Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Компьютерные системы и сети

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему:

**СЕТЕВАЯ КОМПЬЮТЕРНАЯ ИГРА В ЖАНРЕ CТРАТЕГИИ**

БГУИР КП 6-05-0612-01 038 ПЗ

Студент Поливкин В.С.

Руководитель Болтак С.В.

Минск 2025

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc199159579)

[1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 5](#_Toc199159580)

[1.1 Обзор аналогов 5](#_Toc199159581)

[1.2 Постановка задачи 8](#_Toc199159582)

[2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 9](#_Toc199159583)

[2.1 Структура программы 9](#_Toc199159584)

[2.2 Проектирование интерфейса программного средства 10](#_Toc199159585)

[2.3 Проектирование функционала программного средства 12](#_Toc199159586)

[3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА 17](#_Toc199159587)

[3.1 Разработка интерфейсов 17](#_Toc199159588)

[3.2 Реализация функционала юнитов 19](#_Toc199159589)

[3.3 Реализация функционала защитников 21](#_Toc199159590)

[4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 23](#_Toc199159592)

[5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 24](#_Toc199159593)

[5.1 Интерфейс программного средства 24](#_Toc199159594)

[5.2 Управление программным средством 27](#_Toc199159595)

[ЗАКЛЮЧЕНИЕ 29](#_Toc199159596)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 30](#_Toc199159597)

[ПРИЛОЖЕНИЕ А. Исходный код программы 31](#_Toc199159598)

# ВВЕДЕНИЕ

История компьютерных игр берет начало в 1950–1960-х годах, когда первые интерактивные программы, такие как «Tennis for Two» (1958) и «Spacewar!» (1962), заложили основы для развития индустрии. С появлением персональных компьютеров и игровых консолей в 1970–1980-х годах начался стремительный рост жанров, среди которых выделился жанр Tower Defense. Его корни уходят в стратегии реального времени, такие как «Dune II» (1992), где игрокам впервые пришлось защищать базы от волн врагов. В 2000-х годах жанр Tower Defense стал самостоятельным благодаря популярности браузерных игр, таких как «Desktop Tower Defense» (2007), которые привлекли миллионы игроков простотой и стратегической глубиной.

Жанр Tower Defense характеризуется необходимостью защищать базу или территорию, размещая оборонительные сооружения (башни) для уничтожения наступающих врагов. Простота механик в сочетании с тактической сложностью делает этот жанр привлекательным для широкой аудитории. Современные технологии, такие как игровой движок Unity, упростили создание таких игр, позволяя реализовать сложные алгоритмы поведения врагов, разнообразные типы башен и динамичные визуальные эффекты. Развитие мобильных платформ и сетевых технологий также способствовало популяризации жанра, позволяя игрокам соревноваться или сотрудничать в реальном времени.

Актуальность разработки игр в жанре Tower Defense обусловлена их универсальностью: они подходят как для казуальных игроков, так и для тех, кто ищет глубокий стратегический опыт. Такие проекты, как «Kingdom Rush» и «Bloons TD», демонстрируют коммерческий успех и устойчивый интерес аудитории. Кроме того, жанр предоставляет широкие возможности для экспериментов с механиками, такими как улучшение башен, управление ресурсами или внедрение героев с уникальными способностями.

Целью данного курсового проекта является разработка концепции и прототипа игры в жанре Tower Defense на игровом движке Unity, предназначенной для операционной системы Windows. Задачи включают анализ предметной области, проектирование игровых механик, создание интуитивного интерфейса и реализацию базового функционала, соответствующего требованиям жанра. Проект ориентирован на создание увлекательной игры с акцентом на стратегическое планирование и динамичный геймплей, способной привлечь внимание современной аудитории.

# **АНАЛИЗ** ПРЕДМЕТНОЙ ОБЛАСТИ

## Обзор аналогов

Для разработки игры в жанре Tower Defense необходимо изучить успешные проекты, которые задают стандарты качества в жанре, предлагают проверенные механики и демонстрируют подходы к вовлечению аудитории. Анализ аналогов позволяет выявить ключевые элементы успешных игр, включая геймплей, визуальный стиль, техническую реализацию и особенности интерфейса. Ниже представлен обзор трёх популярных игр в жанре Tower Defense, которые послужат ориентирами для проектирования разрабатываемого прототипа.



Рисунок 1.1 – Постер игры «Kingdom Rush»

«Kingdom Rush» (2011, Ironhide Game Studio) — эталон жанра Tower Defense, доступный на ПК, мобильных устройствах и в браузерах. Игрок защищает средневековое королевство от орд врагов (орки, тролли, драконы), размещая башни четырех типов: лучники, маги, артиллерия и казармы. Каждая башня имеет ветку улучшений, позволяющую адаптировать стратегию к различным типам врагов (например, летающим или бронированным). Уникальной особенностью является система героев — управляемых персонажей с уникальными способностями, которые добавляют динамики в геймплей. Уровни имеют разветвленные пути, что требует от игрока тактического планирования.

Игра получила высокие оценки (93% положительных отзывов на Steam из 14,672) благодаря сбалансированному сочетанию простоты управления и стратегической глубины. Визуальный стиль — мультяшная 2D-графика с яркими цветами — делает игру доступной для широкой аудитории. Однако, в отличие от Kingdom Rush, разрабатываемая игра будет ориентирована на минималистичный визуальный стиль, чтобы снизить системные требования.



Рисунок 1.2 – Постер игры «Bloons TD 6»

«Bloons TD 6» (2018, Ninja Kiwi) — одна из самых популярных современных игр жанра, доступная на ПК, iOS и Android. Игрок защищает территорию от волн разноцветных воздушных шаров, используя башни в виде обезьян, каждая из которых обладает уникальными способностями (например, метание дротиков, магические заклинания, взрывы). Ключевая особенность — глубокая система улучшений: каждая башня имеет три ветки апгрейдов, позволяющих создавать специализированные стратегии.

Игра поддерживает кооперативный мультиплеер, что повышает «реиграбельность», а также предлагает сотни уровней с различными режимами сложности. «Bloons TD 6» имеет свыше 250,000 отзывов на Steam (97% положительных), что подтверждает её коммерческий успех. Технически игра использует 2D-графику с оптимизацией для мобильных устройств, а её интерфейс интуитивно понятен даже новичкам. Игра представлена на рисунке 1.2.

«Plants vs. Zombies» (2009, PopCap Games) — культовая игра, которая популяризировала жанр Tower Defense среди казуальных игроков. Игрок защищает дом от зомби, размещая растения с различными способностями (например, стреляющие горохом, замедляющие или взрывающиеся).

Игра выделяется ярким мультяшным стилем, юмористическим подходом и простотой механик, что делает её доступной для всех возрастов. Уровни представляют собой линейные дорожки, а ресурсы (солнце) собираются во время игры, что добавляет элемент управления экономикой. «Plants vs. Zombies» получила 93% положительных отзывов на Steam и остается популярной благодаря своей универсальности. С технической точки зрения игра использует 2D-графику и минимальные системные требования, что делает её примером оптимизации. Для разрабатываемого проекта «Plants vs. Zombies» полезна как пример создания интуитивного интерфейса и баланса между простотой и глубиной. Постер игры представлен на рисунке 1.3.



Рисунок 1.3 – Постер игры «Plants vs. Zombies»

## Постановка задачи

Целью курсового проекта является разработка многопользовательской сетевой компьютерной игры в жанре «база против базы» с элементами стратегии и обороны. Игровой процесс должен осуществляться в реальном времени и предполагает синхронное взаимодействие двух клиентов по локальной сети. Задача проекта — спроектировать и реализовать клиентскую часть игры с использованием игрового движка Unity и сетевого взаимодействия на базе TCP и UDP протоколов, сочетающую элементы стратегии и тактического планирования, минималистичный дизайн и простоту управления. Цель игры — разрушить вражескую базу путём последовательных атак, одновременно защищая свою.

В процессе разработки должны быть реализованы следующие функции:

* Поддержка «мультиплеерного» режима для двух игроков, позволяющего соревноваться в реальном времени через локальную сеть;
* Реализация трех типов башен для каждого игрока, различающихся по функциональности, для защиты базы от юнитов противника;
* Введение экономической системы, в которой игроки начинают с равного количества очков и зарабатывают дополнительные очки за уничтожение юнитов противника башнями;
* Возможность выпуска юнитов, выступающих в роли врагов для базы соперника, с одинаковыми характеристиками для обоих игроков;
* Организация игрового процесса, направленного на защиту своей базы и уничтожение базы противника;
* Интуитивного понятный интерфейс.

Дополнительные функции, которые должны быть реализованы:

* Обеспечение простого управления размещением башен и выпуском юнитов с помощью мыши;
* Добавление настроек игры,;
* Определение победителей;
* Система «ачивок» (достижений).

Проект должен быть реализован с использованием игрового движка Unity, который обеспечит поддержку 3D-графики и базовой игровой логики, необходимой для создания «мультиплеерной» игры. Разработка программного кода будет осуществляться на языке программирования C# в среде Visual Studio 2022, что позволит эффективно реализовать игровые механики и логику взаимодействия. Особое внимание должно уделяться сетевой логике, фильтрации урона «свой-чужой», синхронному отображению объектов. Звуковое сопровождение должно включать эффекты для игровых действий, таких как выстрелы и уничтожение юнитов, а также фоновую музыку в формате .mp3.

# ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

## 2.1 Структура программы

Программа представляет собой многомодульную систему, реализующую многопользовательское взаимодействие двух клиентов в реальном времени. Разработка выполнена на языке C# с использованием игрового движка Unity и собственной реализации сетевого обмена на основе TCP и UDP протоколов. Структура приложения разделена на несколько ключевых функциональных компонентов, обеспечивающих взаимодействие между игроками, синхронизацию действий и реализацию игровой логики.

