МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования

«Гомельский государственный технический университет

имени П.О.Сухого»

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

направление специальности 1-40 05 01-12 Информационные системы и технологии (в игровой индустрии)

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту

по дисциплине «Программирование сетевых приложений»

на тему: «Сетевое игровое приложение в жанре «Битва героев» с использованием стека протоколов *TCP/IP* и игрового движка *Unity*»

Исполнитель: студент группы ИТИ-41

Гусаров М.Д.

Руководитель: заведующий кафедрой

Курочка К.С.

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

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

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

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

Подписи членов комиссии

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

Гомель 2023

**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ 44

1 ИГРОВЫЕ ПРИЛОЖЕНИЯ И СРЕДСТВА ИХ РАЗРАБОТКИ5 5

1.1 Платформа .*NET*Ошибка: источник перекрёстной ссылки не найденюю...ы 5

1.2 Язык программирования C# 6

1.3 Взаимодействие в сетевых приложениях с помощью *TCP*/*IP* 8

1.4 Игровой движок *Unity* 10

2 АРХИТЕКТУРА ИГРОВОГО ПРИЛОЖЕНИЯ 12

2.1 Исходные данные и предполагаемый результат 12

2.2 Основные этапы разработки игрового приложения 12

2.3 Структура игрового приложения 13

2.4 Паттерны проектирования для реализации игрового приложения 15

3 ВЕРИФИКАЦИЯ И ЭКСПЛУАТАЦИЯ ПРИЛОЖЕНИЯОшибка: источник перекрёстной ссылки не найден 18

3.1 Принципы работы игрового приложенияОшибка: источник перекрёстной ссылки не найден 18

3.2 Модульные тесты игрового приложенияОшибка: источник перекрёстной ссылки не найден 23

ЗАКЛЮЧЕНИЕОшибка: источник перекрёстной ссылки не найденюю 25

Список использованных источниковОшибка: источник перекрёстной ссылки не найден 26

ПРИЛОЖЕНИЕ А Листинг игрового приложенияОшибка: источник перекрёстной ссылки не найден 27

ПРИЛОЖЕНИЕ Б Схема «архитектура приложения»Ошибка: источник перекрёстной ссылки не найден 56

**ВВЕДЕНИЕ**

Игровую индустрию в настоящее время воспринимают в качестве сектора экономики, связанного с разработкой, продвижением и продажей компьютерных игр. На данный момент времени создание видеоигр является одним из наиболее крупных сегментов индустрии развлечений. Масштабы индустрии можно сопоставить с кинематографом, а прибыль за последние несколько лет показывает лучшие показатели.

Актуальность темы данного курсового проекта обусловлена тем, что в наше время большое количество людей, соревнуются между собой, доказывая кто является лучше в той или иной сфере. И компьютерные игры хорошо подходят для соревнований друг с другом в свободное время, причём в них играют не только школьники или студенты. Бизнесмены, инженеры, художники – люди самых разных профессий, встречаются среди игроков. Всех их объединяет общая цель – познавать в виртуальных мирах что-то новое, неисследованное, сражаться друг с другом и тем самым испытать удачу и получить удовольствие, как от игрового процесса, так и от победы над своим оппонентом.

В соответствии с поставленной задачей необходимо:

– разработать классы игровой логики с использованием игрового движка *Unity*;

– разработать классы для сетевого взаимодействия и синхронизации игрового процесса средствами *.NET*;

– собрать игровое приложения для операционной системы *Android.*

Приложение должно иметь возможность масштабирования, т.е. можно добавлять новые сущности без сильного изменения уже написанного кода.

**1 ИГРОВЫЕ ПРИЛОЖЕНИЯ И СРЕДСТВА ИХ РАЗРАБОТКИ**

**1.1 Платформа *.NET***

Платформа .*NET* была официально представлена в 2002 году и почти моментально стала основой современной разработки программного обеспечения. Платформа позволяет использовать большое количество языков программирования (включая *C*#, *VB*.*NET* и *F*#) для взаимодействия друг с другом. Код, написанный на C#, может быть использован при написании приложения на другом языке программирования. В 2016 году был официально запущен .*NET* *Core*. Как и .*NET*, .*NET* *Core* позволяет языкам взаимодействовать между собой (хотя поддерживается ограниченное количество языков). Что более важно, эта новая технология больше не ограничена работой в операционной системе *Windows*, но также может работать (и разрабатываться) на *MacOS* и *Linux*. Независимость от платформы открыла C# для гораздо большего числа разработчиков. В 2020 году был выпущен .*NET* 5, «*Core*» в имени был опущен, с целью показать, что эта версия представляет собой унификацию всего .*NET*.

Платформа .*NET* *Core* – это программная платформа для создания веб-приложений и сервисов на операционных системах *Windows*, *MacOS* и *Linux*, а также приложения *WinForms* и *WPF* в операционных системах *Windows* [1].

Основные функции .*NET* *Core*:

– Обратная совместимость с существующим кодом: существующее программное обеспечение, работающее на .*NET* *Framework* может работать с новыми версиями .*NET* *Core*;

– Поддержка множества языков программирования: приложения на .*NET* Core могут создаваться с использованием разных языков программирования: *C*#, *F*#, *VB*.*NET* и других;

– Общий механизм выполнения, используемый всеми языками .*NET* Core: один из аспектов этого механизма – чётко определённый набор типов, который понимает каждый язык .*NET* *Core*;

– Языковая интеграция: .*NET* *Core* поддерживает наследование, обработку исключений и отладку кода на разных языках программирования. Например, вы можете определить базовый класс в *C*# и расширить этот тип в *Visual* *Basic*;

– Обширная библиотека базовых классов: содержит тысячи предопределённых типов, позволяющие создать свои библиотеки, простые консольные приложения, графические приложения и веб-сайты промышленного уровня;

– Упрощённая модель развёртывания: библиотеки .*NET* *Core* не регистрируются в системный реестр. Платформа .*NET* *Core* позволяет использовать несколько версий фреймворка, гармонично сосуществующих на одной машине;

* Расширенная поддержка командной строки: интерфейс командной строки (*CLI*) это кроссплатформенный набор инструментов для разработки и сборки приложений .*NET Core*;

Основные компоненты .*NET*:

– *Common* *Type* *System* или *CTS*. Спецификация полностью описывает все возможные типы данных и все программные конструкции, поддерживаемые средой выполнения. Определяет, как эти сущности могут взаимодействовать между собой, и описывает, как они должны быть представлены в формате метаданных;

– *Common* *Language* *Runtime* – исполняющая среда для байт-кода *CIL* (*MSIL*), в который компилируются программы, написанные на .*NET*-совместимых языках программирования. Основная задача *CLR* – автоматическое обнаружение, загрузка и управление типами .*NET*. Также среда *CLR* заботится о ряде низкоуровневых деталей – управление памятью, обработка потоков, выполнение разных проверок, связанных с безопасностью;

– *Common Language Specification* является спецификацией, связанной с *CTS*, которая определяет подмножество общих типов и программных конструкций, с которыми могут работать все языки программирования .*NET Core*;

– *Base Class Libraries* представляет набор библиотек базовых классов, доступных для всех пользователей .*NET*. *BCL* не только инкапсулирует различные примитивы, но также обеспечивает поддержку ряда сервисов, необходимых для работы большинству реальных приложений.

* 1. **1.2 Язык программирования *C*#**

C#, как объектно-ориентированный язык, имеет схожую с другими языками программирования функциональность. Этот язык поддерживает наследование, полиморфизм, инкапсуляцию, перегрузку операторов, статическую типизацию. Объектно-ориентированный метод позволяет, разбивая одну большую задачу на множество мелких, создавать гибкие и расширяемые приложения [2].

Понятие строго типизированного языка программирования – это неформальный термин, говорящий о сравнительной способности системы типов языка. Строгого общепринятого определения для данного термина не существует. Для C# система типов достаточно богата, чтобы выражать связи между данными, но, когда речь заходит об отношениях между типами, то тут уже возникают вопросы. Тем не менее, язык *C*# по большей мере является типобезопасным: тип выражения, определённый на этапе компиляции, гарантирован статическим анализом. Исключениями являются явные операции преобразования типа и специальный тип *dynamic*, а также небезопасный код, который может привести к компрометации типобезопасности.

Особенности *C*#:

– Автоматическая сборка мусора;

– Значимые типы, допускающие значение *null*;

– Обнаружение и обработка ошибок;

– Синтаксис языка запросов к источнику данных (*LINQ*);

– Все типы наследуются от базового типа *object*.

Программы на *C*# выполняются в среде .*NET*, виртуальной системе выполнения, называемой *CLR*, и наборе библиотек классов.

Исходный код, написанный на *C*#, компилируется в промежуточный язык (*IL*), соответствующий спецификации *CLI*. Код и ресурсы *IL*, такие как растровые изображения и строки, хранятся в сборке, обычно с расширением .*dll*. Сборка содержит манифест, предоставляющий информацию о типах, версии и культуре сборки [2].

Когда программа *C*# выполняется, сборка загружается в среду *CLR*. *CLR* выполняет компиляцию *Just*-*In*-*Time* (*JIT*) для преобразования кода *IL* в собственные машинные инструкции. *CLR* предоставляет другие службы, связанные с автоматической сборкой мусора, обработкой исключений и управлением ресурсами. Код, выполняемый средой *CLR*, иногда называют «управляемым кодом». «Неуправляемый код» компилируется в собственный машинный язык, предназначенный для конкретной платформы.

На рисунке 1.1 представлена схема компиляции и работы приложения на *C#*.

