляет структурированный и расширяемый подход к обнаружению ошибок и восстановлению после них;
* обработка исключений обеспечивает структурированный и расширяемый подход к обнаружению и устранению ошибок;
* лямбда-выражения поддерживают методы функционального программирования;
* синтаксис *LINQ* создает общий шаблон для работы с данными из любого источника;
* языковая поддержка асинхронных операций обеспечивает синтаксис для создания распределенных систем;
* унифицированная система типов.

Все типы *C*#, включая примитивные типы, такие как *int* и *double*, наследуются от одного и того же объекта основного типа. Все типы используют общий набор операций, и значения любого типа могут храниться, передаваться и манипулироваться аналогичным образом. Кроме того, *C*# поддерживает как определяемые пользователем ссылочные типы, так и типы значений. *C*# позволяет динамически размещать объекты и хранить в стеке упрощенные структуры. *C*# поддерживает универсальные методы и типы, обеспечивающие повышенную безопасность типов и производительность. *C*# предоставляет итераторы, которые позволяют разработчикам классов коллекций определять собственное поведение клиентского кода. *C*# делает упор на «управление версиями», чтобы поддерживать совместимость программ и библиотек с течением времени. Проблемы управления версиями существенно повлияли на такие аспекты программирования на *C*#, как отдельные модификаторы *virtual* и *override*, правила разрешения перегрузок методов и поддержка явного объявления элементов интерфейса.

Программы *C*# выполняются в .*NET*, виртуальной системе выполнения, которая вызывает среду *CLR* и набор библиотек классов. Общеязыковая среда выполнения (*CLR*) – это международная стандартная реализация *Common* *Language* *Infrastructure* (*CLI*) от *Microsoft*. *CLI* является основой для создания сред выполнения и разработки, в которых языки и библиотеки прозрачно взаимодействуют друг с другом [4, c. 20].

Исходный код, написанный на *C*#, компилируется в промежуточный язык (*IL*), соответствующий спецификациям *CLI*. Код и ресурсы *IL*, включая растровые изображения и строки, хранятся в сборке, обычно с расширением .*dll*. Сборка включает манифест с информацией о типе, версии и культуре для этой сборки.

При запуске программы *C*# сборка загружается в среду *CLR*. *CLR* выполняет *JIT*-компиляцию *IL*-кода в инструкции машинного языка. Общеязыковая среда выполнения также выполняет другие операции, такие как автоматическая сборка мусора, обработка исключений и управление ресурсами. Код, выполняемый общеязыковой средой выполнения, иногда называют «управляемым кодом». «Неуправляемый код» компилируется в машинный язык, предназначенный для конкретной платформы.

Обеспечение взаимодействия между языками – ключевая особенность .*NET*. Код *IL*, сгенерированный компилятором *C*#, соответствует спецификации общего типа (*CTS*). Код *IL*, созданный из кода *C*#, может взаимодействовать с кодом, созданным из *F*#, *Visual* *Basic*, *C*++ для .*NET*. Существует более 20 других языков, совместимых с *CTS*. Одна сборка может содержать несколько модулей, написанных на разных языках .*NET*, и все типы могут ссылаться друг на друга, как если бы они были написаны на одном языке.

В дополнение к службам времени выполнения .*NET* также включает расширенные библиотеки. Эти библиотеки поддерживают множество различных рабочих нагрузок. Они упорядочены по пространствам имен, которые предоставляют разные полезные возможности: от операций файлового ввода и вывода до управления строками и синтаксического анализа *XML*, от платформ веб-приложений до элементов управления *Windows* *Forms* и *WPF*. Обычно приложение *C*# активно используют библиотеку классов .*NET* для решения типовых задач.

* 1. Клиент-серверное взаимодействие в многопользовательских сетевых приложениях

Передача данных по сети с одного устройства на другое проводится по ряду определенных действий, определяемых сетевыми протоколами. Для облегчения разработки протоколов были созданы сетевые модели – схемы, определяющие общие принципы работы набора сетевых протоколов и способы их взаимодействия друг с другом для осуществления передачи данных по сети. Модель разделяется на уровни, в которых протоколы используют протоколы нижестоящего уровня.

Главной задачей при разработке многопользовательских игр является синхронизация состояний всех пользователей на сервере. Для обеспечения синхронизации клиентам необходимо отправлять серверу обновления с фиксированным интервалом времени. Обновления содержат введенные пользователям данные. После получения данных от всех пользователей, следующим тактом сервер обновляет состояния всех клиентов. На рисунке 1.4 представлено взаимодействие сервера с несколькими клиентами.

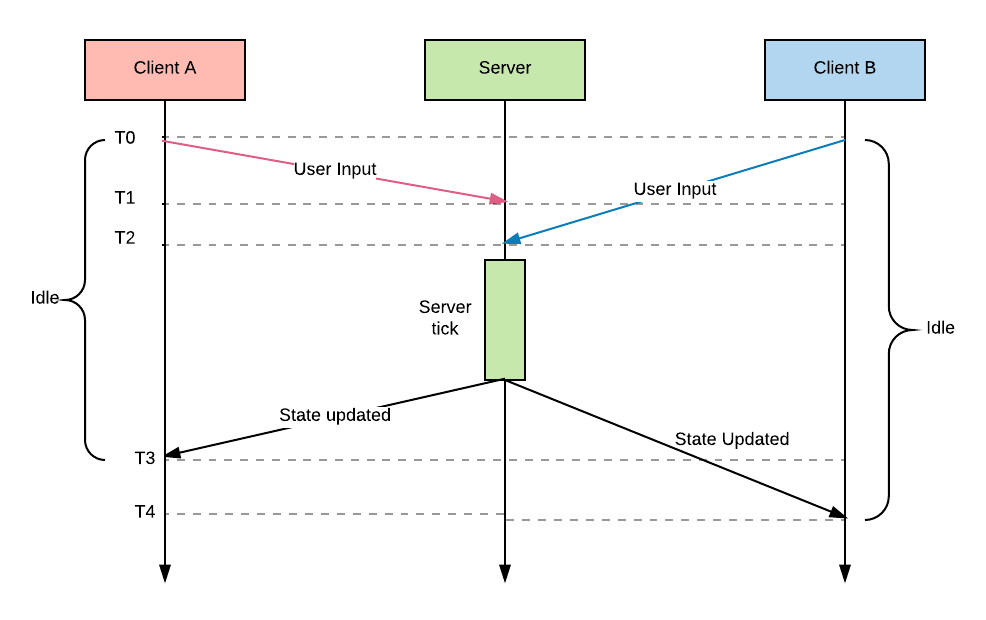


Рисунок 1.4 – Взаимодействие сервера с несколькими клиентами

Отправка данных с фиксированным интервалом времени в лучшем случае имеет высокую скорость обработки данных и возможность обслуживания большего количества пользователей. Однако, если имеется игрок с плохим интернет-соединением, то замедляется обновления у всех пользователей. Решением данной проблемы является использование концепции прогнозирования. Прогнозирование происходит на стороне клиента, когда вместо ожидания ответа от сервера программа прогнозирует исход действий, отправленных серверу [5]. И когда клиент получит данные от сервера, он подменяет их на устройстве. Таким образом, данная концепция минимизирует количество явных зависаний у всех клиентов при плохом качестве интернет-соединении.

* 1. Применение протокола *TCP* в сетевых коммуникациях
     1. *Transmission* *Control* *Protocol* (*TCP*) – один из основных протоколов передачи данных интернета. Предназначен для управления передачей данных интернета. Пакеты в *TCP* называются сегментами [6]. Этот протокол является одним из наиболее распространенных протоколов взаимодействия в сети и гарантирует доставку сообщений и широко используется в различных существующих на сегодняшний день программах.

В стеке протоколов *TCP*/*IP* выполняет функции транспортного уровня модели *OSI*.

Механизм *TCP* предоставляет поток данных с предварительной установкой соединения, осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного пакета, гарантируя тем самым целостность передаваемых данных и уведомление отправителя о результатах передачи.

Реализации *TCP* обычно встроены в ядра ОС. Существуют реализации *TCP*, работающие в пространстве пользователя.

Когда осуществляется передача от компьютера к компьютеру через Интернет, *TCP* работает на верхнем уровне между двумя конечными системами, например, браузером и веб-сервером. *TCP* осуществляет надёжную передачу потока байтов от одного процесса к другому.

На рисунке 1.5 представлено взаимодействие сервера с клиентом по протоколу *TCP*.

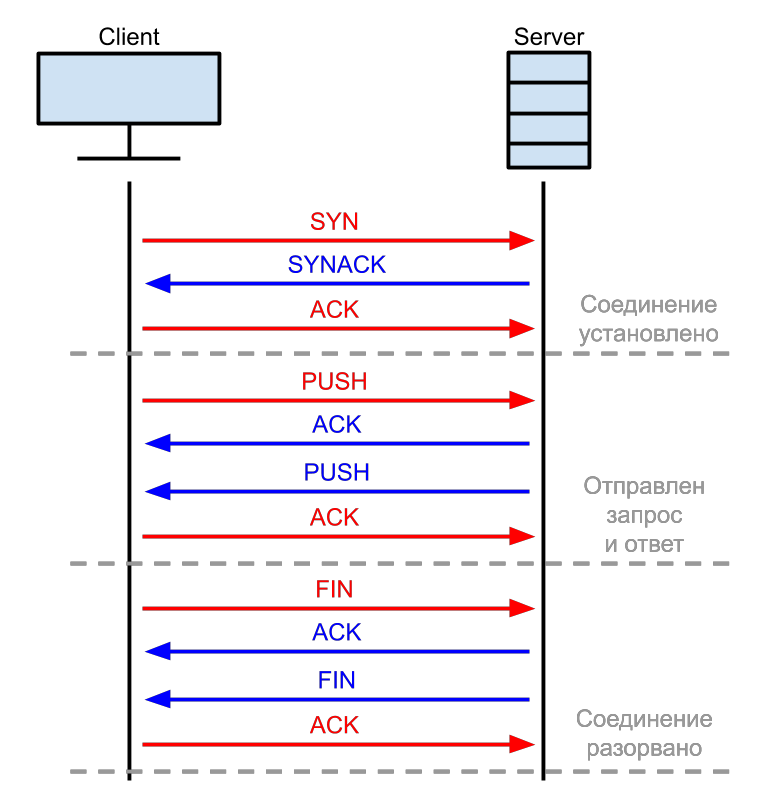


Рисунок 1.5 – Взаимодействие сервера с клиентом по протоколу *TCP*

* + 1. Процесс начала сеанса *TCP* (также называемый «рукопожатие»), состоит из трёх шагов.

1. Клиент, который намеревается установить соединение, посылает серверу сегмент с номером последовательности и флагом *SYN*.

* Сервер получает сегмент, запоминает номер последовательности и пытается создать сокет (буфера и управляющие структуры памяти) для обслуживания нового клиента.
* В случае успеха сервер посылает клиенту сегмент с номером последовательности и флагами *SYN* и *ACK*, и переходит в состояние *SYN*-*RECEIVED*.
* В случае неудачи сервер посылает клиенту сегмент с флагом *RST*.

2. Если клиент получает сегмент с флагом *SYN*, то он запоминает номер последовательности и посылает сегмент с флагом *ACK*.

* Если клиент одновременно получает и флаг *ACK* (что обычно и происходит), то он переходит в состояние *ESTABLISHED*.
* Если клиент получает сегмент с флагом *RST*, то он прекращает попытки соединиться.
* Если клиент не получает ответа в течение 10 секунд, то он повторяет процесс соединения заново.

3. Если сервер в состоянии *SYN*-*RECEIVED* получает сегмент с флагом *ACK*, то он переходит в состояние *ESTABLISHED*.

* В противном случае после тайм-аута он закрывает сокет и переходит в состояние *CLOSED*.

Процесс называется «трёхэтапным рукопожатием», так как, несмотря на то, что возможен процесс установления соединения с использованием четырёх сегментов (*SYN* в сторону сервера, *ACK* в сторону клиента, *SYN* в сторону клиента, *ACK* в сторону сервера), на практике для экономии времени используется три сегмента.

* + 1. Завершение соединения можно рассмотреть в три этапа:

1. Посылка серверу от клиента флага *FIN* на завершение соединения.
2. Сервер посылает клиенту флаги ответа *ACK*, *FIN*, что соединение закрыто.
3. После получения этих флагов клиент закрывает соединение и в подтверждение отправляет серверу *ACK* , что соединение закрыто.

Для работы с протоколом *TCP* в .*NET* предназначены классы *TcpClient* и *TcpListener*. Эти классы строятся поверх класса *System*.*Net*.*Sockets*.*Socket*. *TcpClient* и *TcpListener* упрощают создание клиента и сервера, которые реализуют протокол *TCP*. Если же функциональности этих классов недостаточно, то для более продвинутых и изощренных сценариев можно использовать тот же класс *Socket*.

* + 1. При обмене данными приёмник использует номер последовательности, содержащийся в получаемых сегментах, для восстановления их исходного порядка. Приёмник уведомляет передающую сторону о номере последовательности, до которой он успешно получил данные, включая его в поле «номер подтверждения». Все получаемые данные, относящиеся к промежутку подтверждённых последовательностей, игнорируются. Если полученный сегмент содержит номер последовательности больший, чем ожидаемый, то данные из сегмента буферизируются, но номер подтверждённой последовательности не изменяется. Если впоследствии будет принят сегмент, относящийся к ожидаемому номеру последовательности, то порядок данных будет автоматически восстановлен исходя из номеров последовательностей в сегментах.

Для того, чтобы передающая сторона не отправляла данные интенсивнее, чем их может обработать приёмник, *TCP* содержит средства управления потоком. Для этого используется поле «окно». В сегментах, направляемых от приёмника передающей стороне, в поле «окно» указывается текущий размер приёмного буфера. Передающая сторона сохраняет размер окна и отправляет данных не более, чем указал приёмник. Если приёмник указал нулевой размер окна, то передачи данных в направлении этого узла не происходит, пока приёмник не сообщит о большем размере окна.

В некоторых случаях передающее приложение может явно затребовать передать данные до некоторой последовательности принимающему приложению, не буферизируя их. Для этого используется флаг *PSH*. Если в полученном сегменте обнаруживается флаг *PSH*, то реализация *TCP* отдаёт все буферизированные на текущий момент данные принимающему приложению. «Проталкивание» используется, например, в интерактивных приложениях. В сетевых терминалах нет смысла ожидать ввода пользователя после того, как он закончил набирать команду. Поэтому последний сегмент, содержащий команду, обязан содержать флаг *PSH*, чтобы приложение на принимающей стороне смогло начать её выполнение.

1. основной этап разработки игрового приложения
   1. Компоненты игрового приложения

Для осуществления грамотной разработки и возможности переносимости данного игрового приложения необходимо разделить все программные компоненты на функциональные части, которые будут проектироваться на своих этапах разработки приложения. С использованием методов декомпозиции определяется так называемая иерархическая структура работ. Данная структура разделяет общую задачу на иерархически зависимые подзадачи.

Принцип декомпозиции имеет следующие характеристики:

* разделение проекта на подзадачи;
* описание с необходимой точностью содержание каждой подзадачи по проекту;
* формирование проекта с его задачами в виде иерархической структуры;
* оценка реалистичности выполнения проекта;
* оценка ресурсов.

В ходе применения принципа декомпозиции приложение было разделено на следующие блоки (рисунок 2.1).