Основные модули:

* NetworkConnector — модуль сетевого взаимодействия, реализующий соединение между игроками (режим хост/клиент), обработку TCP-сообщений (надежная передача команд) и UDP-сообщений (быстрая синхронизация, например, координат и выстрелов).
* GameManagerMulty — центральный управляющий скрипт, отвечающий за игровые состояния, логику размещения защитников, обработку попаданий и урона, а также за UI-интерфейс и конечные события (победа/поражение).
* Defender\_AI и Enemy\_AI — скрипты управления искусственным интеллектом защитников и вражеских юнитов, включая поиск целей, стрельбу и движение по маршрутам.
* Weapon и Projectile — компоненты, реализующие стрельбу защитников и врагов, контроль направления и визуализацию попаданий, включая синхронизацию по сети.
* Main\_Menu — модуль отображения и управления главным меню игры.
* Health — универсальный компонент, реализующий здоровье для всех типов объектов (врагов, защитников, башен), а также управление смертью, визуальной шкалой здоровья и начислением очков.
* UI-модуль — обеспечивают отображение интерфейса пользователя: здоровья башен, количества очков, окон победы/поражения.

Каждый игрок управляет собственным экземпляром игры, при этом взаимодействие между ними осуществляется через сетевые сообщения. Вся логика взаимодействия построена с учётом разграничения сторон (игрок A и игрок B), что позволяет каждому клиенту обрабатывать только свой сегмент ответственности (например, урон по своей башне).

Программа предусматривает поддержку синхронного старта, автоматического поиска ближайших целей, динамического размещения защитников и сетевой передачи всех ключевых игровых событий. Отдельное внимание уделено корректному отображению событий на обеих сторонах и предотвращению дублирования урона или ошибок синхронизации.

Таким образом, структура программы соответствует архитектуре распределённого клиентского приложения с высокой степенью взаимодействия и зависимостей между модулями, обеспечивая корректную работу игрового процесса в сетевом режиме.

**2.2 Проектирование интерфейса программного средства**

Пользовательский интерфейс многопользовательской игры в жанре Tower Defense проектируется с учетом принципов минимализма, интуитивности и функциональности, чтобы обеспечить удобное взаимодействие для двух игроков, соревнующихся по локальной сети. Интерфейс разрабатывается с использованием встроенных инструментов Unity UI и включает несколько ключевых экранов, каждый из которых выполняет определенные функции, поддерживая динамичный игровой процесс и предоставляя игрокам необходимую информацию в реальном времени. Дизайн интерфейса ориентирован на простоту, четкость и оптимизацию, чтобы минимизировать отвлекающие элементы и обеспечить быструю адаптацию игроков к управлению.

Главное меню служит отправной точкой для взаимодействия с игрой. Оно содержит кнопки для создания нового матча, подключения к матчу, доступа к настройкам, достижениям и выхода из игры, а также поля для ввода IP-адреса для подключения к локальной сети. Дизайн меню выполнен в минималистичном стиле с использованием нейтральной цветовой палитры и четких шрифтов, чтобы обеспечить читаемость.

На рисунке 2.1 представлен экран главного меню.

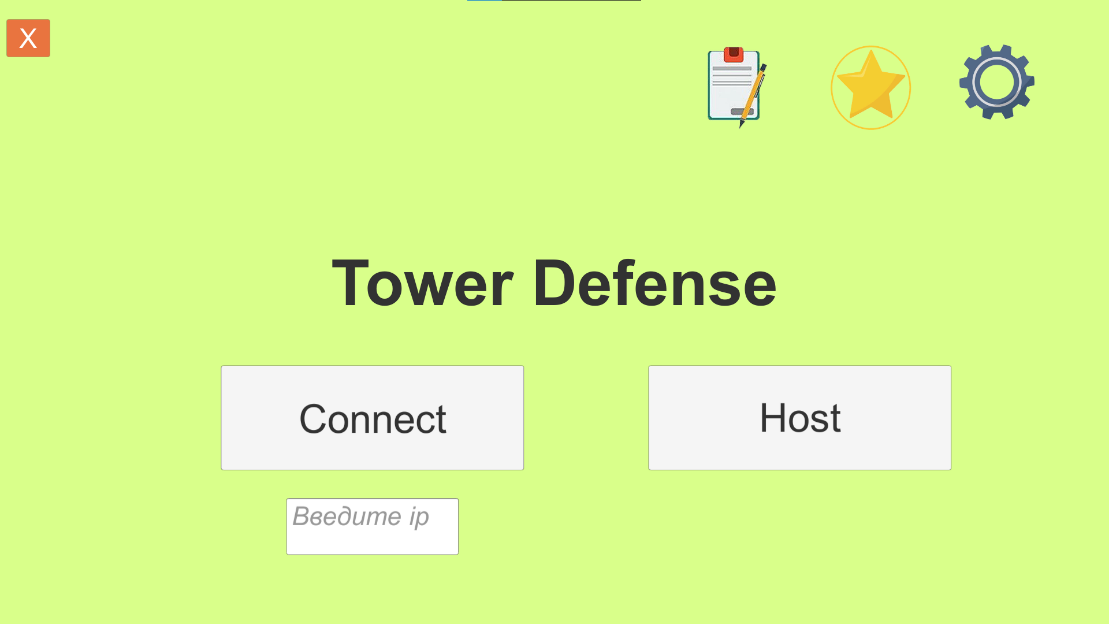


Рисунок 2.1 – Главное меню.

Игровой экран является центральным элементом интерфейса, обеспечивая игроков всей необходимой информацией во время матча. Этот экран отображает ключевые данные: здоровье собственной базы и базы противника, текущее количество очков. Панель управления, расположенная в нижней части экрана, содержит иконки для размещения башен, выпуска юнитов и просмотра их характеристик, таких как стоимость в очках. Визуальные индикаторы, такие как анимации атак башен или мигание шкалы здоровья при получении урона, проектируются для быстрого восприятия, что критично в динамичном «мультиплеерном» режиме. Элементы игрового используют контрастные цвета и простые формы. Игровой экран представлен на рисунке 2.2.

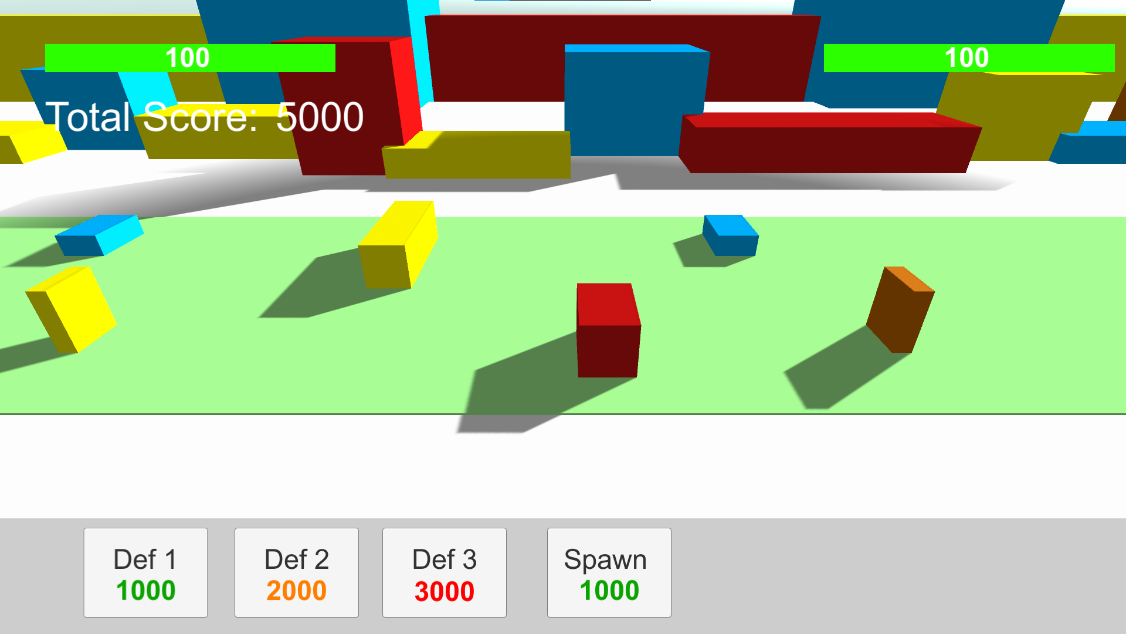


Рисунок 2.2 – Игровой экран.

Экран результатов появляется после завершения матча. Этот экран включает кнопки для начала нового матча, возврата в главное меню, а также возможность просмотра статистики в виде таблицы для анализа стратегии. Дизайн экрана результатов сохраняет минималистичную эстетику, фокусируясь на четком представлении данных. Экран настроек предоставляет игрокам возможность отключать или включать звук. Настройки доступны из главного меню и игрового экрана, чтобы игроки могли вносить изменения без прерывания матча.



Рисунок 2.3 – Экран статистики.

Все элементы интерфейса проектируются с учетом 2D-графики, используя спрайты для иконок, кнопок и индикаторов, что соответствует минималистичному стилю игры. Шрифты выбираются с высокой читаемостью, а цветовая схема избегает избыточной яркости, чтобы не отвлекать от игрового процесса. Интерфейс оптимизирован для управления мышью, что обеспечивает удобство в мультиплеере. Адаптивность интерфейса гарантирует корректное отображение на различных разрешениях экрана, поддерживаемых Windows, а интеграция с Unity UI обеспечивает отзывчивость и минимальную нагрузку на производительность.

## 2.3 Проектирование функционала программного средства

При проектировании программного средства была выбрана концепция мультиплеерной игры в жанре стратегической обороны базы с применением сетевого взаимодействия между двумя игроками. Главной задачей является защита своей башни и одновременное нападение на вражескую, что требует точной синхронизации действий и состояния игровых объектов в реальном времени.

В рамках проекта реализуются следующие ключевые функции:

– размещение защитников игроком на своей стороне карты;

– спавн вражеских юнитов и их автоматическое движение к базе соперника;

– стрельба защитников по вражеским юнитам с прицеливанием и синхронизацией;

– синхронизация состояния здоровья башен и победа/поражение в случае их разрушения;

– учет и расходование игровых ресурсов (монет) при размещении защитников и вызове атакующих юнитов;

– передача всех ключевых событий через сетевое подключение по TCP/UDP между игроками.