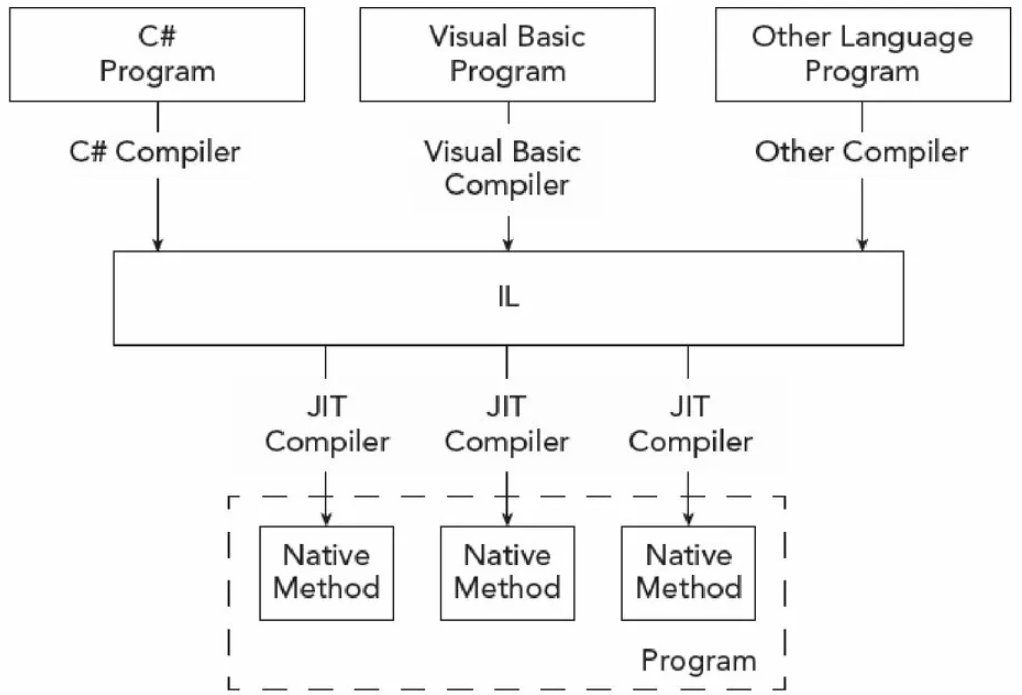


Рисунок 1.1 – Схема компиляции и работы приложения на *C*#

Определяет структуру и поведение любых данных в *C*#. Объявление типа может включать его члены, базовый тип, интерфейсы, которые он реализует, и операции, разрешённые для этого типа. Переменная – это метка, которая ссылается на экземпляр определённого типа.

В C# существуют две разновидности типов: ссылочные типы и типы значений. Переменные типа значений содержат непосредственно данные, а в переменных ссылочных типов хранятся ссылки на нужные данные, которые именуются объектами. Две переменные ссылочного типа могут ссылаться на один и тот же объект, поэтому может случиться так, что операции над одной переменной затронут объект, на который ссылается другая переменная. Каждая переменная типа значения имеет собственную копию данных, и операции над одной переменной не могут затрагивать другую (за исключением переменных параметров *ref*и *out*).

**Идентификатор** – это имя переменной. Идентификатор – это последовательность символов юникода без пробелов. Идентификатор может быть зарезервированным словом *C*#, если он имеет префикс @. При взаимодействии с другими языками в качестве идентификатора может быть полезно использовать зарезервированное слово.

Типы значений в *C*# делятся на простые типы*,*типы перечислений*,*типы структур*,*типы, допускающие значение NULL*,* итипы значений кортежей*.* Ссылочные типы в *C*# подразделяютсянатипы классов*,*типы интерфейсов*,*типы массивовитипы делегатов*.*

Основными понятиями организации в *C*# являются **программы**,**пространстваимён**, **типы**,**члены** и**сборки**. В программе объявляются типы, которые содержат члены. Эти типы можно организовать в пространства имён. Примерами типов являются классы, структуры и интерфейсы. К членам относятся поля, методы, свойства и события. При компиляции программы на *C*# упаковываются в сборки. Сборки обычно имеют расширение .*exe*файла или .*dl*, в зависимости от того, реализуют ли они .*exe*или **библиотеки**соответственно.

**1.3 Взаимодействие в сетевых приложениях c помощью *TCP/IP***

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.

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 [2]. И когда клиент получит данные от сервера, он подменяет их на устройстве. Таким образом, данная концепция минимизирует количество явных зависаний у всех клиентов при низкопроизводительном интернет-соединении.

Клиент-серверное взаимодействие в многопользовательских сетевых приложениях осуществляется с использованием протокола *TCP/IP* который является стандартным протоколом для обмена данными в сети интернет и большинстве сетей. *TCP* (*Transmission Control Protocol*) – это протокол транспортного уровня, который используется для обмена сообщениями между устройствами в сети. Он обеспечивает надежную, упорядоченную и безошибочную доставку данных от источника к получателю, обеспечивая, таким образом, бесперебойное взаимодействие.

В сети файлы не передаются целиком, а дробятся на относительно небольшие сообщения, которые затем передаются другому устройству - получателю, где они собираются обратно в файл. При этом протокол TCP отслеживает передаваемые и принимаемые сегменты, присваивая номера каждому из них. Это обеспечивает упорядоченность передачи данных и возможность восстановления потерянных или поврежденных сегментов [4].

В контексте многопользовательских приложений, протокол *TCP/IP* обычно используется для обмена сообщениями между клиентом (игроком) и сервером (игровым сервером) [5]. Клиенты отправляют запросы на сервер, который в свою очередь обрабатывает эти запросы и отправляет ответы обратно клиентам. Это обеспечивает синхронизацию состояний игры между всеми игроками.

При использовании *TCP/IP* для клиент-серверного взаимодействия в многопользовательских приложениях может возникнуть ряд проблем, таких как задержка или потеря пакетов. Это может быть особенно проблематично для реального времени или быстро меняющихся приложений, таких как онлайн-игры. В таких ситуациях могут быть использованы различные стратегии и техники для управления этими проблемами, включая использование прогнозирования и интерполяции для сглаживания движения и действий в игре.

**1.4 Игровой движок *Unity***

*Unity* – это мощный игровой движок и интегрированная среда разработки, которая позволяет создавать различные типы интерактивных приложений, включая игры, виртуальную реальность и дополненную реальность. *Unity* был выпущен в 2005 году и с тех пор стал одним из самых популярных игровых движков в индустрии.

Игровой движок состоит из следующих компонентов:

**– Физика**: *Unity* предоставляет возможности для работы с физикой, включая работу с твердыми телами и манипулирование ими в *3D* пространстве.

**– Скриптинг**: *Unity* поддерживает программирование игр с использованием скриптинга. Вы можете использовать языки программирования, такие как C# или *JavaScript*, для создания логики и взаимодействия в игре.

**– Аудио**: *Unity* предоставляет возможности для работы с аудио, включая импорт и настройку звука, использование клипов, источников и слушателей.

**– Анимация**: *Unity* обеспечивает возможности для создания анимацией в игре. Вы можете создавать анимации объектов, персонажей и интерфейса пользователя.

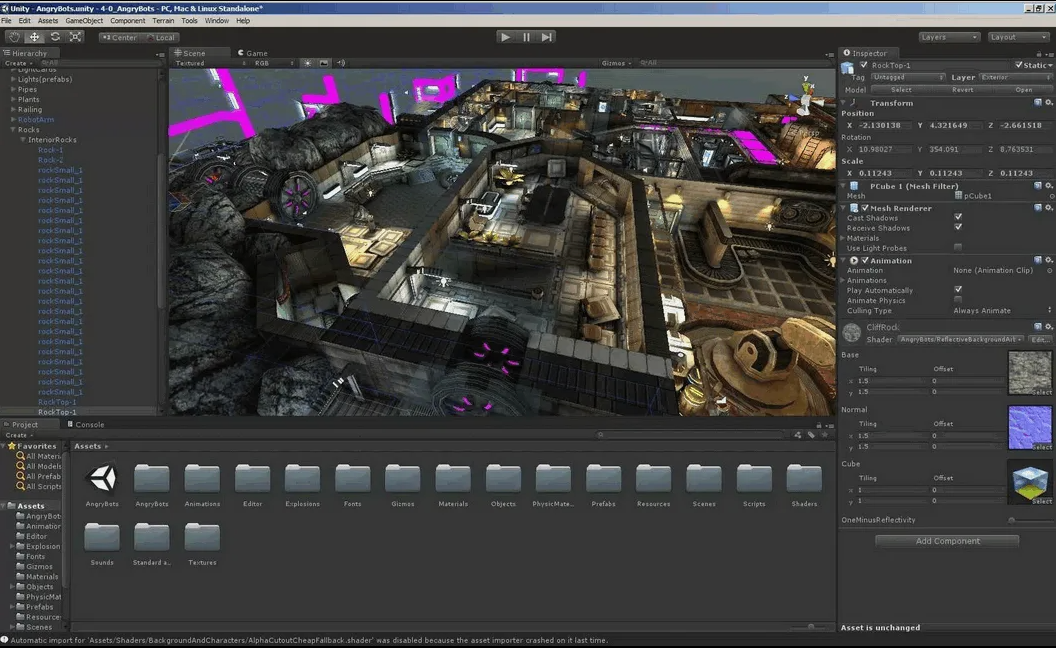
**– *UI***: *Unity* имеет удобную систему пользовательского интерфейса (UI), которая позволяет создавать интерактивные элементы управления и меню в игре.

**– Навигация**: *Unity* предоставляет возможности для работы с навигацией в игре, включая искусственный интеллект и поиск пути

**– Графика**: *Unity* обеспечивает возможности для работы с графикой, включая создание и настройку визуальных эффектов и шейдеров.

*Unity* предоставляет обширную библиотеку встроенных активов и фреймворков, а также поддерживает разработчиков на всех уровнях сложности. Он также предлагает поддержку сообщества и обширный маркетплейс для инструментов и активов, что делает его привлекательным выбором для разработчиков игр [6]. Одна из причин выбрать *Unity* заключается в том, что он бесплатен для использования, если ваш доход от проектов, созданных с его помощью, не превышает установленные лимиты. *Unity* также обладает простым интерфейсом и архитектурой, что делает его легким в использовании даже для начинающих разработчиков.

На рисунке 1.2 представлен интерфейс игрового движка *Unity*.

  
Рисунок 1.2 – Интерфейс игрового движка *Unity*.