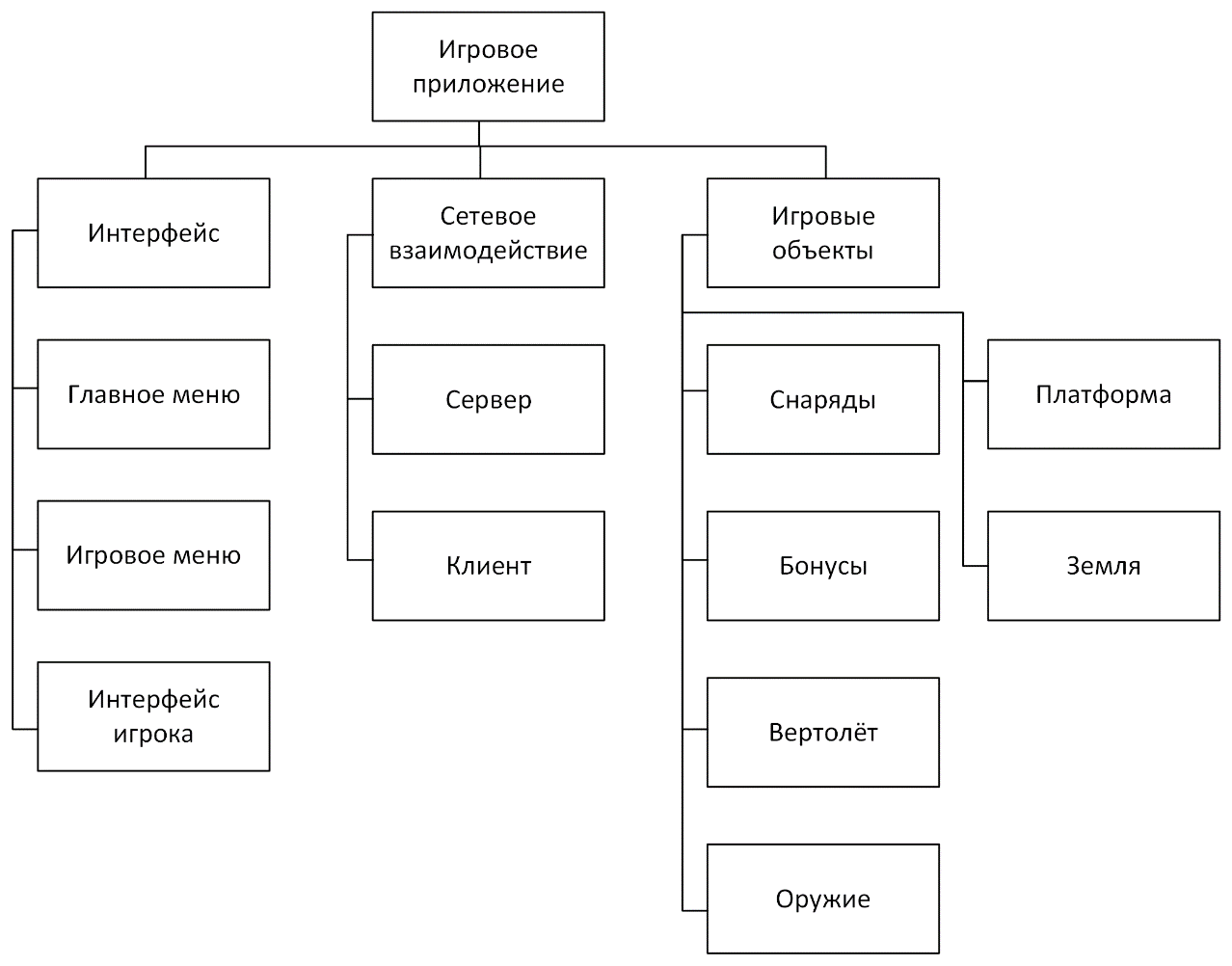


Рисунок 2.1 – Иллюстрация разделения задачи по принципу декомпозиции

Основными задачами при создании игрового приложения станут создание игровых объектов, их взаимодействия в рамках игровой логики и интерфейса игрока средствами *Unity* и *C#*, а также реализация сетевого взаимодействия с помощью протокола *TCP*.

Среди игровых объектов можно выделить снаряды, которые будут перемещаться по определённой физической модели, бонусы, которые будут улучшать характеристики оружия вертолёта и увеличивать показатель здоровья, вертолёт, которых будет главным игровым объектом и которыми будут управлять игроки, оружие, которое будут иметь вертолёты и которое будут улучшать бонусы, давая создаваемым снарядам различные характеристики, а также платформы, которые будут наносить урон вертолётам при столкновении, и землю, столкновение с которой приведёт к поражению в раунде. Для снарядов необходимо сделать реагирование на столкновение с другими объектами и нанесение урона вертолёту. Для бонусов необходимо будет реализовать добавление улучшений к конкретному типу вооружения у вертолёта и восстановление здоровья вертолёта. Для вертолёта необходимо сделать передвижение, ускорение с его восстановлением, переключение выбранного вооружения, стрельбу и перезарядку. Для вооружения необходимо сделать создание снарядов с конкретными характеристиками, которые могут изменяться при применении бонусов и окончании их эффектов. Для платформы и земли необходимо сделать реагирование на столкновение с вертолётами.

Среди объектов интерфейса можно выделить главное меню, которое будет встречать игрока при каждом запуске игрового приложения и в котором игрок сможет создать сервер, присоединиться к уже существующему серверу или выйти из игры, а также интерфейс игрока, который будет отображать текущее здоровье игрока и сообщение о победе/поражении, и игровое меню, в котором игрок сможет выйти в главное меню и продолжить игру.

Среди объектов сетевого взаимодействия можно выделить объект клиента, который будет отвечать за подключение к серверу, отправку ввода клиента серверу и получение информации об игровых объектах на сервере, и объект сервер, который будет принимать ввод клиентов, обновлять позиции игроков и отправлять состояния объектов на сцене.

Задача была разбита на зависящие друг от друга подзадачи. Игровая локация будет представлять собой область, ограниченную по краям экрана платформами и поверхностью земли. В этой игровой области будут находиться платформы-препятствия и появляться различные бонусы. При появлении двух игроков на сцене начнётся раунд, который завершится с уничтожением одного из игроков. Игрокам необходимо для победы уничтожить вертолёт противника, попав по нему выпущенным снарядом из своего вертолёта. Игрок может менять тип снаряда для стрельбы, каждый из которых имеет свою модель поведения и урон. С поражением одного из игроков начнётся новый раунд.

В ходе игрового процесса управляемый игроком вертолёт будет сталкиваться с препятствиями, собирать бонусы, стрелять по своему противнику и получать урон от попадания по нему снарядом.

Для осуществления взаимодействия вертолёта с объектами на сцене в программе будут реализованы следующие функции:

* влияние физики на полёт снаряда и вертолёта;
* столкновение снаряда с вертолётом, другим снарядом, препятствиями и поверхностью земля;
* подбор бонусов при столкновении вертолёта с ними.

Игрок сможет менять выбранное вооружения нажатием на клавиши «*Q*» и «*E*». Будут реализованы три типа снарядов: обычная пуля, которая имеет средние показатели скорости и урона, ракета, которая имеет радиус срабатывания, летит строго прямо и имеет средний урон, и бомба, которая имеет низкую скорость, высокий урон и появляется под вертолётом.

* 1. Игровой движок *Unity* как основной компонент игрового приложения

Для реализации игровой логики требуется игровой движок – проект, разделяющий центральные компоненты игры (например, создание окна отображения и его настройки, создание ввода с клавиатуры, создание счетчика времени и игровых объектов, имеющих определённый набор компонентов для отображения и корректного поведения на сцене) от других логических частей игры. Игровой движок должен в полной мере поддерживать разработку основных механик игр жанра аркада и симулятор. В данном проекте требуются технологии перемещения в 2*D*-пространстве, отображения 2*D*-графики, анимации, распознавание столкновений твердых тел и «триггеров», а также симуляции физики и физических взаимодействий. В данном проекте роль игрового движка выполняет *Unity*, который способен выполнить все поставленные задачи.

Игровой движок предполагает определенный алгоритм работы: подготовка ресурсов для корректной работы приложения, циклический рендеринга объектов на протяжении всей игры и освобождение ресурсов перед закрытием. *Unity* берёт на себя всю работу по отрисовке объектов сцены на экране, подготовке и очистке ресурсов, симуляции физики и распознавание столкновений твёрдых тел. Благодаря этому разработчик может сосредоточиться на создании логики самого игрового приложения.

* 1. Сетевое взаимодействие в игровом приложении

Так как игровой процесс предполагает игру для двух пользователей, в которой необходима общая конечная точка для связи друг с другом, выполнять симуляцию физики, обрабатывать вводимые игроками данные, обновлять состояние игры и иметь единый источник истинной информации, в проекте необходима архитектура для взаимодействия игроков в сети клиент-сервер, то есть, данные между устройствами будут передаваться на сервер, где после обработки и синхронизации будут передаваться обратно клиенту. В данном случае один из игроков будет играть роль хоста, на котором будут проводиться все вычисления, и который будет источником истинной информации. Подключённым клиентам будут отправляться данные об объектах на сцене хоста, а сами клиенты будут отправлять вводимые команды игроков своим вертолётам. В качестве передаваемой информации проект сетевого взаимодействия использует данные в бинарном формате.

Для реализации сетевого взаимодействия в игровом приложении необходимо спроектировать несколько классов, которые будут связывать двух игроков между собой и исполнять функции клиента и сервера.

Для организации данных, которые будут передаваться между клиентом и сервером, необходимо создать класс *Packet*, который будет хранить в себе передаваемую информацию, методы записи и чтения этих данных из буфера, а также кодирования данных под определённым номером, чтобы можно было понять, какой пакет с какой информацией пришёл от клиента или сервера.

Со стороны клиента необходимо реализовать класс *Client*, которыйдолжен отвечать за установку соединения с сервером по протоколу *TCP*, отправлять данные на сервер и обрабатывать приходящие пакеты. Класс *ClientHandle* должен отвечать за обработку приходящих от сервера пакетов для обновления состояния игровых объектов на сцене. Для реализации отправки данных об действии клиента на сервер необходимо спроектировать класс *ClientSend*, в котором будет реализованы функции отправки конкретных пакетов о действии клиента. Для обновления состояния игровых объектов на сцене необходимо создать класс *GameManager*, который будет хранить список всех объектов сцены, а также реализовывать функционал манипуляций с этими объектами.

Со стороны сервера необходимо создать класс *NetworkManager*, который будет реализовывать функционал запуска сервера при загрузке уровня, создание игроков на сцене, старт и рестарт игры, изменение состояния игры и пересылка данных об этих изменениях клиенту. Далее для проекта необходим класс *Server*, который будет реализовывать функционал создания сервера *TCP*, хранить подключённых клиентов и принимать приходящие подключения. Класс *ServerClient* необходим для реализации подключённого на сервере игрока, реализации отправки данных клиенту и обрабатывать приходящие пакеты. Класс *ServerHandle* должен отвечать за обработку приходящих от клиента пакетов для обновления состояния игровых объектов на сцене. Класс *ServerSend* будет реализовывать функционал отправки пакетов клиенту об состоянии игровых объектов на сцене.

* 1. Программная реализация игрового приложения

Перед началом работы над проектом в среду разработки *Visual* *Studio* загружаем необходимые пакеты для работы со средой разработки *Unity*. После этого создадим новый проект 2*D* игры в *Unity*, чтобы начать работу над проектом. Далее необходимо установить дополнительные пакеты *Unity* для упрощения разработки игры. После данных манипуляций можно приступить к разработке классов игрового приложения.

При реализации игрового приложения был создан ряд классов, реализующий функции игровых объектов.

Класс *Bomb* (Приложение А*,* код программы *Bomb.cs*). Данный класс создан для реализации снаряда типа бомбы на стороне хоста, который будет наносить большой урон, иметь низкую скорость и возможность увеличить скорость полёта данного снаряда. Данный класс наследуется от абстрактного класса *Projectile*.

Класс *Bullet* (Приложение А*,* код программы *Bullet.cs*). Данный класс создан для реализации снаряда типа пуля на стороне хоста, который будет наносить средний урон, иметь среднюю скорость и возможность увеличить урон данного снаряда. Данный класс наследуется от абстрактного класса *Projectile*.

Класс *Projectile* (Приложение А*,* код программы *Projectile.cs*). Данный класс создан для реализации абстрактного снаряда на стороне хоста, который будет хранить в себе список всех снарядов на сцене, присваивать новым снарядам *id* и в котором будет реализована базовая логика всех игровых снарядов: реакция на столкновения, перемещение и отправка своего состояния клиенту каждый тик сервера.

Класс *Rocket* (Приложение А*,* код программы *Rocket.cs*). Данный класс создан для реализации снаряда типа ракета на стороне хоста, который будет наносить средний урон, иметь среднюю скорость и возможность увеличить радиус поражения данного снаряда. Данный класс наследуется от абстрактного класса *Projectile*.

Класс *Bonus* (Приложение А*,* код программы *Bonus.cs*). Данный класс создан для реализации абстрактного бонуса на стороне хоста, в котором будет реализована базовая логика реакции на столкновения.

Класс *DamageBonus* (Приложение А*,* код программы *DamageBonus.cs*). Данный класс создан для реализации бонуса урона для пули на стороне хоста. При подборе данного бонуса вертолётом будет накладываться дополнительный урон на каждый выпущенный снаряд данного типа. Данный класс наследуется от абстрактного класса *Bonus*.

Класс *HealthBonus* (Приложение А*,* код программы *HealthBonus.cs*). Данный класс создан для реализации бонуса здоровья на стороне хоста. При подборе данного бонуса вертолётом будут добавлены дополнительные 10 очков здоровья вертолёту. Данный класс наследуется от абстрактного класса *Bonus*.

Класс *RangeBonus* (Приложение А*,* код программы *RangeBonus.cs*). Данный класс создан для реализации бонуса урона для ракеты на стороне хоста. При подборе данного бонуса вертолётом будет увеличиваться радиус поражения ракетой для каждого выпущенного снаряд данного типа. Данный класс наследуется от абстрактного класса *Bonus*.

Класс *SpeedBonus (*Приложение А*,* код программы *SpeedBonus.cs*). Данный класс создан для реализации бонуса урона для бомбы на стороне хоста. При подборе данного бонуса вертолётом будет увеличиваться скорость полёта бомбы для каждого выпущенного снаряд данного типа. Данный класс наследуется от абстрактного класса *Bonus*.

Класс *BonusSpawnController* (Приложение А*,* код программы *BonusSpawnController.cs*). Данный класс создан для реализации контроллера создания бонусов на сцене. Данный контроллер создаёт бонусы на сцене, отправляет клиенту информацию об их создании и подборе вертолётом. В данном классе имеется метод *OnGetBous*(), который отвечает за удаление подобранного бонуса из уровня, а также за отправку сообщения клиенту о подборе данного бонуса.

Класс *HelicopterController* (Приложение А*,* код программы *HelicopterController.cs*). Данный класс создан для реализации контроллера вертолёта. Контроллер отвечает за управление вертолётом, его стрельбу, получение урона, получение дополнительного здоровья и рывок. В классе имеются различные методы, такие как:

* метод *Initialize*(), который отвечает за инициализацию вертолёта игроком: присвоение ему *id* клиента и имя игрока;
* метод *SetInputs*(), который отвечает за управление вертолётом;
* метод *TakeDamage*(), который отвечает за получение урона;
* метод *TakeHealth*(), который отвечает за пополнение очков здоровья бонусом;
* метод *ChangeWeapon*(), который отвечает за смену выбранного вооружения;
* метод *Shoot*(), который отвечает за стрельбу;
* метод *MakeDash*(), который отвечает за кратковременное ускорение вертолёта.

Класс *HostPlayerController* (Приложение А*,* код программы *HostPlayerController.cs*). Данный класс создан для реализации контроллера игрока на стороне хоста. Контроллер отвечает за считывание ввода пользователя-хоста и управление его вертолётом.