**2.3.1** Размещение защитников

Каждый игрок имеет возможность разместить защитников на своей территории, если у него достаточно очков. Игрок кликает по игровой области, выбирает тип защитника и размещает его. После подтверждения действия информация о типе защитника, его координатах и ориентации отправляется по сети, чтобы второй игрок мог отобразить его у себя. До момента размещения башня неактивна и не участвует в бою. Блок-схема логики размещения защитника представлена на рисунке 2.4.

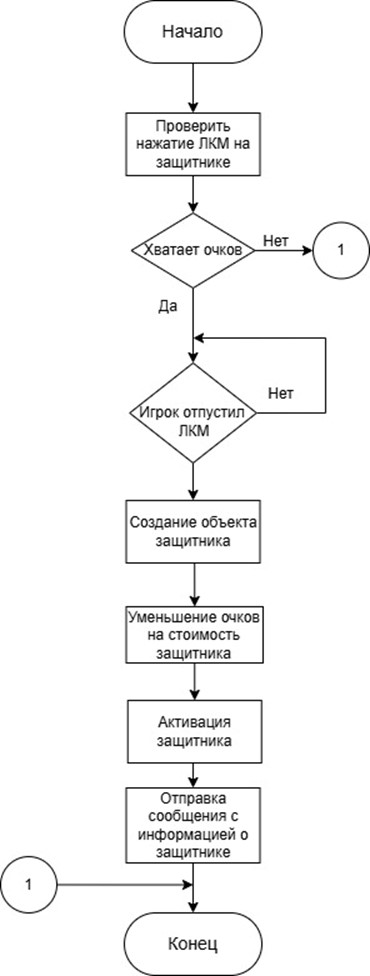


Рисунок 2.4 – Блок-схема метода CreateDefender().

**2.3.2** Спавн атакующих юнитов

Игрок может потратить ресурсы на создание юнита, атакующего вражескую базу. Префабы врагов разделены по принадлежности: часть идет от базы A к базе B, другая — в обратном направлении. При создании юнита его начальная позиция, сторона и уникальный ID отправляются через TCP, чтобы на стороне противника он также был создан.

Блок-схема генерации врагов представлена на рисунке 2.5.

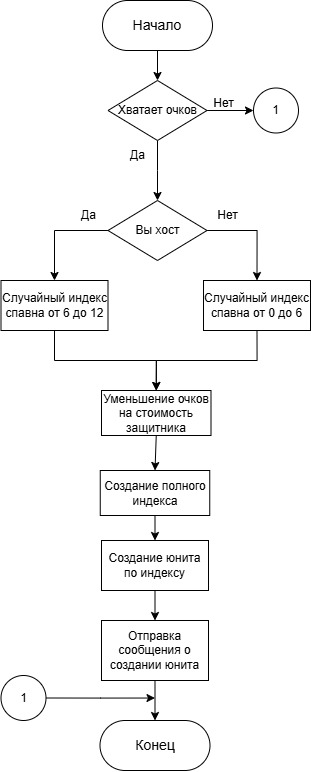


Рисунок 2.5 – Блок-схема метода SpawnOneEnemy().

**2.3.3** Стрельба и обработка урона

Каждый защитник выбирает ближайшего вражеского юнита, вращается в его сторону и периодически выпускает снаряды. Пуля создаётся локально и синхронизируется на стороне второго игрока через UDP. При столкновении пули с юнитом проверяется принадлежность (по стороне), и если цель вражеская — ей наносится урон. Если цель уничтожена, это событие передаётся по TCP, и объект уничтожается у обоих игроков.

Блок-схема стрельбы защитника показана на рисунке 2.6.



Рисунок 2.6 – Блок-схема метода Weapon.Start().

**2.3.4** Синхронизация состояния башен и победа

Башни обеих сторон имеют начальное здоровье. Когда вражеский юнит достигает башни, он наносит ей урон. Только владелец башни обновляет её здоровье локально, а затем по TCP сообщает оппоненту, чтобы у него отобразилось изменение. При падении здоровья до нуля у одного игрока — он видит окно «Поражение», а второй — «Победа».

Блок-схема обработки урона по башне приведена на рисунке 2.7.

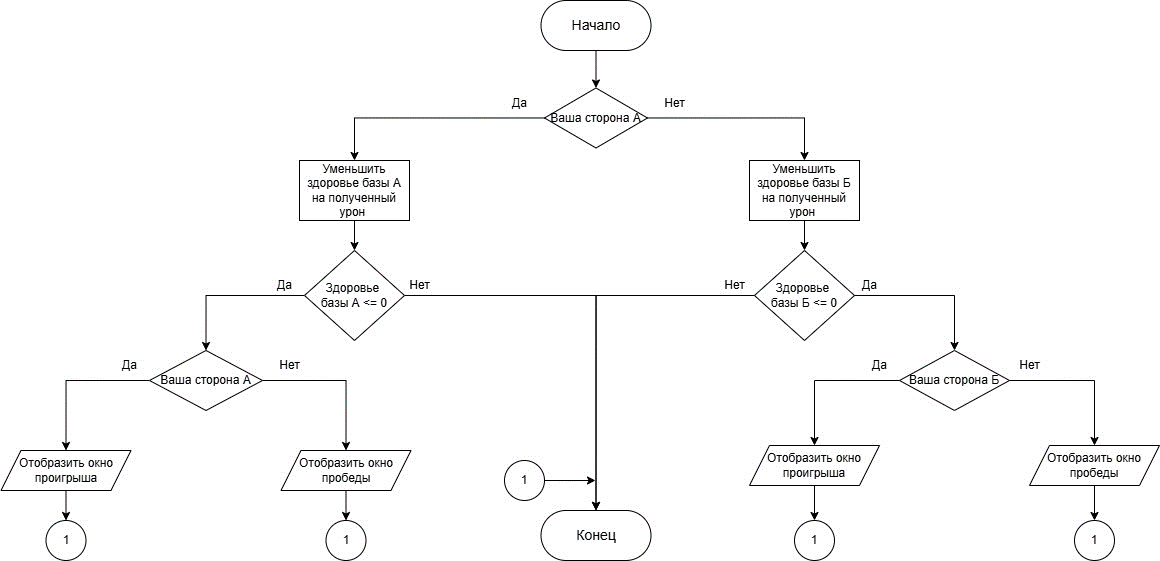


Рисунок 2.6 – Блок-схема метода Reduce\_Tower\_Health().

# 3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

## 3.1 Работа с сетевым взаимодействием

Сетевое взаимодействие является ключевым компонентом программного средства и обеспечивает корректную синхронизацию игровых событий между двумя участниками. В разработанной компьютерной сетевой игре в жанре стратегии используется собственная реализация соединения по протоколам TCP и UDP, что позволяет эффективно разделять обработку критически важных и второстепенных данных. Сервер реализован таким образом, что один из игроков выступает в роли хоста, а второй подключается к нему по IP-адресу. При запуске игры хост запускает TCP и UDP прослушивание, а клиент инициализирует соединение и начинает приём сообщений.

Для организации сетевого взаимодействия применяется модуль NetworkConnector, который управляет инициализацией подключения, обменом сообщениями, а также прослушиванием входящих пакетов. Игрок, запустивший игру первым, становится Host, второй — Client. После подключения к общей игровой сессии происходит загрузка основной сцены и синхронизация начальных параметров. Подключение клиента к хосту происходит посредствам TCP.

Пример запуска хоста:  
public void StartHost()

{

isHost = true;

string localIP = GetLocalIPAddress();

ipDisplayText.text = $"Your IP: {localIP}";

Debug.Log("Set IP text to UI: " + ipDisplayText.text);

tcpListener = new TcpListener(IPAddress.Any, tcpPort);

tcpListener.Start();

tcpThread = new Thread(() =>

{

tcpClient = tcpListener.AcceptTcpClient();

Debug.Log(" Client connected!");

stream = tcpClient.GetStream();

StartTCPListening();

remoteIP = ((IPEndPoint)tcpClient.Client.RemoteEndPoint).Address.ToString();

StartUDP(remoteIP);

SceneManager.LoadScene(gameplaySceneName);

});

tcpThread.IsBackground = true;

tcpThread.Start();

}

Как мы можем заметить хост начинает слушание на своём порту любых адресов и затем подключает клиента в дополнительном потоке. В этом же потоке начинается приём TCP и UDP пакетов.

Пример потока для приёма UDP пакетов:

udpListenThread = new Thread(() =>

{

IPEndPoint remoteEP = new IPEndPoint(IPAddress.Any, udpPort);

while (true)

{

try

{

byte[] data = udpReceiver.Receive(ref remoteEP);

string msg = Encoding.UTF8.GetString(data);

Debug.Log(" UDP Received: " + msg);

OnUDPMessageReceived?.Invoke(msg);

}

}

});

udpListenThread.IsBackground = true;

udpListenThread.Start();

Все полученные UDP пакеты проходят декодировку поскольку пакеты отправляются байтами в кодировке UTF8, дальше уже декодированное сообщение отбавляется на расшифровку, т.к логика значений сообщений реализована через систему префиксов. Похожая логика и при приёме TCP пакетов.

TCP используется для обмена важными событиями, такими как размещение защитников, нанесение урона по базе, уничтожения врага.

Такие события имеют решающее значение для игрового процесса и должны быть доставлены гарантированно и в правильной последовательности. Для их обработки создаются команды в текстовом формате, которые рассылаются противоположной стороне и вызывают соответствующую логику обновления.

UDP применяется для передачи событий, связанных с визуальной частью и не критичных к потере пакетов либо событий который должны быть быстро доставлены, а именно синхронизации выстрелов защитников и позиций юнитов. Это позволяет минимизировать задержки и сохранить плавность геймплея даже при нестабильном соединении. В случае потери одного из пакетов (например, позиции врага), визуальная картина быстро корректируется следующим сообщением.