*Unity* поддерживает разработку игр на различных платформах, включая ПК, консоли и мобильные устройства. Он также предлагает инструменты для создания 2D и 3D игр, а также поддержку *AR* и *VR*. С помощью Unity вы можете создавать широкий спектр игр и приложений, включая *RPG, survival, strategy,* платформеры, *VR*-проекты и многое другое. *Unity* также предлагает сертификации для разработчиков игр и программистов, что делает его одним из немногих признанных в отрасли аккредитаций для разработки игр в целом.

*Unity* является мощным и гибким инструментом для создания игр и интерактивных приложений различных типов. Он предоставляет широкий набор функций, поддерживает различные платформы и имеет большое сообщество разработчиков, что делает его одним из лидеров в игровой индустрии.

**2** **АРХИТЕКТУРА ИГРОВОГО ПРИЛОЖЕНИЯ**

## **2.1 Исходные данные и предполагаемый результат**

Необходимо разработать сетевое игровое приложение «Битва героев», в игру могут играть только два игрока, каждый из которых выбирает себе персонажа, после чего начинается сражение в котором каждый игрок может передвигаться по сцене и использовать свои способности.

Игра будет разрабатываться на игровом движке *Unity* который берет на себя задачу по отрисовки и сборке игры под разные платформы. С помощью *Unity API* так же будет реализовано взаимодействие игровых объектов друг с другом, используя физику и события игрового движка.

Так как игра по сети требуется реализовать сетевое взаимодействие между клиентами. Сетевое взаимодействие состоит из двух частей: отправка данных по сети и синхронизация данных на клиенте. Для отправки данных будут использоваться средства платформы .*NET* и протокол *TCP.* Синхронизацию на клиенте необходимо разработать под игровое приложение.

В итоге должна получиться игра на двоих игроков, которую можно будет запустить на двух мобильных устройствах под ОС *Android*.

## **2.2 Основные этапы разработки игрового приложения**

В ходе разработки структуры программного обеспечения применяется принцип разделения крупных элементов проекта на более мелкие и легко управляемые задачи, что осуществляется по принципу декомпозиции (пример декомпозиции представлен на рисунке 2.1). Данный принцип представлен в виде иерархического дерева, вершина которого надсистема, а уровни – выделенные системы.

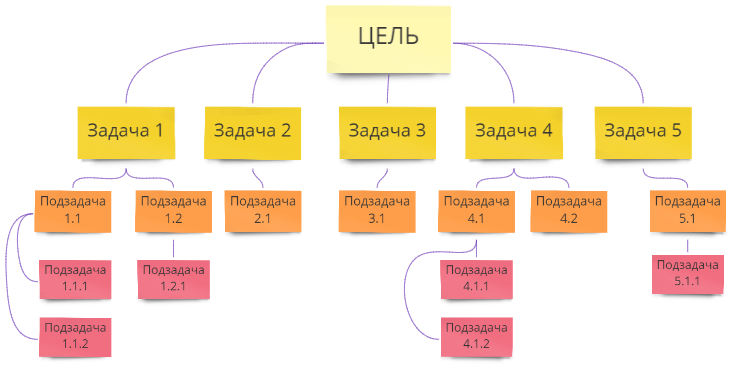


Рисунок 2.1 – Пример принципа декомпозиции

Цели принципа декомпозиции:

* сбора и сохранения информации о структуре приложения, глубина уровней будет варьироваться в зависимости от использования структуры;
* создания листа, который можно будет использовать в процессе тестирования, для этого необходимо достичь очень глубокого разделения.

Для осуществления грамотной разработки и возможности масштабируемости данного игрового приложения необходимо разделить все программные компоненты на части.

Использование принципа декомпозиции позволяет создать модульную схему игрового приложения, что значительно скажется на дальнейшей поддержке кода.

Игровое приложение будет состоять из трёх основных блоков: блок игры с реализованной логикой для игры «Битва героев» с использованием *Unity API* для отрисовки игровых объектов на сцене и их взаимодействия друг с другом, блок с логикой для синхронизации игры по сети, который будет подготавливать данные и управлять синхронизацией игровых объектов на сцене, и последний блок с логикой подключения и отправки данных между клиентами игры.

## **2.3 Структура игрового приложения**

Исходя из принципа декомпозиции, приложения можно разбить на три основных блока: игровая логика, логика синхронизации и сетевая логика.

Для начало необходимо разработать архитектуру сетевого взаимодействия и синхронизации игры. 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.

В качестве архитектуры для игры была выбрана клиент-серверная, где оба клиента смогут отправлять и принимать данные друг другу. Протокол *TCP* позволяет отправлять и принимать данные одновременно. Это необходимо для синхронизации игры, которая будет устроена по следующему принципу: один из игроков возьмет на себя роль сервера (хост), у него будет запущена реальная игровая логика приложения, результат которой будет отправляться второму игроку который будет синхронизировать весь игровой процесс. Но в то же время второй игрок будет отправлять данные о своем вводе (управление персонажем), чтобы у первого игрока в игровой логике производился расчёт финальной игры.

На рисунке 2.2 иллюстрируется схема архитектуры приложения.

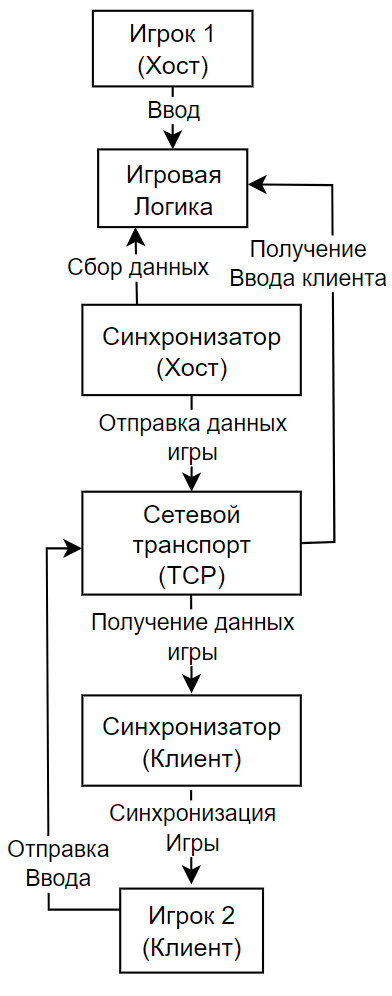


Рисунок 2.2 – Схема архитектуры приложения

Блок игровой логики основан на *Unity API* который помогает реализовать взаимодействие игровых объектов друг с другом. Игровой движок предоставляет возможности, такие как:

* создание игровых компонентов, и предоставление методов, которые будут помогать удобно управлять поведением компонента, а также обрабатывать события, которые случаются с объектом;
* создание и уничтожение игровых объектов, на которых будут хранится компоненты;
* готовые компоненты и структуры, позволяющие описать положение игрового объекта в пространстве, физические компоненты которые помогают создать упрощенную физическую симуляцию, а так же компоненты для отрисовки изображений и трехмерных моделей.

На рисунке 2.3 иллюстрируется схема игровой абстракции движка *Unity*.

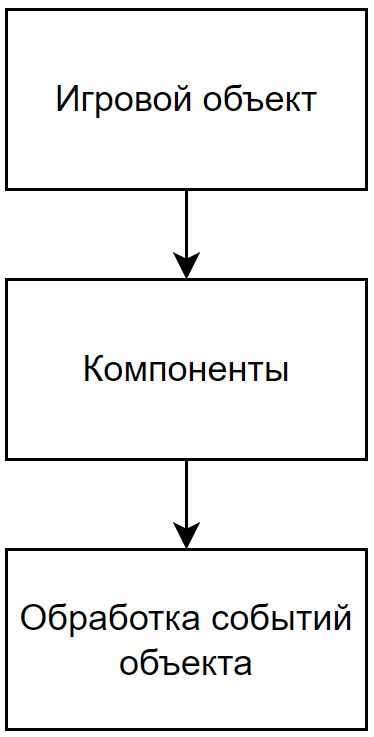


Рисунок 2.3 – Схема игровой абстракции

Каждый компонент будет предоставлять методы обработки событий, которые будут помогать управлять состоянием объекта, например, событие отрисовки кадра, событие создание или удаление объекта, событие включения или отключения объекта. Это поможет простым способом создать разную игровую логику а так же используя компонентный подход комбинировать разные сценарии их использования что даёт простое и гибкое изменение игровой логики.

**2.4 Паттерны проектирования для реализации игрового приложения**

Паттерн «Наблюдатель» является поведенческим паттерном, который позволяет объектам оповещать другие объекты об изменениях своего состояния. В этом паттерне наблюдатели могут свободно подписываться и отписываться от этих оповещений

В C#, паттерн «Наблюдатель» часто встречается, особенно там, где применяется событийная модель отношений между компонентами. Он позволяет отдельным компонентам реагировать на события, происходящие в других компонентах. В игре «Битва героев» паттерн «Наблюдатель» применим к классам персонажей, которыми игроки управляют на сцене, чтобы отслеживать изменения показателей здоровья и нанесения урона.

На рисунке 2.4 иллюстрируется схема паттерна «Наблюдатель».

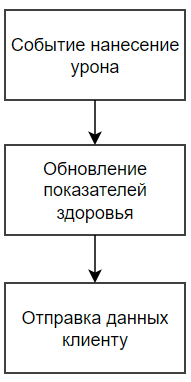


Рисунок 2.4 – Схема паттерна «Наблюдатель»

Фабричный метод – это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов. В классе, который производит продукты, создается пустой фабричный метод. В качестве возвращаемого типа указывается общий интерфейс продукта. Затем код класса просматривается и все участки, создающие продукты, заменяются вызовами фабричного метода, перенося в него код создания различных продуктов.

На рисунке 2.5 иллюстрируется схема паттерна «Фабричный метод».