Класс *PlayerController* (Приложение А*,* код программы *PlayerController.cs*). Данный класс создан для реализации контроллера игрока на стороне клиента. Контроллер отвечает за считывание ввода пользователя-клиента и отправку ввода пользователя на сервер.

Класс *BonusManager* (Приложение А*,* код программы *BonusManager.cs*). Данный класс предоставляет доступ к созданным на уровне бонусам для их удаления и добавления новых.

Класс *GameStateManager* (Приложение А*,* код программы *GameStateManager.cs*). Данный класс предоставляет доступ к состоянию игры и позволяет изменять статус игры. От статуса игры зависит поведение игроков. В данном классе имеется метод *ChangeGameState*(), который отвечает за изменение статуса игры.

Класс *HelicopterManager* (Приложение А*,* код программы *HelicopterManager.cs*). Данный класс создан для реализации объекта вертолёт на стороне клиента. Также данный класс предсказывает состояние вертолёта между типами сервера. В данном классе имеется метод *Initialize*(), который отвечает за инициализацию объекта: присвоение ему *id* клиента на сервере, имени игрока и максимального здоровья, и метод *SetHealth*(), который отвечает за установку здоровья данного вертолёта со стороны сервера.

Класс *MainUIManager* (Приложение А*,* код программы *MainUIManager.cs*). Данный класс создан для реализации функционала главного меню игры: кнопок выхода, присоединения к серверу и создания сервера.

Класс *ProjectileManager* (Приложение А*,* код программы *ProjectileManager.cs*). Данный класс создан для реализации объекта снаряда на стороне клиента. Данный класс имеет методы *Initialize*(), который отвечает за инициализацию объекта (присвоение ему *id* снаряда на сервере), и метод *Explode*(), который отвечает за уничтожение снаряда при попадании. Также данный класс предсказывает состояние снаряда в кадре между тиками сервера.

Класс *UIManager* (Приложение А*,* код программы *UIManager.cs*). Данный класс создан для реализации функционала меню игры: кнопок выхода, вывод здоровья игрока и сообщения о конце игры на экран.

Класс *Weapon* (Приложение А*,* код программы *Weapon.cs*). Данный абстрактный класс создан для реализации базового функционала оружия вертолёта.

Класс *WeaponBomb* (Приложение А*,* код программы *WeaponBomb.cs*). Данный класс создан для реализации функционала оружия вертолёта типа бомба. Данный класс имеет метод *Shoot*(), который отвечает за создание снаряда типа бомба при стрельбе с определёнными характеристиками. Данный класс наследуется от абстрактного класса *Weapon*.

Класс *WeaponBullet* (Приложение А*,* код программы *WeaponBullet.cs*). Данный класс создан для реализации функционала оружия вертолёта типа пуля. Данный класс имеет метод *Shoot*(), который отвечает за создание снаряда типа пуля при стрельбе с определёнными характеристиками. Данный класс наследуется от абстрактного класса *Weapon*.

Класс *WeaponRocket* (Приложение А*,* код программы *WeaponRocket.cs*). Данный класс создан для реализации функционала оружия вертолёта типа ракета. Данный класс имеет метод *Shoot*(), который отвечает за создание снаряда типа ракета при стрельбе с определёнными характеристиками. Данный класс наследуется от абстрактного класса *Weapon*.

Класс *Ground* (Приложение А*,* код программы *Ground.cs*). Данный класс создан для реализации функционала поверхности земли. При столкновении с землёй вертолёт сразу уничтожается и раунд завершается.

Класс *Platform* (Приложение А*,* код программы *Platform.cs*). Данный класс создан для реализации функционала платформы-препятствия. При столкновении с платформой вертолёту наносится определённое количество урона.

Класс *Client* (Приложение А*,* код программы *Client.cs*). Данный класс создан для реализации *TCP*-клиента, подключение к серверу, отправки данных на сервер и обработку приходящих данных. Данный класс имеет методы *ConnectToServer*(), который отвечает за присоединение к серверу, *LeaveServer*(), который отвечает за отсоединение от сервера и загрузку начального меню, и метод *Disconnect*(), который отвечает за отсоединение от сервера.

Класс *ClientHandle* (Приложение А*,* код программы *ClientHandle.cs*). Данный класс создан как обработчик пакетов с сервера. В классе имеются различные методы, такие как:

* метод *Welcome*(), который отвечает за обработку пакета-приветствия от сервера. В данном пакете находится присвоенное клиенту на сервере *id* и сообщение от сервера;
* метод *SpawnPlayer*(), который отвечает за обработку пакета, в котором находятся данные о создании нового вертолёта на сцене;
* метод *PlayerPosition*(), который отвечает за обработку пакета, в котором находятся данные о позиции вертолёта на сцене;
* метод *PlayerRotation*(), который отвечает за обработку пакета, в котором находятся данные о повороте вертолёта на сцене;
* метод *BonusSpawned*(), который отвечает за обработку пакета, в котором находятся данные о создании нового бонуса на сцене;
* метод *BonusPickedUp*(), который отвечает за обработку пакета, в котором находятся данные о подборе бонуса игроком;
* метод *PlayerHealth*(), который отвечает за обработку пакета, в котором находятся данные о показателях здоровья определённого вертолёта;
* метод *SpawnProjectile*(), который отвечает за обработку пакета, в котором находятся данные о создании нового снаряда на сцене;
* метод *ProjectilePosition*(), который отвечает за обработку пакета, в котором находятся данные о состоянии снаряда на сцене;
* метод *ProjectileExplode*(), который отвечает за обработку пакета, в котором находятся данные о столкновении снаряда с объектом на сцене;
* метод *ChangeGameState*(), который отвечает за обработку пакета, в котором находятся данные о смене режима игры.

Класс *ClientSend* (Приложение А*,* код программы *ClientSend.cs*). Данный класс создан для реализации функционала отправки пакетов от клиента серверу. В классе имеются различные методы, такие как:

* метод *WelcomeReceived*(), который отвечает за отправку ответного пакета на приветствие сервера, в котором содержится *id* клиента и имя игрока;
* метод *PlayerMovement*(), который отвечает за отправку пакета с информацией о движении игрока на стороне клиента;
* метод *PlayerShoot*(), который отвечает за отправку пакета с информацией о выстреле игрока со стороны клиента;
* метод *PlayerChangeWeapon*(), который отвечает за отправку пакета с информацией о смене вооружения вертолёта со стороны клиента;
* метод *PlayerDash*(), который отвечает за отправку пакета с информацией о использовании рывка игроком со стороны клиента.

Класс *GameManager* (Приложение А*,* код программы *GameManager.cs*). Данный класс создан для предоставления доступа к объектам на сцене на стороне клиента. В этом классе хранятся списки игроков и снарядов на сцене. В классе имеются различные методы, такие как:

* метод *SpawnPlayer*(), который отвечает за создание объекта-игрока на сцене со стороны клиента;
* метод *SpawnBonus*(), который отвечает за создание бонуса на сцене клиента
* метод *SpawnProjectile*(), который отвечает за создание снаряда на сцене клиента;
* метод *GameStateChanged*(), который отвечает за изменение статуса игры.

Класс *NetworkManager* (Приложение А*,* код программы *NetworkManager.cs*). Данный класс создан для реализации функции запуска сервера и начала игры. Данный класс имеет методы *StartGame*(), который отвечает за начало игры, и *RestartGame*(), который отвечает за перезапуск игры.

Класс *Server* (Приложение А*,* код программы *Server.cs*). Данный класс создан для реализации функции *TCP-*сервера в приложении и принятие подключений к нему. Данный класс имеет методы *Start*(), который отвечает за создание сервера и начало прослушивания подключений, и *Stop*(), который отвечает за закрытие сервера.

Класс *ServerClient* (Приложение А*,* код программы *ServerClient.cs*). Данный класс создан для реализации функции подключённого к серверу *TCP-*клиента в приложении, отправка ему и принятие от него данных и их обработку. Данный класс имеет методы *SendIntoGame*(), который вызывается при получении ответа на приветствие сервера от клиента и инициирует его игрока, и *Disconnect*(), который отвечает за отключение клиента.

Класс *ServerHandle* (Приложение А*,* код программы *ServerHandle.cs*). Данный класс представляет собой обработчик приходящих от клиента пакетов. В классе имеются различные методы, такие как:

* метод *WelcomeReceived*(), который отвечает за обработку ответа на приветствие сервера и добавляет подключившегося клиента как нового игрока;
* метод *PlayerMovement*(), который отвечает за обработку пакета от клиента, в котором содержится информация о его передвижении;
* метод *PlayerShoot*(), который отвечает за обработку пакета от клиента, в котором содержится информация о выстреле данного игрока;
* метод *PlayerChangeWeapon*(), который отвечает за обработку пакета с информацией о смене вооружения игрока;
* метод *PlayerDash*(), который отвечает за обработку пакета с информацией о применённом игроком ускорении.

1. верификация и апробация игрового приложения «битва на вертолётах»
   1. Принцип работы игрового приложения

При запуске игрового приложения запускается движок *Unity* на котором открывается стартовая сцена, на которой появляется необходимый для работы приложения объект – *ThreadManager* (Приложение А, код программы для *ThreadManager.cs*), после чего открывается сцена главного меню игры. В меню, которое появится на экране, игрок может выбрать несколько вариантов действий с игрой:

* начать играть;
* выйти из игры.

При нажатии на кнопку «*Exit*» произойдёт выход из игрового приложения с помощью метода *OnExitButton*() (Приложение А, код программы для *MainUIManager.cs*).

При нажатии на кнопку «*Play*» с помощью метода *OnPlayButton*() (Приложение А, код программы для *MainUIManager.cs*) откроется меню, в котором игрок может ввести своё имя, создать сервер или присоединиться к уже существующему серверу, введя его *IP* и порт. При нажатии на кнопку «*Host Game*» произойдёт вызов метода *OnHostGameButton*() (Приложение А, код программы для *MainUIManager.cs*). игрок создаст сервер с выбранным портом (или портом по стандарту, если игрок ничего не вводил). При нажатии на кнопку «*Join Game*» произойдёт вызов метода *OnJoinGameButton*() (Приложение А, код программы для *MainUIManager.cs*), после которого игрок попытается подключиться к серверу по выбранному *IP* и порту.

После выбора начала игры начнётся загрузка главной сцены с игрой. Если игрок создал свой сервер, то на сцене будут загружены необходимый для этого объект – *NetworkManager*, а сам игрок появится на сцене в виде синего вертолёта и будет ожидать подключения второго игрока. В левом верхнем углу экрана начнёт отображаться здоровье игрока. Когда второй игрок подключится к серверу, то на сцене появится второй вертолёт, но уже красного цвета.

Если игрок выбрал подключиться к серверу, то на сцене появится необходимый для работы игры объект – *GameManager*, а на сцене появятся два вертолёта синего и красного цветов. Игрок управляет синим вертолётом.

После небольшой паузы игроки смогут двигаться и стрелять. Если здоровья игрока падает до нуля в результате столкновения с препятствиями, снарядами или землёй, то на экране появляется сообщение о победе того или иного игрока, а сами игроки теряют управление над их вертолётами. Через некоторое время вертолёты перемещаются на стартовые позиции с полным показателем здоровья. Через некоторое время игроки снова могут управлять вертолётами и они начинают новый раунд.

Игра будет продолжаться до тех пор, пока один из игроков не выйдет из игры. Если выйдет присоединившийся игрок, то хост будет ожидать подключение второго игрока. Если выйдет хост, то подключившийся игрок выйдет в главное меню автоматически. При выходе из игры закрывается движок *Unity* и само приложение.

* 1. Результат верификации игрового приложения

При старте игры в окне приложения отображается главное меню, которое позволяет начать игру или выйти из приложения. Главное меню приложения представлено на рисунке 3.1:



Рисунок 3.1 – Главное меню игры

При нажатии на кнопку «Играть» появляется панель с созданием сервера и подключением к уже существующему. Игрок может ввести определённый порт для своего сервера, к которому будут подключаться другие игроки, или оставить стандартный. Экран приложения меню подключения представлен на рисунке 3.2:



Рисунок 3.2 – Создание сервера и присоединение к игре

При подключении к существующему серверу игрок должен указать *IP*-адрес сервера и порт, если он был изменён хостом на свой. Также в главном меню выводятся подсказки по управлению.

При запуске сервера загружается главная сцена, где игрок появляется в виде синего вертолёта и ожидает подключения второго игрока. На рисунке 3.3 представлен экран с ожидающим хостом.

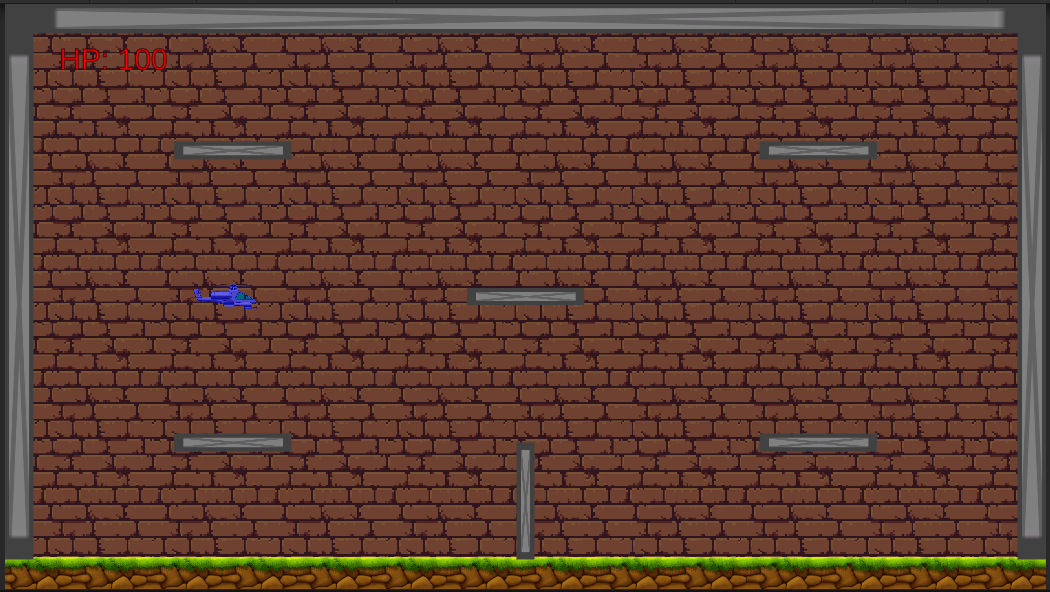


Рисунок 3.3 – Ожидающий хост

При подключении второго игрока на сцене появляется второй вертолёт, но уже красного цвета. Через некоторое время вертолёт начинает спускаться на землю, а игрок может им управлять. Игроку необходимо уничтожить второго игрока, стреляя по нему снарядами разных типов, при этом не столкнувшись с землёй и не потерям всё своё здоровье. На рисунке 3.4 представлен экран с идущей игрой.