Как уже говорилось выше система сообщений разбирается по префиксам:

* SPAWN\_DEFENDER — размещение защитника;
* DEF\_SHOOT — выстрел защитника;
* TOWER\_HIT — попадание по башне;
* ENEMY\_POS — позиция врага;
* SPAWN\_ENEMY\_GLOBAL — создание юнита;
* ENEMY\_DEAD — уничтожение врага.

Каждое сообщение имеет заранее определённый формат. Например:

SPAWN\_DEFENDER;1;10.5;0;3.2;90;A

Где 1 — индекс защитника, далее идут координаты, угол поворота и сторона, которой принадлежит защитник. Остальные префиксные сообщения формируются похожим образом.

На стороне приёмника каждый тип сообщения обрабатывается в методах OnTCPMessage(string) или OnUDPMessage(string). Внутри этих методов происходит парсинг строки и выполнение соответствующих действий — создание объекта, запуск анимации или обновление состояния.Пример:

void SpawnMsg(string message)

{

if (message.StartsWith("SPAWN\_DEFENDER"))

{

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

if (parts.Length < 7) return;

int defIndex = int.Parse(parts[1]);

float x = float.Parse(parts[2]);

float y = float.Parse(parts[3]);

float z = float.Parse(parts[4]);

float rotY = float.Parse(parts[5]);

string side = parts[6];

GameManagerMulty gm = FindObjectOfType<GameManagerMulty>();

if (gm != null)

{

Vector3 pos = new Vector3(x, y, z);

Quaternion rot = Quaternion.Euler(0, rotY, 0);

gm.SpawnDefenderRemote(defIndex, pos, rot, side);

}

}

}

## 3.2 Реализация функционала юнитов

В рамках разработки программного средства были реализованы вражеские юниты, которые представляют собой основную угрозу для базы противника. Функциональность юнитов включает в себя автономное перемещение, атаку башни, получение урона и синхронизацию с другим игроком через сеть.

Каждый вражеский юнит обладает следующими характеристиками: принадлежность к одной из сторон — A или B, маршрут движения от своей базы к базе противника(всего двенадцать маршрутов, шесть от базы А к базе Б, ещё шесть наоборот), способность атаковать башню противника при достижении цели, визуальные и анимационные компоненты для отображения состояния (например, смерть, стрельба).

После спавна на определённой точке юнит начинает движение по заранее определённому маршруту, реализованному с использованием компонента NavMeshAgent. При достижении позиции вблизи башни происходит остановка и активация боевого поведения.

Атака осуществляется в режиме «дальнего боя»: юнит запускает снаряд в сторону башни, нанося урон по таймеру (в соответствии с интервалом стрельбы). Поведение управляется скриптом Weapon, который активирует выстрелы при обнаружении цели в радиусе. Пример стрельбы:

if (canShoot)

{

if (shootingMode == ShootingMode.Gun)

{

var bullet = Instantiate(projectile, shootPoint.position, shootPoint.rotation);

var health = GetComponent<Health>();

if (bullet.TryGetComponent<Projectile>(out var proj) && health != null)

{

proj.GetComponent<Projectile>().SendMessage("SetSide", health.side, SendMessageOptions.DontRequireReceiver);

}

if (bullet.TryGetComponent(out Projectile owner))

owner.isOwner = isOwner;

if (isOwner)

{

Vector3 shootPos = shootPoint.position;

Vector3 shootDir = shootPoint.forward;

string msg = $"DEF\_SHOOT;{shootPos.x:F2};{shootPos.y:F2};{shootPos.z:F2};{shootDir.x:F2};{shootDir.y:F2};{shootDir.z:F2}";

NetworkConnector.Instance.SendUDP(msg);

}

bullet.GetComponent<Rigidbody>().AddForce(shootPoint.forward \* force);

При попадании снаряда защитника в юнита активируется метод ApplyDamage, который уменьшает количество очков здоровья. В случае достижения нуля — юнит уничтожается, играется визуальный эффект и начисляются очки стороне, уничтожившей врага.

Особенностью реализации стало разделение ответственности: каждый игрок управляет только «своими» юнитами, а другой получает лишь отображение (визуальную копию) их состояния. Это позволило значительно снизить объём сетевого трафика и упростить логику. Юниты получают уникальный Id, по которому осуществляется их синхронизация.Уникальный id формируется из трёх компонентов: globalEnemyIndex (число от 0 до 11,которое указывает на каком маршруте юнит был создан), spawnIndex (число от 0 до 11 по которому можно определить какой стороне принадлежит юнит), enemyId (комбинированный идентификатор который представляет собой два предыдущих числа и ещё одно в диапазоне от 1000 до 9999,нужен для минимизации сличая создания юнитов с одинаковыми id).

## 3.3 Реализация функционала защитников

Защитники — это ключевой элемент обороны базы игрока. Они автоматически атакуют вражеские юниты, выполняя функцию стационарных турелей. Игрок может устанавливать защитников вручную в любое место на карте, при этом каждый защитник привязан к стороне игрока (A или B) и не взаимодействует со своими союзными юнитами.

Размещение происходит в два этапа:

1. Игрок нажимает кнопку выбора защитника и начинает перетаскивание;
2. После отпускания мыши, защитник размещается в указанной точке, и его функциональность активируется.

void CreateDefender()

{

if (isDraging)

{

Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

if (Physics.Raycast(ray, out RaycastHit hit, 1000))

{

if (hit.transform.CompareTag("Ground"))

{

if (canInstantiate)

{

canInstantiate = false;

currentDraggingDefender = Instantiate(defenders[currentDefender - 1], hit.point, Quaternion.identity);

Renderer rend = currentDraggingDefender.GetComponentInChildren<Renderer>();

if (rend != null)

{

Color color = (mySide == "A") ? Color.blue : Color.red;

rend.material.color = color;

}

if (currentDraggingDefender.TryGetComponent(out Defender\_AI ai))

ai.mySide = this.mySide;

createdDefenders.Add(currentDraggingDefender);

createdDefCounts++;

if (currentDraggingDefender.TryGetComponent(out Weapon weapon))

weapon.canShoot = false;

}

else if (currentDraggingDefender != null)

{

currentDraggingDefender.transform.position = hit.point;

Vector3 lookPos = FindClosestPoints("Center Point").position - currentDraggingDefender.transform.position;

lookPos.y = 0;

currentDraggingDefender.transform.rotation = Quaternion.Slerp(currentDraggingDefender.transform.rotation, Quaternion.LookRotation(lookPos), Time.deltaTime \* 1000);

}

}

}

}

}

До завершения размещения защита неактивна — это предотвращает преждевременную стрельбу.

После установки защитник окрашивается в соответствующий цвет стороны игрока и получает параметр mySide, по которому в дальнейшем фильтруются цели.

Каждый защитник имеет автоматическую систему прицеливания (Defender\_AI), которая периодически ищет ближайшего вражеского юнита, принадлежащего противоположной стороне. Если цель обнаружена и находится в радиусе действия, включается стрельба через компонент Weapon.

Снаряды, выпущенные защитником, наносят урон только противникам. Это реализуется через фильтрацию по параметру side в компоненте Health у целей. Урон по союзным юнитам невозможен.

Для синхронной визуализации выстрелов на стороне другого игрока используется протокол UDP. При каждом выстреле отправляется сообщение DEF\_SHOOT, содержащее координаты и направление. Другой клиент воспроизводит идентичный снаряд в том же направлении.

# ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

В этом разделе описывается процесс тестирования игры «Tower Defense Game», который включает в себя цели, этапы и методы, применяемые для подтверждения корректности ее функционирования. Основной задачей тестирования является выявление ошибок и подтверждение соответствия проекта заданным требованиям.

Для проверки работоспособности было проведено ручное тестистирование.

Цели:

– Проверить корректность работы игровых механик (движение юнитов, взаимодействие башен и врагов, получение урона базой).

– Убедиться в стабильности сетевого взаимодействия

– Оценить интуитивность интерфейса.

– Проверить производительность.

– Выявить и устранить возможные ошибки влияющие на игровой процесс.

Было проведено несколько матчей, в которых были обнаружены незначительные ошибки синхронизации позиции врагов ,которые заключались в неправильной расшифровке полученных сообщений , но они были быстро исправлены и остались лишь некоторые предупреждения, связанные с реорганизацией структуры проекта. Пример предупреждения представлен на рисунке 4.1.



Рисунок 4.1 – Предупреждение об отсутствии сцены.

# РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Руководство пользователя предназначено для игроков, использующих программное средство «Tower Defense Game» на операционной системе Windows. Оно описывает интерфейс игры, управление и основные шаги для начала игры, обеспечивая простое освоение.

## **5.1 Интерфейс программного средства**

Дизайн меню выполнен в минималистичном стиле с использованием нейтральной цветовой палитры и четких шрифтов, чтобы обеспечить читаемость.. Главное меню содержит кнопки для создания нового матча(Host), подключения к матчу(Connect), доступа к настройкам(Шестерёнка), достижениям(Звезда),статистике(Блокнот с ручкой ) и выхода из игры(белый крестик в красном прямоугольнике), а также поля для ввода IP-адреса для подключения к локальной сети. Фон представляет собой монотонный экран.