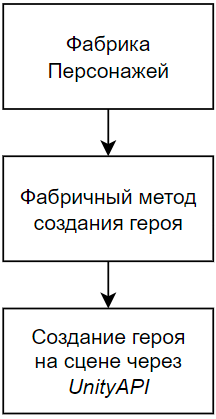


Рисунок 2.5 – Схема паттерна «Фабричный метод»

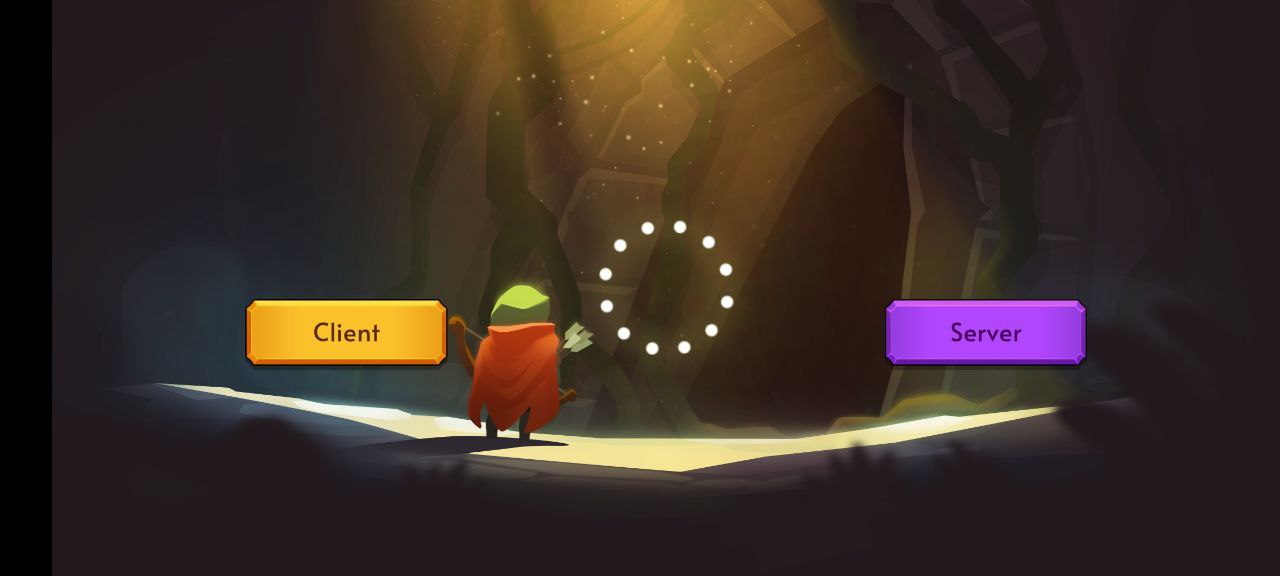
В игре паттерн «Фабричный метод» применим с созданию персонажей, это сделает процесс их создание более управляемым что упросит добавление новых персонажей. Главной особенностью данного метода является то, что он избавляет проектировщика от необходимости встраивать в код зависящие от приложения классы, а также позволяет установить связь между параллельными иерархиями классов.

**3** **ВЕРИФИКАЦИЯ И ЭКСПЛУАТАЦИЯ ПРИЛОЖЕНИЯ**

**3.1 Принципы работы игрового приложения**

При запуске приложения загружается сцена лобби, на которой пользователь выбирает кем он будет в сети. У пользователя есть выбор стать либо клиентом, либо сервером. Если пользователь выберет стать сервером, то будет вызван метод *StartAsServer* который запустит *TCP* сервер и будет ожидать подключения клиента. Если пользователь выберет стать клиентом, то будет вызван метод *StartAsClient* который начнет искать запущенный сервер в локальной сети и как только найдет подключится к серверу. Как только игроки подключаться они смогут отправлять друг другу сообщения в сети с помощью класса *TcpTransport,* который обеспечивает двухстороннюю коммуникацию.

На рисунке 3.1 иллюстрируется сцена лобби.

  
Рисунок 3.1 – Сцена лобби

Как только игроки подключились на сцене будет показано меню выбора персонажей, где каждый игрок сможет выбрать своего персонажа. Всего в игре представлено четыре персонажа у каждого из которых сто единиц здоровья и есть по три своих уникальных способностей.

На рисунке 3.2 иллюстрируется сцена выбора персонажей.

  
Рисунок 3.2 – Сцена выбора персонажей

Первый персонаж это роботизированный Киборг. Первая способность это удар булавой, который наносит пять единиц урона при попадании во врага. Вторая способность это выстрел во врага несколькими зарядами огня, которые в сумме наносят двадцать единиц урона при попадании всех зарядов. Третья способность это мгновенный выстрел лазером в сторону противника от которого из-за скорости сложно увернуться и наносящий десять единиц урона.

На рисунке 3.3 иллюстрируется персонаж Киборг.



Рисунок 3.3 – Игровой персонаж Киборг

Второй персонаж это воин Самурай, его первая способность «Удар Катаны» наносящий пять единиц урона в попавшего врага. Вторая способность это выстрел во врага дымовой шашкой, которая наносит небольшой урон и прячет врага в дымовой завесе что дезориентирует противника. Последняя способность самурая это «огонь Самурая», воин выпускает заряд синего пламени прямо в противника нанося ему урон в десять единиц здоровья.

На рисунке 3.4 иллюстрируется персонаж Самурай.



Рисунок 3.4 – Игровой персонаж Киборг

Третий персонаж это тёмный Лорд, первая способность это удар магическим мечем, который нанесёт врагу пять единиц урона. Второй способностью темного Лорда является выстрел снежного шара, который способен нанести двенадцать единиц урона. Третья способность лорда называется «Огонь Тьмы», персонаж запускает в сторону противника несколько горящих осколков, каждый их которых наносит по пять единиц урона.

На рисунке 3.5 иллюстрируется персонаж Лорд.



Рисунок 3.5 – Игровой персонаж Лорд

Последний четвертый персонаж «Абориген», первой способностью которого является удар копьем, наносящий пять единиц урона. Вторая способность это выстрел во врага ядовитым дротиком, который наносит десять единиц урона. Последней способностью персонажа является выстрел ядовитыми шарами в сторону врага, каждый из которых наносит по шесть единиц урона.

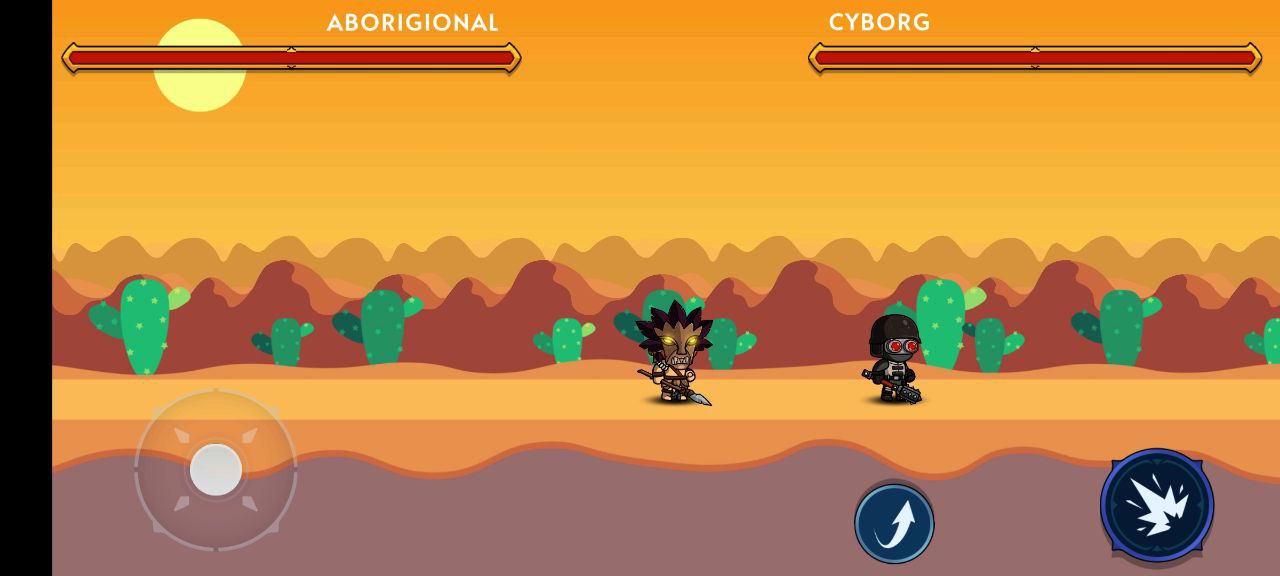
На рисунке 3.6 иллюстрируется персонаж Лорд.



Рисунок 3.6 – Игровой персонаж Лорд

После того как игрок изменяет своего персонажа, второму игроку отправляется данные об изменении персонажа, и второй игрок синхронизирует изменение персонажа. Так же у каждого игрока есть кнопка со статусом, готов ли он играть, если оба игрока будут готовы то начнется загрузка сцены с битвой.

На рисунке 3.7 иллюстрируется сцена битвы.

  
Рисунок 3.7 – Сцена битвы

Для синхронизации игровых объектов был разработан класс *NetworkRecorder*, который на сервере будет записывать и подготавливать данные к отправке, а так же синхронизировать игру на клиенте при получении данных. На сцене битвы сервер создаст двух персонажей с помощью фабричного метода *CreateHero,* после чего созданные персонажи будут отображаться на сцене и с помощью метода *TrackChanges* класса *NetworkRecorder* изменения игрового объекта персонажа будут записываться и отправляться клиенту.

Паттерн «Наблюдатель» был реализован в классе персонажа *Hero,* который содержал событие о получении урона, и когда на сервере срабатывало событие, сервер изменял показатели здоровья и отправлял клиенту для синхронизации.

Игра заканчивалась когда у одного из игроков здоровье было равно нулю, после чего показывался результат игры и игроки возвращались в лобби и могли ещё раз выбрать персонажей и сыграть ещё раз.

На рисунке 3.8-3.9 иллюстрируется результаты игры.