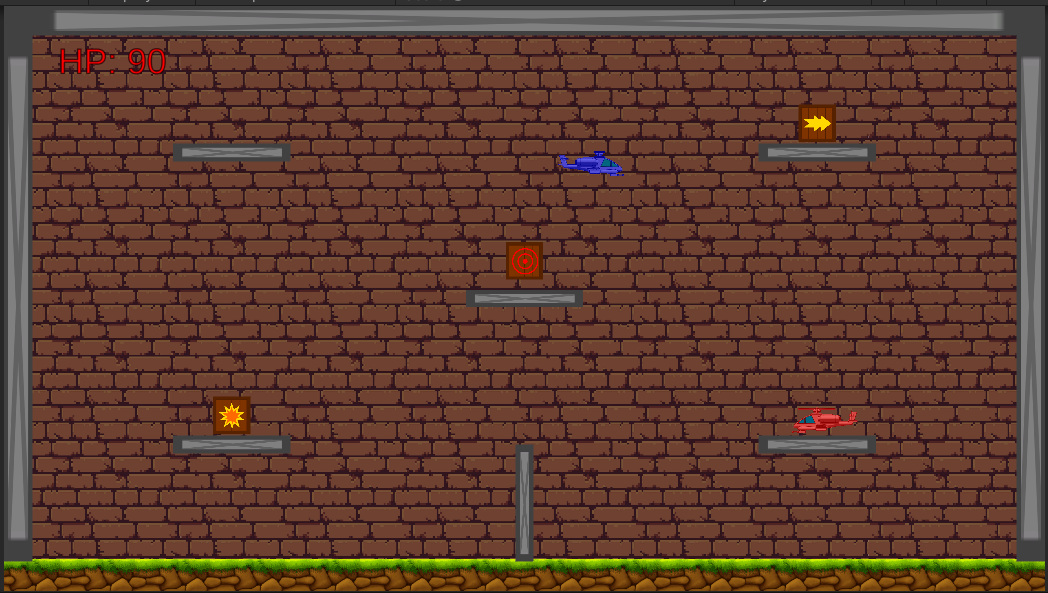


Рисунок 3.4 – Игровой процесс

На сцене будут появляться бонусы, которые будут восстанавливать здоровье вертолёта и временно улучшать характеристики вооружения.

В процессе игры можно подбирать 4 вида бонусов, каждый из которых имеет свой эффект:

* бонус здоровья восстанавливает 20 единиц здоровья игрока;
* бонус урона увеличивает каждый раз урон от пули на 10 единиц;
* бонус скорости увеличивает максимальную скорость полёта бомбы на 10 единиц;
* бонус радиуса поражения один раз увеличивает радиус поражения ракеты.

На рисунке 3.5 представлены создаваемые в сцене бонусы:

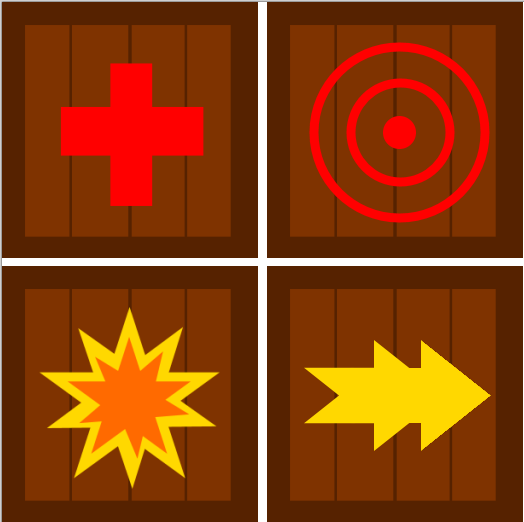


Рисунок 3.5 – Создаваемые бонусы

При столкновении с платформами у игрока будет отниматься 5 очков здоровья, а также он будет отскакивать от них в противоположном его движению направлении. При движении нов вертолёта направлен в сторону движения и поворачивается по часовой стрелке, если движется вправо, и против часовой, если движется влево. Так игрок может регулировать направление полёта снарядов от параллельного земле до 45 градусов к ней.

Если один из игроков потерял все свои очки здоровья, то игра приостанавливается, игроки теряют управление, а на экране появляется надпись с именем победителя.

Данное игровое приложение рассчитано для игры двух игроков на разных устройствах по сети, оно не требует больших вычислительных мощностей и имеет возможность для дальнейших нововведений и улучшений. В игру можно добавить новые визуальные и звуковые эффекты, улучшить модели кораблей, музыку и графику.

ЗАКЛЮЧЕНИЕ

Результатом проделанной работы является игровое приложение жанра аркада и симулятор для игры двух пользователей по сети интернет. Каждый игрок управляет с клавиатуры вертолётом, и их задача заключается в уничтожении друг друга.

При выполнении поставленной задачи был проведён аналитический обзор научной литературы и интернет-источников в области компьютерной графики и информационных технологий, был проведён аналитический обзор игрового жанра «Аркада» и «Симулятор», особенностей разработки многопользовательских игр и их средств разработки. Были отобраны максимально удачные способы решения поставленной задачи, использованы новые идеи для реализации различных геймплей-механик (физика полёта вертолёта и снарядов, улучшение вооружения бонусами и др.), а также реализован понятный и удобный интерфейс.

Выбраны средства разработки программного обеспечения и программные интерфейсы для работы с графикой и моделями, которые способствуют быстрой и качественной разработке игровых приложений.

В процессе подготовки к разработке были рассмотрены преимущества и недостатки среды разработки *Unity* и языка программирования *C*#. В процессе анализа рассматривалась проблема синхронизации игрового времени между пользователями и определялись способы её решения. Затем была определена архитектура, алгоритм работы и способ сетевой коммуникации в игровом приложении. Так как игровое приложение предусмотрено для двух игроков, где важен единый источник истинной информации, было принято решение использовать реализацию клиент-серверного соединения с помощью протокола *TCP*, где один из игроков выступает хостом.

Для достижения принципов ООП игровое приложение было разработано посредством проектирования его архитектуры. Для программной реализации приложения были использованы среда разработки *Unity* 2021, *Visual Studio* 2022 и язык программирования *C*#.

Было разработана игровое приложение «Бой на вертолётах» в жанре аркадный симулятор, позволяющее игроку сразиться с другим игроком на вертолётах. Были выявлены важные аспекты в проектирования и реализации игрового приложения. Данное игровое приложение рассчитано на аудиторию двух данных жанров, способствует развитию логики, пространственного и тактического мышления и инициативе. Данное игровое приложение отличается малыми системными требованиями, понятным интерфейсом и увлекательностью для пользователя.

Созданное игровое приложение послужит хорошим времяпрепровождением для двух пользователей.

Проект выполнен самостоятельно, проверена в системе «Антиплагиат». Процент оригинальности составляет 86%. Цитирования обозначены ссылками на публикации, указанные в «Списке источников».

СПИСОК ИСТОЧНИКОВ

1. Видеоигра-симулятор | *Wikipedia* [Электронный ресурс] – Режим доступа: *https*://*en.wikipedia.org/wiki/Simulation\_video\_game* – Дата доступа: 10.10.2023.
2. Аркада (жанр) | *Wikipedia* [Электронный ресурс] – Режим доступа: *https*:// *ru.wikipedia.org/wiki/Аркада\_(жанр)* – Дата доступа: 10.10.2023.
3. *Unity* (игровой движок) | *Wikipedia* [Электронный ресурс] – Режим доступа: *https*://*ru.wikipedia.org/wiki/Unity\_(игровой\_движок)* – Дата доступа: 10.10.2023.
4. *CLR* *via* *C*#. Программирование на платформе *Microsoft* .*NET* *Framework* 4.0 на языке *C*# / Джефри Рихтер – СПб.: Питер, 2013. – 896 с.
5. Синхронизация данных в мультиплеерных играх [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/328702/ – Дата доступа: 10.10.2023.
6. *TCP* | *Wikipedia* [Электронный ресурс] – Режим доступа: *https*:// *ru.wikipedia.org/wiki/TCP* – Дата доступа: 10.10.2023.
7. *Unity* в действии/ Джозеф Хокинг – СПб.: Питер, 2019. – 344 с.
8. *Unity* и *С#.* Геймдевотидеидореализации / Бонд Джереми Гибсон – СПб.: Питер, 2019. – 928 с.

ПРИЛОЖЕНИЕ А

(обязательное)

**Исходный код приложения**

**Код программы для класса *Bomb.cs*:**

using UnityEngine;

public class Bomb : Projectile

{

public float MaxSpeed

{

get => \_maxSpeed;

set => \_maxSpeed = value;

}

public float HelicopterSpeedX

{

get => \_helicopterSpeedX;

set => \_helicopterSpeedX = value;

}

private float \_helicopterSpeedX;

protected override void Awake()

{

base.Awake();

}

protected override void Start()

{

\_rigidbody.velocity = \_maxSpeed \* Mathf.Abs(\_helicopterSpeedX) \* transform.right;

}

protected override void FixedUpdate()

{

\_rigidbody.velocity = Vector2.ClampMagnitude(\_rigidbody.velocity, \_maxSpeed);

base.FixedUpdate();

}

}

**Код программы для класса *Bullet.cs*:**

using UnityEngine;

public class Bullet : Projectile

{

public float Damage

{

get => \_damage;

set => \_damage = value;

}

protected override void FixedUpdate()

{

\_rigidbody.velocity = Vector2.ClampMagnitude(\_rigidbody.velocity, \_maxSpeed);

base.FixedUpdate();

}

}

**Код программы для класса *Prpjectile.cs*:**

using System.Collections.Generic;

using UnityEngine;

[RequireComponent (typeof(Rigidbody2D))]

public abstract class Projectile : MonoBehaviour

{

public static Dictionary<int, Projectile> Projectiles = new Dictionary<int, Projectile>();

private static int NextProjectileId = 1;

public int Id;

public Vector2 ProjectileVelocity => \_rigidbody.velocity;

[SerializeField] protected float \_maxSpeed;

[SerializeField] protected float \_damage = 10;

protected Rigidbody2D \_rigidbody;

protected virtual void Awake()

{

Id = NextProjectileId;

NextProjectileId++;

Projectiles.Add(Id, this);

\_rigidbody = GetComponent<Rigidbody2D>();

}

protected virtual void Start()

{

\_rigidbody.velocity = \_maxSpeed \* transform.right;

}

protected virtual void FixedUpdate()

{

ServerSend.ProjectilePosition(this);

}

private void OnTriggerEnter2D(Collider2D collision)

{

Explode(collision);

}

private void Explode(Collider2D collision)

{

ServerSend.ProjectileExplode(this);

if (collision.gameObject.TryGetComponent(out HelicopterController controller))

{

controller.TakeDamage(\_damage);

}

Projectiles.Remove(Id);

Destroy(gameObject);

}

}

**Код программы для класса *Rocket.cs*:**

using UnityEngine;

[RequireComponent(typeof(CircleCollider2D))]

public class Rocket : Projectile

{

public float Range

{

get => GetComponent<CircleCollider2D>().radius;

set => GetComponent<CircleCollider2D>().radius = value;

}

}

**Код программы для класса *Bonus.cs*:**

using UnityEngine;

public abstract class Bonus : MonoBehaviour

{

private void OnTriggerEnter2D(Collider2D collision)

{

Debug.Log("Trigger!");

if (collision.TryGetComponent(out HelicopterController controller))

{

BonusSpawnController.Instance.OnGetBous(transform.position);

GetBonus(controller);

Destroy(gameObject);

}

}

protected abstract void GetBonus(HelicopterController helicopter);

}

**Код программы для класса *DamageBonus.cs*:**

using System.Collections;

using UnityEngine;

public class DamageBonus : Bonus

{

[SerializeField] private int \_damageBonus = 10;

[SerializeField] private float \_bonusTime = 5;

protected override void GetBonus(HelicopterController helicopter)

{

WeaponBullet weapon = helicopter.Weapons[WeaponType.Bullet] as WeaponBullet;

weapon.Damage += \_damageBonus;

if (helicopter.BonusTimers.TryGetValue(WeaponType.Bullet, out Coroutine coroutine))

{

helicopter.StopCoroutine(coroutine);

helicopter.BonusTimers[WeaponType.Bullet] = helicopter.StartCoroutine(CleanWeaponCoroutine(\_bonusTime, weapon));

}

else

{

helicopter.BonusTimers.Add(WeaponType.Bullet, helicopter.StartCoroutine(CleanWeaponCoroutine(\_bonusTime, weapon)));

}

}

private IEnumerator CleanWeaponCoroutine(float bonusTime, WeaponBullet weapon)

{

yield return new WaitForSeconds(bonusTime);

weapon.Damage = weapon.DefaultDamage;

}

}

**Код программы для класса *HealthBonus.cs*:**

using UnityEngine;

public class HealthBonus : Bonus

{

[SerializeField] private int \_health = 10;

protected override void GetBonus(HelicopterController helicopter)

{

helicopter.TakeHealth(\_health);

}

}

**Код программы для класса *RangeBonus.cs*:**

using System.Collections;

using UnityEngine;

public class RangeBonus : Bonus

{

[SerializeField] private float \_rangeBonus = .5f;

[SerializeField] private float \_bonusTime = 5;

protected override void GetBonus(HelicopterController helicopter)

{

WeaponRocket weapon = helicopter.Weapons[WeaponType.Rocket] as WeaponRocket;

weapon.Range = weapon.DefaultRange + \_rangeBonus;

if (helicopter.BonusTimers.TryGetValue(WeaponType.Rocket, out Coroutine coroutine))

{

helicopter.StopCoroutine(coroutine);

helicopter.BonusTimers[WeaponType.Rocket] = helicopter.StartCoroutine(CleanWeaponCoroutine(\_bonusTime, weapon));

}

else

{

helicopter.BonusTimers.Add(WeaponType.Rocket, helicopter.StartCoroutine(CleanWeaponCoroutine(\_bonusTime, weapon)));

}

}

private IEnumerator CleanWeaponCoroutine(float bonusTime, WeaponRocket weapon)

{

yield return new WaitForSeconds(bonusTime);

weapon.Range = weapon.DefaultRange;

}

}

**Код программы для класса *SpeedBonus.cs*:**

using System.Collections;

using UnityEngine;

public class SpeedBonus : Bonus

{

[SerializeField] private int \_speedBonus = 10;

[SerializeField] private float \_bonusTime = 5;

protected override void GetBonus(HelicopterController helicopter)

{

WeaponBomb weapon = helicopter.Weapons[WeaponType.Bomb] as WeaponBomb;

weapon.Speed += \_speedBonus;

if (helicopter.BonusTimers.TryGetValue(WeaponType.Bomb, out Coroutine coroutine))

{

helicopter.StopCoroutine(coroutine);

helicopter.BonusTimers[WeaponType.Bomb] = helicopter.StartCoroutine(CleanWeaponCoroutine(\_bonusTime, weapon));

}

else

{

helicopter.BonusTimers.Add(WeaponType.Bomb, helicopter.StartCoroutine(CleanWeaponCoroutine(\_bonusTime, weapon)));

}

}

private IEnumerator CleanWeaponCoroutine(float bonusTime, WeaponBomb weapon)

{

yield return new WaitForSeconds(bonusTime);

weapon.Speed = weapon.DefaultSpeed;

}

}

**Код программы для класса *BonusSpawnController.cs*:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class BonusSpawnController : MonoBehaviour

{

public static BonusSpawnController Instance { get; private set; }

public List<Bonus> BonusList => \_bonusList;

[SerializeField] private List<Bonus> \_bonusList;

[SerializeField] private float \_spawnRate = 10f;

[SerializeField] private float \_spawnDelay = 5f;

private int \_spawnedBonusCount = 0;

private int \_spawnpointsCount;

private void Awake()

{

if (Instance == null)

{

Instance = this;

}

else if (Instance != this)

{

Destroy(this);

}

DontDestroyOnLoad(this);

}

public void OnLevelWasLoaded(int level)

{

if (level == 2)

{

\_spawnpointsCount = BonusManager.Instance.SpawnPointTransforms.Count;

GameStateManager.Instance.OnChangeState += GameStateChanged;

}

}

private void GameStateChanged(GameState state)

{

if (state is GameState.Play)

{

StopAllCoroutines();

StartCoroutine(SpawnBonusCoroutine(\_spawnDelay, \_spawnRate));

}

else

{

StopAllCoroutines();

}

}

private IEnumerator SpawnBonusCoroutine(float spawnDelay, float spawnRate)

{

yield return new WaitForSeconds(spawnDelay);

bool spawned = false;

Vector3 position;

while (true)

{

if (\_spawnedBonusCount != \_spawnpointsCount)

{

while (!spawned)

