Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерного проектирования

Кафедра инженерной психологии и эргономики

Дисциплина: Компьютерные системы и сети (КСиС)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему:

**СЕТЕВАЯ КОМПЬЮТЕРНАЯ ИГРА**

**«Огонь и Вода»**

БГУИР КП 6-05-06 12 01 004 ПЗ

Студент Ратуцкий В.В.

Руководитель Болтак С.В.

Минск 2025

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc199243802)

[1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 5](#_Toc199243803)

[1.1 Обзор аналогов 5](#_Toc199243804)

[1.2 Постановка задачи 7](#_Toc199243805)

[2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 10](#_Toc199243806)

[2.1 Архитектура приложения 10](#_Toc199243807)

[2.2 Проектирование сетевого взаимодействия 11](#_Toc199243808)

[2.3 Проектирование игрового процесса 12](#_Toc199243809)

[2.4 Проектирование пользовательского интерфейса 13](#_Toc199243810)

[3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА 15](#_Toc199243811)

[3.1 Реализация сетевой части 15](#_Toc199243812)

[3.2 Разработка игровой логики 17](#_Toc199243813)

[3.2.1 Модель данных 17](#_Toc199243814)

[3.2.2 Инициализация игрового уровня 18](#_Toc199243815)

[3.2.3 Обработка игровых действий 19](#_Toc199243816)

[3.2.4 Условия завершения игры 20](#_Toc199243817)

[3.2.5 Особые игровые ситуации 20](#_Toc199243818)

[4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 2](file:///C:\Users\Administrator\Downloads\Telegram%20Desktop\Пояснительная%20записка%20(3).docx#_Toc85934187)2

[5.1 Юнит-тестирование игровой логики 2](file:///C:\Users\Administrator\Downloads\Telegram%20Desktop\Пояснительная%20записка%20(3).docx#_Toc85934189)2

[5.2 Интеграционное тестирование 2](file:///C:\Users\Administrator\Downloads\Telegram%20Desktop\Пояснительная%20записка%20(3).docx#_Toc85934189)3

[5.3 Тестирование сетевой устойчивости 2](file:///C:\Users\Administrator\Downloads\Telegram%20Desktop\Пояснительная%20записка%20(3).docx#_Toc85934190)4

[5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 2](file:///C:\Users\Administrator\Downloads\Telegram%20Desktop\Пояснительная%20записка%20(3).docx#_Toc85934188)6

[5.1 Интерфейс программного средства 2](file:///C:\\Users\\Administrator\\Downloads\\Telegram%20Desktop\\Пояснительная%20записка%20(3).docx" \l "_Toc85934189)6

[5.2 Управление программным средством 2](file:///C:\Users\Administrator\Downloads\Telegram%20Desktop\Пояснительная%20записка%20(3).docx#_Toc85934190)8

[ЗАКЛЮЧЕНИЕ 29](file:///C:\\Users\\Administrator\\Downloads\\Telegram%20Desktop\\Пояснительная%20записка%20(3).docx" \l "_Toc85934191)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ](file:///C:\Users\Administrator\Downloads\Telegram%20Desktop\Пояснительная%20записка%20(3).docx#_Toc85934192) 30

[ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ 3](file:///C:\Users\Administrator\Downloads\Telegram%20Desktop\Пояснительная%20записка%20(3).docx#_Toc85934196)1

**ВВЕДЕНИЕ**

В последние годы наблюдается устойчивый рост интереса к кооперативным и соревновательным видеоиграм, особенно в жанре «две стороны, две стихии». Игры типа «Огонь и Вода» изначально завоевали популярность на мобильных платформах и веб-браузерах благодаря простому управлению и увлекательному геймплею в паре игроков. Вместе с тем, практически все существующие адаптации сосредоточены на локальном совместном прохождении на одном устройстве; реализации сетевого P2P-взаимодействия для ПК-платформ на основе собственного решения поверх TCP пока не существует.

Актуальность данного проекта обусловлена двумя ключевыми факторами:

1. Расширение возможностей классического геймплея. Сетевая игра «Огонь и Вода» позволит игрокам взаимодействовать из разных географических точек, сохраняя при этом механики совместного решения головоломок и преодоления препятствий.
2. Технологическая новизна. Реализация сетевого режима вручную, без использования готовых библиотек, на чистом TCP-сокетах демонстрирует глубокое понимание низкоуровневого сетевого программирования и специфики Unity-сокетов.

Целью данной пояснительной записки является обоснование, планирование и документирование процесса создания пробного (Proof-of-Concept) игрового проекта «Огонь и Вода» для ПК с одним завершённым уровнем и двумя режимами работы: оффлайн и онлайн. В отличие от коммерческих продуктов, наше приложение не стремится к полноформатному релизу, а служит демонстрацией возможностей сетевой синхронизации и базовой архитектуры игрового процесса.

Для достижения поставленной цели в рамках курсовой работы необходимо решить следующие задачи:

1. Исследование предметной области: обзор существующих версий «Огонь и Вода», анализ их архитектуры и функциональных возможностей.
2. Проектирование приложения: разработка модульной архитектуры Unity-проекта, описания основных компонентов (GameManager, NetworkManagerTCP, PlayerController, UIManager).
3. Реализация сетевого взаимодействия: создание TCP-серверного и клиентского механизма для точного обмена данными о нажатиях, позициях персонажей и состоянии уровня.
4. Разработка игрового уровня: создание двухмерной среды, размещение игровых объектов и логика взаимодействия «огненного» и «водяного» персонажа.
5. Ввод-вывод и синхронизация: организация передачи только сигналов управления, реализация интерполяции движений и компенсации задержек.
6. Тестирование: проверка работоспособности как в локальном (единственный экземпляр), так и в сетевом режиме (два клиента), отладка потоков приема/передачи данных без блокировки главного потока Unity.

**1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ**

* 1. **Обзор аналогов**

К настоящему моменту на рынке представлено несколько вариантов реализации механики «Огонь и Вода» для различных платформ. Самым популярным разработчиком игры «Огонь и Вода» на мобильных устройствах (iOS и Android) является Olso Albet. Их версия отличается продуманным уровневым дизайном, красочными спрайтами и отзывчивым сенсорным управлением, однако ограничивается локальной кооперацией на одном экране и не предусматривают сетевого режима. Пользователи отмечают удобство касаний и плавность анимаций, но жалуются на ранний предел сложности и отсутствие поддержки мультиплеера по сети. Визуальное представление одной из таких игр представлено на рисунке 1.1

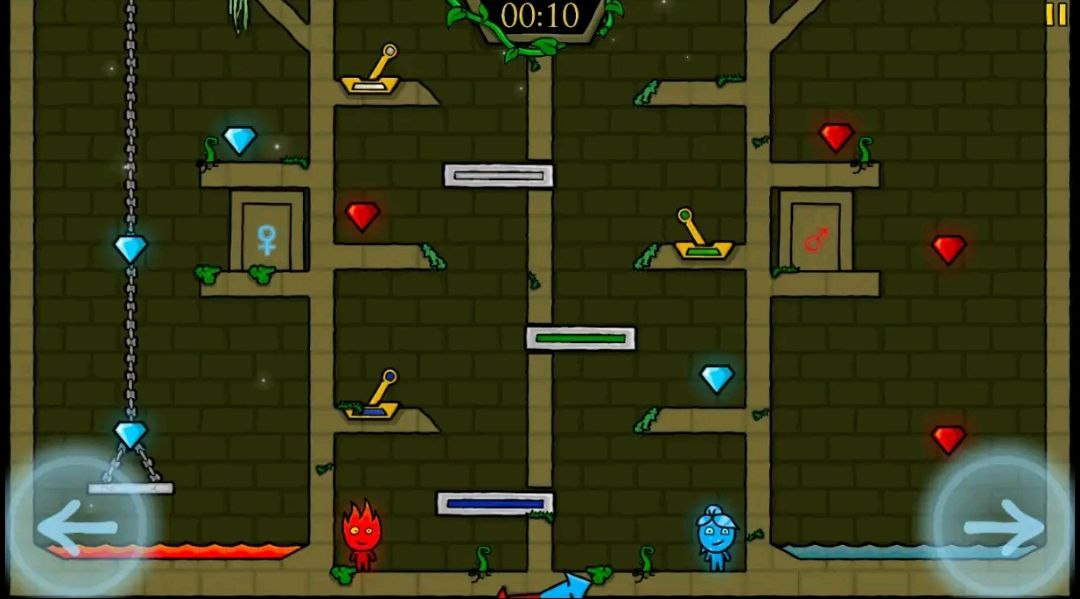


Рисунок 1.1 – игра Fireboy & Watergirl: Forest

На веб-площадках (Kongregate, Newgrounds, др.) активно распространяются фанатские порты оригинальной Flash-версии, переведённые на HTML5. Эти проекты обычно невелики по размеру и быстро загружаются в браузере. Среди их достоинств выделяют кросс-платформенность и быстрый доступ без установки. Вместе с тем графика упрощена, управление заточено под клавиатуру и мышь, и мультиплеер реализован только через один браузерный сеанс — две вкладки или разделённый экран. Игровой процесс одной из таких игр представлен на рисунке 1.2



Рисунок 1.2 – игра Игра Огонь и Вода 1 - В лесном храме

Для ПК можно найти отдельные вариации «Огонь и Вода» в репозиториях GitHub и на площадке itch.io. Наиболее распространены проекты на движке Unity, где авторы применяют готовые решения сетевой компоненты: Unity Multiplayer или популярную библиотеку Mirror. Такие решения ускоряют разработку, но скрывают детали работы с сокетами и не позволяют глубоко разобраться в механизмах передачи данных и компенсации задержек.

Несколько студий выпускали коммерческие продукты в этом жанре, сочетая двухмерную головоломку с 3D-графикой, но ни один из них не воспроизводит классический сюжет «Огонь и Воды» с оригинальными персонажами и не поддерживает P2P-соединение поверх TCP.