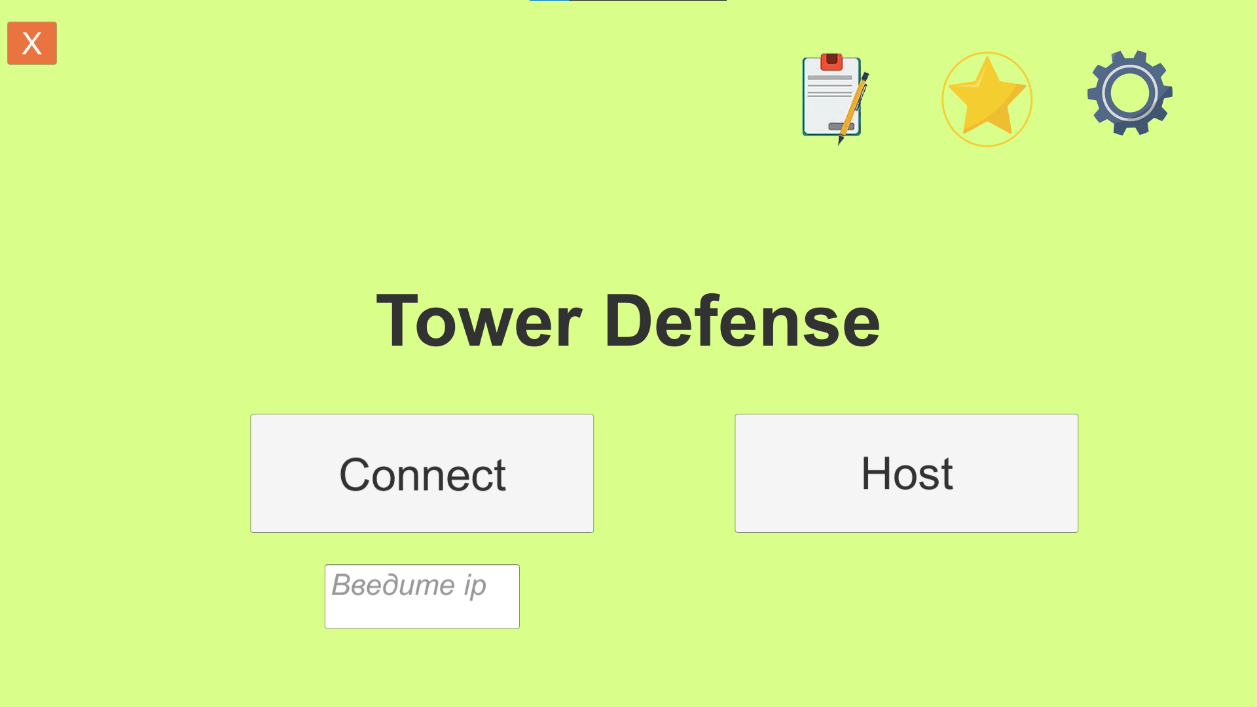


Рисунок 5.1 – Главное меню.

Меню настроек представляет из себя экран ,который появляется поверх основного и имеет два элемента типа «checkbox» для постобработки и включения/отключения звука, а также кнопку для закрытия этого экрана.

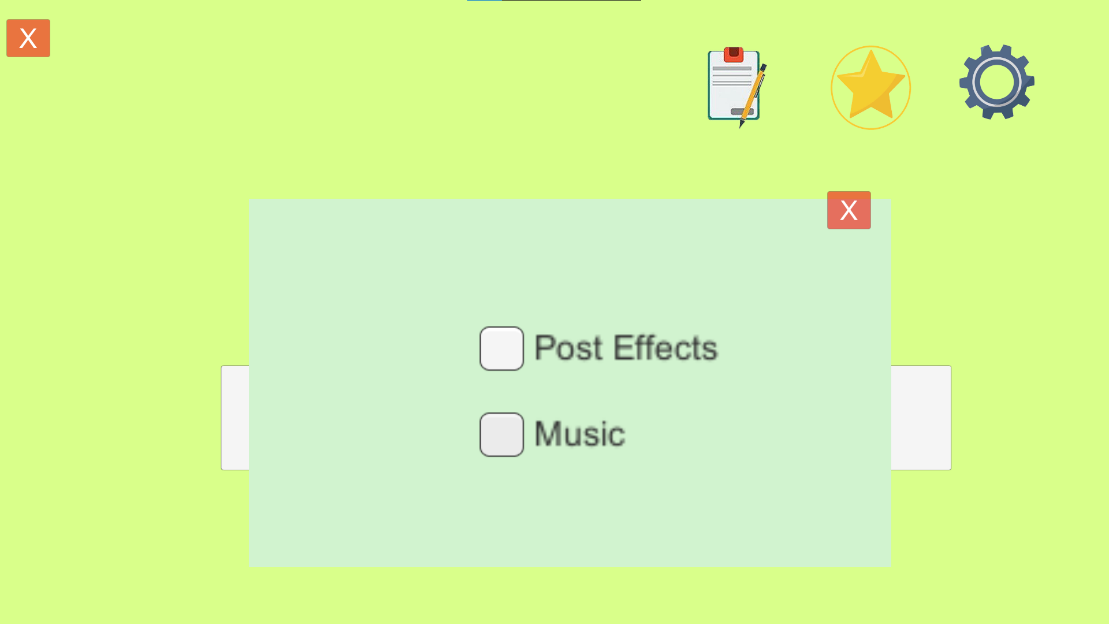


Рисунок 5.2 – Меню настроек.

Меню достижений представляет из себя экран ,который появляется поверх основного и имеет множества достижений которые достиг игрок, а также кнопку для закрытия этого экрана.



Рисунок 5.3 – Меню достижений.

Меню статистики представляет из себя экран ,который появляется поверх основного и имеет статистику по общим убийствам и очкам игрока, а также кнопку для закрытия этого экрана.



Рисунок 5.4 – Меню статистики.

Игровой экран является центральным элементом интерфейса. На этом экран отображает ключевые данные: здоровье собственной базы и базы противника, текущее количество очков(Total Score). Панель управления, расположенная в нижней части экрана, содержит кнопки для размещения башен(Def 1, Def 2, Def 3), выпуска юнитов(Spawn) и просмотра их характеристик,таких как стоимость в очках. Визуальные индикаторы, такие как анимации атак башен или уменьшение шкалы здоровья при получении урона, проектируются для быстрого восприятия, что критично в динамичном «мультиплеерном» режиме.

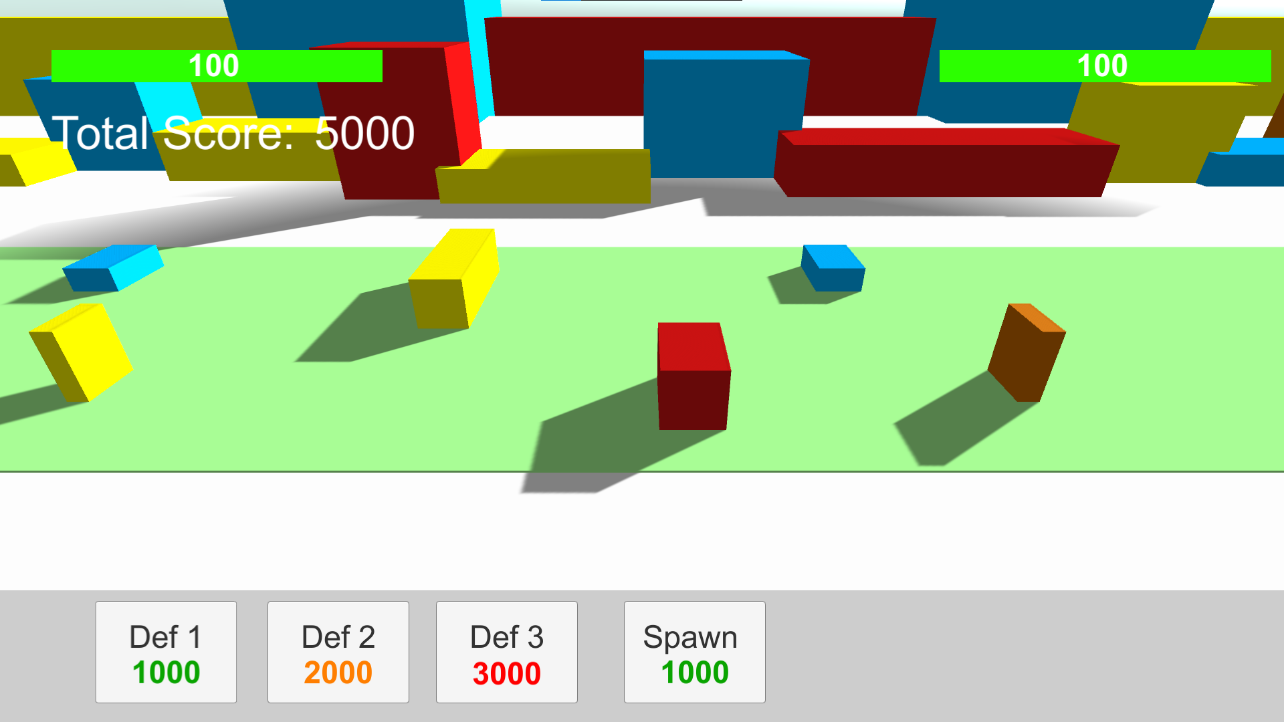


Рисунок 5.5 – Игровой экран.

Экран результатов появляется после завершения матча. Этот экран включает кнопки для начала нового матча, возврата в главное меню.



Рисунок 5.6 – Экран поражения.

## **5.2 Управление программным средством**

Для запуска игры убедитесь, что присутствует интернет-подключение и два игрока находятся в локальной сети. Чтобы запустить игру, нужно открыть TowerDefenseGame.exe файл. После запуска вы увидите меню (рисунок 5.1).

Для создания матча нужно нажать на кнопку «Host» и вам покажут ваш IP-адрес (Рисунок 5.7). Если же вы хотите подключиться, то вы вводите IP-адрес хоста в поле где написано «введите ip» и жмёте на кнопку «Connect».