{

position = BonusManager.Instance.SpawnPointTransforms[Random.Range(0, \_spawnpointsCount)].position;

if (!BonusManager.Instance.Bonuses.ContainsKey(position))

{

\_spawnedBonusCount++;

Bonus bonus = GetRandomBonus();

BonusManager.Instance.Bonuses.Add(position, Instantiate(bonus, position, Quaternion.identity).gameObject);

Debug.Log(\_bonusList.IndexOf(bonus));

ServerSend.BonusSpawned(position, \_bonusList.IndexOf(bonus));

break;

}

}

}

yield return new WaitForSeconds(spawnRate);

}

}

private Bonus GetRandomBonus()

{

return \_bonusList[Random.Range(0, \_bonusList.Count)];

}

public void OnGetBous(Vector3 position)

{

BonusManager.Instance.Bonuses.Remove(position);

\_spawnedBonusCount--;

ServerSend.BonusPickedUp(position);

}

}

**Код программы для класса *HelicopterController.cs*:**

using System;

using System.Collections.Generic;

using UnityEngine;

[RequireComponent (typeof(Rigidbody2D))]

public class HelicopterController : MonoBehaviour

{

public int Id;

public string Username;

public Dictionary<WeaponType, Weapon> Weapons

{

get => \_weapons;

set => \_weapons = value;

}

public Dictionary<WeaponType, Coroutine> BonusTimers

{

get => \_bonusTimers;

set => \_bonusTimers = value;

}

public float Health => \_health;

public Vector2 PlayerSpeed => \_rigidbody.velocity;

[SerializeField] private float \_maxSpeed = 10.0f;

[SerializeField] private float \_deltaSpeed = 10.0f;

[SerializeField] private float \_maxHeals = 100;

private Dictionary<WeaponType, Weapon> \_weapons = new Dictionary<WeaponType, Weapon>();

private Dictionary<WeaponType, Coroutine> \_bonusTimers = new Dictionary<WeaponType, Coroutine>();

private Rigidbody2D \_rigidbody;

private WeaponType \_currentWeapon = WeaponType.Bullet;

private float \_deltaX, \_deltaY, \_lastShootTime = 0;

private float \_health;

private bool \_isLeftDirection;

[SerializeField] private float \_deshForse = 2f;

[SerializeField] private float \_staminaRecoveryRate = 20f;

private float \_maxStamina = 100f;

private float \_currentStamina;

public void Initialize(int id, string username)

{

Id = id;

Username = username;

}

private void Awake()

{

\_rigidbody = GetComponent<Rigidbody2D>();

\_health = \_maxHeals;

\_weapons.Add(WeaponType.Bullet, GetComponent<WeaponBullet>());

\_weapons.Add(WeaponType.Rocket, GetComponent<WeaponRocket>());

\_weapons.Add(WeaponType.Bomb, GetComponent<WeaponBomb>());

}

private void Start()

{

GameStateManager.Instance.OnChangeState += GameStateShanged;

}

private void FixedUpdate()

{

Move();

\_currentStamina += \_staminaRecoveryRate \* Time.fixedDeltaTime;

\_currentStamina = Mathf.Clamp(\_currentStamina, 0f, \_maxStamina);

}

public void SetInputs(float deltaX, float deltaY)

{

\_deltaX = deltaX;

\_deltaY = deltaY;

}

public void TakeDamage(float damage)

{

if (\_health > 0)

{

SetHealth(\_health - damage);

}

}

public void TakeHealth(int health)

{

SetHealth(\_health + health);

}

public void ChangeWeapon(int index)

{

int current = (int)\_currentWeapon + index;

if (current < 0)

{

current = Enum.GetNames(typeof(WeaponType)).Length - 1;

}

else if (current >= Enum.GetNames(typeof(WeaponType)).Length)

{

current = 0;

}

\_currentWeapon = (WeaponType)current;

Debug.Log(\_currentWeapon);

}

public void Shoot()

{

if (\_lastShootTime <= Time.time)

{

\_lastShootTime = Time.time + \_weapons[\_currentWeapon].ReloadingTime;

ServerSend.SpawnProjectile(\_weapons[\_currentWeapon].Shoot(transform.position, transform.rotation), (int)\_currentWeapon);

}

}

public void MakeDash()

{

if (\_currentStamina == \_maxStamina && GameStateManager.Instance.State is GameState.Play)

{

\_rigidbody.AddForce(\_rigidbody.velocity.normalized \* \_deshForse, ForceMode2D.Impulse);

\_currentStamina = 0;

}

}

private void Move()

{

if (GameStateManager.Instance.State == GameState.Play)

{

\_rigidbody.velocity += new Vector2(\_deltaX, \_deltaY) \* \_deltaSpeed \* Time.deltaTime;

\_rigidbody.velocity = Vector2.ClampMagnitude(\_rigidbody.velocity, \_maxSpeed);

}

\_isLeftDirection = Id == 0 ? \_rigidbody.velocity.x < 0 : \_rigidbody.velocity.x <= 0;

if (\_isLeftDirection)

{

transform.rotation = Quaternion.Euler(new Vector3(0, 180, 45 \* \_deltaX));

}

else

{

transform.rotation = Quaternion.Euler(new Vector3(0, 0, -45 \* \_deltaX));

}

ServerSend.PlayerPosition(this);

ServerSend.PlayerRotation(this);

}

private void GameStateShanged(GameState state)

{

switch (state)

{

case GameState.Start:

\_rigidbody.constraints = RigidbodyConstraints2D.FreezeAll;

SetHealth(\_maxHeals);

if (Id == 0)

{

\_rigidbody.position = NetworkManager.Instance.HostSpawnPosition.position;

}

else

{

\_rigidbody.position = NetworkManager.Instance.PlayerSpawnPosition.position;

}

break;

case GameState.Play:

\_rigidbody.constraints = RigidbodyConstraints2D.FreezeRotation;

\_rigidbody.WakeUp();

break;

case GameState.End:

\_rigidbody.constraints = RigidbodyConstraints2D.FreezeAll;

break;

}

}

private void SetHealth(float health)

{

\_health = health;

ServerSend.PlayerHealth(this);

if (\_health <= 0)

{

NetworkManager.Instance.RestartGame();

}

if (Id == 0)

{

UIManager.Instance.HealthText.text = "HP: " + \_health.ToString("0.#");

}

}

}

**Код программы для класса *HostPlayerController.cs*:**

using UnityEngine;

public class HostPlayerController : MonoBehaviour

{

public HelicopterController controller;

private float \_deltaX, \_deltaY;

private void Awake()

{

controller = GetComponent<HelicopterController>();

}

private void Update()

{

if (Input.GetKeyDown(KeyCode.Q))

{

controller.ChangeWeapon(-1);

}

else if (Input.GetKeyDown(KeyCode.E))

{

controller.ChangeWeapon(1);

}

if (Input.GetKeyDown(KeyCode.Space))

{

controller.MakeDash();

}

if (GameStateManager.Instance.State is GameState.Play)

{

if (Input.GetButtonDown("Fire1"))

{

controller.Shoot();

}

\_deltaX = Input.GetAxis("Horizontal");

\_deltaY = Input.GetAxis("Vertical");

controller.SetInputs(\_deltaX, \_deltaY);

}

}

}

**Код программы для класса *PlayerController.cs*:**

using UnityEngine;

public class PlayerController : MonoBehaviour

{

float \_deltaX, \_deltaY;

bool \_isPlayerPressedQ, \_isPlayerPressedE, \_isPlayerShoot, \_isPlayerDash;

private void Update()

{

if (Input.GetKeyDown(KeyCode.Q))

{

\_isPlayerPressedQ = true;

\_isPlayerPressedE = false;

}

else if (Input.GetKeyDown(KeyCode.E))

{

\_isPlayerPressedQ = false;

\_isPlayerPressedE = true;

}

if (GameStateManager.Instance.State is GameState.Play)

{

if (Input.GetKeyDown(KeyCode.Mouse0))

{

\_isPlayerShoot = true;

//ClientSend.PlayerShoot();

}

if (Input.GetKeyDown(KeyCode.Space))

{

\_isPlayerDash = true;

//ClientSend.PlayerDash();

}

\_deltaX = Input.GetAxis("Horizontal");

\_deltaY = Input.GetAxis("Vertical");

}

}

private void FixedUpdate()

{

if (GameStateManager.Instance.State == GameState.Play)

{

SendInputToServer();

}

}

private void SendInputToServer()

{

ClientSend.PlayerMovement(\_deltaX, \_deltaY);

if (\_isPlayerPressedQ)

{

\_isPlayerPressedQ = false;

ClientSend.PlayerChangeWeapon(-1);

}

else if (\_isPlayerPressedE)

{

\_isPlayerPressedE = false;

ClientSend.PlayerChangeWeapon(1);

}

if (\_isPlayerShoot)

{

\_isPlayerShoot = false;

ClientSend.PlayerShoot();

}

if (\_isPlayerDash)

{

\_isPlayerDash = false;

ClientSend.PlayerDash();

}

}

}

**Код программы для класса *BonusManager.cs*:**

using System.Collections.Generic;

using UnityEngine;

public class BonusManager : MonoBehaviour

{

public static BonusManager Instance { get; private set; }

public List<Transform> SpawnPointTransforms => \_spawnPointTransforms;

public Dictionary<Vector3, GameObject> Bonuses = new Dictionary<Vector3, GameObject>();

[SerializeField] private List<Transform> \_spawnPointTransforms;

private void Awake()

{

if (Instance == null)

{

Instance = this;

}

else if (Instance != this)

{

Destroy(this);

}

}

}

**Код программы для класса *GameStateManager.cs*:**

using UnityEngine;

public class GameStateManager : MonoBehaviour

{

public static GameStateManager Instance { get; private set; }

public GameState State => \_state;

public delegate void GameStateDelegate(GameState state);

public event GameStateDelegate OnChangeState;

[SerializeField]

private GameState \_state = GameState.Start;

private void Awake()

{

if (Instance == null)

{

Instance = this;

}

else if (Instance != this)

{

Destroy(this);

}

}

public void ChangeGameState(GameState state)

{

\_state = state;

Debug.Log($"State: {\_state}");

OnChangeState?.Invoke(\_state);

}

}

**Код программы для класса *HelicopterManager.cs*:**

using UnityEngine;

public class HelicopterManager : MonoBehaviour

{

public int Id;

public string Username;

public float Health;

public float MaxHealth;

public Vector2 PlayerSpeed;

private void Update()

{

transform.position += (Vector3)PlayerSpeed \* Time.deltaTime;

}

public void Initialize(int id, string username)

{

Id = id;

Username = username;

Health = MaxHealth;

}

public void SetHealth(float health)

{

Health = health;

if (Id == Client.Instance.MyId)

{

UIManager.Instance.SetHealth(Health);

}

}

}

**Код программы для класса *MainUIManager.cs*:**

using System;

using TMPro;

using UnityEngine;

using UnityEngine.SceneManagement;

public class MainUIManager : MonoBehaviour

{

public GameObject StartMenu;

public GameObject ConnectionMenu;

public TMP\_InputField UsernameField;

public TMP\_InputField IpField;

public TMP\_InputField PortField;

public NetworkManager NetworkManagerPrefab;

public GameObject ClientManagerPrefab;

public GameObject GameManagerPrefab;

public GameObject BonusSpawnControllerPrefab;

public void OnPlayButton()

{

StartMenu.SetActive(false);

ConnectionMenu.SetActive(true);

}

public void OnExitButton()

{

Application.Quit();

}

public void OnReturnButton()

{

StartMenu.SetActive(true);

ConnectionMenu.SetActive(false);

}

public void OnHostGameButton()

{

NetworkManager manager = Instantiate(NetworkManagerPrefab);

manager.HostName = string.IsNullOrEmpty(UsernameField.text) ? "Host" : UsernameField.text;

Instantiate(BonusSpawnControllerPrefab);

SceneManager.LoadScene(2);

}

public void OnJoinGameButton()

{

try

{

Instantiate(ClientManagerPrefab);

string ip = string.IsNullOrEmpty(IpField.text) ? "127.0.0.1" : IpField.text;

int port = string.IsNullOrEmpty(PortField.text) ? 26950 : int.Parse(PortField.text);

Client.Instance.Username = string.IsNullOrEmpty(UsernameField.text) ? "Client" : UsernameField.text;

SceneManager.LoadScene(2);

Client.Instance.ConnectToServer(ip, port);

Instantiate(GameManagerPrefab);

}

catch (Exception e)

{

Debug.LogError($"Error: {e}");

Cursor.visible = true;

Cursor.lockState = CursorLockMode.None;

SceneManager.LoadScene(1);

}

}

}

**Код программы для класса *ProjectileManager.cs*:**

using UnityEngine;

public class ProjectileManager : MonoBehaviour

{

public int Id;

public Vector2 ProjectileVelocity;

public void Update()

{

transform.position += (Vector3)ProjectileVelocity \* Time.deltaTime;

}

public void Initialize(int id)

{

Id = id;

}

public void Explode(Vector3 position)

{

transform.position = position;

//Instantiate(explosionPrefab, transform.position, Quaternion.identity);

GameManager.Projectiles.Remove(Id);

Destroy(gameObject);

}

}

**Код программы для класса *UIManager.cs*:**

using TMPro;

using UnityEngine;

using UnityEngine.SceneManagement;

public class UIManager : MonoBehaviour

{

public static UIManager Instance;

public TMP\_Text HealthText;

public TMP\_Text EndGameText;

public GameObject MenuPanel;

private void Awake()

{

if (Instance == null)

{

Instance = this;

}

else if (Instance != this)

{

Debug.Log("Instance already exists, destroying object!");

Destroy(this);

}

GameStateManager.Instance.OnChangeState += GameStateShanged;

SetCursoreMode(false);

}

public void Update()

{

if (Input.GetKeyDown(KeyCode.Escape))

{

ToggleCursoreMode();

}

}

public void SetHealth(float hp)

{

HealthText.text = "HP: " + hp;

}

public void OnContinueButton()

{

MenuPanel.SetActive(false);

ToggleCursoreMode();

}

public void OnExitButton()

{

if (Client.Instance != null)

{

Client.Instance.LeaveServer();

Destroy(Client.Instance.gameObject);

Destroy(GameManager.Instance.gameObject);

}

if (NetworkManager.Instance != null)

{

Destroy(NetworkManager.Instance.gameObject);

Destroy(BonusSpawnController.Instance.gameObject);

}

SetCursoreMode(true);

SceneManager.LoadScene(1);

}

public void SetCursoreMode(bool cursoreMode)

{

if (cursoreMode)

{

Cursor.lockState = CursorLockMode.None;

Cursor.visible = true;

}

else

{

Cursor.lockState = CursorLockMode.Locked;

Cursor.visible = false;

}

}

private void ToggleCursoreMode()

{

if (Cursor.lockState == CursorLockMode.None)

{

MenuPanel.SetActive(false);

Cursor.lockState = CursorLockMode.Locked;

Cursor.visible = false;

}

else

{

MenuPanel.SetActive(true);

Cursor.lockState = CursorLockMode.None;

Cursor.visible = true;

}

}

private void GameStateShanged(GameState state)

{

if (state is GameState.End)

{

EndGameText.gameObject.SetActive(true);

}

else

{

EndGameText.gameObject.SetActive(false);

}

}

}

**Код программы для класса *Weapon.cs*:**

using UnityEngine;

public abstract class Weapon : MonoBehaviour

{

public Projectile AmmoPrefab => \_ammoPrefab;

public float ReloadingTime => \_reloadingTime;

[SerializeField] protected Projectile \_ammoPrefab;

[SerializeField] private float \_reloadingTime;

public abstract Projectile Shoot(Vector2 position, Quaternion rotation);

