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:

РАЗРАБОТКА СЕТЕВОЙ КОМПЬЮТЕРНОЙ ИГРЫ «КОСМИЧЕСКАЯ ОДИССЕЯ»

БГУИР КП 6-05 0612-01 007 ПЗ

Студент Ёщик Ю. А.

Руководитель Болтак С.В.

Минск 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Интернет и компьютерные технологии стали важной частью нашей повседневной жизни, открывая новые горизонты для общения и досуга. Одним из самых захватывающих способов проведения времени являются сетевые компьютерные игры, которые позволяют игрокам со всего мира объединяться, взаимодействовать и соревноваться в реальном времени. В последние годы наблюдается устойчивый рост интереса к разработке таких игр, что связано с развитием языков программирования и инструментов, упрощающих создание интерактивных приложений. Эти технологии не только обогащают игровой опыт, но и способствуют созданию сообществ, где игроки могут обмениваться опытом и впечатлениями.

Одним из самых популярных жанров игр являются гонки. Гонки представляют собой захватывающий и динамичный способ развлечения, в котором игроки соревнуются на различных транспортных средствах, таких как автомобили, мотоциклы или даже космические корабли. Эти игры требуют от игроков навыков управления, стратегического мышления и быстрой реакции, что делает каждую гонку уникальной и напряженной. Разнообразие трасс и условий, в которых проходят гонки, добавляет игре элемент непредсказуемости и делает каждое задание увлекательным.

В данном курсовом проекте будет представлена разработка сетевой компьютерной игры «Космическая одиссея». Это игра в жанре гонок, происходящая в захватывающем космическом окружении. Игроки смогут управлять космическими кораблями и соревноваться друг с другом на пути к победе. Разработка данной игры предполагает создание игрового окружения и реализацию управления движением, что требует тщательной проработки механик и физики в игре.

Для реализации отображения игроков в режиме реального времени важно обеспечить быструю передачу данных между пользователями. Наиболее подходящий для этого протокол – RUDP (Reliable User Datagram Protocol) [1].

Использование Unity [2] в разработке данной игры позволит создать впечатляющую графику, а также обеспечит гибкость в реализации сетевых функций. Язык программирования C# идеально подходит для работы в Unity благодаря своей простоте и мощным возможностям. C# позволяет быстро реализовывать идеи и делиться ими с командой, что значительно ускоряет процесс разработки.

Целью данного курсового проекта является разработка программного средства – сетевой игры «Космическая одиссея».

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Обзор аналогов

Существует множество игр, предлагающих уникальный опыт в жанре гонок. Все они предоставляют пользователям схожий функционал, включая управление транспортными средствами, возможность соревноваться с другими игроками и разнообразие трасс. Однако каждая игра привносит что-то новое в игровой процесс, будь то уникальная механика, стиль графики или особенности кастомизации, что делает каждую из них по-своему привлекательной и интересной.

В первую очередь нужно обратить внимание на сетевую игру «Wipeout Series» [3]. Интерфейс данной игры представлен на рисунке 1.1.

Рисунок 1.1 – Интерфейс сетевой игры «Wipeout Series»

Wipeout – это серия аркадных гонок, действие которых происходит в футуристическом мире с высокоскоростными антигравитационными транспортными средствами, что создает ощущение невероятной скорости и динамики. Игроки соревнуются на различных трассах, используя ускорители и оружие для противодействия соперникам, что добавляет элемент стратегии в каждую гонку. Яркая графика и электронная музыка создают захватывающую атмосферу, которая погружает игроков в мир высоких технологий и адреналина, заставляя их чувствовать себя частью захватывающего соревнования. Также игра предлагает возможность кастомизации кораблей, что позволяет игрокам адаптировать транспортные средства под свой стиль игры и предпочтения, делая каждую гонку уникальной и персонализированной.

Далее стоит рассмотреть игру «Star Wars: Episode I Racer» [4]. На рисунке 1.2 представлен ее интерфейс.