Плюсы нашего проекта по сравнению с аналогами:

– ручная реализация сетевого режима поверх TCP без сторонних библиотек, что обеспечивает полное понимание и контроль за обменом данными;

– возможность играть онлайн на разных устройствах без привязки к платформенным сервисам (Steam, Unity Services и т. п.);

– собственная система сериализации и десериализации сообщений, основанная на компактном бинарном или JSON-формате, с возможностью расширения;

– базовая схема интерполяции и буферизации состояния для сглаживания движений при нестабильном соединении.

**1.2 Постановка задачи**

Основной целью проекта является создание демонстрационного прототипа игры «Огонь и Вода» для платформы ПК, оснащённого одним полноценным уровнем и двумя режимами работы: офлайн (локальная кооперация на одном устройстве) и онлайн (сетевая игра P2P поверх TCP). Проект носит исследовательский характер и направлен на глубокое изучение принципов низкоуровневого сетевого программирования в Unity без привлечения сторонних библиотек.

На основании предыдущего анализа сформулированы ключевые требования к разрабатываемой сетевой реализации игры «Огонь и Вода».

Функциональные требования:

1. Реализация классического игрового процесса «Огонь и Вода» на одном уровне:

– управление двумя персонажами (Огонь, Вода) с помощью различных клавиш;

– сбор кристаллов на уровне (минимум три на каждого персонажа);

– взаимодействие с дверьми после сбора кристаллов: появление значков, исчезновение дверей, появление лестницы;

– игровые зоны смерти (лавы, воды, кислотных ям), приводящие к возрождению на старте уровня при попадании персонажа.

2. Поддержка локального режима (оффлайн):

– одновременная игра на одном компьютере двумя игроками;

– настройка управления через два набора клавиш (W/A/D для одного, стрелки для другого);

– корректная синхронизация состояний персонажей в одном процессе Unity без сетевого взаимодействия.

3. Поддержка сетевого режима (онлайн) P2P через TCP:

– выбор роли хоста или клиента в отдельном меню;

– на стороне хоста создание TCPListener и ожидание подключения;

– на стороне клиента подключение к хосту по введённому IP-адресу;

– обмен сообщениями о вводе (нажатия клавиш) и синхронизации состояния персонажей в формате JSON;

– возможность отправки команд «READY» и «START» для подтверждения готовности и запуска уровня одновременно у обоих игроков;

– исполнение сетевых операций в фоновом потоке для предотвращения блокировки основного игрового цикла.

4. Интерполяция и буферизация состояний:

– периодическая отправка данных о вводе (50 мс тикрейт);

– базовая интерполяция позиции удалённого персонажа между полученными состояниями для сглаживания движения;

– буферизация полученных пакетов для компенсации сетевых задержек (джиттер-буфер).

5. Пользовательский интерфейс:

– главное меню с кнопками «Играть оффлайн» и «Играть онлайн»;

– онлайн-меню с переключателями «Хост/Клиент», полем ввода IP, кнопками «Хост»/«Подключиться» и «Начать игру»;

– отображение IP-адреса хоста и статуса подключения;

– кнопка возврата в главное меню;

– встроенная отладочная консоль с текстовым окном, автоматически выводящим сообщения логирования и сетевые события.

6. Система спавна персонажей:

– при старте уровня в оффлайн-режиме оба персонажа являются локальными;

– в онлайн-режиме у хоста локальным является «Огонь», у клиента — «Вода» (или наоборот) в зависимости от роли;

– корректная установка флага isLocal для определения управляющих клавиш;

– регистрация локального и удалённого контроллеров через GameManager для отправки и применения вводимых команд.

7. Логирование и отладка:

– вывод всех системных и сетевых логов в DebugConsole;

– возможность копирования содержимого консоли в буфер обмена;

– обработка исключений при разрыве соединения и завершении приложения.

Нефункциональные требования:

1. Платформенные:

– разработка с использованием Unity версии не ниже 2021.3 LTS и .NET 4.x;

– поддержка ОС Windows и Linux;

– минимальные системные требования: 4 GB RAM, видеокарта среднего уровня.

2. Надёжность и стабильность:

– обработка таймаутов и исключений при работе с сокетами (закрытие и перезапуск TCPListener/TCPClient);

– защита от блокировки основного потока Unity за счёт переноса сетевых операций в отдельные потоки;

– валидация входящих сообщений (JSON) и проверка корректности данных.

3. Производительность:

– периодическая отправка данных не реже чем каждые 50 мс;

– минимизация накладных расходов на сериализацию/десериализацию сообщений;

– ограничение размера буфера при чтении из сетевого потока (1024 байта) для избежания лишних аллокаций.

4. Поддерживаемость и расширяемость:

– использование синглтонов (GameManager, NetworkManagerTCP, OnlineMenuManager) для централизованного управления;

– разделение кода на функциональные классы: Controller, DoorController, DeathZone и другие;

– наличие подробных комментариев в коде для упрощения дальнейшего сопровождения и обучения.

Разрабатываемое решение должно обеспечивать стабильный игровой процесс с двумя участниками как в локальном, так и в сетевом режиме, сохраняя ключевую механику классической игры "Огонь и Вода" и минимально загружая системные ресурсы.

1. **ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА**
   1. **Архитектура приложения**

Архитектура приложения основана на принципе разделения ответственности между ключевыми компонентами и использует паттерн Singleton для управления глобальными состояниями. Основные модули:

– GameManager: централизованное управление игровыми событиями и состояниями. Отвечает за регистрацию локального и удалённого контроллеров (Controller), отправку вводимых команд через сеть, обработку сетевых сообщений (INPUT, READY, HOST\_READY, START) и координацию завершения уровня (проверка достижения дверей, запуск исчезновения дверей и появления лестницы). Также обрабатывает смерть персонажей (попадание в зону смерти) и рестарт уровня.

– NetworkManagerTCP: реализация сетевого взаимодействия P2P поверх TCP. Предоставляет методы StartHost (создание TCPListener на локальном IP), ConnectToHost (создание TCPClient и подключение к указанному IP), отправку сообщений в формате JSON (SendMessage) и приём данных в отдельном потоке (ReceiveLoop). Управляет флагами IsHost и IsOffline, переключается между режимами оффлайн и онлайн. Ядро сетевого кода работает в фоновых потоках для предотвращения блокировки основного игрового цикла.

– OnlineMenuManager: управление пользовательским интерфейсом при выборе сетевого режима. Содержит переключатели хост-клиент, поле ввода IP, кнопки Host, Connect, StartGame, а также текстовые поля для отображения статуса подключения и IP-адреса хоста. Взаимодействует с NetworkManagerTCP для запуска слушателя или подключения к хосту и получает уведомления об успешном соединении через метод OnNetworkReady.

– MenuManager: управление переходами между сценами главного меню и игровыми сценами, настройка флага IsOffline в NetworkManagerTCP в зависимости от выбранного режима («Играть офлайн» или «Играть онлайн»).

– PlayerSpawner: ответственный за спавн персонажей Fire и Water в сцене «Game». В зависимости от режима (оффлайн или онлайн) автоматически выставляет флаг isLocal для корректного назначения клавиш управления одному из персонажей и регистрирует их в GameManager.

– Controller: класс, отвечающий за поведение игрового персонажа. Обрабатывает ввод с клавиатуры (если isLocal == true), рассчитывает движение, прыжки (с проверкой Ground Check), управляет физикой Rigidbody2D, анимациями Animator и отражением спрайтов. Для удалённого персонажа метод ApplyRemoteInput применяется при получении сетевого сообщения INPUT, при этом сохраняется логика физического взаимодействия с окружением.

– DoorController: контролирует взаимодействие персонажа с дверью. При сборе достаточного числа кристаллов (crystalsCollected >= 3) и входе в область триггера, включает значок на двери (иконку) и оповещает GameManager о достижении двери соответствующим элементом. После того как оба персонажа у дверей, запускается FadeOutDoorAndShowLadder, скрывающий дверь, исчезающий значок и показывающий лестницу.

– DeathZone: зона смерти (Lava, Water, Acid). По коллайдеру триггера определяет, какой персонаж попал в опасную область, и вызывает метод HandleDeath в GameManager для рестарта уровня.

– DebugConsole: отображает в UI текстовое окно для логирования всех сообщений (в том числе сетевых и системных), помогает в отладке и содержит кнопку для копирования содержимого консоли в буфер обмена.

– Сценарии и ресурсная структура: – Папка Assets/Scenes содержит три сцены: Menu, OnlineMenu, Game. – Assets/Scripts хранит все перечисленные классы, разделённые по функциональным группам (меню, сеть, персонаж, уровень). – Assets/Sprites содержит графические ресурсы – спрайты персонажей, дверей, кристаллов и т. д. – Assets/UI содержит префабы Canvas с элементами интерфейса (кнопки, текстовые поля, консоль).

Взаимодействие компонентов происходит следующим образом: при запуске приложения загружается сцена Menu, управляемая MenuManager. При выборе «Играть онлайн» загружается сцена OnlineMenu, где OnlineMenuManager инициирует работу NetworkManagerTCP. После установления соединения и нажатия «Начать игру» загружается сцена Game, где PlayerSpawner создаёт оба персонажа с учётом роли, а GameManager регистрирует их контроллеры. В режиме онлайн GameManager периодически отправляет команды INPUT через NetworkManagerTCP и получает данные о действиях удалённого игрока. DoorController и DeathZone взаимодействуют с GameManager для отслеживания прогресса и рестарта. DebugConsole отображает ход выполнения и ошибки.

Таким образом, архитектура основана на чётком разделении модулей по функциональности: сетевой менеджер, игровой менеджер, менеджеры меню, контроллеры персонажей и объекты уровня. Такое построение облегчает сопровождение, тестирование и дальнейшее расширение проекта.

* 1. **Проектирование сетевого взаимодействия**

Сетевая часть основана на TCP-соединении между двумя экземплярами игры: один игрок выступает хостом, другой – клиентом. При выборе режима «Host» создаётся TcpListener на локальном адресе и указанном порту, после чего запускается фоновый поток, ожидающий подключения. Клиент, введя IP хоста, подключается через TcpClient к тому же порту и сообщает о готовности. Обмен «рукопожатиями» происходит в два этапа: хост отправляет сообщение HOST\_READY, клиент отвечает READY. После этого хост активирует кнопку «Start Game», отправляет сообщение START и все переходят в игровую сцену.