}

**Код программы для класса *WeaponBomb.cs*:**

using UnityEngine;

public class WeaponBomb : Weapon

{

public float DefaultSpeed => \_defaultSpeed;

public float Speed

{

get => \_speed;

set => \_speed = value;

}

[SerializeField] private float \_defaultSpeed;

[SerializeField] private Rigidbody2D \_rigidbody;

private float \_speed;

private void Awake()

{

\_speed = \_defaultSpeed;

}

public override Projectile Shoot(Vector2 position, Quaternion rotation)

{

Bomb bomb = Instantiate((Bomb)\_ammoPrefab, position - (Vector2)transform.up \* .5f, rotation);

bomb.MaxSpeed = \_speed;

bomb.HelicopterSpeedX = \_rigidbody.velocity.x;

return bomb;

}

}

**Код программы для класса *WeaponBullet.cs*:**

using UnityEngine;

public class WeaponBullet : Weapon

{

public int DefaultDamage => \_defaultDamage;

public int Damage

{

get => \_damage;

set => \_damage = value;

}

[SerializeField] private int \_defaultDamage;

private int \_damage;

private void Awake()

{

\_damage = \_defaultDamage;

}

public override Projectile Shoot(Vector2 position, Quaternion rotation)

{

Bullet bullet = Instantiate(\_ammoPrefab, position + (Vector2)transform.right \* .8f, rotation) as Bullet;

bullet.Damage = \_damage;

return bullet;

}

}

**Код программы для класса *WeaponRocket.cs*:**

using System.ComponentModel.Design.Serialization;

using UnityEngine;

public class WeaponRocket : Weapon

{

public float DefaultRange => \_defaultRange;

public float Range

{

get => \_range;

set => \_range = value;

}

[SerializeField] private float \_defaultRange = 0.5f;

private float \_range;

private void Awake()

{

\_range = \_defaultRange;

}

public override Projectile Shoot(Vector2 position, Quaternion rotation)

{

Rocket rocket = Instantiate(\_ammoPrefab, position + (Vector2)transform.right \* 1.5f, rotation) as Rocket;

rocket.Range = \_range;

return rocket;

}

}

**Код программы для *WeaponType.cs*:**

public enum WeaponType

{

Bullet = 0,

Rocket,

Bomb

}

**Код программы для *GameState.cs*:**

public enum GameState

{

Start,

Play,

End

}

**Код программы для класса *Ground.cs*:**

using UnityEngine;

public class Ground : MonoBehaviour

{

private void OnCollisionEnter2D(Collision2D collision)

{

if (collision.gameObject.TryGetComponent(out HelicopterController controller))

{

controller.TakeDamage(controller.Health);

}

}

}

**Код программы для класса *Platform.cs*:**

using UnityEngine;

public class Platform : MonoBehaviour

{

[SerializeField] private float \_damage = 10;

private void OnCollisionEnter2D(Collision2D collision)

{

if (collision.gameObject.TryGetComponent(out HelicopterController controller))

{

controller.TakeDamage(\_damage);

}

}

}

**Код программы для класса *StartSceneController.cs*:**

using UnityEngine;

using UnityEngine.SceneManagement;

public class StartSceneController : MonoBehaviour

{

// Start is called before the first frame update

void Start()

{

SceneManager.LoadScene(1);

}

}

**Код программы для класса *Client.cs*:**

using System.Collections.Generic;

using UnityEngine;

using System.Net.Sockets;

using System;

using UnityEngine.SceneManagement;

public class Client : MonoBehaviour

{

public static Client Instance;

public static int dataBufferSize = 4096;

public string Username = "Player";

public string Ip = "127.0.0.1";

public int Port = 26950;

public int MyId = 0;

public TCP Tcp;

private bool \_isConnected = false;

private delegate void PacketHandler(Packet packet);

private static Dictionary<int, PacketHandler> packetHandlers;

private void OnApplicationQuit()

{

Disconnect();

}

private void Awake()

{

if (Instance == null)

{

Instance = this;

}

else if (Instance != this)

{

Debug.Log("Instance already exists, destroying object!");

Destroy(gameObject);

}

Tcp = new TCP();

DontDestroyOnLoad(gameObject);

}

public void ConnectToServer()

{

InitializeClientData();

\_isConnected = true;

Tcp.Connect();

}

public void ConnectToServer(string ip, int port)

{

Ip = ip;

Port = port;

ConnectToServer();

}

public class TCP

{

public TcpClient Socket;

private NetworkStream \_stream;

private Packet \_receivedData;

private byte[] \_receiveBuffer;

public void Connect()

{

Socket = new TcpClient

{

ReceiveBufferSize = dataBufferSize,

SendBufferSize = dataBufferSize

};

\_receiveBuffer = new byte[dataBufferSize];

Socket.BeginConnect(Instance.Ip, Instance.Port, ConnectCallback, Socket);

}

public void SendData(Packet packet)

{

try

{

if (Socket != null)

{

\_stream.BeginWrite(packet.ToArray(), 0, packet.Length(), null, null);

}

}

catch(Exception e)

{

Debug.Log($"Error sending data to server via TCP: {e}");

}

}

private void ConnectCallback(IAsyncResult result)

{

Socket.EndConnect(result);

if (!Socket.Connected)

{

return;

}

\_stream = Socket.GetStream();

\_receivedData = new Packet();

\_stream.BeginRead(\_receiveBuffer, 0, dataBufferSize, ReceiveCallback, null);

}

private void ReceiveCallback(IAsyncResult result)

{

try

{

int byteLength = \_stream.EndRead(result);

if (byteLength <= 0)

{

//Instance.Disconnect();

Instance.LeaveServer();

return;

}

byte[] data = new byte[byteLength];

Array.Copy(\_receiveBuffer, data, byteLength);

\_receivedData.Reset(HandleData(data));

\_stream.BeginRead(\_receiveBuffer, 0, dataBufferSize, ReceiveCallback, null);

}

catch (Exception e)

{

Debug.Log($"Error receiving TCP data: {e}");

Disconnect();

}

}

private bool HandleData(byte[] data)

{

int packetLength = 0;

\_receivedData.SetBytes(data);

if (\_receivedData.UnreadLength() >= 4)

{

packetLength = \_receivedData.ReadInt();

if (packetLength <= 0)

{

return true;

}

}

while (packetLength > 0 && packetLength <= \_receivedData.UnreadLength())

{

byte[] packetBytes = \_receivedData.ReadBytes(packetLength);

ThreadManager.ExecuteOnMainThread(() =>

{

using(Packet packet = new Packet(packetBytes))

{

int packetId = packet.ReadInt();

packetHandlers[packetId](packet);

}

});

packetLength = 0;

if (\_receivedData.UnreadLength() >= 4)

{

packetLength = \_receivedData.ReadInt();

if (packetLength <= 0)

{

return true;

}

}

}

if (packetLength < 1)

{

return true;

}

return false;

}

private void Disconnect()

{

Instance.Disconnect();

\_stream = null;

\_receivedData = null;

\_receiveBuffer = null;

Socket = null;

}

}

private void InitializeClientData()

{

packetHandlers = new Dictionary<int, PacketHandler>()

{

{(int)ServerPackets.welcome, ClientHandle.Welcome},

{(int)ServerPackets.spawnPlayer, ClientHandle.SpawnPlayer},

{(int)ServerPackets.playerPosition, ClientHandle.PlayerPosition},

{(int)ServerPackets.playerRotation, ClientHandle.PlayerRotation},

{(int)ServerPackets.bonusSpawned, ClientHandle.BonusSpawned},

{(int)ServerPackets.bonusPickedUp, ClientHandle.BonusPickedUp},

{(int)ServerPackets.playerHealth, ClientHandle.PlayerHealth},

{(int)ServerPackets.spawnProjectile, ClientHandle.SpawnProjectile},

{(int)ServerPackets.projectilePosition, ClientHandle.ProjectilePosition},

{(int)ServerPackets.projectileExplode, ClientHandle.ProjectileExplode},

{(int)ServerPackets.changeGameState, ClientHandle.ChangeGameState}

};

Debug.Log("Initialize packets.");

}

public void LeaveServer()

{

if (\_isConnected)

{

Disconnect();

ThreadManager.ExecuteOnMainThread(() =>

{

UIManager.Instance.SetCursoreMode(true);

SceneManager.LoadScene(1);

});

}

}

public void Disconnect()

{

if (\_isConnected)

{

\_isConnected = false;

Tcp.Socket.Close();

Debug.Log("Disconnected from server.");

}

}

}

**Код программы для класса *ClientHandle.cs*:**

using System;

using UnityEngine;

public class ClientHandle : MonoBehaviour

{

public static void Welcome(Packet packet)

{

string msg = packet.ReadString();

int myId = packet.ReadInt();

Debug.Log($"Message from server: {msg}");

Client.Instance.MyId = myId;

ClientSend.WelcomeReceived();

}

public static void SpawnPlayer(Packet packet)

{

int id = packet.ReadInt();

string username = packet.ReadString();

Vector3 position = packet.ReadVector3();

Quaternion rotation = packet.ReadQuaternion();

GameManager.Instance.SpawnPlayer(id, username, position, rotation);

}

public static void PlayerPosition(Packet packet)

{

int id = packet.ReadInt();

Vector3 position = packet.ReadVector3();

Vector2 playerSpeed = packet.ReadVector2();

if (GameManager.Players.ContainsKey(id))

{

GameManager.Players[id].transform.position = position;

GameManager.Players[id].PlayerSpeed = playerSpeed;

}

}

public static void PlayerRotation(Packet packet)

{

int id = packet.ReadInt();

Quaternion rotation = packet.ReadQuaternion();

if (GameManager.Players.ContainsKey(id))

{

GameManager.Players[id].transform.rotation = rotation;

}

}

public static void BonusSpawned(Packet packet)

{

Vector3 position = packet.ReadVector3();

int type = packet.ReadInt();

GameManager.Instance.SpawnBonus(position, type);

}

public static void BonusPickedUp(Packet packet)

{

Vector3 position = packet.ReadVector3();

GameManager.Instance.PickUpBonus(position);

}

public static void PlayerHealth(Packet packet)

{

int id = packet.ReadInt();

float health = packet.ReadFloat();

if (GameManager.Players.ContainsKey(id))

{

GameManager.Players[id].SetHealth(health);

}

}

public static void SpawnProjectile(Packet packet)

{

int projectileId = packet.ReadInt();

Vector3 position = packet.ReadVector3();

Quaternion rotation = packet.ReadQuaternion();

int type = packet.ReadInt();

GameManager.Instance.SpawnProjectile(projectileId, position, rotation, type);

}

public static void ProjectilePosition(Packet packet)

{

int projectileId = packet.ReadInt();

Vector3 position = packet.ReadVector3();

Quaternion rotation = packet.ReadQuaternion();

Vector2 velocity = packet.ReadVector2();

GameManager.Projectiles[projectileId].transform.position = position;

GameManager.Projectiles[projectileId].transform.rotation = rotation;

GameManager.Projectiles[projectileId].ProjectileVelocity = velocity;

}

public static void ProjectileExplode(Packet packet)

{

int projectileId = packet.ReadInt();

Vector3 position = packet.ReadVector3();

if (GameManager.Projectiles.ContainsKey(projectileId))

{

GameManager.Projectiles[projectileId].Explode(position);

}

}

public static void ChangeGameState(Packet packet)

{

int index = packet.ReadInt();

GameState state = (GameState)Enum.ToObject(typeof(GameState), index);

GameStateManager.Instance.ChangeGameState(state);

}

}

**Код программы для класса *ClientSend.cs*:**

using UnityEngine;

public class ClientSend : MonoBehaviour

{

private static void SendTCPData(Packet packet)

{

packet.WriteLength();

Client.Instance.Tcp.SendData(packet);

}

#region Packets

public static void WelcomeReceived()

{

using (Packet packet = new Packet((int)ClientPackets.welcomeReceived))

{

packet.Write(Client.Instance.MyId);

packet.Write(Client.Instance.Username);

SendTCPData(packet);

}

}

public static void PlayerMovement(float deltaX, float deltaY)

{

using (Packet packet = new Packet((int)ClientPackets.playerMovement))

{

packet.Write(deltaX);

packet.Write(deltaY);

SendTCPData(packet);

}

}

public static void PlayerShoot()

{

using (Packet packet = new Packet((int)ClientPackets.playerShoot))

{

SendTCPData(packet);

}

}

public static void PlayerChangeWeapon(int index)

{

using (Packet packet = new Packet((int)ClientPackets.playerChangeWeapon))

{

packet.Write(index);

SendTCPData(packet);

}

}

public static void PlayerDash()

{

using (Packet packet = new Packet((int)ClientPackets.playerDash))

{

SendTCPData(packet);

}

}

#endregion

}

**Код программы для класса *GameManager.cs*:**

using System.Collections.Generic;

using UnityEngine;

public class GameManager : MonoBehaviour

{

public static GameManager Instance;

public static Dictionary<int, HelicopterManager> Players = new Dictionary<int, HelicopterManager>();

public static Dictionary<int, ProjectileManager> Projectiles = new Dictionary<int, ProjectileManager>();

public GameObject PlayerPrefab;

public GameObject EnemyPlayerPrefab;

public List<GameObject> BonusList = new List<GameObject>();

public List<ProjectileManager> ProjectilePrefabs = new List<ProjectileManager>();

private void Awake()

{

if (Instance == null)

{

Instance = this;

}

else if (Instance != this)

{

Debug.Log("Instance already exists, destroying object!");

Destroy(this);

}

DontDestroyOnLoad(gameObject);

}

private void OnDestroy()

{

Players.Clear();

Projectiles.Clear();

}

public void OnLevelWasLoaded(int level)

{

if (level == 2)

{

GameStateManager.Instance.OnChangeState += GameStateChanged;

}

}

public void SpawnPlayer(int id, string username, Vector3 position, Quaternion rotation)

{

GameObject player;

if (id == Client.Instance.MyId)

{

player = Instantiate(PlayerPrefab, position, rotation);

player.gameObject.AddComponent<PlayerController>();

}

else

{

player = Instantiate(EnemyPlayerPrefab, position, rotation);

}

HelicopterManager manager = player.GetComponent<HelicopterManager>();

manager.Initialize(id, username);

Players.Add(id, manager);

}

public void SpawnBonus(Vector3 position, int type)

{

BonusManager.Instance.Bonuses.Add(position, Instantiate(BonusList[type], position, Quaternion.identity));

}

public void PickUpBonus(Vector3 position)

{

if (BonusManager.Instance.Bonuses.ContainsKey(position))

{

Destroy(BonusManager.Instance.Bonuses[position]);

BonusManager.Instance.Bonuses.Remove(position);

}

}

public void SpawnProjectile(int id, Vector3 position, Quaternion rotation, int type)

{

ProjectileManager projectile = Instantiate(ProjectilePrefabs[type], position, rotation);

projectile.Initialize(id);

Projectiles.Add(id, projectile);

}

private void GameStateChanged(GameState state)

{

if (state is GameState.End)

{

if (Players[Client.Instance.MyId].Health > 0)

{

UIManager.Instance.EndGameText.text = "You win!";

UIManager.Instance.EndGameText.color = Color.yellow;

}

else

{

UIManager.Instance.EndGameText.text = Players[0].Username + " win!";

UIManager.Instance.EndGameText.color = Color.gray;

}

}

}

}

**Код программы для класса *NetworkManager.cs*:**

using System.Collections;

using UnityEngine;

public class NetworkManager : MonoBehaviour

{

public static NetworkManager Instance;

public GameObject PlayerPrefab;

public GameObject EnemyPlayerPrefab;

public Transform HostSpawnPosition;

public Transform PlayerSpawnPosition;

public string HostName = "Host";

private void Awake()

{

if (Instance == null)

{

Instance = this;

}

else if (Instance != this)

{

Debug.Log("Instance already exists, destroying object!");