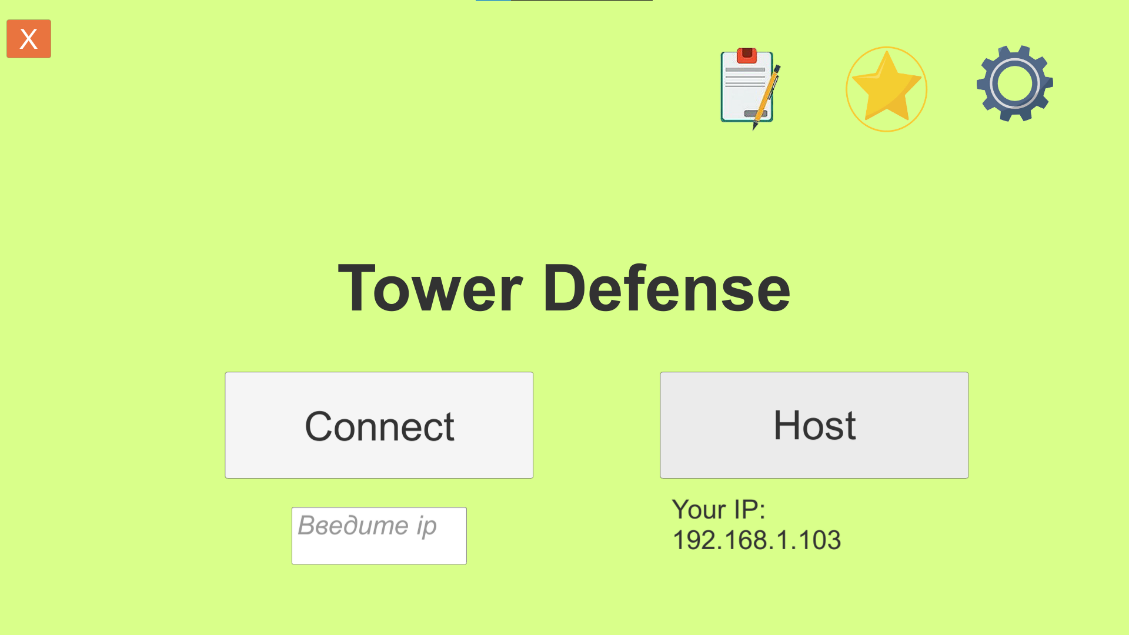


Рисунок 5.7 – Главное меню при создании матча.

Навигация по интерфейсу происходит с помощью:

– Мыши – выбор кнопок меню, полей для ввода,чекбоксов.

– Клавишой Esc (вызов меню).

В игровом процессе используются мышь для создания юнитов, упрвление камерой путём перетягивания, управление позицией башни путём перетаскивания её на доступную позицию.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта была разработана 3D-мультиплеерная игра в жанре Tower Defense на движке Unity с использованием языка программирования C#. Главной задачей стало создание сетевого взаимодействия между двумя игроками без применения сторонних библиотек, что позволило глубже погрузиться в принципы передачи и обработки данных через протоколы TCP и UDP. Разработка включала реализацию спавна юнитов, передачи их позиции, синхронизации стрельбы, обработки урона и состояния здоровья баз. Были реализованы две стороны конфликта — база A и база B, каждая из которых имеет собственную команду, экономику, защиту и систему управления. Игроки могут строить защитные башни, накапливать и тратить ресурсы, а также отправлять в бой атакующих юнитов, взаимодействующих с элементами другой стороны.

Игровой процесс синхронизируется между клиентами: один из игроков отвечает за контроль юнита, передавая его позицию другому, в то время как на стороне второго игрока юнит двигается автономно, получая корректирующие данные. Это позволило достичь устойчивой передачи информации и воспроизведения боевых действий без необходимости в централизованном сервере. Были решены ключевые задачи оптимизации сетевого взаимодействия, обеспечения плавности движения юнитов и независимой работы логики защиты и нападения на каждой стороне. Игра сопровождается минималистичным визуальным оформлением, четко различающим стороны конфликта с помощью цветовой маркировки, а также предоставляет простой и интуитивно понятный пользовательский интерфейс.

Разработанный прототип подтвердил свою работоспособность при тестировании, показал стабильную синхронизацию действий игроков и продемонстрировал потенциал для расширения. В дальнейшем проект может быть дополнен поддержкой нескольких игроков, расширенной системой прокачки, новыми типами юнитов и картами, а также возможностью игры на мобильных устройствах. Полученный опыт стал ценным практическим результатом изучения сетевых технологий в игровой разработке и позволил на практике реализовать базовые принципы построения мультиплеерных игр.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Документация по Visual Studio [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/ru-ru/visualstudio/ide/?view=vs-2022.  
 [2] Полное руководство по языку программирования С# – METANIT – [Электронный ресурс]. – Режим доступа: https://metanit.com/sharp/tutorial/

[3] Даны, Р. Основы C# и .NET: Полное руководство / Р. Даны. – М.: Диалектика, 2021. – 740 с.

[4] Holzner S. Unity 2022 Game Development Bible, 2022. — 1000 с.

[5] Гетц, Джо. C# 8.0: Новые возможности и примеры использования / Джо Гетц. – М.: Издательский дом «Солон-Пресс», 2021. – 536 с.

[6] Официальная документация Unity [Электронный ресурс]. – Режим доступа: https://docs.unity3d.com/Manual/index.html.

[7] Nystrom R. Game Development with Unity and C#, 2021. — 512 с.

[8]Стандарт предприятия [Электронный ресурс]. – Режим доступа: СТП\_2024.pdf.

[9] Окли, Дж. Сетевое программирование для разработчиков игр / Дж. Окли. – СПб.: Питер, 2019. – 432 с.

[10] Роджерс, С. Level Up! Руководство по созданию отличных видеоигр / С. Роджерс. – М.: Манн, Иванов и Фербер, 2020. – 528 с.

# ПРИЛОЖЕНИЕ А. Исходный код программы

NetworkConnector.cs:

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading;

using UnityEngine;

using UnityEngine.SceneManagement;

using UnityEngine.UI;

public class NetworkConnector : MonoBehaviour

{

[Header("UI")]

public Text ipDisplayText;

public InputField ipInputField;

[Header("Scene")]

public string gameplaySceneName = "Main\_Gameplay\_Multi";

private TcpClient tcpClient;

private TcpListener tcpListener;

private NetworkStream stream;

private Thread tcpThread;

private Thread listenThread;

public static Action<string> OnMessageReceived;

private UdpClient udpSender;

private UdpClient udpReceiver;

private Thread udpListenThread;

public static Action<string> OnUDPMessageReceived;

private const int tcpPort = 5555;

private const int udpPort = 5556;

public bool isHost = false;

private string remoteIP;

public void StartHost()

{

isHost = true;

string localIP = GetLocalIPAddress();

ipDisplayText.text = $"Your IP: {localIP}";

Debug.Log("Set IP text to UI: " + ipDisplayText.text);

tcpListener = new TcpListener(IPAddress.Any, tcpPort);

tcpListener.Start();

tcpThread = new Thread(() =>

{

tcpClient = tcpListener.AcceptTcpClient();

Debug.Log(" Client connected!");

stream = tcpClient.GetStream();

StartTCPListening();

remoteIP = ((IPEndPoint)tcpClient.Client.RemoteEndPoint).Address.ToString();

StartUDP(remoteIP);

SceneManager.LoadScene(gameplaySceneName);

});

tcpThread.IsBackground = true;

tcpThread.Start();

}

public void ConnectToHost()

{

isHost = false;

remoteIP = ipInputField.text.Trim();

tcpClient = new TcpClient();

try

{

tcpClient.Connect(remoteIP, tcpPort);

Debug.Log(" Connected to host!");

stream = tcpClient.GetStream();

StartTCPListening();

StartUDP(remoteIP);

SceneManager.LoadScene(gameplaySceneName);

}

catch (Exception ex)

{

Debug.LogError(" Connection failed: " + ex.Message);

}

}

void StartTCPListening()

{

listenThread = new Thread(() =>

{

byte[] buffer = new byte[1024];

while (tcpClient.Connected)

{

try

{

int bytesRead = stream.Read(buffer, 0, buffer.Length);

if (bytesRead > 0)

{

string message = Encoding.UTF8.GetString(buffer, 0, bytesRead);

Debug.Log(" TCP Received: " + message);

OnMessageReceived?.Invoke(message);

}

}

catch (Exception ex)

{

Debug.LogError("TCP listening error: " + ex.Message);

break;

}

}

});

listenThread.IsBackground = true;

listenThread.Start();

}

public void SendMessageToPeer(string message)

{

if (tcpClient != null && tcpClient.Connected)

{

byte[] data = Encoding.UTF8.GetBytes(message);

stream.Write(data, 0, data.Length);

stream.Flush();

Debug.Log(" TCP Sent: " + message);

}

}

void StartUDP(string ip)

{

try

{

udpSender = new UdpClient();

udpSender.Connect(ip, udpPort);

udpReceiver = new UdpClient(udpPort);

udpListenThread = new Thread(() =>

{

IPEndPoint remoteEP = new IPEndPoint(IPAddress.Any, udpPort);

while (true)

{

try

{

byte[] data = udpReceiver.Receive(ref remoteEP);

string msg = Encoding.UTF8.GetString(data);

Debug.Log(" UDP Received: " + msg);

OnUDPMessageReceived?.Invoke(msg);

}

catch (Exception ex)

{

Debug.LogError("UDP listen error: " + ex.Message);

break;

}

}

});

udpListenThread.IsBackground = true;

udpListenThread.Start();

}

catch (Exception ex)

{

Debug.LogError("UDP init error: " + ex.Message);

}

}

public void SendUDP(string message)

{

if (udpSender != null)

{

byte[] data = Encoding.UTF8.GetBytes(message);

udpSender.Send(data, data.Length);

Debug.Log(" UDP Sent: " + message);

}

}

private string GetLocalIPAddress()

{

string localIP = "Not found";

var host = Dns.GetHostEntry(Dns.GetHostName());

foreach (var ip in host.AddressList)

{

if (ip.AddressFamily == AddressFamily.InterNetwork)

{

localIP = ip.ToString();

break;

}

}

return localIP;

}

void Awake()

{

if (FindObjectsOfType<NetworkConnector>().Length > 1)

{

Destroy(gameObject);

return;

}

DontDestroyOnLoad(gameObject);

}

void OnApplicationQuit()

{

try

{

listenThread?.Abort();

tcpThread?.Abort();

stream?.Close();

tcpClient?.Close();

tcpListener?.Stop();

udpListenThread?.Abort();

udpSender?.Close();

udpReceiver?.Close();