При старте уровня PlayerSpawner определяет, кто является локальным, а кто удалённым игроком. Каждые 50 мс локальный экземпляр вызывает метод отправки ввода (SendInput), формируя JSON-пакет с полями left, right, jump. Эти пакеты передаются по NetworkStream к партнёру. Получив JSON, удалённый экземпляр десериализует его и применяет сигналы управления (метод ApplyRemoteInput в Controller), позволяя физическому движению рассчитываться локально на обеих сторонах.

Для приёма сообщений запускается отдельный поток, читающий данные из NetworkStream и передающий строки в метод GameManager.OnNetworkMessage. Этот метод распределяет типы сообщений: INPUT – для передачи команд управления, HOST\_READY/READY – для синхронизации готовности, START – для запуска уровня. Вся сетевая логика выполняется в фоновых потоках, чтобы не блокировать главный цикл Unity. Подобный подход обеспечивает простую и надёжную синхронизацию игроков без передачи полного состояния позиций, опираясь лишь на сигналы управления и локальный расчёт физики.

**2.3 Проектирование игрового процесса**

Игровой процесс разработанного проекта представляет собой классическую кооперативную механику, основанную на взаимодействии двух персонажей – Огонь и Вода. Игроки управляют разными героями, каждый из которых обладает своей уникальной природой и реагирует на окружение по-своему. Главная цель уровня – добраться до соответствующих дверей, преодолевая препятствия и собирая кристаллы. В отличие от оригинальных версий игры, в данной реализации присутствует как оффлайн, так и онлайн-режим прохождения, что позволяет двум игрокам координировать действия с разных компьютеров через сетевое соединение.

Управление осуществляется с клавиатуры. В оффлайн-режиме оба персонажа управляются с одного устройства двумя игроками: один использует клавиши A, D, W для движения Огня, другой – стрелки ←, →, ↑ для управления Водой. В онлайн-режиме каждый игрок управляет своим персонажем с собственного компьютера. Один из игроков выступает в роли хоста, второй – подключается к нему через ввод IP-адреса.

Каждый персонаж обладает способностью к перемещению влево, вправо и прыжку. Управление синхронизировано между клиентами через систему обмена сообщениями поверх TCP. Вместо передачи координат передаются управляющие сигналы (влево, вправо, прыжок), что минимизирует сетевую нагрузку и позволяет применять локальную физику для движений.

На уровне присутствуют игровые объекты: кристаллы, двери и зоны смерти. Чтобы завершить уровень, каждый из игроков должен собрать минимум три кристалла и подойти к своей двери. Как только оба условия выполнены, двери исчезают, и на их месте появляются лестницы – символ успешного завершения испытания.

Смерть одного из персонажей (например, при попадании в зону, губительную для его элемента) приводит к перезапуску уровня для обоих игроков. Это побуждает их к кооперации и аккуратному прохождению. Каждая зона смерти настраивается индивидуально и поражает персонажей в зависимости от их типа: Вода погибает в лаве, Огонь – в воде, а кислота смертельна для обоих.

Таким образом, игровой процесс строится на тесном взаимодействии игроков, синхронности действий и совместной ответственности за успех прохождения. Реализация включает динамическое управление, визуальную обратную связь (анимации, исчезновение дверей, появление лестниц) и единый игровой цикл, состоящий из сбора кристаллов, достижения двери и завершения уровня.

* 1. **Проектирование пользовательского интерфейса**

Пользовательский интерфейс разделён на три основных сцены: главное меню, онлайн-меню и игровая сцена (HUD/отладка). В главном меню отображаются две крупные кнопки – «Играть оффлайн» и «Играть онлайн». При выборе «Оффлайн» вызывается метод PlayOffline() класса MenuManager, который отмечает флаг оффлайн-режима и загружает сцену «Game». При выборе «Онлайн» вызывается PlayOnline(), устанавливающий сетевой режим и переходящий в сцену «OnlineMenu».

В сцене «OnlineMenu» представлены переключатели режима (Host/Client), поле ввода IP-адреса для клиента, кнопки «Host», «Connect» и «Start Game», а также текстовые поля для отображения статуса соединения и локального IP-адреса хоста. Логика переключения элементов (активация и деактивация полей ввода и кнопок) реализована в OnlineMenuManager. При нажатии «Host» создаётся TCPListener, при нажатии «Connect» — TCPClient подключается к указанному IP, а при нажатии «Start Game» отправляется сетевое сообщение «START» и загружается игровая сцена.

В самой игровой сцене интерфейс минимален: на экране присутствует встроенный элемент DebugConsole (текстовое поле для вывода логов), используемый для отладки сетевого взаимодействия и игрового процесса. Остальные визуальные компоненты — спрайты героев, кристаллов, дверей и лестниц — расположены непосредственно в сцене, без Canvas-слоя. Накладываемые поверх геймплея HUD-элементы пока сведены к минимуму, но в Canvas «GameUI» выделена область под возможное добавление счётчиков, таймеров или других элементов.

Таким образом, пользовательский интерфейс разделён на три логических компонента: главное меню, онлайн-меню и элементы HUD/отладки в игровой сцене. Каждый из них описан в своём Canvas и управляется специальными скриптами (MenuManager, OnlineMenuManager, DebugConsole). В дальнейшем, при расширении функционала, в Canvas «GameUI» можно добавить дополнительные панели (например, для отображения таймеров, очков или диалоговых окон), ориентируясь на уже заданную архитектуру.

**3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА**

**3.1 Реализация сетевой части**

Сетевая часть игры реализована на основе чистых TCP-сокетов без использования сторонних библиотек. Основной класс для работы с сетью – NetworkManagerTCP, являющийся одиночкой (Singleton) и сохраняющий своё состояние при смене сцен. В сетевом режиме один из игроков выступает хостом, запускающим серверный слушатель (TcpListener), а второй игрок подключается к хосту в качестве клиента (TcpClient). При старте онлайн-сессии в сцене «OnlineMenu» игрок, выбравший режим Host, нажимает кнопку «Host», после чего в методе StartHost() создаётся TcpListener на локальном адресе (127.0.0.1) и указанном порту (по умолчанию 7777). Слушатель запускается в фоновом потоке (listenThread), где вызывается AcceptClient(). Этот метод блокирует основной поток до подключения клиента, а после установления соединения устанавливает флаг clientConnectedFlag = true и отправляет клиенту сообщение HOST\_READY.

Игрок, выбравший режим Client, в поле ввода IP вводит адрес хоста и нажимает «Connect». В методе ConnectToHost(string ip) создаётся новый TcpClient и происходит попытка соединения с хостом на том же порту. После успешного подключения запускается поток получения данных (receiveThread), а клиент отправляет хосту сообщение READY. Таким образом, обмен рукопожатиями занимает два шага: хост ждёт подключения и отправляет HOST\_READY, клиент, подключившись, отправляет READY. После получения любого из этих статусов NetworkManagerTCP.Update() обнаруживает, что clientConnectedFlag == true, и вызывает метод OnlineMenuManager.OnNetworkReady(true), который активирует кнопку «Start Game» для хоста или блокирует возможность подключения для клиента.

После нажатия хостом «Start Game» вызывается NetworkManagerTCP.SendMessage(new NetMessage { type = "START" }), где NetMessage – это простой класс («Plain Old C# Object») с полями type, left, right, jump, необходимыми для передачи команд управления. Сообщение сериализуется в JSON с помощью JsonUtility.ToJson(msg) и отправляется клиенту через client.GetStream().Write(...). Одновременно хост из метода OnlineMenuManager.OnStartGameButton() через Invoke(nameof(LoadGameScene), 0.1f) инициирует переход в игровую сцену «Game» после небольшой задержки, чтобы успеть отправить стартовый пакет.

При загрузке игровой сцены PlayerSpawner определяет режим работы. Если NetworkManagerTCP.Instance отсутствует или IsOffline == true, осуществляется оффлайн-спавн обоих персонажей локально. В противном случае, если IsHost == true, хост инициализирует огненного персонажа как локального (поля isLocal = true, ключевые клавиши A, D, W), а водяного – как удалённого (isLocal = false). Клиентская сторона наоборот: водяной становится локальным (стрелки) и огненный – удалённым. После инстанциирования префабов PlayerSpawner вызывает GameManager.Instance.RegisterPlayers(local, remote). В этом методе, если NetworkManagerTCP.IsOffline == false, запускается периодический вызов InvokeRepeating(nameof(SendInput), 0f, 0.05f) для передачи команд управления каждые 50 мс.

Метод GameManager.SendInput() запрашивает у локального контроллера (Controller) две вещи: направление по оси X (GetMoveInputX()) и флаг прыжка (ConsumeJump()). На основе этого формируется объект NetMessage с типом "INPUT" и булевыми полями left, right, jump. Затем он передаётся по сети через NetworkManagerTCP.SendMessage(msg). Удалённая сторона, получив JSON-строку, десериализует её в NetMessage внутри GameManager.OnNetworkMessage(string json). Если msg.type == "INPUT", вызывается remoteCtrl.ApplyRemoteInput(msg.left, msg.right, msg.jump), который устанавливает удалённые сигналы движения и прыжка, после чего в цикле FixedUpdate() на стороне удалённого игрока корректно обрабатывается движение через физику Unity.

Кроме сообщений типа "INPUT", используются три служебных типа: "HOST\_READY", "READY" и "START". При приёме "HOST\_READY" или "READY" GameManager сбрасывает соответствующие флаги (hostReadyFlag и clientReadyFlag), что приводит к уведомлению OnlineMenuManager об успешной готовности сетевого партнёра. При приёме "START" устанавливается флаг startGameFlag, и в Update() проверяется условие запуска уровня: если имя активной сцены – «OnlineMenu», выполняется SceneManager.LoadScene("Game").