Рисунок 1.2 – Интерфейс сетевой игры «Star Wars: Episode I Racer»

«Star Wars: Episode I Racer» – это игра, которая предлагает игрокам стать участниками гонок на различных планетах, каждая из которых имеет свои уникальные трассы и условия, что добавляет разнообразие и делает каждую гонку неповторимой.

Следующим аналогом является игра «Redout» [5]. На рисунке 1.3 представлен ее интерфейс.

Рисунок 1.3 – Интерфейс сетевой игры «Redout»

«Redout» – это высокоскоростная аркадная гонка, которая предлагает сложные трассы и разнообразные режимы гонок, включая одиночные и

многопользовательские соревнования. Высокая скорость и реалистичная физика управления создают динамичный и напряжённый игровой процесс, который требует от игроков быстрой реакции и стратегического мышления.

1.2 Постановка задачи

В рамках данной курсовой работы необходимо разработать сетевую игру «Космическая одиссея». В перечень задач при разработке игры входит:

- изучение концепции космических гонок;
- проведение анализа существующих реализаций сетевых игр;
- реализация игрового интерфейса;
- определение функциональных требований к игре.

Основные функциональные требования игры включают реализацию механики гонок:

- создание игровой сессии;
- реализация механики старта гонки;
- настройка физики космических кораблей;
- управление процессом гонки;
- определение победителя;
- реализация механики окончания гонки и отображения результатов.

Помимо основных функций игрового процесса необходимо реализовать удобный интерфейс:

- экран подключения к гонке;
- основной игровой интерфейс: отображение текущего положения в гонке;
- экран завершения гонки.

За автоматизацию игровых процессов будет отвечать сервер, который реализует следующие функции:

- подключение нескольких игроков к одной игровой сессии;
- синхронизация игровых данных между клиентами и сервером;
- обработка игровых событий.

Для разработки игры будет использоваться язык программирования C# и игровая платформа Unity, что обеспечит высокое качество графики и продвинутый сетевой функционал. Unity поддерживает физику и анимацию, что позволит реализовать реалистичное управление космическими кораблями. Также планируется использование Asset Store [6] для интеграции готовых ресурсов, таких как модели и текстуры, что ускорит процесс разработки и повысит качество итогового продукта.

2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

2.1 Структура программы

При разработке программного обеспечения предпочтение будет отдано клиент-серверной архитектуре, как наиболее подходящей для данного типа приложений. Такой выбор обусловлен необходимостью наличия централизованного узла, который регулирует игровое состояние и гарантирует одинаковое представление игры у всех участников. Сервер выступает единым источником истины, что исключает расхождение в игровом состоянии между разными клиентами. Серверная часть берет на себя все ключевые вычисления и принятие решений: обработка физики движения, расчет столкновений, определение пересечения финишной линии, подсчет пройденных кругов. Это обеспечивает защиту от жульничества, так как все важные решения принимаются на сервере, это исключает возможность подделки игровых событий или параметров движения со стороны игроков.

Сетевое взаимодействие будет реализовано на основе гибридного протокола RUDP. Он содержит в себе низкие задержки от UDP и возможность подтверждения доставки от TCP. Гибкость данного протокола позволяет самостоятельно выбирать, какие сообщения требуют подтверждения доставки. Это позволяет оптимизировать сетевой трафик, гарантируя доставку только действительно важных данных.

При разработке сетевого взаимодействия будет реализовано три основных модуля:

– Server – ядро системы, реализующее серверную логику, обработку подключений клиентов, синхронизацию игрового состояния, и трансляцию данных;

– Client – модуль конечного пользователя, обеспечивающий подключение к серверу, обработку входящих данных и отображение интерфейса;

– MessageSystem – промежуточный модуль, обеспечивающий взаимодействие между клиентом и сервером.

В модулях будут реализованы шесть основных классов: RaceManager, Player, PlayerMovement, PlayerController, NetworkManager, UIManager.

Класс RaceManager отвечает за логику гонки: начало и завершение заездов, контроль прохождения кругов, обработку финиша и возрождение игрока в случае столкновения с трассой.

Класс Player на клиентской стороне отвечает за отображение игрока, на серверной за хранение данных игрока.

Класс PlayerMovement отвечает за расчет физики движения на основе данных клиента и обработку столкновений.