}

catch (Exception e)

{

Debug.LogWarning("Cleanup failed: " + e.Message);

}

}

}

EnemySpawnReciver.cs:

using System.Globalization;

using UnityEngine;

public class EnemySyncReceiver : MonoBehaviour

{

void OnEnable()

{

NetworkConnector.OnUDPMessageReceived += HandleUDP;

}

void OnDisable()

{

NetworkConnector.OnUDPMessageReceived -= HandleUDP;

}

void HandleUDP(string msg)

{

if (!msg.StartsWith("ENEMY\_POS;")) return;

string[] parts = msg.Split(';');

if (parts.Length < 5) return;

string id = parts[1];

if (!EnemySyncTracker.All.TryGetValue(id, out var tracker))

return;

if (tracker.isOwner) return;

float x = float.Parse(parts[2].Replace(',', '.'), CultureInfo.InvariantCulture);

float y = float.Parse(parts[3].Replace(',', '.'), CultureInfo.InvariantCulture);

float z = float.Parse(parts[4].Replace(',', '.'), CultureInfo.InvariantCulture);

tracker.SetRemotePosition(new Vector3(x, y, z));

}

}

EnemySyncTracker:  
  
using System.Globalization;

using UnityEngine;

public class EnemySyncReceiver : MonoBehaviour

{

void OnEnable()

{

NetworkConnector.OnUDPMessageReceived += HandleUDP;

}

void OnDisable()

{

NetworkConnector.OnUDPMessageReceived -= HandleUDP;

}

void HandleUDP(string msg)

{

if (!msg.StartsWith("ENEMY\_POS;")) return;

string[] parts = msg.Split(';');

if (parts.Length < 5) return;

string id = parts[1];

if (!EnemySyncTracker.All.TryGetValue(id, out var tracker))

return;

if (tracker.isOwner) return;

float x = float.Parse(parts[2].Replace(',', '.'), CultureInfo.InvariantCulture);

float y = float.Parse(parts[3].Replace(',', '.'), CultureInfo.InvariantCulture);

float z = float.Parse(parts[4].Replace(',', '.'), CultureInfo.InvariantCulture);

tracker.SetRemotePosition(new Vector3(x, y, z));

}

}

EnemyRegistry.cs:  
  
using UnityEngine;

public class EnemyRegistry : MonoBehaviour

{

public static EnemyRegistry Instance;

[Header("Общий список из 12 врагов")]

public GameObject[] allEnemies; // 0–5 = B→A, 6–11 = A→B

void Awake()

{

if (Instance == null)

{

Instance = this;

DontDestroyOnLoad(gameObject);

}

else

{

Destroy(gameObject);

}

}

public GameObject GetEnemy(int index)

{

if (index < 0 || index >= allEnemies.Length)

{

Debug.LogError($" EnemyRegistry: индекс вне диапазона ({index})");

return null;

}

return allEnemies[index];

}

}

GameManager.cs:  
  
using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class GameManagerMulty : MonoBehaviour

{

[Header("Defenders")]

public GameObject[] defenders;

public int[] defendersPrice;

private List<GameObject> createdDefenders = new();

private int createdDefCounts = 0;

[Header("UI")]

public Text coinsText;

public Slider towerHealthSliderA;

public Slider towerHealthSliderB;

public Text towerHealthTextA;

public Text towerHealthTextB;

[Header("Tower Settings")]

public int towerHealthA = 100;

public int towerHealthB = 100;

public int towerDamage = 1;

[Header("UI Windows")]

public GameObject gameLostWindow;

public GameObject gameWinWindow;

[HideInInspector] public int currentDefender;

[HideInInspector] public bool canInstantiate = true;

[HideInInspector] public bool isDraging = false;

[HideInInspector] public bool dragOnViewSpace = true;

private bool purchasedCurrentItem = true;

private int localCoins = 5000;

private string mySide;

public string MySide => mySide;

void Start()

{

mySide = FindObjectOfType<NetworkConnector>().isHost ? "A" : "B";

currentDefender = 1;

purchasedCurrentItem = true;

localCoins = 5000;

coinsText.text = localCoins.ToString();

createdDefenders = new List<GameObject>();

canInstantiate = true;

UpdateTowerUI();

}

void Update()

{

CreateDefender();

EndDraging();

}

public void SetDefenderID(int id)

{

currentDefender = id;

}

public void SetDraging(bool dragState)

{

dragOnViewSpace = false;

if (localCoins >= defendersPrice[currentDefender - 1])

{

isDraging = dragState;

canInstantiate = true;

purchasedCurrentItem = false;

}

}

public void EndDraging()

{

if (Input.GetKeyUp(KeyCode.Mouse0))

{

dragOnViewSpace = true;

isDraging = false;

if (!purchasedCurrentItem)

{

localCoins -= defendersPrice[currentDefender - 1];

purchasedCurrentItem = true;

coinsText.text = localCoins.ToString();

}

}

}

void CreateDefender()

{

if (isDraging)

{

if (canInstantiate)

{

Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

if (Physics.Raycast(ray, out RaycastHit hit, 1000))

{

if (hit.transform.CompareTag("Ground"))

{

canInstantiate = false;

GameObject newDef = Instantiate(defenders[currentDefender - 1], hit.point, Quaternion.identity);

createdDefenders.Add(newDef);

createdDefCounts++;

}

}

}

else

{

Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

if (Physics.Raycast(ray, out RaycastHit hit, 1000))

{

if (hit.transform.CompareTag("Ground"))

{

GameObject current = createdDefenders[createdDefCounts - 1];

current.transform.position = hit.point;

Vector3 lookPos = FindClosestPoints("Center Point").position - current.transform.position;

lookPos.y = 0;

current.transform.rotation = Quaternion.Slerp(current.transform.rotation, Quaternion.LookRotation(lookPos), Time.deltaTime \* 1000);

}

}

}

}

}

public void ReduceCoins(int value)

{

localCoins -= value;

coinsText.text = localCoins.ToString();

}

public void AddCoins(int value)

{

localCoins += value;

coinsText.text = localCoins.ToString();

}

public void Game\_Lost()

{

Time.timeScale = 0;

gameLostWindow.SetActive(true);

}

public void You\_Win()

{

Time.timeScale = 0;

gameWinWindow.SetActive(true);

}

public void Reduce\_Tower\_Health(string side, int value)

{

if (side == "A")

{

towerHealthA -= value;

towerHealthSliderA.value = towerHealthA;

towerHealthTextA.text = towerHealthA.ToString();

if (towerHealthA <= 0) Game\_Lost();

}

else

{

towerHealthB -= value;

towerHealthSliderB.value = towerHealthB;

towerHealthTextB.text = towerHealthB.ToString();

if (towerHealthB <= 0) Game\_Lost();

}

}

void UpdateTowerUI()

{

towerHealthSliderA.value = towerHealthA;

towerHealthSliderB.value = towerHealthB;

towerHealthTextA.text = towerHealthA.ToString();

towerHealthTextB.text = towerHealthB.ToString();

}

Transform FindClosestPoints(string tag)

{

GameObject[] points = GameObject.FindGameObjectsWithTag(tag);

float closestDist = Mathf.Infinity;

GameObject closest = null;

Vector3 currentPos = createdDefenders[createdDefCounts - 1].transform.position;

foreach (GameObject go in points)

{

float dist = (go.transform.position - currentPos).sqrMagnitude;

if (dist < closestDist)

{

closestDist = dist;

closest = go;

}

}

return closest?.transform;

}

}

Health.cs:  
  
using System.Collections;

using UnityEngine;

public enum TargetType { Enemy, Tower, Defender }

public class Health : MonoBehaviour

{

public TargetType targetType;

public string side = "A"; // A или B

public int healthValue = 100;

public int destroyAwardedCoins = 100;

public GameObject damageParticle;

public float destroyDelay = 0;

public MeshRenderer healthColor;

private GameManagerMulty gManager;

private bool isDead;

void Start()

{

gManager = GameObject.FindObjectOfType<GameManagerMulty>();

if (healthColor)

healthColor.material.color = Color.green;

}

public void ApplyDamage(int damage)

{

healthValue -= damage;

if (healthColor)

{

if (healthValue > 0)

healthColor.transform.localScale = new Vector3(healthColor.transform.localScale.x, healthColor.transform.localScale.y, healthValue);

else

healthColor.transform.localScale = Vector3.zero;

}

if (targetType == TargetType.Enemy && healthValue <= 0)

{

if (GetComponent<CapsuleCollider>()) GetComponent<CapsuleCollider>().enabled = false;

if (GetComponent<Weapon>()) GetComponent<Weapon>().canShoot = false;

gManager.AddCoins(destroyAwardedCoins);

if (damageParticle)

Instantiate(damageParticle, transform.position, transform.rotation);

if (healthColor && healthColor.transform.parent) Destroy(healthColor.transform.parent.gameObject);

if (GetComponent<Weapon>()) GetComponent<Weapon>().shootingDelay = 999f;

if (!isDead)

{

isDead = true;

StartCoroutine(Destroy\_Delay());

}

}

if (targetType == TargetType.Defender && healthValue <= 0)

{

if (!isDead)

{

isDead = true;

StartCoroutine(Destroy\_Delay());

}

}

}

IEnumerator Destroy\_Delay()

{

yield return new WaitForSeconds(destroyDelay);

Destroy(gameObject);

}

}

Weapon.cs:  
  
using System.Collections;

using UnityEngine;

public enum ShootingMode

{

Gun, Sword

}

public class Weapon : MonoBehaviour

{

[Header("General Settings")]

public ShootingMode shootingMode = ShootingMode.Gun;

public GameObject projectile;

public Transform shootPoint;

public float force = 100f;

public float shootingDelay = 1f;

[Header("Sword Settings")]

public int SwordDamage = 1;

[Header("Additional Options")]

public GameObject secondProjectile;

public Transform secondShootPoint;