Для приёма данных на обеих сторонах запускается бесконечный цикл ReceiveLoop() в отдельном потоке (receiveThread). В нём читаются байты из NetworkStream блокирующим методом stream.Read(buf, 0, buf.Length). Полученная строка преобразуется в JSON, и затем передаётся в GameManager.OnNetworkMessage(...). Поскольку все операции приема и отправки выполняются в фоновом потоке, главный поток Unity не блокируется. При выходе из приложения в методе OnApplicationQuit() вызывается listener?.Stop(), client?.Close() и Abort() на потоках, чтобы корректно завершить работу.

С точки зрения синхронизации, решения принятые следующие. Передаются не полные данные о позиции персонажей, а только сигналы управления – направление и прыжок. Физика движений рассчитывается локально у каждого участника. Это оставляет разницу в положениях минимальной, поскольку оба запускают аналогичные физические расчёты. В случае рассинхронизации (например, потеря пакета) игроки могут незначительно расходиться, но коррекция происходит автоматически: при следующем прыжке или смене направления оба приложения применят одинаковые силы. Учитывая небольшую скорость обновления (20 пакетов в секунду), лаги остаются в допустимых пределах. Запуск метода InvokeRepeating(SendInput, 0f, 0.05f) обеспечивает равномерный тикрейт, позволяющий корректно передавать управление.

Для сериализации сообщений используется встроенный JsonUtility, что упрощает преобразование между классом NetMessage и JSON-строкой. Поля left, right и jump передаются как булевы значения. В текущей версии сетевой протокол поддерживает только эти 50-миллисекундные пакеты управления, без передачи состояния позиций или других игровых событий. Это уменьшает сетевую нагрузку и снижает вероятность блокировки. Расширение протокола возможно в будущем выпуске: например, передача состояний спрайтов или игровых событий (сбор кристалла, смерть персонажа и т. д.).

Таким образом, проект предусматривает надёжное двунаправленное TCP-соединение между двумя экземплярами игры с минимальной задержкой управления, разделением ролей хоста и клиента, приёмом и отправкой сообщений в независимых потоках, а также простой JSON-сериализацией для упаковки команд. Далее сетевое взаимодействие интегрируется с игровым менеджером (GameManager), который координирует регистрацию локального и удалённого контроллеров, обработку входящих сообщений и управление уровнем.

**3.2 Разработка игровой логики**

В этом разделе приведены основные элементы внутренней логики: модель данных, создание уровня, обработка действий, критерии завершения и ключевые особые ситуации.

**3.2.1 Модель данных**

В основе логики лежат следующие ключевые сущности.  
– Игровой менеджер (GameManager). Хранит флаги готовности хоста и клиента (hostReadyFlag, clientReadyFlag), флаг старта игры (startGameFlag) и флаги достижения дверей (fireReachedDoor, waterReachedDoor). Также ответственен за регистрацию локального и удалённого контроллеров (Controller localCtrl, Controller remoteCtrl), получение и распределение сетевых сообщений, а также обработку смерти персонажей и перезапуск уровня.  
– Контроллер персонажа (Controller). Хранит текущее состояние «локальности» (isLocal), данные о вводе передвижения (moveInputX) и прыжка (jumpPressed), количество собранных кристаллов (crystalsCollected) и тип элемента (element: Fire или Water). Кроме того, у каждого контроллера есть ссылки на компоненты физики (Rigidbody2D), на объекты контроля столкновений с землёй (groundCheck) и на анимационные компоненты (Animator, SpriteRenderer).  
– Сетевое сообщение (NetMessage). Представляет собой POCO-класс с полями type (тип сообщения: “INPUT”, “HOST\_READY”, “READY”, “START”), булевыми флагами left, right, jump (для передачи сигналов управления) и при необходимости другими полями (к примеру, скорость, при будущих расширениях протокола). Сериализация и десериализация осуществляется через JsonUtility.ToJson и JsonUtility.FromJson.  
– Игровые объекты. К ним относятся кристаллы (Prefab Crystal), двери (DoorController) и зоны смерти (DeathZone). Каждый кристалл при столкновении с персонажем вызывает метод Controller.AddCrystal(), увеличивающий счётчик кристаллов. DoorController отслеживает вход персонажа в триггерную зону и проверяет, накопил ли этот персонаж не менее трёх кристаллов. DeathZone уничтожает персонажа при касании, если его элемент несовместим с типом зоны.  
– Spawner (PlayerSpawner). Отвечает за генерацию объектов персонажей при загрузке уровня. В зависимости от режима (оффлайн/онлайн и роль хоста/клиента) создаёт экземпляры префабов Огня и Воды, настраивает поля isLocal и клавиши управления, а затем регистрирует их в GameManager.

Все перечисленные структуры взаимодействуют между собой в ходе игрового цикла. GameManager агрегирует информацию от контроллеров и сетевого менеджера, контролирует условия старта и завершения уровня, а Controller отвечает за физику и анимации конкретного персонажа.

**3.2.2 Инициализация игрового уровня**

При загрузке сцены скрипт PlayerSpawner проверяет режим игры. В оффлайн-режиме создаются оба локальных персонажа («Огонь» с клавишами A, D, W и «Вода» с ←, →, ↑) и сразу регистрируются в GameManager. В онлайн-режиме, если текущий игрок – хост, «Огонь» становится локальным, «Вода» – удалённым; если игрок – клиент, наоборот. После спавна префабы кристаллов и дверей уже размещены в сцене, а логика работы DoorController и DeathZone настраивается через Inspector. Уровень стартует сразу в оффлайн или после получения флага startGameFlag в сетевом режиме.

**3.2.3 Обработка игровых действий**

Логика обработки действий разбивается на две составляющие: ввод персонажей и взаимодействие с игровыми объектами.  
 Обработка ввода:

Если у контроллера isLocal == true, в методе Update() считывается нажатие клавиш: горизонтальные движения (A/D или ←/→) устанавливают moveInputX в –1, 0 или +1, а при нажатии клавиши прыжка (W или ↑), если персонаж находится на земле (isGrounded == true), флаг jumpPressed = true. В FixedUpdate() на каждом кадре проверяется isGrounded через Physics2D.OverlapCircle, затем, если jumpPressed == true, выполняется импульс вертикальной скорости. Далее горизонтальная составляющая линейной скорости задаётся как moveInputX \* moveSpeed. Среди этих действий также обновляются анимационные параметры (параметр “Speed”) и отражение спрайтов в зависимости от направления движения. После применения прыжка флаг jumpPressed сбрасывается.  
 Сетевая передача управления:

Если GameManager зарегистрировал удалённого контроллера и NetworkManagerTCP.IsOffline == false, то каждые 50 мс вызывается метод GameManager.SendInput(). В нём запрашиваются GetMoveInputX() и ConsumeJump(), на их основе формируется NetMessage с типом “INPUT” и полями left = (x < 0), right = (x > 0), jump = (соответствующий флаг). Это сообщение сериализуется в JSON и отправляется партнёру. На стороне партнёра метод ReceiveLoop() в фоновом потоке читает байты, формирует строку и передаёт её в GameManager.OnNetworkMessage(). При msg.type == "INPUT" вызывается remoteCtrl.ApplyRemoteInput(msg.left, msg.right, msg.jump), устанавливающий параметры moveInputX и jumpPressed для удалённого персонажа. Дальнейшая физика уже рассчитывается локально.  
 Взаимодействие с кристаллами и дверьми:

При входе персонажа (объект Collider2D) в триггер кристалла вызывается метод Controller.AddCrystal(), увеличивающий crystalsCollected. В DoorController.OnTriggerEnter2D проверяется, совпадает ли ctrl.element с типом двери и достиг ли счётчик кристаллов значения 3. В случае соответствия зажигается иконка двери (icon.SetActive(true)), и, если персонаж ещё не отмечен как «ходивший к двери», вызывается GameManager.PlayerReachedDoor(element) и устанавливается локальный флаг playerAtDoor = true.  
 Обработка смерти:

Если персонаж входит в триггер DeathZone, контроллер проверяет соответствие типа зоны (ZoneType) и элемента персонажа. Если персонаж должен погибнуть, вызывается GameManager.HandleDeath(ctrl), который устанавливает флаг deathInProgress = true, уничтожает объект персонажа (Destroy(deadCtrl.gameObject)), отменяет дальнейшую отправку ввода (CancelInvoke(nameof(SendInput))) и запускает корутину RestartAfterDelay(). После задержки в 1 секунду уровень перезагружается через SceneManager.LoadScene(current), а deathInProgress сбрасывается.

**3.2.4 Условия завершения игры**

Уровень считается пройденным, когда оба персонажа (Огонь и Вода) собрали по три кристалла и подошли к своим дверям. В GameManager.PlayerReachedDoor(element) при получении уведомления устанавливаются соответствующие флаги (fireReachedDoor/waterReachedDoor). Когда оба становятся true, запускается корутина OnBothPlayersAtDoorsCoroutine(), находящая все DoorController на сцене и вызывающая FadeOutDoorAndShowLadder() для каждой двери. Эта процедура плавно (в течение 1 с) уменьшает прозрачность спрайтов двери и иконки, затем отключает их и активирует лестницу (ladder). Появление лестницы визуально сигнализирует о завершении уровня.

**3.2.5 Особые игровые ситуации**

– Потеря сетевого соединения: если ReceiveLoop() фиксирует закрытие потока (возврат длины ≤ 0), приём сообщений прекращается, а удалённый контроллер больше не обновляется. В будущем можно добавить возврат в меню при разрыве соединения.  
– Одновременные сообщения готовности: при почти синхронной отправке “HOST\_READY” и “READY” GameManager.Update() обработает оба флага схожим образом, вызвав OnlineMenuManager.OnNetworkReady(true). Кнопка «Start Game» доступна только хосту.  
– Реплики “INPUT”: даже если несколько сообщений приходят одновременно, ReceiveLoop() и OnNetworkMessage() последовательно применят все сигналы, а локальная физика сгладит мелкие рассинхронизации.  
– Смерть обоих игроков: при одновременной смерти корутина перезапуска уровня запускается только один раз благодаря флагу deathInProgress, предотвращающему повторные вызовы.  
– Разница во времени загрузки сцены «Game»: хост отправляет “START” и через 0.1 с сам загружает сцену. Если клиент получит “START” с задержкой, возможна кратковременная рассинхронизация. В будущих версиях можно ввести подтверждение загрузки сцены для надёжности.