Класс PlayerController отвечает за обработку и отправку ввода игрока.

Класс NetworkManager на клиентской стороне отвечает за подключение к серверу, отправку и получение данных, на серверной за управление сетевым подключением, обработку и отправку сообщений.

Класс UIManager отвечает за отображение холстов ввода имени и финиша.

Рассмотрим поля, методы и события основных классов в таблице 2.1.

Таблица 2.1 – Основные классы игры

Название класса	Основные поля	Ключевые методы	Обрабатываемые сообщения
RaceManager	playerLaps: Dictionary<ushort, int> totalLaps: int raceFinished: bool winnerId: ushort raceStarted: bool	RespawnPlayer() OnPlayerConnect() StartRace()	PlayerRespawned CountdownUpdate RaceStateUpdate RaceFinished
Player	Id: ushort IsLocal: bool username: string	Spawn()	playerSpawned playerMovement input
PlayerMovement	inputs: [] bool canMove: bool	EnableMovement() GetInputDirection() ProcessMovement() ProcessRotation()	playerMovement
PlayerController	inputs: [] bool canInput: bool	EnableInput() SendInput()	input
NetworkManager (клиент)	ip: string port: ushort	Connect() DidConnect() FailedToConnect() DidDisconnect()	Все клиентские сообщения
NetworkManager (сервер)	port: ushort maxClientCount: ushort	PlayerLeft()	Все системные сообщения
UIManager	connectUI: GameObject finishUI: GameObject raceUI: GameObject	ConnectClicked() BackToMain() SendName() ShowWinner()	RaceStateUpdate name

Помимо основных классов будут вспомогательные:

– MessageExtentions – класс, расширяющий отправку сообщения, позволяющий отправлять и принимать Vector3 как аргумент;

– FinishLine – класс, отвечающий за обработку пересечения финишной черты.

2.2 Проектирование интерфейса программного средства

Проектирование интерфейса будет осуществлятся на игровом движке Unity, который предоставляет инструменты для 2D/3D, анимации, работы с физикой и мультимедиа.

2.2.1 Подключение к гонке

Холст подключения к гонке представлен на рисунке 2.1, он содержит поле для ввода имени и кнопку «Подключиться».