Рисунок 3.8 – Экран победы



Рисунок 3.9 – Экран поражения

**3.2 Модульные тесты игрового приложения**

Для проверки корректной работы приложения были созданы юнит-тесты. В классе тестирования *HeroBasic* проверяются основа изменений показателей персонажа, такие как:

– проверка времени восстановления ударов героев в методе *FirstAbilityCooldownTest;*

– проверка времени восстановления второй способности героев *SecondAbilityCooldownTest;*

– проверка нанесния урона по герою в методе *TakeDamageTest;*

– проверка времени восстановления третей способности героев *ThirdAbilityCooldownTest.*

В классе тестирования *NetworkTes*ts проверяются работа с сетью, такие как:

– проверка на локальное подключение в методе *TcpLocalConnectionTest*, создаётся два экземпляра *TcpTransport* первый становится сервером а второй подключаться к нему;

– проверка на локальную потерю соединения в методе *TcpLocalDisconnectionTest*, создаётся два экземпляра *TcpTransport* первый становится сервером а второй подключаться к нему, после чего сервер останавливается и высвобождает ресурсы, идёт проверка на вызов события отсоединения у клиента;

– проверка на локальную отправку сообщения в методе *TcpLocalDisconnectionTest*, создаётся два экземпляра *TcpTransport* первый становится сервером а второй подключаться к нему, после чего отправляются сообщение друг другу и проверяется корректность сообщения.

В классе тестирования *SyncTest* проверяются работа синхронизацией компонентов, такие как:

– проверка изменения позиции у *Transfrom* в методе *TransformChangeTest,* создается экземпляр компонента и меняется позиция, после чего идет проверка получения изменения и проверка после применения изменения;

– проверка изменения активности у *GameObject* в методе *ActiveGameObjectTest,* создается экземпляр компонента и меняется активность объекта, после чего идет проверка получения изменения и проверка после применения изменения;

– проверка вызова методов у *Animator* в методе *AnimatorTest,* для проверки правильной синхронизации вызовов изменения анимации.

Успешное прохождение всех тестов изображено на рисунке 3.10

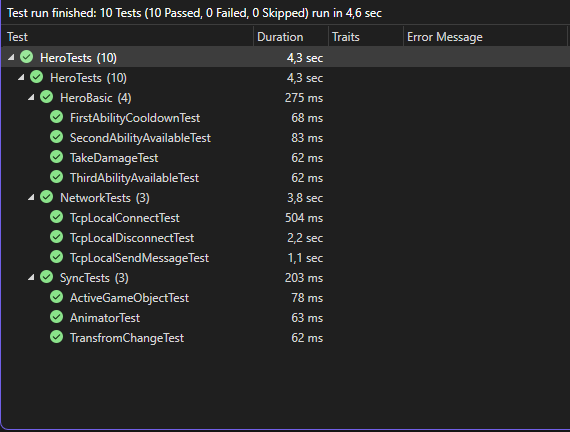
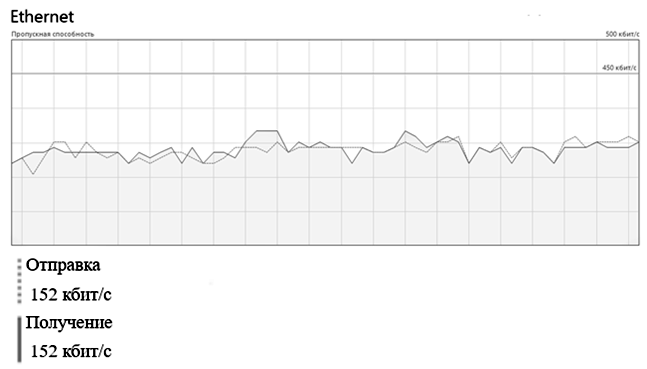


Рисунок 3.10 – Выполнение тестов

Также в процессе тестирования анализируется сетевое взаимодействие между устройствами и объем передаваемых данных. В результате определяется минимальная пропускная способность сети, приблизительно равная 150 килобитам в секунду.

На рисунке 3.11 представлена иллюстрация с пропускной способностью сети при работе приложения.

Рисунок 3.11 – Пропускная способность сети при работе приложения  


**ЗАКЛЮЧЕНИЕ**

В процессе разработки курсового проекта произошло более глубокое ознакомление с теоретической и практической областью сетевых приложений. Были изучены базовые концепции построения сетевых приложений, позволяющие объединять множество устройств, с целью повышения производительности и качества.

Для реализации игрового приложения в соответствии с принципами ООП, было проведено проектирование архитектуры приложения. В ходе проектирования были применены такие паттерны проектирования, как: «фабричный метод» для создания игровых персонажей и «наблюдатель» для отслеживания изменения показателей здоровья и способностей игровых персонажей. Для программной реализации приложения были использованы: среда разработки *Visual Studio*, язык программирования *C*# игровой движок *Unity*.

Так как игровое приложение рассчитано на двух игроков, в качестве архитектура сетевого взаимодействия используется реализация двунаправленного клиент-серверного соединения с помощью протокола *TCP*. Основными этапом являлась разработка проекта игровой логики, реализующей обмен данных пользователей по сети с решением проблемы синхронизации игрового времени, а также реализацией обработки сетевых ошибок и предотвращения конфликтов, связанных с подключением других пользователей к существующей сессии игры.

Игра разработана под операционную систему *Android.* Данная игра подходит для любой возвратной категории и гендерной принадлежности. Поскольку взаимодействие с окружающим миром осуществляется без каких-либо затруднений для игроков.

Получившаяся игра является средством развлечения и рассчитана на широкую аудиторию пользователей. Она способствует развитию реакции, логики, повышению внимания и концентрации.

Курсовая работа проверена системой «Антиплагиат». Оригинальность со-ставила 91,74 процентов. Цитирования обозначены ссылками на публикации, указанные в «Списке использованных источников».

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. *Troelsen, Pro* C# 9 *with* .*NET* 5: *Foundational Principles and Practices in Programming.* 10th *edition / Andrew Troelsen, Phillip Japikse*. – *Apress Media, LLC*, 2021. – 1383 p.
2. Обзор C# [Электронный ресурс]. – Режим доступа: *https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/* (дата доступа: 11.10.2023).
3. Синхронизация данных в мультиплеерных играх [Электронный ресурс]. – Режим доступа: *https*://*habr*.*com*/*ru*/*post*/328702/ – (дата доступа: 11.10.2023).
4. Отличия *TCP* и *UDP*-протоколов [Электронный ресурс]. – Режим доступа: *https://selectel.ru/blog/tcp-vs-udp/* (дата доступа: 11.10.2023).
5. Программирование сокетов: Клиентское/серверное приложение *TCP* [Электронный ресурс]. – Режим доступа: *https://www.developer.com/web-services/socket-programming-tcp-client-server-application/* (дата доступа: 11.10.2023).
6. Что такое *Unity* [Электронный ресурс]. – Режим доступа: *https://gamedevacademy.org/what-is-unity* (дата доступа: 11.10.2023).

**ПРИЛОЖЕНИЕ А**

(обязательное)

**Листинг программы**

**Hero****.cs:**

using Assets.HeroEditor.Common.CharacterScripts;

using System;

using System.Threading.Tasks;

using UnityEngine;

using UnitySynchronization;

namespace Assets.Scripts

{

public class Hero : MonoBehaviour, IDamageable

{

private const float HitRadius = 0.5f;

private const float HitCooldown = 0.4f;

private readonly TimeSpan hitRegistrationDelay = TimeSpan.FromMilliseconds(100);

public Character Character;

public Projectile FirstAbilityProjectile;

public Projectile SecondAbilityProjectile;

public int FirstAbilityCountShoot;

public int SecondAbilityCountShoot;

private Hero opponentHero;

private HeroStats stats;

private IInputService inputService;

private CharacterMovement characterMovement;

private WeaponControls weaponControls;

private Transform weaponEdge;

private NetworkRecorder recorder;

private float lastHitTime;

private bool isDummy;

public event Action<float> OnTakeDamage;

public GameHeroState State { get; set; }

public bool FirstAbilityAvailable { get; private set; }

public bool SecondAbilityAvailable { get; private set; }

private void Awake()

{

recorder = GameApplication.GetInstance<NetworkRecorder>();

if (recorder != null)

{

recorder.RegisterPrefab(FirstAbilityProjectile.gameObject);

recorder.RegisterPrefab(SecondAbilityProjectile.gameObject);

}

}

public void Init(Hero opponentHero, HeroStats stats, IInputService inputService)

{

this.stats = stats;

this.opponentHero = opponentHero;

weaponEdge = FindWeaponEdge();

weaponControls = Character.GetComponent<WeaponControls>();

if (inputService != null)

{

this.inputService = inputService;

inputService.OnHit += HandleOnHit;

inputService.OnFirstAbilityUse += HandleOnFirstAbilityUse;

inputService.OnSecondAbilityUse += HandleOnSecondAbilityUse;

characterMovement = GetComponent<CharacterMovement>();

characterMovement.Init(inputService);

}

}

private void Update()

{

if (isDummy || State == null || stats == null)

{

return;

}

lastHitTime += Time.deltaTime;

FirstAbilityAvailable = CheckFirstAbilityAvailable(stats, State);

SecondAbilityAvailable = CheckSecondAbilityAvailable(stats, State);

}

private void OnDestroy()

{

if (inputService != null)

{

inputService.OnHit -= HandleOnHit;

inputService.OnFirstAbilityUse -= HandleOnFirstAbilityUse;

inputService.OnSecondAbilityUse -= HandleOnSecondAbilityUse;

}

}

private Transform FindWeaponEdge()

{

int needSkip = 1;

int skipCount = 0;

foreach (Transform transform in transform.GetComponentsInChildren<Transform>())

{

if (transform.name == "Edge")

{

if (needSkip != skipCount)

{

skipCount++;

continue;

}

return transform;

}

}

throw new Exception("not found edge");

}

public void MakeDummy()

{

isDummy = true;

GetComponent<Rigidbody2D>().bodyType = RigidbodyType2D.Static;