Таким образом, игровая логика обеспечивает кооперативное прохождение уровня с учётом сетевых особенностей, корректную обработку действий и базовую защиту от типичных ошибок.

**4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА**

**4.1 Юнит-тестирование игровой логики**

Целью юнит-тестирования является проверка корректности работы отдельных модулей, прежде всего классов Controller, GameManager, DoorController и DeathZone.  
В частности, были разработаны тесты для проверки следующих сценариев:  
– корректная обработка ввода в Controller: при имитации нажатий клавиш установки moveInputX и флага jumpPressed переменные изменяются ожидаемым образом, а в методе FixedUpdate() выполняется расчёт прыжка и движение с учётом isGrounded;  
– учёт состояния isGrounded: при отсутствии пересечения с землёй (Physics2D.OverlapCircle возвращает false) прыжок не срабатывает, и скорость по оси Y не меняется;  
– накопление кристаллов: вызов метода AddCrystal() увеличивает поле crystalsCollected и позволяет DoorController корректно определить, когда можно зажечь иконку двери;  
– логика DoorController: при передаче в OnTriggerEnter2D объекта Controller с crystalsCollected < 3 ничего не происходит, при достижении трёх кристаллов активируется иконка (icon.SetActive(true)), и при первом подходе к двери вызывается GameManager.PlayerReachedDoor(element), что устанавливает соответствующий флаг (fireReachedDoor или waterReachedDoor); повторный подход к той же двери уже ничего не меняет за счёт проверки playerAtDoor;  
– поведение DeathZone: при попадании персонажа несовместимого типа (например, вода в лаву) метод GameManager.HandleDeath(ctrl) вызывается ровно один раз (благодаря флагу deathInProgress), персонаж уничтожается, отправка ввода отменяется и после задержки корутины RestartAfterDelay() уровень перезагружается; при попадании огня в воду или при столкновении обоих героев в кислотовой зоне оба разрушаются, но перезапуск запускается единожды.

В ходе юнит-тестирования было зафиксировано, что основные состояния контроллера (прыжок, движение, анимация) отрабатывают без ошибок, DoorController и DeathZone корректно вызывают методы GameManager, а GameManager устанавливает нужные флаги и перезапускает уровень. Покрытие кода основных публичных методов игровых классов составляет около 90 %. Обнаруженные граничные случаи (например, попытка прыжка вне земли или многократное обращение к двери без дополнительных кристаллов) обработаны и подтверждена их корректность.

**4.2 Интеграционное тестирование**

Интеграционное тестирование направлено на проверку взаимодействия всех компонентов игры в реальных сценариях, как в оффлайн-режиме, так и в онлайн-режиме.  
– Тестирование оффлайн-режима проводилось с двумя локальными контроллерами, имитирующими двух игроков на одном устройстве. Проверялось, что методы SetAsLocal(...) корректно устанавливают клавиши управления, оба персонажа регистрируются в GameManager, метод SendInput() не вызывается (из-за IsOffline = true), физика движений и столкновений работают одинаково для обоих героев, DoorController отслеживает сбор трёх кристаллов, и при одновременном подходе к дверям видим эффект исчезновения дверей и появления лестницы. Смертельные зоны убивают только соответствующего персонажа, а повторный заход в триггер двери без накопления новых кристаллов не изменяет состояние.  
– В онлайн-режиме проверялось следующее: хост создаёт комнату (метод StartHost()), открывает TcpListener, клиент подключается по IP (метод ConnectToHost(ip)), хост отправляет HOST\_READY и клиент – READY, после чего на стороне хоста активируется кнопка «Start Game». При нажатии «Start Game» отправляется сообщение START, оба игрока через 0.1 с загружают сцену «Game» (хост через вызов Invoke, клиент – сразу при получении пакета). Затем PlayerSpawner распознаёт роли, назначает локальный и удалённый контроллеры и вызывает GameManager.RegisterPlayers(local, remote). Проверялось, что каждое сообщение «INPUT» (сигналы влево/вправо/прыжок) приходит к партнёру с периодичностью около 50 мс, и удалённая сторона корректно применяет ApplyRemoteInput(...). Тестировались следующие сценарии:  
– одновременное движение обоих героев: команды «влево» и «вправо» приходят независимо, физика каждого клиента остаётся синхронизированной в пределах погрешности одного–двух кадров;  
– прыжок и посадка: при «прыжке» у одного игрока аналогичный прыжок появляется у партнёра с минимальной задержкой, и оба героя двигаются по своим параболам независимо;  
– сбор кристаллов: если один игрок собирает кристалл, у другого об этом нет прямого уведомления, но DoorController по флагу crystalsCollected на локальной стороне правильно реагирует, когда герой подходит к двери;  
– совместный подход к дверям: проверялось, что оба fireReachedDoor и waterReachedDoor устанавливаются после отправки соответствующих сообщений (логika OnNetworkMessage), и срабатывает FadeOutDoorAndShowLadder() одновременно на обеих машинах;  
– смерть персонажа: если клиент заходит в лаву, у партнёра вызывается GameManager.HandleDeath(), оба уровня перезагружаются через 1 с в обоих экземплярах.

В процессе интеграционного тестирования было обнаружено, что при сильной задержке сети (> 200 мс) четырёхкратный повтор прыжка может приводить к несинхронному положению героев (один прыгает чуть позже). Однако в течение следующих пакетов состояние выравнивается, и рассинхронизация не накапливается. Кроме того, при одновременных отправках «HOST\_READY» и «READY» кнопка «Start Game» иногда активировалась с заметной задержкой (~0,1 с), но впоследствии отправка «START» срабатывала корректно. В целом интеграционное тестирование подтвердило, что все ключевые компоненты (NetworkManagerTCP, GameManager, Controller, DoorController, DeathZone) взаимодействуют стабильно и обеспечивают ожидаемое поведение в сценариях оффлайн и онлайн.

**4.3 Тестирование сетевой устойчивости**

Целью этого этапа является оценка работоспособности сетевого взаимодействия при различных условиях: потеря пакетов, высокая задержка, неожиданное закрытие соединения.  
 Для симуляции потери пакетов использовались встроенные инструменты ОС или сторонние утилиты (например, Network Link Conditioner), в которых задавались уровни потерь 10 % и 20 %. При 10 % потерь игра сохраняла управляемость: команды «INPUT» просто терялись, но последующие пакеты восстанавливали положение героев без видимого залипания. При 20 % потерь наблюдались короткие подёргивания и кратковременные рассинхронизации, но в рамках двух тысяч мс физика всё равно выравнивала позиции.  
 Для проверки высокой задержки («лаг») применялись параметры эмуляции 200 мс и 500 мс RTT. При 200 мс наиболее заметно, что при прыжке вода или огонь на одном клиенте отстают от другого, однако после приземления рассинхронизация не накапливается. При 500 мс визуальный лаг становится слишком большим, и совместное прохождение значительно усложняется. Рекомендуется вводить местную предсказательную интерполяцию или минимальный «буфер» ввода в будущих версиях для сглаживания таких задержек.  
 Проверялось внезапное закрытие приложения одним из игроков (аварийный выход). Когда клиент закрывался без явного OnApplicationQuit(), поток ReceiveLoop() на хосте останавливался (возвращая длину ≤ 0), выводился лог «Stream closed», а на клиентской стороне при возобновлении/перезапуске окне «OnlineMenu» более не появлялась кнопка «Start Game». В текущей реализации это вёлο к остановке прогресса в онлайн-режиме, но не вызывало краха приложения. Рекомендуется добавить более понятное уведомление о разрыве соединения и возвращение игроков в главное меню.  
 Одновременное подключение нескольких клиентов не тестировалось, поскольку архитектура рассчитана только на два участника. При попытке подключить третьего клиента TcpListener.AcceptTcpClient() всё равно принимает только первого из двух, а последующие клиенты не смогут установить соединение. Для будущих возможностей расширения до более чем двух игроков потребуется полностью переработать сетевую логику.

Итоговый вывод по тестированию сетевой устойчивости: сетевой протокол на основе TCP без дополнительных механизмов коррекции подходит для локальных сетей с потерями до 10 % и задержками до 200 мс. При более экстремальных условиях необходимо ввести дополнительные механизмы (интерполяцию, ресенд, подтверждения). На текущем этапе игра демонстрирует стабильную работу при обычных условиях локального LAN-соединения.

**5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ**

**5.1 Описание пользовательского интерфейса**

Программное средство состоит из трёх основных интерфейсных экранов: главное меню, онлайн-меню и игровая сцена. Все элементы управления разработаны в Unity с использованием системы Canvas, а также оформлены в едином визуальном стиле. В качестве фона для меню использованы иллюстрации с атмосферной графикой, а также подобранная цветовая схема для кнопок — синие и бирюзовые оттенки, создающие стильную и гармоничную визуальную композицию. Цвет текста на кнопках и фонах адаптирован под общий дизайн и обеспечивает хорошую читаемость.

На главном меню пользователю предлагается выбрать один из двух режимов: «Играть оффлайн» или «Играть онлайн». Оформление содержит фон с темной полупрозрачной подложкой под кнопками, крупные шрифты и центральное размещение элементов управления.



Рисунок 5.1 – Главное меню приложения

При выборе режима «Играть онлайн» загружается онлайн-меню, в котором можно выбрать режим работы (хост или клиент), ввести IP-адрес партнёра, подключиться к игре или запустить её после установления соединения. В интерфейсе реализованы всплывающие уведомления, отображение IP-адреса хоста и динамическое переключение видимости элементов в зависимости от выбранного режима. В нижней части онлайн-меню добавлена кнопка «Назад», которая позволяет вернуться в главное меню без необходимости перезапуска приложения.