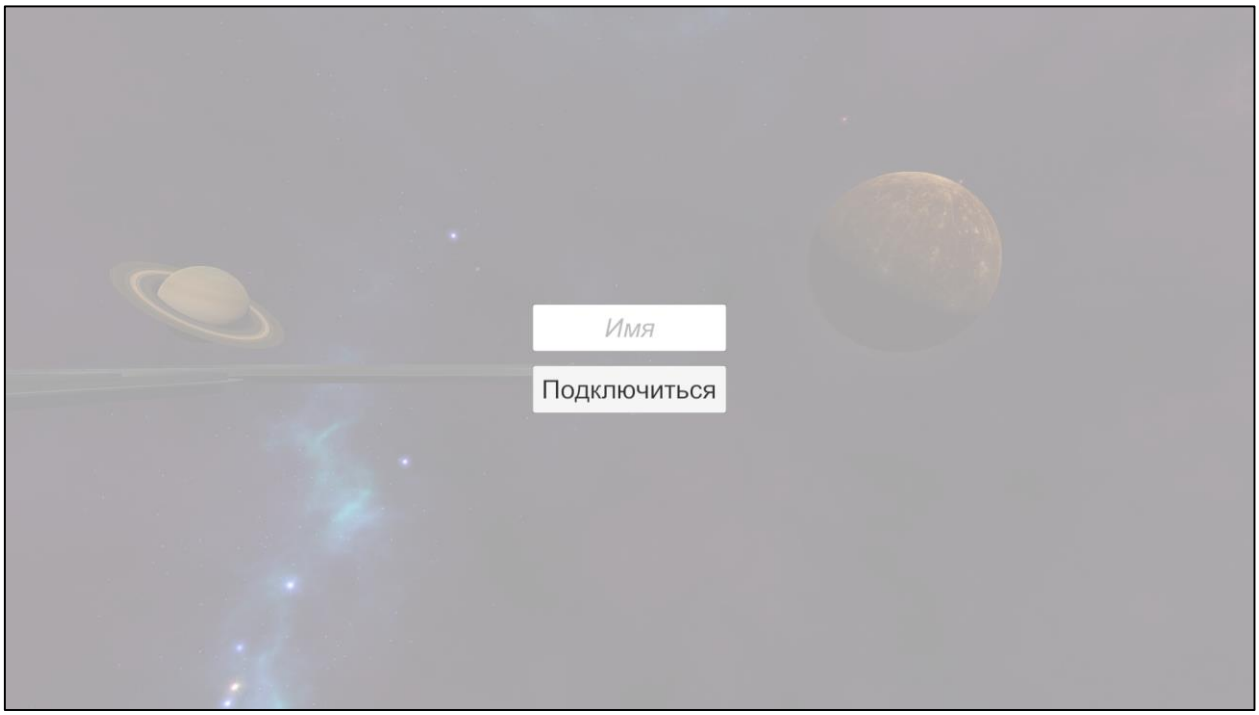

Рисунок 2.1 – Холст подключения к гонке

После ввода имени и нажатия на кнопку, игрок подключается к гонке и ожидает второго для начала гонки.

2.2.2 Игровая локация

После успешного подключения игрок видит изображение, представленное на рисунке 2.2. Когда второй игрок подключается, игроки могут начать движение по карте, изображенной на рисунке 2.3.

После пересечения одним из игроков финишной черты, оба видят холст, изображенный на рисунке 2.4.