Destroy(gameObject);

}

DontDestroyOnLoad(gameObject);

}

public void OnLevelWasLoaded(int level)

{

if (level == 2)

{

GameStateManager.Instance.OnChangeState += GameStateChanged;

QualitySettings.vSyncCount = 0;

Application.targetFrameRate = 30;

Server.Start(1, 26950);

HostSpawnPosition = GameObject.FindWithTag("HostSpawn").transform;

PlayerSpawnPosition = GameObject.FindWithTag("PlayerSpawn").transform;

Server.HostPlayer = Instantiate(PlayerPrefab, HostSpawnPosition.position, Quaternion.identity).GetComponent<HelicopterController>();

Server.HostPlayer.Initialize(0, HostName);

Server.HostPlayer.gameObject.AddComponent<HostPlayerController>();

GameStateManager.Instance.ChangeGameState(GameState.Start);

}

}

private void OnApplicationQuit()

{

Server.Stop();

}

private void OnDestroy()

{

UIManager.Instance.SetCursoreMode(true);

Server.Stop();

}

public HelicopterController InstantiatePlayer()

{

return Instantiate(EnemyPlayerPrefab, PlayerSpawnPosition.position, Quaternion.identity).GetComponent<HelicopterController>();

}

public void StartGame()

{

StartCoroutine(RespawnCoroutine());

}

public void RestartGame()

{

GameStateManager.Instance.ChangeGameState(GameState.End);

StartCoroutine(RespawnCoroutine());

}

private IEnumerator RespawnCoroutine()

{

yield return new WaitForSeconds(3f);

GameStateManager.Instance.ChangeGameState(GameState.Start);

yield return new WaitForSeconds(3f);

GameStateManager.Instance.ChangeGameState(GameState.Play);

}

private void GameStateChanged(GameState state)

{

ServerSend.ChangeGameState(state);

if (state is GameState.End)

{

if (Server.HostPlayer.Health > 0)

{

UIManager.Instance.EndGameText.text = "You win!";

UIManager.Instance.EndGameText.color = Color.yellow;

}

else

{

UIManager.Instance.EndGameText.text = Server.Clients[1].Player.Username + " win!";

UIManager.Instance.EndGameText.color = Color.gray;

}

}

}

}

**Код программы для класса *Server.cs*:**

using System;

using System.Collections.Generic;

using System.Net.Sockets;

using System.Net;

using UnityEngine;

public class Server

{

public static int MaxPlayers { get; private set; }

public static int Port { get; private set; }

public static Dictionary<int, ServerClient> Clients = new Dictionary<int, ServerClient>();

public delegate void PacketHandler(int fromClient, Packet packet);

public static Dictionary<int, PacketHandler> PacketHandlers;

public static HelicopterController HostPlayer;

private static TcpListener \_tcpListener;

public static void Start(int maxPlayers, int port)

{

MaxPlayers = maxPlayers;

Port = port;

Debug.Log("Starting server...");

InitializeServerData();

\_tcpListener = new TcpListener(IPAddress.Any, Port);

\_tcpListener.Start();

\_tcpListener.BeginAcceptTcpClient(new AsyncCallback(TCPConnectCallback), null);

Debug.Log($"Server started on {Port}.");

}

private static void TCPConnectCallback(IAsyncResult result)

{

try

{

TcpClient client = \_tcpListener.EndAcceptTcpClient(result);

\_tcpListener.BeginAcceptTcpClient(new AsyncCallback(TCPConnectCallback), null);

Debug.Log($"Incoming connection from {client.Client.RemoteEndPoint}...");

for (int i = 1; i <= MaxPlayers; i++)

{

if (Clients[i].Tcp.Socket == null)

{

Clients[i].Tcp.Connect(client);

return;

}

}

Debug.Log($"{client.Client.RemoteEndPoint} failed to connect: Server full!");

}

catch (Exception e)

{

Debug.Log(e);

}

}

private static void InitializeServerData()

{

Clients.Clear();

for (int i = 1; i <= MaxPlayers; i++)

{

Clients.Add(i, new ServerClient(i));

}

PacketHandlers = new Dictionary<int, PacketHandler>()

{

{(int)ClientPackets.welcomeReceived, ServerHandle.WelcomeReceived },

{(int)ClientPackets.playerMovement, ServerHandle.PlayerMovement },

{(int)ClientPackets.playerShoot, ServerHandle.PlayerShoot },

{(int)ClientPackets.playerChangeWeapon, ServerHandle.PlayerChangeWeapon },

{(int)ClientPackets.playerDash, ServerHandle.PlayerDash },

};

Debug.Log("Initialized packets.");

}

public static void Stop()

{

for (int i = 1; i <= MaxPlayers; i++)

{

if (Clients[i].Tcp.Socket != null)

{

Clients[i].Tcp.Disconnect();

}

}

\_tcpListener.Stop();

}

}

**Код программы для класса *ServerClient.cs*:**

using System;

using System.Net.Sockets;

using UnityEngine;

public class ServerClient

{

public static int DataBufferSize = 4096;

public int Id;

public HelicopterController Player;

public TCP Tcp;

public ServerClient(int clientId)

{

Id = clientId;

Tcp = new TCP(Id);

}

public class TCP

{

public TcpClient Socket;

private readonly int \_id;

private NetworkStream \_stream;

private Packet \_receivedData;

private byte[] \_receiveBuffer;

public TCP(int id)

{

\_id = id;

}

public void Connect(TcpClient socket)

{

Socket = socket;

Socket.ReceiveBufferSize = DataBufferSize;

Socket.SendBufferSize = DataBufferSize;

\_stream = Socket.GetStream();

\_receivedData = new Packet();

\_receiveBuffer = new byte[DataBufferSize];

\_stream.BeginRead(\_receiveBuffer, 0, DataBufferSize, ReceiveCallback, null);

ServerSend.Welcome(\_id, "Welcome to the server!");

}

public void SendData(Packet packet)

{

try

{

if (Socket != null)

{

\_stream.BeginWrite(packet.ToArray(), 0, packet.Length(), null, null);

}

}

catch (Exception e)

{

Debug.Log($"Error sending data to player {\_id} via TCP: {e}");

}

}

private void ReceiveCallback(IAsyncResult result)

{

try

{

int byteLength = \_stream.EndRead(result);

if (byteLength <= 0)

{

Server.Clients[\_id].Disconnect();

return;

}

byte[] data = new byte[byteLength];

Array.Copy(\_receiveBuffer, data, byteLength);

\_receivedData.Reset(HandleData(data));

\_stream.BeginRead(\_receiveBuffer, 0, DataBufferSize, ReceiveCallback, null);

}

catch (Exception e)

{

Debug.Log($"Error receiving TCP data: {e}");

Server.Clients[\_id].Disconnect();

}

}

private bool HandleData(byte[] data)

{

int packetLength = 0;

\_receivedData.SetBytes(data);

if (\_receivedData.UnreadLength() >= 4)

{

packetLength = \_receivedData.ReadInt();

if (packetLength <= 0)

{

return true;

}

}

while (packetLength > 0 && packetLength <= \_receivedData.UnreadLength())

{

byte[] packetBytes = \_receivedData.ReadBytes(packetLength);

ThreadManager.ExecuteOnMainThread(() =>

{

using (Packet packet = new Packet(packetBytes))

{

int packetId = packet.ReadInt();

Server.PacketHandlers[packetId](\_id, packet);

}

});

packetLength = 0;

if (\_receivedData.UnreadLength() >= 4)

{

packetLength = \_receivedData.ReadInt();

if (packetLength <= 0)

{

return true;

}

}

}

if (packetLength < 1)

{

return true;

}

return false;

}

public void Disconnect()

{

Socket.Close();

\_stream = null;

\_receivedData = null;

\_receiveBuffer = null;

Socket = null;

}

}

public void SendIntoGame(string playerName)

{

ServerSend.SpawnPlayer(Id, Server.HostPlayer);

Player = NetworkManager.Instance.InstantiatePlayer();

Player.Initialize(Id, playerName);

foreach (ServerClient client in Server.Clients.Values)

{

if (client.Player != null)

{

if (client.Id != Id)

{

ServerSend.SpawnPlayer(Id, client.Player);

}

}

}

foreach (ServerClient client in Server.Clients.Values)

{

if (client.Player != null)

{

ServerSend.SpawnPlayer(client.Id, Player);

}

}

foreach (GameObject bonus in BonusManager.Instance.Bonuses.Values)

{

ServerSend.BonusSpawned(bonus.transform.position, BonusSpawnController.Instance.BonusList.IndexOf(bonus.GetComponent<Bonus>()));

}

NetworkManager.Instance.StartGame();

}

public void Disconnect()

{

try

{

Debug.Log($"{Tcp.Socket.Client.RemoteEndPoint} has disconnected.");

ThreadManager.ExecuteOnMainThread(() =>

{

UnityEngine.Object.Destroy(Player.gameObject);

NetworkManager.Instance.StopAllCoroutines();

GameStateManager.Instance.ChangeGameState(GameState.Start);

Player = null;

});

Tcp.Disconnect();

//ServerSend.PlayerDisconnected(Id); //Нет надобности для двух игроков

}

catch (Exception e)

{

Debug.LogException(e);

}

}

}

**Код программы для класса *ServerHandle.cs*:**

using System;

using UnityEngine;

public class ServerHandle

{

public static void WelcomeReceived(int fromClient, Packet packet)

{

try

{

int clientIdCheck = packet.ReadInt();

string username = packet.ReadString();

Debug.Log($"{Server.Clients[fromClient].Tcp.Socket.Client.RemoteEndPoint} connected successfully and is now player {fromClient}.");

if (fromClient != clientIdCheck)

{

Debug.Log($"Player \"{username}\" (ID: {fromClient}) has assumed the wrong client ID ({clientIdCheck})!");

}

Server.Clients[fromClient].SendIntoGame(username);

}

catch (Exception e)

{

Debug.Log(e);

}

}

public static void PlayerMovement(int fromClient, Packet packet)

{

float deltaX = packet.ReadFloat();

float deltaY = packet.ReadFloat();

Server.Clients[fromClient].Player.SetInputs(deltaX, deltaY);

}

public static void PlayerShoot(int fromClient, Packet packet)

{

Server.Clients[fromClient].Player.Shoot();

}

public static void PlayerChangeWeapon(int fromClient, Packet packet)

{

int index = packet.ReadInt();

Server.Clients[fromClient].Player.ChangeWeapon(index);

}

public static void PlayerDash(int fromClient, Packet packet)

{

Server.Clients[fromClient].Player.MakeDash();

}

}

**Код программы для класса *ServerSend.cs*:**

using UnityEngine;

public class ServerSend

{

#region SendTCPData

private static void SendTCPData(int toClient, Packet packet)

{

packet.WriteLength();

Server.Clients[toClient].Tcp.SendData(packet);

}

private static void SendTCPDataToAll(Packet packet)

{

packet.WriteLength();

for (int i = 1; i <= Server.MaxPlayers; i++)

{

Server.Clients[i].Tcp.SendData(packet);

}

}

private static void SendTCPDataToAll(int exceptClient, Packet packet)

{

packet.WriteLength();

for (int i = 1; i <= Server.MaxPlayers; i++)

{

if (i != exceptClient)

{

Server.Clients[i].Tcp.SendData(packet);

}

}

}

#endregion

#region Pockets

public static void Welcome(int toClient, string msg)

{

using (Packet packet = new Packet((int)ServerPackets.welcome))

{

packet.Write(msg);

packet.Write(toClient);

SendTCPData(toClient, packet);

}

}

/\*public static void PlayerDisconnected(int playerId)

{

using (Packet packet = new Packet((int)ServerPackets.playerDisconnected))

{

packet.Write(playerId);

SendTCPDataToAll(packet);

}

}\*/

public static void SpawnPlayer(int toClient, HelicopterController player)

{

using (Packet packet = new Packet((int)ServerPackets.spawnPlayer))

{

packet.Write(player.Id);

packet.Write(player.Username);

packet.Write(player.transform.position);

packet.Write(player.transform.rotation);

SendTCPData(toClient, packet);

}

}

public static void PlayerPosition(HelicopterController player)

{

using (Packet packet = new Packet((int)ServerPackets.playerPosition))

{

packet.Write(player.Id);

packet.Write(player.transform.position);

packet.Write(player.PlayerSpeed);

SendTCPDataToAll(packet);

}

}

public static void PlayerRotation(HelicopterController player)

{

using (Packet packet = new Packet((int)ServerPackets.playerRotation))

{

packet.Write(player.Id);

packet.Write(player.transform.rotation);

SendTCPDataToAll(packet);

}

}

public static void BonusSpawned(Vector3 position, int bonusType)

{

using (Packet packet = new Packet((int)ServerPackets.bonusSpawned))

{

packet.Write(position);

packet.Write(bonusType);

SendTCPDataToAll(packet);

}

}

public static void BonusPickedUp(Vector3 position)

{

using (Packet packet = new Packet((int)ServerPackets.bonusPickedUp))

{

packet.Write(position);

SendTCPDataToAll(packet);

}

}

public static void PlayerHealth(HelicopterController player)

{

using (Packet packet = new Packet((int)ServerPackets.playerHealth))

{

packet.Write(player.Id);

packet.Write(player.Health);

SendTCPDataToAll(packet);

}

}

public static void SpawnProjectile(Projectile projectile, int type)

{

using (Packet packet = new Packet((int)ServerPackets.spawnProjectile))

{

packet.Write(projectile.Id);

packet.Write(projectile.transform.position);

packet.Write(projectile.transform.rotation);

packet.Write(type);

SendTCPDataToAll(packet);

}

}

public static void ProjectilePosition(Projectile projectile)

{

using (Packet packet = new Packet((int)ServerPackets.projectilePosition))

{

packet.Write(projectile.Id);

packet.Write(projectile.transform.position);

packet.Write(projectile.transform.rotation);

packet.Write(projectile.ProjectileVelocity);

SendTCPDataToAll(packet);

}

}

public static void ProjectileExplode(Projectile projectile)

{

using (Packet packet = new Packet((int)ServerPackets.projectileExplode))

{

packet.Write(projectile.Id);

packet.Write(projectile.transform.position);

SendTCPDataToAll(packet);

}

}

public static void ChangeGameState(GameState state)

{

using (Packet packet = new Packet((int)ServerPackets.changeGameState))

{

int index = (int)state;

packet.Write(index);

SendTCPDataToAll(packet);

}

}

#endregion

}

**Код программы для класса *Packet.cs*:**

using System;

using System.Collections.Generic;

using System.Text;

using UnityEngine;

/// <summary>Sent from server to client.</summary>

public enum ServerPackets

{

welcome = 1,

spawnPlayer,

playerPosition,

playerRotation,

bonusSpawned,

bonusPickedUp,

spawnProjectile,

projectilePosition,

projectileExplode,

playerHealth,

changeGameState

}

/// <summary>Sent from client to server.</summary>

public enum ClientPackets

{

welcomeReceived = 1,

playerMovement,

playerShoot,

playerChangeWeapon,

playerDash

}

public class Packet : IDisposable

{

private List<byte> buffer;

private byte[] readableBuffer;

private int readPos;

/// <summary>Creates a new empty packet (without an ID).</summary>

public Packet()

{

buffer = new List<byte>(); // Intitialize buffer

readPos = 0; // Set readPos to 0

}

/// <summary>Creates a new packet with a given ID. Used for sending.</summary>

/// <param name="\_id">The packet ID.</param>

public Packet(int \_id)

{

buffer = new List<byte>(); // Intitialize buffer

readPos = 0; // Set readPos to 0

Write(\_id); // Write packet id to the buffer

}

/// <summary>Creates a packet from which data can be read. Used for receiving.</summary>

/// <param name="\_data">The bytes to add to the packet.</param>

public Packet(byte[] \_data)