GetComponent<CharacterMovement>().enabled = false;

}

public void TakeDamage(float damage)

{

OnTakeDamage?.Invoke(damage);

}

private async void HandleOnHit()

{

if (lastHitTime < HitCooldown)

{

return;

}

lastHitTime = 0;

weaponControls.Hit();

await Task.Delay(hitRegistrationDelay);

IDamageable.DamageCircleZone(weaponEdge.position, HitRadius, stats.HitDamage, transform);

}

private void HandleOnFirstAbilityUse()

{

if (!FirstAbilityAvailable)

{

return;

}

ShootProjectile(FirstAbilityProjectile, FirstAbilityCountShoot);

State.FirstAbilityMana = 0;

}

private void HandleOnSecondAbilityUse()

{

if (!SecondAbilityAvailable)

{

return;

}

ShootProjectile(SecondAbilityProjectile, SecondAbilityCountShoot);

State.SecondAbilityMana = 0;

}

private async void ShootProjectile(Projectile prefab, int countShoot)

{

for (int i = 0; i < countShoot; i++)

{

Vector3 offset = new(0, 1.5f, 0);

Projectile projectile = Instantiate(prefab, transform.position + offset, Quaternion.identity);

projectile.Init(opponentHero.transform, this);

recorder.TrackChanges(projectile.gameObject);

await Task.Delay(200);

}

}

private bool CheckFirstAbilityAvailable(HeroStats stats, GameHeroState heroState)

{

bool abilityAvailable = heroState.FirstAbilityMana >= stats.FirstAbilityManaPrice;

if (!abilityAvailable)

{

heroState.FirstAbilityMana += Time.deltaTime;

}

return abilityAvailable;

}

private bool CheckSecondAbilityAvailable(HeroStats stats, GameHeroState heroState)

{

bool abilityAvailable = heroState.SecondAbilityMana >= stats.SecondAbilityManaPrice;

if (!abilityAvailable)

{

heroState.SecondAbilityMana += Time.deltaTime;

}

return abilityAvailable;

}

private void OnDrawGizmos()

{

if (Character == null)

{

return;

}

if (weaponEdge == null)

weaponEdge = FindWeaponEdge();

Gizmos.color = Color.red;

Gizmos.DrawWireSphere(weaponEdge.position, HitRadius);

}

}

}

**TcpTransport****.cs:**

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

using System.Collections.Generic;

using System.Threading.Tasks;

using Cysharp.Threading.Tasks;

using UnityEngine;

namespace Transports

{

public class TcpTransport : IDisposable, ITransport

{

private const int BufferSize = 1024 \* 1024 \* 10;

private const string MessageSeparator = "!";

private const string MessageCodeSeparator = "@";

private const int HeartbeatCode = int.MinValue;

private const int DisconnectCode = int.MinValue + 1;

private readonly TimeSpan HeartbeatTimestep = TimeSpan.FromMilliseconds(1000);

private readonly TimeSpan ConnectTimeout = TimeSpan.FromMilliseconds(3000);

private readonly int port;

private readonly Queue<string> messages = new();

private readonly byte[] buffer = new byte[BufferSize];

private bool disposed;

private Socket socket;

private Socket connectedSocket;

private IPAddress hostAddress;

private string tempMessage;

public bool Connected { get; private set; }

public bool IsServer { get; private set; }

public bool IsClient => !IsServer;

public event Action OnConnected;

public event Action OnDisconnected;

public event Action<int, string> OnMessageReceived;

public TcpTransport(int port)

{

this.port = port;

Task.Run(HeartbeatLoop).Forget();

Task.Run(HandleMessagesLoop).Forget();

}

public async Task<bool> ConnectServer(IPAddress serverAddress)

{

try

{

bool connecting = await Connect(serverAddress);

if (connecting)

{

Task.Run(StartClient).Forget();

}

return connecting;

}

catch (Exception error)

{

Debug.LogWarning("Connect Server Fail! " + error.ToString());

return false;

}

}

public async Task StartServer()

{

IPAddress localAddress = ITransport.LocalAddress;

Debug.Log($"Start Server {localAddress}:{port}");

Dispose(socket);

socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

socket.Bind(new IPEndPoint(localAddress, port));

socket.Listen(100);

IsServer = true;

try

{

while (!disposed)

{

try

{

connectedSocket = await socket.AcceptAsync();

OnConnecting();

while (connectedSocket.Connected)

{

if (disposed) return;

int received = await connectedSocket.ReceiveAsync(buffer, SocketFlags.None);

SocketReceivedMessage(Encoding.UTF8.GetString(buffer, 0, received));

}

InvokeDisconnecting(string.Empty);

}

catch (Exception error)

{

InvokeDisconnecting("Server Loop " + error.ToString());

}

}

}

catch (ObjectDisposedException)

{

return;

}

}

private async Task StartClient()

{

IsServer = false;

Debug.Log($"Start as Client");

try

{

while (!disposed)

{

while (!socket.Connected)

{

if (disposed) return;

await Connect(hostAddress);

}

connectedSocket = socket;

OnConnecting();

try

{

while (socket.Connected)

{

int received = await socket.ReceiveAsync(buffer, SocketFlags.None);

SocketReceivedMessage(Encoding.UTF8.GetString(buffer, 0, received));

}

}

catch (Exception error)

{

InvokeDisconnecting("Client Loop" + error.ToString());

}

}

}

catch (ObjectDisposedException)

{

return;

}

}

private async Task HeartbeatLoop()

{

try

{

while (!disposed)

{

if (disposed) return;

await Task.Delay(HeartbeatTimestep);

if (!Connected) continue;

try

{

SendMessage(HeartbeatCode, string.Empty);

}

catch (Exception error)

{

InvokeDisconnecting("Heartbeat Loop " + error.ToString());

}

}

}

catch (ObjectDisposedException)

{

return;

}

}

private async Task HandleMessagesLoop()

{

while (!disposed)

{

while (messages.Count == 0)

{

if (disposed) return;

await Task.Yield();

}

try

{

string message = messages.Dequeue();

if (string.IsNullOrEmpty(message))

{

continue;

}

await connectedSocket.SendAsync(Encoding.UTF8.GetBytes(message), SocketFlags.None);

}

catch (Exception error)

{

InvokeDisconnecting("Handle Messages Loop " + error.ToString());

}

}

}

public void SendMessage(string message) => SendMessage(0, message);

public void SendMessage(int code, string message)

{

if (Connected)

{

messages.Enqueue(code + MessageCodeSeparator + message + MessageSeparator);

}

else

{

Debug.LogWarning("Not connected, message not send");

}

}

public void Dispose()

{

disposed = true;

Dispose(socket);

Dispose(connectedSocket);

}

private void Dispose(Socket socket)

{

if (socket == null)

return;

try

{

socket.Shutdown(SocketShutdown.Both);

}

catch { }

try

{

socket.Close();

}

catch { }

try

{

socket.Dispose();

}

catch { }

}

private async Task<bool> Connect(IPAddress address)

{

try

{

Dispose(socket);

socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

Debug.Log($"Connecting to {address}:{port}");

Task connectTask = socket.ConnectAsync(address, port);

Task first = await Task.WhenAny(connectTask, Task.Delay(ConnectTimeout));

if (first != connectTask || !socket.Connected)

{

throw new SocketException();

}

hostAddress = address;

Debug.Log($"Connecting success!");

return true;

}

catch (Exception error)

{

Dispose(socket);

socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

Debug.LogWarning("Connecting fail! " + error.Message);

return false;

}

}

private async void OnConnecting()

{

if (Connected)

{

return;

}

Connected = true;

try

{

await UniTask.SwitchToMainThread();

OnConnected?.Invoke();

}

catch (Exception error)

{

Debug.LogException(error);

}

}

private async void InvokeDisconnecting(string reason)

{

if (!Connected)

{

return;

}

Debug.LogWarning("Disconnected due to " + reason);

try

{

SendMessage(DisconnectCode, string.Empty);

}

catch (Exception error)

{

Debug.LogWarning(error);

}

Connected = false;

try

{

await UniTask.SwitchToMainThread();

OnDisconnected?.Invoke();

}

catch (Exception error)

{

Debug.LogException(error);

}

}

private void SocketReceivedMessage(string message)

{

message = tempMessage + message;

tempMessage = string.Empty;

if (message.Contains(MessageSeparator))

{

string[] parts = message.Split(MessageSeparator);

bool isFullMessage = message.EndsWith(MessageSeparator);

for (int i = 0; i < parts.Length; i++)

{

string part = parts[i];

bool itsLastPart = parts.Length - 1 == i;

if (itsLastPart && !isFullMessage)

{

tempMessage += part;

}

else

{

ReceivedMessage(part);

}

}

return;

}

else

{

tempMessage = message;

}

}

private async void ReceivedMessage(string fullMessage)

{

if (string.IsNullOrWhiteSpace(fullMessage))

{

return;

}

try

{

string[] messageParts = fullMessage.Split(MessageCodeSeparator);

int code = int.Parse(messageParts[0]);

if (code == HeartbeatCode)

return;

if (code == DisconnectCode)

{

InvokeDisconnecting(string.Empty);

return;

}

string message = messageParts[1];

await UniTask.SwitchToMainThread();

OnMessageReceived?.Invoke(code, message);

}

catch (Exception error)

{

Debug.LogException(error);

}

}

}

}

**NetworkRecorder****.cs:**

using System;

using System.Linq;

using System.Collections.Generic;

using UnityEngine;

namespace UnitySynchronization

{

public class NetworkRecorder

{

private const int InstantiateChangeCode = 0;

private const int DestroyChangeCode = 1;

private const int GameObjectChangeCode = 2;

private readonly List<GameObjectID> prefabs = new();

private readonly List<GameObjectSource> gameObjects = new();

private readonly List<GameObjectSource> rootGameObjects = new();

public event Action<Change> OnChanged;

public bool SyncPhysics { get; set; } = true;

public int GetInstanceId(GameObject gameObject) => gameObjects.First(x => x.gameObject == gameObject).InstanceID;

public GameObject FindGameObject(int instanceId) => gameObjects.First(x => x.InstanceID == instanceId).gameObject;

public void TrackChanges(params GameObject[] gameObjects) => Array.ForEach(gameObjects, x => TrackChanges(x));

private Change DestroyChange(int instanceId) => new(DestroyChangeCode, instanceId.ToString());

private Change InstantiateChange(InstantiateChangeArgs args) => new(InstantiateChangeCode, args.ToBinaryString());

private Change GameObjectChange(GameObjectChangeArgs args) => new(GameObjectChangeCode, args.ToBinaryString());

public void RegisterPrefab(GameObject prefab)