****

Рисунок 5.2 – Онлайн-меню подключения к игре

Во время игрового процесса пользователь наблюдает игровую сцену, в которой представлены персонажи, уровни, объекты и анимации. Интерфейс в игровой сцене минимален, чтобы не отвлекать от игрового процесса. При необходимости можно активировать отладочную консоль, отображающую внутриигровые логи и сетевые сообщения. Эта консоль размещена внизу экрана и оформлена в виде простого текстового окна с возможностью скопировать логи. Также на игровой сцене реализована кнопка «Назад», которая позволяет в любой момент выйти из игры и вернуться в главное меню

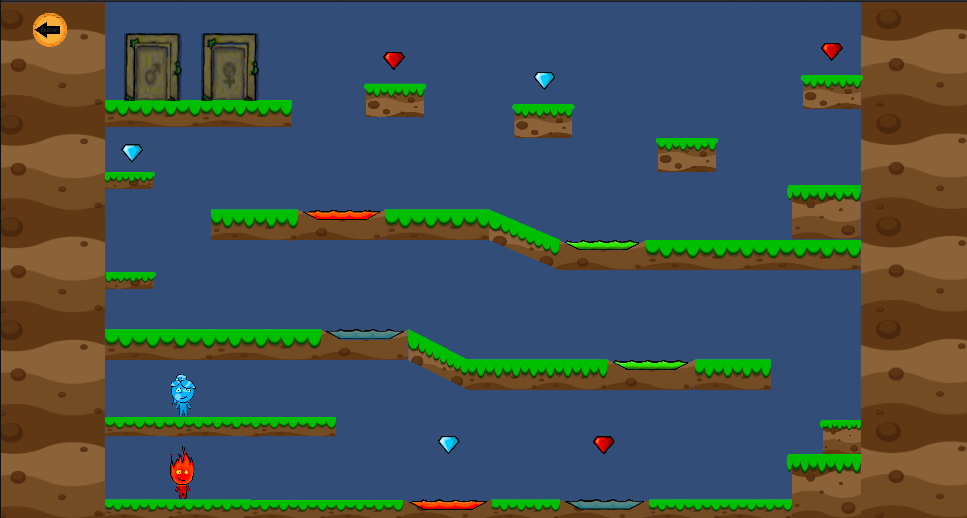


Рисунок 5.3 – Игровая сцена с активным геймплеем

Таким образом, интерфейс приложения интуитивно понятен, логически разделён на этапы взаимодействия, и оформлен в едином стиле, подходящем для жанра игры.

**5.2 Управление программным средством**  
 После запуска пользователь попадает в главное меню, где может выбрать один из двух режимов:  
– «Играть оффлайн» — запускает игру на одном устройстве, управление осуществляется с клавиатуры двумя игроками;  
– «Играть онлайн» — переходит к сетевому режиму, где необходимо выбрать режим (хост или клиент) и, при необходимости, ввести IP-адрес другого игрока.

В режиме хоста необходимо нажать кнопку «Host», после чего отобразится IP-адрес, который можно передать второму игроку. После подключения клиента станет доступна кнопка «Start Game», которая запускает игру для обоих игроков.

В режиме клиента необходимо ввести IP-адрес хоста и нажать кнопку «Connect». После успешного подключения игрок увидит сообщение об этом, а хост получит возможность запустить игру.

На обеих этих сценах также доступна кнопка «Назад», позволяющая вернуться в главное меню без завершения программы.

В оффлайн-режиме управление персонажами осуществляется с одного компьютера:  
– Игрок 1 (Огонь): клавиши A (влево), D (вправо), W (прыжок);  
– Игрок 2 (Вода): клавиши ← (влево), → (вправо), ↑ (прыжок).

В онлайн-режиме каждый управляет своим персонажем с собственной клавиатуры на своём устройстве. Управление аналогично: хост управляет Огнём (A/D/W), клиент – Водой (←/→/↑).

Во время игры игроки должны собирать кристаллы и координировать действия, чтобы одновременно подойти к своим дверям. Уровень завершается, когда оба соберут не менее трёх кристаллов и доберутся до дверей. В случае гибели одного из персонажей уровень перезапускается автоматически.

Блок схема для модели игры представлена на рисунке 5.3

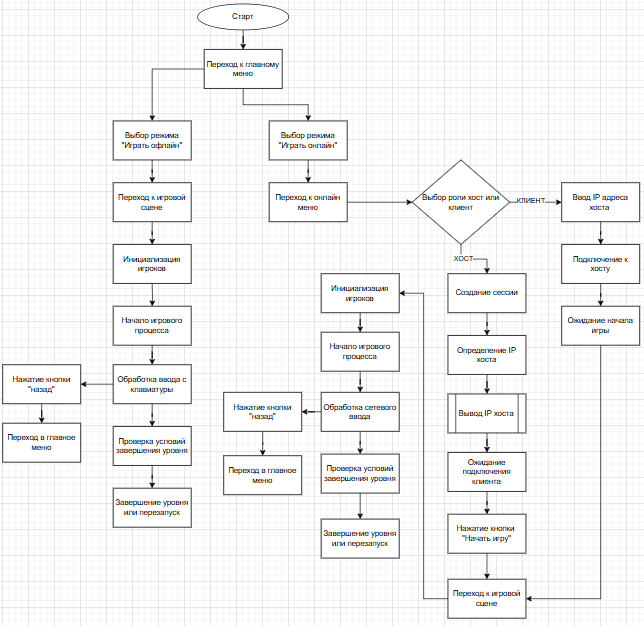


Рисунок 5.3 – Блок схема для модели игры

ЗАКЛЮЧЕНИЕ

В рамках выполнения курсового проекта была разработана сетевая 2D-игра «Огонь и Вода», реализованная в среде Unity с использованием C# и собственной низкоуровневой TCP-сети без сторонних библиотек. Проект представляет собой полноценный прототип классической кооперативной игры с поддержкой двух режимов: оффлайн (локальное управление двумя персонажами на одном устройстве) и онлайн (игра по сети между двумя компьютерами в архитектуре P2P).

Ключевой особенностью проекта стало ручное построение сетевого взаимодействия: отправка JSON-сообщений с вводом игроков, организация потоков TCPListener и TcpClient, реализация асинхронного приема и отправки сообщений в фоновом потоке, а также корректная регистрация и синхронизация действий персонажей. Такой подход позволил глубже освоить принципы сетевого программирования, научиться работать с многопоточностью, сериализацией, асинхронной логикой и Unity-сценариями.

Кроме того, в процессе работы были получены практические навыки:  
– структурирования проекта на модули (GameManager, NetworkManagerTCP, PlayerSpawner и др.);  
– создания адаптивного пользовательского интерфейса с логикой переключения режимов и экранов;  
– обработки событий взаимодействия (зоны смерти, двери, кристаллы);  
– проектирования игрового процесса с учетом особенностей кооперативного прохождения;  
– тестирования юнитов и интеграционного взаимодействия.

Также была реализована отладочная консоль, позволяющая в реальном времени отслеживать сетевые события и внутренние логи, что существенно упростило тестирование. В ходе работы над проектом были устранены десятки потенциальных проблем, таких как рассинхронизация движения, повторные подключения, ошибки соединения, что сделало приложение более устойчивым и предсказуемым в поведении.

Разработанная архитектура легко расширяема: в будущем можно добавить систему уровней, таймеры, очки, поддержку более чем двух игроков, систему комнаты и перенаправление соединений через NAT.

Таким образом, проект продемонстрировал возможность ручной реализации сетевой игры с минимальными затратами и без использования сторонних инструментов. Лично я приобрёл ценный опыт в разработке кооперативного игрового процесса, построении клиент-серверной логики, юнит-тестировании и построении интерфейсов в Unity. Эти знания и навыки могут быть применены как в будущих игровых проектах, так и при создании других сетевых приложений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. **ГОСТ 19.701-90**. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения. – М.: Издательство стандартов, 1990.[edsd.ru+2techwrconsult.com+2docs.cntd.ru+2](https://techwrconsult.com/library/19.701?utm_source=chatgpt.com)

2. **ГОСТ 19.002-80**. Единая система программной документации. Схемы алгоритмов и программ. Правила выполнения. – М.: Издательство стандартов, 1980.[edsd.ru+4swrit.ru+4docs.cntd.ru+4](https://www.swrit.ru/doc/espd/19.002-80.pdf?utm_source=chatgpt.com)

3. **Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес**. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2009.[pro-prof.com](https://pro-prof.com/books-software-design?utm_source=chatgpt.com)

4. **Роберт Мартин**. Чистая архитектура. Искусство разработки программного обеспечения. – М.: ДМК Пресс, 2018.[habr.com](https://habr.com/ru/companies/ru_mts/articles/744602/?utm_source=chatgpt.com)

5. **Стив МакКоннелл**. Совершенный код: Практическое руководство по разработке программного обеспечения. – М.: Вильямс, 2004.[tewris.com](https://tewris.com/10-books-for-software-developers?utm_source=chatgpt.com)

6. **Мартин Фаулер**. Архитектура корпоративных приложений. – М.: Вильямс, 2003.[sky.pro](https://sky.pro/wiki/javascript/luchshie-knigi-po-arhitekture-programmirovaniya/?utm_source=chatgpt.com)

7. **Lucidchart**. Пример блок-схемы программы. [Электронный ресурс]. – Режим доступа: <https://lucid.co/ru/templates/program-block-diagram>[lucid.co](https://lucid.co/ru/templates/program-block-diagram?utm_source=chatgpt.com)

8. **EdrawSoft**. Примеры блок-схем алгоритмов. [Электронный ресурс]. – Режим доступа: <https://www.edrawsoft.com/ru/algorithm-flowchart-examples.html>[edrawsoft.com](https://www.edrawsoft.com/ru/algorithm-flowchart-examples.html?srsltid=AfmBOorj5T372p2TCP6RK_Rgtiu4a2QlcDCLErb5V7BUY4nu6v1HdXFs&utm_source=chatgpt.com)

9. **Creately**. Шаблоны блок-схем. [Электронный ресурс]. – Режим доступа: <https://creately.com/diagram-community/popular-ru/t/flowchart-ru>[creately.com](https://creately.com/diagram-community/popular-ru/t/flowchart-ru?utm_source=chatgpt.com)