{

buffer = new List<byte>(); // Intitialize buffer

readPos = 0; // Set readPos to 0

SetBytes(\_data);

}

#region Functions

/// <summary>Sets the packet's content and prepares it to be read.</summary>

/// <param name="\_data">The bytes to add to the packet.</param>

public void SetBytes(byte[] \_data)

{

Write(\_data);

readableBuffer = buffer.ToArray();

}

/// <summary>Inserts the length of the packet's content at the start of the buffer.</summary>

public void WriteLength()

{

buffer.InsertRange(0, BitConverter.GetBytes(buffer.Count)); // Insert the byte length of the packet at the very beginning

}

/// <summary>Inserts the given int at the start of the buffer.</summary>

/// <param name="\_value">The int to insert.</param>

public void InsertInt(int \_value)

{

buffer.InsertRange(0, BitConverter.GetBytes(\_value)); // Insert the int at the start of the buffer

}

/// <summary>Gets the packet's content in array form.</summary>

public byte[] ToArray()

{

readableBuffer = buffer.ToArray();

return readableBuffer;

}

/// <summary>Gets the length of the packet's content.</summary>

public int Length()

{

return buffer.Count; // Return the length of buffer

}

/// <summary>Gets the length of the unread data contained in the packet.</summary>

public int UnreadLength()

{

return Length() - readPos; // Return the remaining length (unread)

}

/// <summary>Resets the packet instance to allow it to be reused.</summary>

/// <param name="\_shouldReset">Whether or not to reset the packet.</param>

public void Reset(bool \_shouldReset = true)

{

if (\_shouldReset)

{

buffer.Clear(); // Clear buffer

readableBuffer = null;

readPos = 0; // Reset readPos

}

else

{

readPos -= 4; // "Unread" the last read int

}

}

#endregion

#region Write Data

/// <summary>Adds a byte to the packet.</summary>

/// <param name="\_value">The byte to add.</param>

public void Write(byte \_value)

{

buffer.Add(\_value);

}

/// <summary>Adds an array of bytes to the packet.</summary>

/// <param name="\_value">The byte array to add.</param>

public void Write(byte[] \_value)

{

buffer.AddRange(\_value);

}

/// <summary>Adds a short to the packet.</summary>

/// <param name="\_value">The short to add.</param>

public void Write(short \_value)

{

buffer.AddRange(BitConverter.GetBytes(\_value));

}

/// <summary>Adds an int to the packet.</summary>

/// <param name="\_value">The int to add.</param>

public void Write(int \_value)

{

buffer.AddRange(BitConverter.GetBytes(\_value));

}

/// <summary>Adds a long to the packet.</summary>

/// <param name="\_value">The long to add.</param>

public void Write(long \_value)

{

buffer.AddRange(BitConverter.GetBytes(\_value));

}

/// <summary>Adds a float to the packet.</summary>

/// <param name="\_value">The float to add.</param>

public void Write(float \_value)

{

buffer.AddRange(BitConverter.GetBytes(\_value));

}

/// <summary>Adds a bool to the packet.</summary>

/// <param name="\_value">The bool to add.</param>

public void Write(bool \_value)

{

buffer.AddRange(BitConverter.GetBytes(\_value));

}

/// <summary>Adds a string to the packet.</summary>

/// <param name="\_value">The string to add.</param>

public void Write(string \_value)

{

Write(\_value.Length); // Add the length of the string to the packet

buffer.AddRange(Encoding.ASCII.GetBytes(\_value)); // Add the string itself

}

/// <summary>Adds a Wector2 to the packet.</summary>

/// <param name="\_value">The Wector2 to add.</param>

public void Write(Vector2 \_value)

{

Write(\_value.x);

Write(\_value.y);

}

/// <summary>Adds a Wector3 to the packet.</summary>

/// <param name="\_value">The Wector3 to add.</param>

public void Write(Vector3 \_value)

{

Write(\_value.x);

Write(\_value.y);

Write(\_value.z);

}

/// <summary>Adds a Quaternion to the packet.</summary>

/// <param name="\_value">The Quaternion to add.</param>

public void Write(Quaternion \_value)

{

Write(\_value.x);

Write(\_value.y);

Write(\_value.z);

Write(\_value.w);

}

#endregion

#region Read Data

/// <summary>Reads a byte from the packet.</summary>

/// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>

public byte ReadByte(bool \_moveReadPos = true)

{

if (buffer.Count > readPos)

{

// If there are unread bytes

byte \_value = readableBuffer[readPos]; // Get the byte at readPos' position

if (\_moveReadPos)

{

// If \_moveReadPos is true

readPos += 1; // Increase readPos by 1

}

return \_value; // Return the byte

}

else

{

throw new Exception("Could not read value of type 'byte'!");

}

}

/// <summary>Reads an array of bytes from the packet.</summary>

/// <param name="\_length">The length of the byte array.</param>

/// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>

public byte[] ReadBytes(int \_length, bool \_moveReadPos = true)

{

if (buffer.Count > readPos)

{

// If there are unread bytes

byte[] \_value = buffer.GetRange(readPos, \_length).ToArray(); // Get the bytes at readPos' position with a range of \_length

if (\_moveReadPos)

{

// If \_moveReadPos is true

readPos += \_length; // Increase readPos by \_length

}

return \_value; // Return the bytes

}

else

{

throw new Exception("Could not read value of type 'byte[]'!");

}

}

/// <summary>Reads a short from the packet.</summary>

/// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>

public short ReadShort(bool \_moveReadPos = true)

{

if (buffer.Count > readPos)

{

// If there are unread bytes

short \_value = BitConverter.ToInt16(readableBuffer, readPos); // Convert the bytes to a short

if (\_moveReadPos)

{

// If \_moveReadPos is true and there are unread bytes

readPos += 2; // Increase readPos by 2

}

return \_value; // Return the short

}

else

{

throw new Exception("Could not read value of type 'short'!");

}

}

/// <summary>Reads an int from the packet.</summary>

/// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>

public int ReadInt(bool \_moveReadPos = true)

{

if (buffer.Count > readPos)

{

// If there are unread bytes

int \_value = BitConverter.ToInt32(readableBuffer, readPos); // Convert the bytes to an int

if (\_moveReadPos)

{

// If \_moveReadPos is true

readPos += 4; // Increase readPos by 4

}

return \_value; // Return the int

}

else

{

throw new Exception("Could not read value of type 'int'!");

}

}

/// <summary>Reads a long from the packet.</summary>

/// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>

public long ReadLong(bool \_moveReadPos = true)

{

if (buffer.Count > readPos)

{

// If there are unread bytes

long \_value = BitConverter.ToInt64(readableBuffer, readPos); // Convert the bytes to a long

if (\_moveReadPos)

{

// If \_moveReadPos is true

readPos += 8; // Increase readPos by 8

}

return \_value; // Return the long

}

else

{

throw new Exception("Could not read value of type 'long'!");

}

}

/// <summary>Reads a float from the packet.</summary>

/// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>

public float ReadFloat(bool \_moveReadPos = true)

{

if (buffer.Count > readPos)

{

// If there are unread bytes

float \_value = BitConverter.ToSingle(readableBuffer, readPos); // Convert the bytes to a float

if (\_moveReadPos)

{

// If \_moveReadPos is true

readPos += 4; // Increase readPos by 4

}

return \_value; // Return the float

}

else

{

throw new Exception("Could not read value of type 'float'!");

}

}

/// <summary>Reads a bool from the packet.</summary>

/// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>

public bool ReadBool(bool \_moveReadPos = true)

{

if (buffer.Count > readPos)

{

// If there are unread bytes

bool \_value = BitConverter.ToBoolean(readableBuffer, readPos); // Convert the bytes to a bool

if (\_moveReadPos)

{

// If \_moveReadPos is true

readPos += 1; // Increase readPos by 1

}

return \_value; // Return the bool

}

else

{

throw new Exception("Could not read value of type 'bool'!");

}

}

/// <summary>Reads a string from the packet.</summary>

/// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>

public string ReadString(bool \_moveReadPos = true)

{

try

{

int \_length = ReadInt(); // Get the length of the string

string \_value = Encoding.ASCII.GetString(readableBuffer, readPos, \_length); // Convert the bytes to a string

if (\_moveReadPos && \_value.Length > 0)

{

// If \_moveReadPos is true string is not empty

readPos += \_length; // Increase readPos by the length of the string

}

return \_value; // Return the string

}

catch

{

throw new Exception("Could not read value of type 'string'!");

}

}

/// <summary>Reads a Vector2 from the packet.</summary>

/// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>

public Vector2 ReadVector2(bool \_moveReadPos = true)

{

return new Vector2(ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos));

}

/// <summary>Reads a Vector3 from the packet.</summary>

/// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>

public Vector3 ReadVector3(bool \_moveReadPos = true)

{

return new Vector3(ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos));

}

/// <summary>Reads a Quaternion from the packet.</summary>

/// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>

public Quaternion ReadQuaternion(bool \_moveReadPos = true)

{

return new Quaternion(ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos));

}

#endregion

private bool disposed = false;

protected virtual void Dispose(bool \_disposing)

{

if (!disposed)

{

if (\_disposing)

{

buffer = null;

readableBuffer = null;

readPos = 0;

}

disposed = true;

}

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

}

**Код программы для класса *ThreadManager.cs*:**

using System;

using System.Collections.Generic;

using UnityEngine;

public class ThreadManager : MonoBehaviour

{

private static readonly List<Action> executeOnMainThread = new List<Action>();

private static readonly List<Action> executeCopiedOnMainThread = new List<Action>();

private static bool actionToExecuteOnMainThread = false;

private void Awake()

{

DontDestroyOnLoad(gameObject);

}

private void Update()

{

UpdateMain();

}

/// <summary>Sets an action to be executed on the main thread.</summary>

/// <param name="\_action">The action to be executed on the main thread.</param>

public static void ExecuteOnMainThread(Action \_action)

{

if (\_action == null)

{

Debug.Log("No action to execute on main thread!");

return;

}

lock (executeOnMainThread)

{

executeOnMainThread.Add(\_action);

actionToExecuteOnMainThread = true;

}

}

/// <summary>Executes all code meant to run on the main thread. NOTE: Call this ONLY from the main thread.</summary>

public static void UpdateMain()

{

if (actionToExecuteOnMainThread)

{

executeCopiedOnMainThread.Clear();

lock (executeOnMainThread)

{

executeCopiedOnMainThread.AddRange(executeOnMainThread);

executeOnMainThread.Clear();

actionToExecuteOnMainThread = false;

}

for (int i = 0; i < executeCopiedOnMainThread.Count; i++)

{

executeCopiedOnMainThread[i]();

}

}

}

}

**ПРИЛОЖЕНИЕ Б**

(обязательное)

**Руководство пользователя**

1. Введение.

Разработанное программное приложение предназначено для запуска на ОС не ниже *Windows* 7. Игра обладает минимальным порогом вхождения для пользователей.

Игровое приложение обладает следующим функционалом:

* возможность игры двоих пользователей на разных устройствах по сети;
* игровой уровень в виде уровня с препятствиями и бонусами;
* передвижение игроков;
* генерация бонусов.

Для использования программного приложения пользователь должен быть ознакомлен со следующей информацией:

* настоящим руководством пользователя;
* правилами использования ЭВМ.

1. Назначение и условия применения.

Разработанное программное приложение предназначено для игры двух игроков на разных устройствах по сети. Разработанное игровое приложение предназначено для развития внимания и реакции, а также для развлечения. Также развивает концентрацию и внимание и значительно улучшает память, позволяя запоминать всё большие объёмы информации.

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

* центральный процессор *Intel* *Core* 2 *Duo* *c* тактовой частотой 2.30 МГц или более;
* наличие клавиатуры, мыши и монитора *SVGA* с разрешением не менее 1640 на 750 пикселей;
* операционная система *Windows* 7 и выше;
* 100 Мб оперативной памяти;
* скорость интернет-соединения не ниже 100 килобит в секунду.

1. Подготовка к работе.

Приложение запускается путём открытия файла *Helicopter*.*exe* от имени администратора. Также на компьютере должны быть стабильное подключение к Интернету и установленные драйвера для видеокарты, сетевой карты. Если все инструкции соблюдены, и приложение не выдаёт никаких сообщений об ошибках, значит, программа работает исправно.

1. Описание операций.

При запуске приложения открывается интерфейс, который выводит изображение главного меню с предложением начать игру и выйти из приложения. При нажатии на кнопку «*Play*» перед игроком открывается меню с возможностью создать сервер и присоединиться к уже существующему. Игрок передвигается с помощью клавиш *W*, *A*, *S*, *D,* переключаетвооружение на клавиши *Q* и *E* и стреляет при нажатии на *ЛКМ*.

Первый игрок управляет синим персонажем, а второй игрок – красным. Цель каждого игрока заключается в уничтожении другого игрока. В процессе игры можно передвигаться по горизонтали и по вертикали. Появляющиеся бонусы можно подбирать. Столкновение с препятствиями нанесёт игроку урон, а столкновение с землёй приведёт к проигрышу.

В процессе игры можно подбирать бонусы четырёх типов:

* бонус урона;
* бонус здоровья;
* бонус радиуса поражения;
* бонус скорости.

1. Аварийные ситуации

Чтобы избежать ошибок при использовании программы, необходимо соблюдать порядок действий и условия пользования, описанные в пункте 3, данного руководства пользователя.

В случае непредвиденного «зависания» программы рекомендуется завершить процесс в диспетчере задач и запустить снова.

1. Рекомендации по освоению.

Заранее изучить работу с клавиатурой персонального компьютера. Запомнить расположение клавиш, необходимых для игры.

ПРИЛОЖЕНИЕ В

(обязательное)

**Руководство** **программиста**

1. Назначения и условия применения программы.

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

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

* центральный процессор *Intel* *Core* 2 *Duo* *c* тактовой частотой 2.30 МГц или более;
* наличие клавиатуры, мыши и монитора *SVGA* с разрешением не менее 1640 на 750 пикселей;
* операционная система *Windows* 7 и выше;
* 100 Мб оперативной памяти;
* скорость интернет-соединения не ниже 100 килобит в секунду.

1. Характеристики программы.

Приложение необходимо запускать от имени администратора. Для запуска решения необходима среда разработки *Visual* *Studio* с установленными фреймворком .*NET* и *Unity* 2021.3.32f1.

1. Обращение к программе.

Приложение запускается путём открытия файла *Helicopter*.*exe*, находящегося в папке *Helicopter*.

1. Входные и выходные данные.

В данной программе в качестве входных данных используется ввод с клавиатуры кнопок управления игровым процессом. В качестве выходных выступает окно отображения игры.

1. Сообщения.

При окончании игры выводится победивший игрок. В процессе работы приложения здоровье игрока выводится в информационный лейбл игрока.

ПРИЛОЖЕНИЕ Г

(обязательное)

**Руководство системного программиста**

1. Общие сведения о программе.

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

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

* центральный процессор *Intel* *Core* 2 *Duo* *c* тактовой частотой 2.30 МГц или более;
* наличие клавиатуры, мыши и монитора *SVGA* с разрешением не менее 1640 на 750 пикселей;
* операционная система *Windows* 7 и выше;
* 100 Мб оперативной памяти;
* скорость интернет-соединения не ниже 100 килобит в секунду.

1. Структура программы.

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

1. Настройка программы.

Приложение необходимо запускать от имени администратора. Для запуска решения необходима среда разработки *Visual* *Studio* с установленными фреймворком .*NET* и *Unity* 2021.3.32f1.

1. Дополнительные возможности.

Приложение является узконаправленным и не имеет дополнительных возможностей.

1. Сообщение системному программисту.

Проект игровой логики можно дополнить, добавив новые уровни, бонусы и вооружение.

ПРИЛОЖЕНИЕ Д

(обязательное)

**Схема структуры приложения**