{

if (prefab.TryGetComponent(out GameObjectID gameObjectID))

{

prefabs.Add(gameObjectID);

}

else

{

throw new Exception("Prefab not contains GameObjectID!");

}

}

public void TrackChanges(GameObject gameObject)

{

if (gameObject.TryGetComponent(out GameObjectID gameObjectID))

{

GameObjectSource source = new(gameObject);

rootGameObjects.Add(source);

gameObjects.Add(source);

foreach (GameObjectSource childSource in source.Childs)

{

gameObjects.Add(childSource);

}

List<GameObjectChange> changes = source.CheckChanges();

OnChanged?.Invoke(InstantiateChange(new InstantiateChangeArgs(gameObjectID.Id, source, changes)));

}

else

{

throw new Exception("Root object don't have GameObjectID Component!");

}

}

public void CheckChanges()

{

List<GameObjectSource> destroyedSources = new();

foreach (GameObjectSource source in rootGameObjects)

{

if (source.gameObject == null)

{

destroyedSources.Add(source);

}

}

foreach (GameObjectSource source in destroyedSources)

{

rootGameObjects.Remove(source);

foreach (GameObjectSource childSource in source.Childs)

{

gameObjects.Remove(childSource);

}

OnChanged?.Invoke(DestroyChange(source.InstanceID));

}

foreach (GameObjectSource source in rootGameObjects)

{

List<GameObjectChange> changes = source.CheckChanges();

if (changes.Count > 0)

{

OnChanged?.Invoke(GameObjectChange(new GameObjectChangeArgs(source.InstanceID, changes)));

}

}

}

public void SyncUpdate() => SyncUpdate(false);

private void SyncUpdate(bool initialize)

{

foreach (GameObjectSource source in rootGameObjects)

{

source.Sync(initialize);

}

}

public void SyncChange(Change change)

{

if (change.Id == InstantiateChangeCode)

{

InstantiateChangeArgs args = change.Value.DeserializeBinaryString<InstantiateChangeArgs>();

GameObject prefab = prefabs.First(x => x.Id == args.PrefabGameObjectId).gameObject;

GameObject gameObject = UnityEngine.Object.Instantiate(prefab);

if (!SyncPhysics)

{

foreach (Rigidbody2D rigidbody in gameObject.GetComponentsInChildren<Rigidbody2D>())

{

rigidbody.bodyType = RigidbodyType2D.Kinematic;

}

}

GameObjectSource source = new(gameObject, args.InstantiatedSource);

source.ApplyChange(args.InitialChanges);

source.Sync(true);

rootGameObjects.Add(source);

gameObjects.Add(source);

foreach (GameObjectSource childSource in source.Childs)

{

gameObjects.Add(childSource);

}

}

else if (change.Id == DestroyChangeCode)

{

int destroyInstanceId = change.Value.Parse<int>();

GameObjectSource source = rootGameObjects.FirstOrDefault(gameObject => gameObject.InstanceID == destroyInstanceId);

if (source != null)

{

UnityEngine.Object.Destroy(source.gameObject);

rootGameObjects.Remove(source);

foreach (GameObjectSource childSource in source.Childs)

{

gameObjects.Remove(childSource);

}

}

}

else if (change.Id == GameObjectChangeCode)

{

GameObjectChangeArgs args = change.Value.DeserializeBinaryString<GameObjectChangeArgs>();

GameObjectSource source = gameObjects.FirstOrDefault(x => x.InstanceID == args.InstanceId);

source?.ApplyChange(args.Changes);

}

else

{

throw new NotImplementedException();

}

}

}

}

**NetworkTransform****.cs:**

using UnityEngine;

using System.Collections.Generic;

namespace UnitySynchronization

{

[RequireComponent(typeof(GameObjectID)), DisallowMultipleComponent]

public class NetworkTransform : NetworkComponent

{

private const int CodePosX = 0;

private const int CodePosY = 1;

private const int CodePosZ = 2;

private const int CodeScaleX = 3;

private const int CodeScaleY = 4;

private const int CodeScaleZ = 5;

private Vector3 position;

private Vector3 scale;

public override void ApplyChange(Change change)

{

switch (change.Id)

{

default: throw new System.NotImplementedException();

case CodePosX: position.x = change.Value.ToShortFloat(); break;

case CodePosY: position.y = change.Value.ToShortFloat(); break;

case CodePosZ: position.z = change.Value.ToShortFloat(); break;

case CodeScaleX: scale.x = change.Value.ToShortFloat(); break;

case CodeScaleY: scale.y = change.Value.ToShortFloat(); break;

case CodeScaleZ: scale.z = change.Value.ToShortFloat(); break;

}

}

public override List<Change> CheckChanges()

{

List<Change> result = new();

if (position.x != transform.position.x)

{

position.x = transform.position.x;

result.Add(new Change(CodePosX, position.x.ToShortString()));

}

if (position.y != transform.position.y)

{

position.y = transform.position.y;

result.Add(new Change(CodePosY, position.y.ToShortString()));

}

if (position.z != transform.position.z)

{

position.z = transform.position.z;

result.Add(new Change(CodePosZ, position.z.ToShortString()));

}

if (scale.x != transform.localScale.x)

{

scale.x = transform.localScale.x;

result.Add(new Change(CodeScaleX, scale.x.ToShortString()));

}

if (scale.y != transform.localScale.y)

{

scale.y = transform.localScale.y;

result.Add(new Change(CodeScaleY, scale.y.ToShortString()));

}

if (scale.z != transform.localScale.z)

{

scale.z = transform.localScale.z;

result.Add(new Change(CodeScaleZ, scale.z.ToShortString()));

}

return result;

}

public override void Sync(bool initialize)

{

Vector3 resultPosition;

if (initialize)

{

resultPosition = position;

}

else

{

Vector3 velocity = position - transform.position;

float delta = Time.deltaTime \* 10f;

float magnitude = Mathf.Clamp(velocity.magnitude, 1f, float.MaxValue);

resultPosition = Vector3.Lerp(transform.position, position, delta \* magnitude);

}

transform.position = resultPosition;

transform.localScale = new Vector3(scale.x, scale.y, scale.z);

}

}

}

**GameScene****.cs:**

using Assets.Scripts;

using Cinemachine;

using DG.Tweening;

using System.Threading.Tasks;

using UnityEngine;

using UnityEngine.SceneManagement;

using UnityEngine.UI;

using UnitySynchronization;

namespace Assets

{

public class GameScene : MonoBehaviour

{

public static GameSceneArgs SceneArgs { get; set; }

private const int TickRate = 40;

public GameUI UI;

public Transform FirstSpawn;

public Transform SecondSpawn;

public Image Fade;

public CinemachineVirtualCamera virtualCamera;

public GameObject WinScreen;

public GameObject LoseScreen;

private GameState state;

private HeroRepo heroRepo;

private ITransport transport;

private NetworkRecorder recorder;

private NetworkInput networkInput;

private string lastState;

public bool GameOver { get; private set; }

public Hero MyHero { get; set; }

public Hero OpponentHero { get; set; }

public HeroStats MyHeroStats { get; set; }

public HeroStats OpponentHeroStats { get; set; }

public GameHeroState MyHeroState => transport.IsServer ? state.ServerHero : state.ClientHero;

public GameHeroState OpponentHeroState => transport.IsClient ? state.ServerHero : state.ClientHero;

private void Awake()

{

IInputService inputService = UI.CreateInputService();

state = new GameState();

networkInput = new NetworkInput();

recorder = new NetworkRecorder();

GameApplication.Register(recorder);

GameApplication.Register(inputService);

GameApplication.Register(UI);

heroRepo = GameApplication.GetInstance<HeroRepo>();

transport = GameApplication.GetInstance<ITransport>();

recorder.SyncPhysics = transport.IsServer;

transport.OnDisconnected += OnDisconnected;

transport.OnMessageReceived += OnMessageReceived;

networkInput.OnNetworkEvent += NetworkInputOnEvent;

recorder.OnChanged += OnGameObjectsChanged;

foreach (var prefab in heroRepo.GetAllPrefabs())

{

recorder.RegisterPrefab(prefab.gameObject);

}

MyHeroStats = heroRepo.GetStats(SceneArgs.MyLobbyState.SelectedHero);

OpponentHeroStats = heroRepo.GetStats(SceneArgs.OpponentLobbyState.SelectedHero);

MyHeroState.Health = MyHeroStats.MaxHealth;

OpponentHeroState.Health = OpponentHeroStats.MaxHealth;

if (transport.IsServer)

{

MyHero = heroRepo.CreateHero(SceneArgs.MyLobbyState.SelectedHero, null);

OpponentHero = heroRepo.CreateHero(SceneArgs.OpponentLobbyState.SelectedHero, null);

MyHero.transform.position = FirstSpawn.position;

MyHero.OnTakeDamage += (damage) => MyHeroState.Health -= damage;

MyHero.Init(OpponentHero, MyHeroStats, GameApplication.GetInstance<IInputService>());

virtualCamera.Follow = MyHero.transform;

OpponentHero.transform.position = SecondSpawn.position;

OpponentHero.OnTakeDamage += (damage) => OpponentHeroState.Health -= damage;

OpponentHero.Init(MyHero, heroRepo.GetStats(SceneArgs.OpponentLobbyState.SelectedHero), networkInput);

recorder.TrackChanges(MyHero.gameObject, OpponentHero.gameObject);

transport.SendMessage(GameState.SyncHeroIdCode, recorder.GetInstanceId(OpponentHero.gameObject).ToString());