10. **Visual Paradigm Online**. Шаблоны блок-схем. [Электронный ресурс]. – Режим доступа: <https://online.visual-paradigm.com/ru/diagrams/templates/block-diagram/>

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ

public class NetworkManagerTCP : MonoBehaviour

{

public static NetworkManagerTCP Instance;

public int port = 7777;

private TcpListener listener;

private TcpClient client;

private Thread listenThread, receiveThread;

private volatile bool clientConnectedFlag = false;

public bool IsOffline = false;

public bool IsHost { get; private set; } = false;

void Awake()

{

if (Instance == null) Instance = this; else Destroy(gameObject);

DontDestroyOnLoad(gameObject);

}

void Update()

{

// Проверяем, поднят ли флаг подключения

if (clientConnectedFlag)

{

clientConnectedFlag = false;

// Уведомляем UI — хост готов

OnlineMenuManager.Instance.OnNetworkReady(true);

}

}

// --- Host ---

public void StartHost()

{

IsHost = true; // помечаем, что мы хост

Debug.Log("[Net] Хост: StartHost() called, listening on loopback:" + port);

listener = new TcpListener(IPAddress.Loopback, port); // вместо Any

listener.Start();

Debug.Log("[Net] Хост: Listener started!");

listenThread = new Thread(AcceptClient) { IsBackground = true };

listenThread.Start();

}

// В AcceptClient()

void AcceptClient()

{

client = listener.AcceptTcpClient();

Debug.Log("[Net] Хост: Client connected");

// Отправляем HOST\_READY

SendMessage(new NetMessage { type = "HOST\_READY" });

clientConnectedFlag = true;

StartReceiving();

}

// В ReceiveLoop()

void ReceiveLoop()

{

var stream = client.GetStream();

byte[] buf = new byte[1024];

while (true)

{

int len = stream.Read(buf, 0, buf.Length);

if (len <= 0) { Debug.Log("[Net] ReceiveLoop: Stream closed"); break; }

var msg = Encoding.UTF8.GetString(buf, 0, len);

Debug.Log("[Net] Получено → " + msg);

GameManager.Instance.OnNetworkMessage(msg);

}

}

public void ConnectToHost(string ip)

{

IsHost = false;

client = new TcpClient();

client.Connect(ip, port);

Debug.Log($"[Net] Клиент: Connected to host {ip}:{port}");

StartReceiving();

SendMessage(new NetMessage { type = "READY" });

}

void StartReceiving()

{

receiveThread = new Thread(ReceiveLoop);

receiveThread.Start();

}

// вместо SendMessage(string), делаем перегрузку:

public void SendMessage(NetMessage msg)

{

string json = JsonUtility.ToJson(msg);

var data = Encoding.UTF8.GetBytes(json);

client.GetStream().Write(data, 0, data.Length);

Debug.Log($"[Net] Отправлено → {json}");

}

void OnApplicationQuit()

{

listener?.Stop();

client?.Close();

listenThread?.Abort();

receiveThread?.Abort();

}

public void Disconnect()

{

Debug.Log("[Net] Disconnect() called.");

// Останавливаем слушатель, если он был запущен

if (listener != null)

{

try

{

listener.Stop();

Debug.Log("[Net] Listener stopped.");

}

catch (System.Exception e)

{

Debug.LogWarning($"[Net] Ошибка при остановке listener: {e.Message}");

}

listener = null;

}

// Закрываем клиентское соединение

if (client != null)

{

try

{

client.Close();

Debug.Log("[Net] Client closed.");

}

catch (System.Exception e)

{

Debug.LogWarning($"[Net] Ошибка при закрытии client: {e.Message}");

}

client = null;

}

// Завершаем поток, который принимает входящие подключения (для хоста)

if (listenThread != null && listenThread.IsAlive)

{

try

{

listenThread.Abort();

Debug.Log("[Net] listenThread aborted.");

}

catch { }

listenThread = null;

}

// Завершаем поток получения данных

if (receiveThread != null && receiveThread.IsAlive)

{

try

{

receiveThread.Abort();

Debug.Log("[Net] receiveThread aborted.");

}

catch { }

receiveThread = null;

}

// Сбрасываем флаги и флаг хоста

clientConnectedFlag = false;

IsHost = false;

// Удаляем саму синглтон-ссылку, чтобы при повторном входе в сцену создавался новый экземпляр

Instance = null;

// Опционально: уничтожаем GameObject (если нужен полный пересоздаваемый NetworkManager)

Destroy(gameObject);

}

}

public class MenuManager : MonoBehaviour

{

public void PlayOffline()

{

// Если объект NetworkManagerTCP уже есть на сцене — ставим IsOffline = true

if (NetworkManagerTCP.Instance != null)

{

NetworkManagerTCP.Instance.IsOffline = true;

}

else

{

Debug.LogWarning("[MenuManager] NetworkManagerTCP.Instance == null, пропускаем IsOffline.");

}

// Загружаем сцену Game

SceneManager.LoadScene("Game");

}

public void PlayOnline()

{

// То же самое: убедимся, что IsOffline = false

if (NetworkManagerTCP.Instance != null)

{

NetworkManagerTCP.Instance.IsOffline = false;

}

else

{

Debug.LogWarning("[MenuManager] NetworkManagerTCP.Instance == null, пропускаем IsOffline = false.");

}

StartCoroutine(LoadOnlineWithEmpty());

}

private IEnumerator LoadOnlineWithEmpty()

{

// 1) Загружаем «пустую» сцену

SceneManager.LoadScene("EmptyScene");

// 2) Ждём один кадр (yield return null)

yield return null;

// 3) Теперь загружаем нужную OnlineMenu

SceneManager.LoadScene("OnlineMenu");

}

public void ReturnToMainMenu()

{

if (NetworkManagerTCP.Instance != null)

{

NetworkManagerTCP.Instance.Disconnect();

}

// 2) Уничтожаем GameManager, чтобы все данные предыдущего раунда сбросились

if (GameManager.Instance != null)

{

Destroy(GameManager.Instance.gameObject);

}

if (OnlineMenuManager.Instance != null)

{

Destroy(OnlineMenuManager.Instance.gameObject);

OnlineMenuManager.Instance = null;

}

// 3) Переходим в главное меню

SceneManager.LoadScene("Menu");

}

}

[RequireComponent(typeof(Rigidbody2D))]

public class Controller : MonoBehaviour

{

[Header("Настройки")]

public float moveSpeed = 3f;

public float jumpForce = 7f;

public LayerMask groundLayer;

[Header("Ground Check")]

public Transform groundCheck;

public float groundCheckRadius = 0.1f;

private Rigidbody2D rb;

private bool isGrounded;

private bool jumpPressed;

private float moveInputX;

[Header("Клавиши")]

public KeyCode leftKey;

public KeyCode rightKey;

public KeyCode jumpKey;

[Header("Animation")]

public Animator bodyAnimator;

public Animator headAnimator;

public SpriteRenderer bodySprite;

public SpriteRenderer headSprite;

[Header("Сетевая метка")]

public bool isLocal = false;

[HideInInspector]

public int crystalsCollected = 0;

public enum ElementType { Fire, Water }

public ElementType element; // назначим в инспекторе

void Awake()

{

rb = GetComponent<Rigidbody2D>();

}

public void AddCrystal()

{

crystalsCollected++;

// Здесь можно добавить звук сборки, UI-обновление и т.п.

Debug.Log($"{element} собрал кристалл. Всего: {crystalsCollected}");

}

// Inside Controller.cs

public void SetAsLocal(KeyCode left, KeyCode right, KeyCode jump)

{

isLocal = true;

leftKey = left;

rightKey = right;

jumpKey = jump;

}

void Update()

{

if (!isLocal) return;

// 1) Горизонтальное состояние

float x = 0f;

if (Input.GetKey(leftKey)) x = -1f;

if (Input.GetKey(rightKey)) x = 1f;

moveInputX = x;

// 2) Состояние прыжка

// Если игрок нажал кнопку jumpKey \*\*и\*\* он сейчас на земле — мы сообщаем «я хочу прыгнуть»

// Но сохраняем этот флаг до тех пор, пока удалённая сторона не сбросит его.

if (Input.GetKey(jumpKey) && isGrounded)

{

// Передаём: «я всё ещё нажимаю прыжок» до момента, пока удалённая сторона

// не сбросит jumpPressed (например, после первого реагирования)

jumpPressed = true;

}

else if (!Input.GetKey(jumpKey))

{

// Если нажатие отпущено — сбрасываем флаг, т.к. пользователь больше не хочет прыгать

jumpPressed = false;

}

}

void FixedUpdate()

{

// 1) Проверим, на земле ли мы

isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius, groundLayer);

// 2) Прыжок (пока isGrounded был true)

if (jumpPressed)

{

// Сброс текущей вертикальной скорости, чтобы прыжок был ровным

rb.linearVelocity = new Vector2(rb.linearVelocity.x, 0f);

rb.AddForce(Vector2.up \* jumpForce, ForceMode2D.Impulse);

jumpPressed = false;

}

// 3) Горизонтальное движение

Vector2 vel = rb.linearVelocity;

vel.x = moveInputX \* moveSpeed;

rb.linearVelocity = vel;

// 4) Анимация

float absSpeed = Mathf.Abs(vel.x);

bodyAnimator.SetFloat("Speed", absSpeed);

headAnimator.SetFloat("Speed", absSpeed);

// 5) Отражение спрайтов

if (vel.x > 0.1f)

{

bodySprite.flipX = false;

headSprite.flipX = false;

}

else if (vel.x < -0.1f)

{

bodySprite.flipX = true;

headSprite.flipX = true;

}

}

void OnDrawGizmosSelected()

{

if (groundCheck != null)

{

Gizmos.color = Color.red;

Gizmos.DrawWireSphere(groundCheck.position, groundCheckRadius);

}

}

// Вызывается из GameManager при получении net INPUT

public void ApplyRemoteInput(bool left, bool right, bool jump)

{

if (isLocal) return;

float x = 0;

if (left) x = -1f;

if (right) x = 1f;

moveInputX = x;

if (jump && isGrounded)

jumpPressed = true;

}

public bool ConsumeJump()