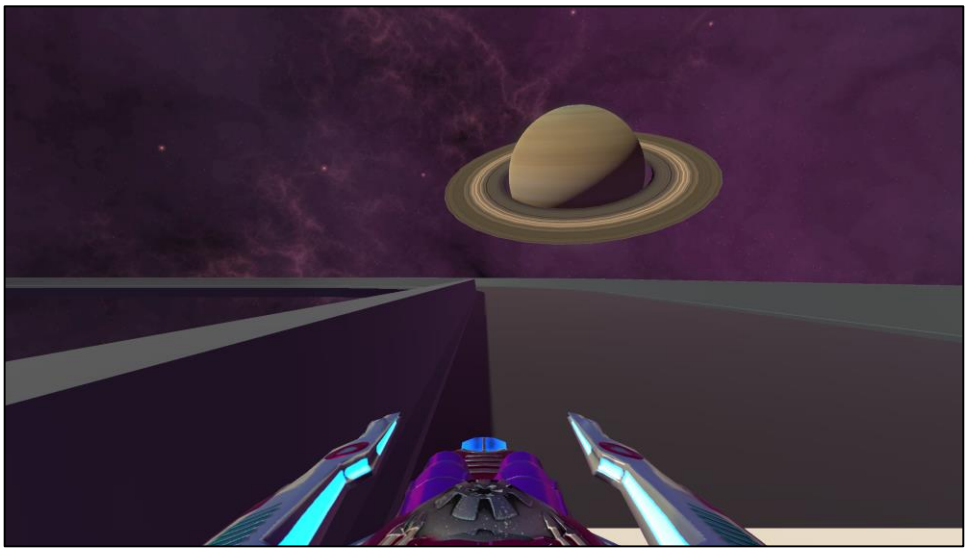

Рисунок 2.2 – Ожидание начала гонки

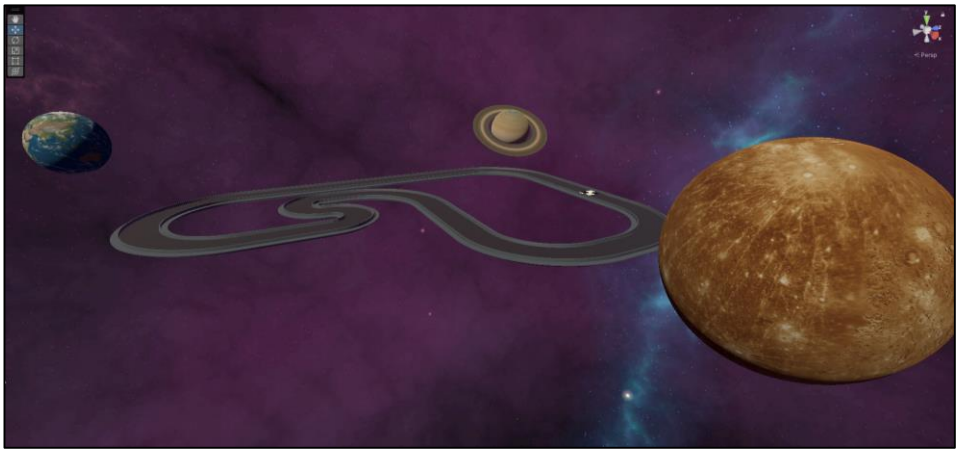

Рисунок 2.3 – Игровая карта

Рисунок 2.4 – Холст с отображением победителя гонки

На рисунке 2.5 представлена 3D-модель игрока.

Рисунок 2.5 – Модель игрока

Модели для неба, планет, трассы и космического корабля были скачаны из Asset Store для ускорения процесса разработки.

2.3 Проектирование функционала программного средства

Для обеспечения корректной работы игры необходимо реализовать основной функционал:

– синхронизация игрового состояния;
– управление игровым циклом;
– обработка физики и коллизий;
– сетевой протокол.

Рассмотрим основные игровые алгоритмы:

– синхронизация игрового состояний;
– старт гонки;
– обработка финиша;

Алгоритм синхронизации игрового состояния представляет собой представляет собой непрерывный циклический процесс, обеспечивающий согласованное отображение игрового мира для всех подключенных участников. Клиенты отправляют на сервер данные о своем вводе, сервер их принимает и обрабатывает, вычисляет новое состояние и отправляет клиентам. На клиентской стороне данные интерполируются для более плавного движения.

Алгоритм синхронизации игрового состояния представлен на рисунке 2.6 в виде блок-схемы.

Рисунок 2.6 – Блок-схема алгоритма синхронизации данных

Алгоритм начала гонки также представляет собой циклический процесс, который выполняется, пока статус гонки не достигнет состояния «GO!». Изначально проверяется, не была ли гонка запущена до этого, в случае, если была, цикл завершается, далее проверка на количество игроков, если их меньше двух, цикл начинается заново. Когда подключено два пользователя, начинается обратный отсчет от пяти до одного, чтоб оба пользователя успели прогрузиться, далее состояние гонки изменяется на «GO!», пользователям разрешается двигаться и состояние гонки отправляется клиентам. После

завершения данного цикла сервер приступает к последующей обработке состояний пользователей.

Алгоритм представлен в виде блок-схемы на рисунке 2.7.