[Header("Sound Settings")]

public AudioClip fireClip;

AudioSource audioSource;

[HideInInspector] public bool canShoot = false;

GameManagerMulty gameManager;

IEnumerator Start()

{

audioSource = GetComponent<AudioSource>();

gameManager = FindObjectOfType<GameManagerMulty>();

while (true)

{

yield return new WaitForSeconds(shootingDelay);

if (canShoot)

{

if (shootingMode == ShootingMode.Gun)

{

GameObject bullet = Instantiate(projectile, shootPoint.position, shootPoint.rotation);

bullet.GetComponent<Rigidbody>().AddForce(shootPoint.forward \* force);

if (secondProjectile)

{

GameObject bullet2 = Instantiate(secondProjectile, secondShootPoint.position, secondShootPoint.rotation);

bullet2.GetComponent<Rigidbody>().AddForce(secondShootPoint.forward \* force);

}

if (audioSource && fireClip)

audioSource.PlayOneShot(fireClip);

if (TryGetComponent(out AnimationList animList) && animList.actor)

animList.actor.CrossFade(animList.fireClip);

}

if (shootingMode == ShootingMode.Sword)

{

if (TryGetComponent(out AnimationList animList) && animList.actor)

animList.actor.CrossFade(animList.fireClip);

gameManager.Reduce\_Tower\_Health(gameManager.MySide, SwordDamage);

}

}

}

}

}

Projectile.cs:  
  
using System.Collections;

using UnityEngine;

public enum ShootingMode

{

Gun, Sword

}

public class Weapon : MonoBehaviour

{

[Header("General Settings")]

public ShootingMode shootingMode = ShootingMode.Gun;

public GameObject projectile;

public Transform shootPoint;

public float force = 100f;

public float shootingDelay = 1f;

[Header("Sword Settings")]

public int SwordDamage = 1;

[Header("Additional Options")]

public GameObject secondProjectile;

public Transform secondShootPoint;

[Header("Sound Settings")]

public AudioClip fireClip;

AudioSource audioSource;

[HideInInspector] public bool canShoot = false;

GameManagerMulty gameManager;

IEnumerator Start()

{

audioSource = GetComponent<AudioSource>();

gameManager = FindObjectOfType<GameManagerMulty>();

while (true)

{

yield return new WaitForSeconds(shootingDelay);

if (canShoot)

{

if (shootingMode == ShootingMode.Gun)

{

GameObject bullet = Instantiate(projectile, shootPoint.position, shootPoint.rotation);

bullet.GetComponent<Rigidbody>().AddForce(shootPoint.forward \* force);

if (secondProjectile)

{

GameObject bullet2 = Instantiate(secondProjectile, secondShootPoint.position, secondShootPoint.rotation);

bullet2.GetComponent<Rigidbody>().AddForce(secondShootPoint.forward \* force);

}

if (audioSource && fireClip)

audioSource.PlayOneShot(fireClip);

if (TryGetComponent(out AnimationList animList) && animList.actor)

animList.actor.CrossFade(animList.fireClip);

}

if (shootingMode == ShootingMode.Sword)

{

if (TryGetComponent(out AnimationList animList) && animList.actor)

animList.actor.CrossFade(animList.fireClip);

gameManager.Reduce\_Tower\_Health(gameManager.MySide, SwordDamage);

}

}

}

}

}

CameraPan.cs:

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class CameraPanMulty : MonoBehaviour

{

[Space(5)]

[Header("Camera Pan System")]

public float mouseSensitivity = 1.0f;

Vector3 lastPosition;

public float limitLeft, limitRight;

GameManagerMulty gameManager;

private void Start()

{

gameManager = GameObject.FindObjectOfType<GameManagerMulty>();

}

void Update()

{

if (Input.GetKeyDown(KeyCode.Mouse0))

{

lastPosition = Input.mousePosition;

}

if (Input.GetKey(KeyCode.Mouse0))

{

if (gameManager.dragOnViewSpace)

{

Vector3 delta = Input.mousePosition - lastPosition;

Vector3 CameralastLocation = gameObject.transform.position;

transform.Translate(delta.x \* mouseSensitivity, 0, 0);

if (transform.position.x > limitRight)

transform.position = new Vector3(CameralastLocation.x, transform.position.y, transform.position.z);

if (transform.position.x < limitLeft)

transform.position = new Vector3(CameralastLocation.x, transform.position.y, transform.position.z);

lastPosition = Input.mousePosition;

}

}

}

}

NavMover.cs:  
  
using UnityEngine;

using System.Collections.Generic;

public class NavMover : MonoBehaviour

{

public string waypointName;

public float moveSpeed = 2.5f;

public float reachDistance = 0.1f;

private List<Transform> points = new List<Transform>();

private int currentPoint = 0;

public bool reachedToEnd = false;

void Start()

{

var pathObject = GameObject.Find(waypointName);

if (pathObject == null)

{

Debug.LogError($"[NavMover] ❌ Не найден путь: {waypointName}");

return;

}

var path = pathObject.GetComponent<WaypointSystem>();

if (path == null)

{

Debug.LogError($"[NavMover] ❌ У объекта {waypointName} нет компонента WaypointSystem");

return;

}

points = path.waypoints;

if (points.Count == 0)

{

Debug.LogError("[NavMover] ❌ Нет точек пути!");

return;

}

transform.position = points[0].position;

currentPoint = 1;

}

void Update()

{

if (reachedToEnd || points.Count == 0 || currentPoint >= points.Count)

return;

Vector3 target = points[currentPoint].position;

Vector3 direction = (target - transform.position).normalized;

float distance = Vector3.Distance(transform.position, target);

transform.position += direction \* moveSpeed \* Time.deltaTime;

if (distance < reachDistance)

{

currentPoint++;

if (currentPoint >= points.Count)

{

reachedToEnd = true;

OnReachEnd();

}

}

}

void OnReachEnd()

{

Debug.Log($"[NavMover] {gameObject.name} достиг конца пути");

}

}

Enemy\_Ai:

using System.Collections;

using UnityEngine;

public class Enemy\_AI : MonoBehaviour

{

public Transform gunHead;

public float dampingSpeed = 1f;

public string mySide = "A";

public float shootingDistance = 30f;

public bool shootOnlyTower = true;

Quaternion originalRotation;

bool isActive;

Transform target;

void Start()

{

originalRotation = gunHead.rotation;

StartCoroutine(AIUpdate());

}

IEnumerator AIUpdate()

{

while (true)

{

target = FindClosestEnemy();

if (target != null && Vector3.Distance(transform.position, target.position) <= shootingDistance)

{

if (shootOnlyTower)

{

if (GetComponent<NavMover>().reachedToEnd)

{

GetComponent<Weapon>().canShoot = true;

isActive = true;

}

}

else

{

GetComponent<Weapon>().canShoot = true;

isActive = true;

}

}

else

{

GetComponent<Weapon>().canShoot = false;

isActive = false;

}

yield return new WaitForSeconds(0.3f);

}

}

void Update()

{

if (isActive && target != null)

{

Vector3 lookPos = target.position - gunHead.position;

lookPos.y = 0;

Quaternion rotation = Quaternion.LookRotation(lookPos);

gunHead.rotation = Quaternion.Slerp(gunHead.rotation, rotation, Time.deltaTime \* dampingSpeed);

}

}

GameObject closest;

Transform FindClosestEnemy()

{

string tagToSearch = mySide == "A" ? "Tower\_B" : "Tower\_A";

GameObject[] gos = GameObject.FindGameObjectsWithTag(tagToSearch);

float distance = Mathf.Infinity;

Vector3 position = transform.position;

foreach (GameObject go in gos)

{

float curDistance = (go.transform.position - position).sqrMagnitude;

if (curDistance < distance)

{

closest = go;

distance = curDistance;

}

}

return closest ? closest.transform : null;

}

}

Defender\_Ai:

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Defender\_AI : MonoBehaviour

{

public Transform gunHead;

public float dampingSpeed = 10f;

public string targetTag = "Enemy";

public float shootingDistance = 30f;

[Header("Seek Animation")]

public bool playAnimationClip;

public float seekSpeed = 50f;

public float rotateAngle = 70f;

Vector3 originalRotation;

bool isActive;

Transform target;

IEnumerator Start()

{

originalRotation = gunHead.localRotation.eulerAngles;

while (true)

{

target = FindClosestEnemy();

if(target)

{

if (Vector3.Distance(transform.position, target.position) <= shootingDistance)

{

GetComponent<Weapon>().canShoot = true;

isActive = true;

}

else

{

GetComponent<Weapon>().canShoot = false;

isActive = false;

}

}

else

{

GetComponent<Weapon>().canShoot = false;

isActive = false;

}

yield return new WaitForSeconds(0.3f);

}

}

void Update()

{

if (isActive)

{

if (target)

{

Vector3 lookPos = target.position - gunHead.position;

lookPos.y = 0;

Quaternion rotation = Quaternion.LookRotation(lookPos);

gunHead.rotation = Quaternion.Slerp(gunHead.rotation, rotation, Time.deltaTime \* dampingSpeed);

}

}

else

{

if(playAnimationClip)

{

if (GetComponent<AnimationList>().actor)

{

GetComponent<AnimationList>().actor.CrossFade(GetComponent<AnimationList>().seekClip);

}

}

else

{

gunHead.localRotation = Quaternion.Euler(originalRotation.x, Mathf.PingPong(Time.time \* seekSpeed, rotateAngle \* 2) - rotateAngle, 1f);

}

}

}

GameObject closest;

Transform FindClosestEnemy()

{

GameObject[] gos;

gos = GameObject.FindGameObjectsWithTag(targetTag);

if (gos.Length == 0)

return null;

float distance = Mathf.Infinity;

Vector3 position = transform.position;

foreach (GameObject go in gos)

{

Vector3 diff = go.transform.position - position;

float curDistance = diff.sqrMagnitude;

if (curDistance < distance)

{

closest = go;

distance = curDistance;

}

}

return closest.transform;

}

}