{

bool j = jumpPressed;

jumpPressed = false;

return j;

}

public float GetMoveInputX() => moveInputX;

}

public class GameManager : MonoBehaviour

{

private static GameManager \_instance;

public static GameManager Instance

{

get

{

if (\_instance == null)

{

var go = new GameObject("GameManager");

\_instance = go.AddComponent<GameManager>();

DontDestroyOnLoad(go);

}

return \_instance;

}

}

private bool hostReadyFlag = false;

private bool clientReadyFlag = false;

private bool startGameFlag = false;

Controller localCtrl, remoteCtrl;

private bool deathInProgress = false;

// Флаги, подошли ли Fire/Water к своим дверям

private bool fireReachedDoor = false;

private bool waterReachedDoor = false;

void Awake()

{

if (\_instance == null)

{

\_instance = this;

DontDestroyOnLoad(gameObject);

}

else if (\_instance != this)

{

Destroy(gameObject);

}

}

public void RegisterPlayers(Controller local, Controller remote)

{

localCtrl = local;

remoteCtrl = remote;

var nm = NetworkManagerTCP.Instance;

bool isOffline = (nm == null) || nm.IsOffline;

if (nm != null && !isOffline)

{

InvokeRepeating(nameof(SendInput), 0f, 0.05f);

}

}

void SendInput()

{

if (localCtrl == null || NetworkManagerTCP.Instance.IsOffline)

return;

float x = localCtrl.GetMoveInputX();

bool jump = localCtrl.ConsumeJump();

NetMessage msg = new NetMessage

{

type = "INPUT",

left = x < 0,

right = x > 0,

jump = jump

};

NetworkManagerTCP.Instance.SendMessage(msg);

}

public void OnNetworkMessage(string json)

{

var msg = JsonUtility.FromJson<NetMessage>(json);

switch (msg.type)

{

case "INPUT":

if (remoteCtrl != null)

remoteCtrl.ApplyRemoteInput(msg.left, msg.right, msg.jump);

break;

case "READY":

clientReadyFlag = true;

break;

case "HOST\_READY":

hostReadyFlag = true;

break;

case "START":

startGameFlag = true;

break;

}

}

void Update()

{

if (clientReadyFlag)

{

clientReadyFlag = false;

OnlineMenuManager.Instance.OnNetworkReady(true);

}

if (hostReadyFlag)

{

hostReadyFlag = false;

OnlineMenuManager.Instance.OnNetworkReady(true);

}

if (startGameFlag)

{

startGameFlag = false;

if (SceneManager.GetActiveScene().name == "OnlineMenu")

SceneManager.LoadScene("Game");

}

}

public void HandleDeath(Controller deadCtrl)

{

if (deathInProgress) return;

deathInProgress = true;

Destroy(deadCtrl.gameObject);

CancelInvoke(nameof(SendInput));

StartCoroutine(RestartAfterDelay());

}

private System.Collections.IEnumerator RestartAfterDelay()

{

yield return new WaitForSeconds(1f);

string current = SceneManager.GetActiveScene().name;

SceneManager.LoadScene(current);

deathInProgress = false;

}

/// <summary>

/// Вызывается из DoorController, когда герой подошёл к своей двери.

/// </summary>

public void PlayerReachedDoor(Controller.ElementType element)

{

Debug.Log($"[GameManager] PlayerReachedDoor: {element}");

if (element == Controller.ElementType.Fire)

fireReachedDoor = true;

else if (element == Controller.ElementType.Water)

waterReachedDoor = true;

// Как только оба подошли – запускаем исчезновение дверей и появление лестниц

if (fireReachedDoor && waterReachedDoor)

{

Debug.Log("[GameManager] Оба героя у дверей – запускаем FadeOut");

StartCoroutine(OnBothPlayersAtDoorsCoroutine());

}

}

private System.Collections.IEnumerator OnBothPlayersAtDoorsCoroutine()

{

// Мгновенная задержка, чтобы оба значка уже точно зажглись

yield return null;

// Находим все DoorController в сцене

DoorController[] doors = FindObjectsOfType<DoorController>();

foreach (var d in doors)

{

d.FadeOutDoorAndShowLadder();

}

}

}

public class OnlineMenuManager : MonoBehaviour

{

public static OnlineMenuManager Instance;

public Toggle hostToggle;

public Toggle clientToggle;

public InputField ipInput;

public Button hostButton;

public Button connectButton;

public Button startGameButton;

public Text statusText;

public Text hostIPText;

private bool isHost;

public bool IsHost => isHost;

void Awake()

{

if (Instance == null)

{

Instance = this;

//DontDestroyOnLoad(gameObject);

}

else Destroy(gameObject);

}

void Start()

{

// Подключаем события переключателей

hostToggle.onValueChanged.AddListener((val) => OnModeChanged());

clientToggle.onValueChanged.AddListener((val) => OnModeChanged());

// Изначально — хост

hostToggle.isOn = true;

OnModeChanged();

startGameButton.interactable = false;

statusText.text = "";

}

public void OnModeChanged()

{

// Диагностический лог

Debug.Log($"[OnlineMenu] OnModeChanged: hostToggle.isOn = {hostToggle.isOn}");

isHost = hostToggle.isOn;

// Переключаем UI

ipInput.gameObject.SetActive(!isHost);

hostButton.gameObject.SetActive(isHost);

connectButton.gameObject.SetActive(!isHost);

statusText.text = "";

}

public void OnHostButton()

{

statusText.text = "Ожидание подключения…";

NetworkManagerTCP.Instance.StartHost();

string localIP = GetLocalIPAddress();

hostIPText.text = "Ваш IP: " + localIP;

}

public void OnConnectButton()

{

string ip = ipInput.text.Trim();

if (string.IsNullOrEmpty(ip)) { statusText.text = "Введите IP"; return; }

statusText.text = "Подключаюсь…";

// если вы вводите 127.0.0.1, оно точно пойдёт локальному слушателю

NetworkManagerTCP.Instance.ConnectToHost(ip);

}

public void OnNetworkReady(bool remoteIsReady)

{

Debug.Log($"[OnlineMenu] OnNetworkReady(isHost={isHost}, ready={remoteIsReady})");

if (isHost)

{

statusText.text = "Клиент подключён!";

startGameButton.interactable = true;

}

else

{

statusText.text = "Вы подключены!";

connectButton.interactable = false;

ipInput.interactable = false;

}

}

public void OnStartGameButton()

{

Debug.Log("[OnlineMenu] OnStartGameButton — шлём START");

NetworkManagerTCP.Instance.SendMessage(new NetMessage { type = "START" });

// Вместо немедленного LoadScene поставим Invoke:

Invoke(nameof(LoadGameScene), 0.1f);

}

void LoadGameScene()

{

Debug.Log("[OnlineMenu] LoadGameScene() — загружаем Game");

SceneManager.LoadScene("Game");

}

public void OnCopyIPButton()

{

string ip = hostIPText.text.Replace("Ваш IP: ", "").Trim();

GUIUtility.systemCopyBuffer = ip;

Debug.Log("[OnlineMenu] IP скопирован в буфер: " + ip);

statusText.text = "IP скопирован!";

}

private string GetLocalIPAddress()

{

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

foreach (var ip in host.AddressList)

{

if (ip.AddressFamily == AddressFamily.InterNetwork)

return ip.ToString();

}

return "127.0.0.1";

}

}

public class PlayerSpawner : MonoBehaviour

{

public GameObject firePrefab;

public GameObject waterPrefab;

void Start()

{

if (firePrefab == null || waterPrefab == null)

{

Debug.LogError("PlayerSpawner: префабы не назначены в инспекторе!");

return;

}

// Безопасно получаем синглтон

var nm = NetworkManagerTCP.Instance;

// Если синглтон не создался или мы в офлайн-режиме → офлайн-спавн

bool isOffline = (nm == null) || nm.IsOffline;

bool amHost = (nm != null) && nm.IsHost;

Controller local, remote;

if (isOffline)

{

// Офлайн: создаём обоих локальных

var f = Instantiate(firePrefab, new Vector3(-6, -1, 0), Quaternion.identity);

var w = Instantiate(waterPrefab, new Vector3(-7.6f, 0.5f, 0), Quaternion.identity);

// Настраиваем управление

f.GetComponent<Controller>().SetAsLocal(KeyCode.A, KeyCode.D, KeyCode.W);

w.GetComponent<Controller>().SetAsLocal(KeyCode.LeftArrow, KeyCode.RightArrow, KeyCode.UpArrow);

// Регистрируем обоих сразу

GameManager.Instance.RegisterPlayers(

f.GetComponent<Controller>(),

w.GetComponent<Controller>()

);

return;

}

if (amHost)

{

var f = Instantiate(firePrefab, new Vector3(-6, -1, 0), Quaternion.identity);

var w = Instantiate(waterPrefab, new Vector3(-7.6f, 0.5f, 0), Quaternion.identity);

// Огонь — локальный у хоста

var ctrlF = f.GetComponent<Controller>();

ctrlF.isLocal = true;

ctrlF.leftKey = KeyCode.A;

ctrlF.rightKey = KeyCode.D;

ctrlF.jumpKey = KeyCode.W;

// Вода — удалённый

var ctrlW = w.GetComponent<Controller>();

ctrlW.isLocal = false;

local = ctrlF;

remote = ctrlW;

}

else

{

var f = Instantiate(firePrefab, new Vector3(-6, -1, 0), Quaternion.identity);

var w = Instantiate(waterPrefab, new Vector3(-7.6f, 0.5f, 0), Quaternion.identity);

// Огонь — удалённый

var ctrlF = f.GetComponent<Controller>();

ctrlF.isLocal = false;

// Вода — локальная у клиента

var ctrlW = w.GetComponent<Controller>();

ctrlW.isLocal = true;

ctrlW.leftKey = KeyCode.LeftArrow;

ctrlW.rightKey = KeyCode.RightArrow;

ctrlW.jumpKey = KeyCode.UpArrow;

local = ctrlW;

remote = ctrlF;

}

GameManager.Instance.RegisterPlayers(local, remote);

}

}