}

else

{

networkInput.RecordFrom(GameApplication.GetInstance<IInputService>());

}

UI.SetLeftHeroName(MyHeroStats.HeroName);

UI.SetRightHeroName(OpponentHeroStats.HeroName);

StartCoroutine(GameApplication.TickRateCoroutine(TickRate, OnTick));

Fade.color = Color.black;

Fade.DOColor(Color.clear, 1);

}

private void OnDestroy()

{

transport.OnDisconnected -= OnDisconnected;

transport.OnMessageReceived -= OnMessageReceived;

networkInput.OnNetworkEvent -= NetworkInputOnEvent;

recorder.OnChanged -= OnGameObjectsChanged;

}

private void Update()

{

CheckGameOver();

if (transport.IsClient)

{

networkInput.Update();

recorder.SyncUpdate();

}

UpdateUI();

}

private void OnTick()

{

if (transport.IsServer)

{

recorder.CheckChanges();

string stateData = state.ToBinaryString();

if (stateData != lastState)

{

transport.SendMessage(GameState.SyncStateCode, stateData);

}

lastState = stateData;

}

}

private void UpdateUI()

{

UI.LeftHeroHealth = MyHeroState.Health;

UI.RightHeroHealth = OpponentHeroState.Health;

if (MyHero != null)

{

MyHero.State = MyHeroState;

UI.SetFirstAbilityAvailable(MyHero.FirstAbilityAvailable);

UI.SetSecondAbilityAvailable(MyHero.SecondAbilityAvailable);

}

if (OpponentHero != null)

{

OpponentHero.State = OpponentHeroState;

}

}

private void OnDisconnected()

{

Debug.Log("OnDisconnected");

SceneManager.LoadScene(0);

}

private void OnMessageReceived(int code, string message)

{

switch (code)

{

case GameState.SyncStateCode:

{

state = message.DeserializeBinaryString<GameState>();

break;

}

case GameState.SyncHeroIdCode:

{

int myHeroInstanceID = int.Parse(message);

MyHero = recorder.FindGameObject(myHeroInstanceID)

.GetComponent<Hero>();

MyHero.Init(OpponentHero, MyHeroStats, null);

virtualCamera.Follow = MyHero.transform;

break;

}

case GameState.SyncInputCode:

{

networkInput.SyncInput(message);

break;

}

case GameState.SyncObjectChangedCode:

{

recorder.SyncChange(message);

break;

}

default: throw new System.NotImplementedException();

}

}

private void NetworkInputOnEvent(string inputEvent)

{

transport.SendMessage(GameState.SyncInputCode, inputEvent);

}

private void OnGameObjectsChanged(Change change)

{

string sendChange = change.ToString();

transport.SendMessage(GameState.SyncObjectChangedCode, sendChange);

}

private async void ShowGameResult()

{

UI.Hide();

Time.timeScale = 0.1f;

virtualCamera.m\_Lens.FieldOfView = 33f;

bool itsWin = OpponentHeroState.Health <= 0;

Tween fadeTween = Fade.DOColor(Color.black, 3f);

fadeTween.SetUpdate(true);

await fadeTween.AsyncWaitForCompletion();

Time.timeScale = 1f;

LoseScreen.SetActive(!itsWin);

WinScreen.SetActive(itsWin);

await Task.Delay(3000);

SceneManager.LoadScene(0);

}

private void CheckGameOver()

{

if (GameOver)

{

return;

}

else

{

GameOver = MyHeroState.Health <= 0 || OpponentHeroState.Health <= 0;

if (GameOver)

{

ShowGameResult();

return;

}

}

}

}

}

**LobbyScene****.cs:**

using Assets.Scripts;

using UnityEngine;

using UnityEngine.UI;

using DG.Tweening;

using UnityEngine.SceneManagement;

using Assets;

using System.Threading.Tasks;

using Transports;

using System.Net;

using System;

public class LobbyScene : MonoBehaviour

{

private static bool FirstRun = true;

private const int StateSyncCode = 50;

public Image Fade;

public GameObject Loading;

public GameObject StartAsServerButton;

public GameObject StartAsClientButton;

public HeroRepo heroRepo;

public HeroSelector MyHeroView;

public HeroSelector OpponentHeroView;

private LobbyState myState;

private LobbyState opponentState;

private ITransport transport;

private INetworkDiscover discover;

private async void Awake()

{

myState = new LobbyState();

transport = GameApplication.GetInstance<ITransport>();

transport ??= new TcpTransport(3005);

discover = new UdpDiscover("hero", 7777);

MyHeroView.OnHeroChanged += MyHeroViewOnHeroChanged;

MyHeroView.OnReadyChanged += MyHeroViewOnReadyChanged;

transport.OnConnected += OnConnected;

transport.OnDisconnected += OnDisconnected;

transport.OnMessageReceived += OnMessageReceived;

GameApplication.Register(heroRepo);

GameApplication.Register(transport);

MyHeroView.Init(heroRepo);

OpponentHeroView.Init(heroRepo);

if (transport.Connected)

{

await Task.Delay(1000);

OnConnected();

}

if (FirstRun)

{

FirstRun = false;

OnFirstRun();

}

}

private void OnFirstRun()

{

StartAsClientButton.SetActive(true);

StartAsServerButton.SetActive(true);

}

public void StartAsServer()

{

StartAsClientButton.SetActive(false);

StartAsServerButton.SetActive(false);

Task.Run(discover.StartBroadcast).Forget();

Task.Run(transport.StartServer).Forget();

}

public async void StartAsClient()

{

StartAsClientButton.SetActive(false);

StartAsServerButton.SetActive(false);

bool connected = await transport.ConnectServer(ITransport.LocalAddress);

if (connected)

{

return;

}

Debug.Log("Discovery host");

IPAddress hostAddress = await discover.SearchClient(TimeSpan.FromSeconds(15));

if (hostAddress == null)

{

#if UNITY\_EDITOR

UnityEditor.EditorApplication.ExitPlaymode();

#endif

Application.Quit();

}

else

{

Debug.Log($"Host {hostAddress}");

do

{

connected = await transport.ConnectServer(hostAddress);

await Task.Delay(1000);

}

while (!connected);

}

}

private void OnDestroy()

{

transport.OnConnected -= OnConnected;

transport.OnDisconnected -= OnDisconnected;

transport.OnMessageReceived -= OnMessageReceived;

discover?.Dispose();

}

private void OnConnected()

{

Debug.Log("OnConnected");

MyHeroView.SetHero(myState.SelectedHero);

MyHeroView.SetReady(myState.IsReady);

Loading.SetActive(false);

MyHeroView.gameObject.SetActive(true);

OpponentHeroView.gameObject.SetActive(true);

}

private void OnDisconnected()

{

Debug.Log("OnDisconnected");

MyHeroView.gameObject.SetActive(false);

OpponentHeroView.gameObject.SetActive(false);

Loading.SetActive(true);

}

private async void OnLobbyStateChanged()

{

if (myState.IsReady && opponentState.IsReady)

{

GameScene.SceneArgs = new GameSceneArgs(myState, opponentState);

MyHeroView.DisableAll();

OpponentHeroView.DisableAll();

await Fade.DOColor(Color.black, 1).AsyncWaitForCompletion();

SceneManager.LoadScene(1);

}

}

private void MyStateChanged()

{

transport.SendMessage(StateSyncCode, myState.ToBinaryString());

OnLobbyStateChanged();

}

private void OnMessageReceived(int code, string message)

{

if (code == StateSyncCode)

OpponentStateChange(message.DeserializeBinaryString<LobbyState>());

}

private void OpponentStateChange(LobbyState state)

{

opponentState = state;

OpponentHeroView.SetHero(opponentState.SelectedHero);

OpponentHeroView.SetReady(opponentState.IsReady);

OnLobbyStateChanged();

}

private void MyHeroViewOnReadyChanged(bool ready)

{

myState.IsReady = ready;

MyStateChanged();

}

private void MyHeroViewOnHeroChanged(HeroType hero)

{

myState.SelectedHero = hero;

MyStateChanged();

}

}

**GameUI****.cs:**

using Assets.Scripts;

using TMPro;

using UnityEngine;

using UnityEngine.UI;

public class GameUI : MonoBehaviour

{

public MobileInput MobileInput;

public GameObject StandaloneInput;

public GameObject StandaloneFirstAbility;

public GameObject StandaloneSecondAbility;

public GameObject MobileFirstAbility;

public GameObject MobileSecondAbility;

public TextMeshProUGUI FirstPlayerName;

public TextMeshProUGUI SecondPlayerName;

public Slider FirstPlayerSlider;

public Slider SecondPlayerSlider;

public IInputService CreateInputService()

{

bool mobile = Application.isMobilePlatform;

IInputService inputService = mobile ? MobileInput : new StandaloneInputService();

MobileInput.gameObject.SetActive(mobile);

StandaloneInput.SetActive(!mobile);

return inputService;

}

public void Hide()

{

MobileInput.gameObject.SetActive(false);

StandaloneInput.SetActive(false);

}

public void SetFirstAbilityAvailable(bool available)

{

StandaloneFirstAbility.SetActive(available);

MobileFirstAbility.SetActive(available);

}

public void SetSecondAbilityAvailable(bool available)

{

StandaloneSecondAbility.SetActive(available);

MobileSecondAbility.SetActive(available);

}

public float LeftHeroHealth

{

get => FirstPlayerSlider.value \* HeroRepo.MaxHealth;

set => FirstPlayerSlider.value = value / HeroRepo.MaxHealth;

}

public float RightHeroHealth

{

get => SecondPlayerSlider.value \* HeroRepo.MaxHealth;

set => SecondPlayerSlider.value = value / HeroRepo.MaxHealth;

}

public void SetLeftHeroName(string name)

{

FirstPlayerName.text = name;

}

public void SetRightHeroName(string name)

{

SecondPlayerName.text = name;

}

}