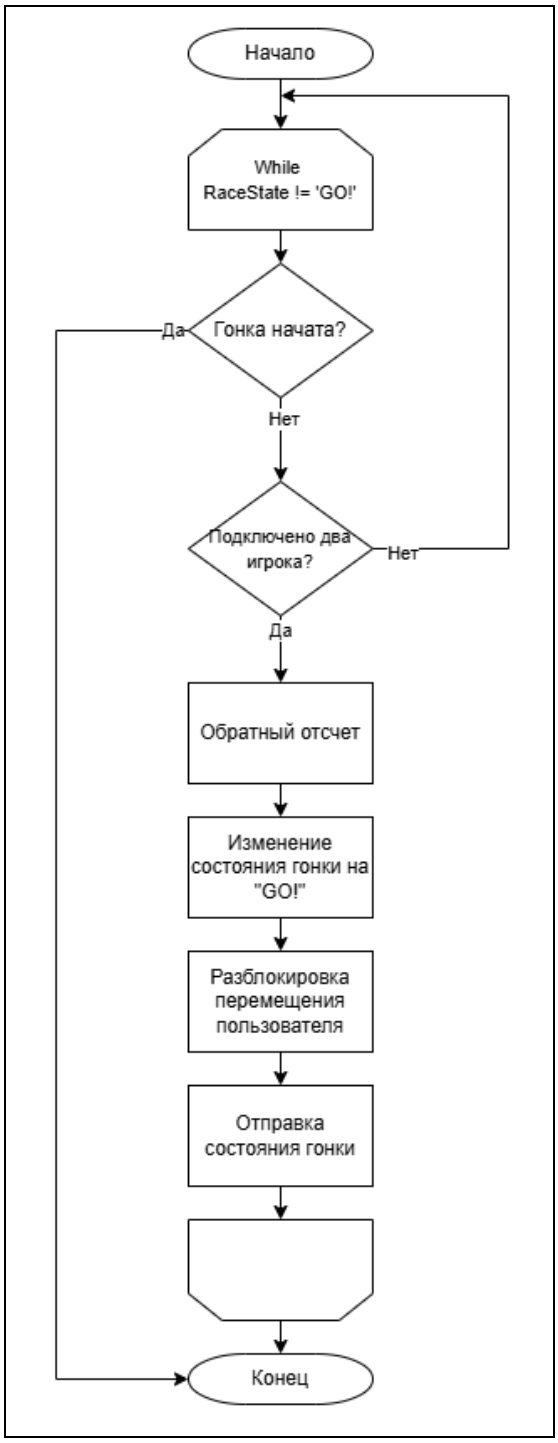

Рисунок 2.7 – Блок-схема алгоритма начала гонки

Алгоритм обработки финиша проверяет, пересек ли пользователь финишную черту. Для этого на черте размещен триггерный коллайдер. Когда игрок его касается, вызывается метод OnPlayerCrossedFinishLine(), в котором проверяется, достаточное ли количество кругов проехал игрок, который

пересек черту, если количество достаточное, то флаг raceFinished = true и передается сообщение с идентификатором игрока для вывода имени победителя на холст.

Алгоритм представлен на рисунке 2.8 в виде блок-схемы.

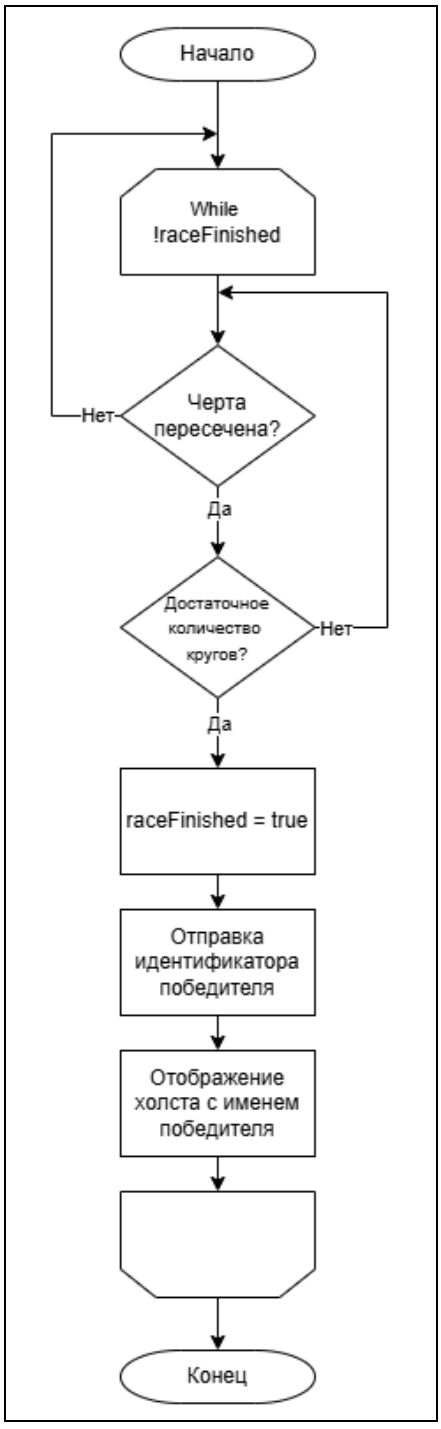

Рисунок 2.8 – Блок-схема алгоритма обработки финиша

Полный алгоритм работы программы представлен в виде блок-схемы на чертеже ГУИР.502520.007 ПД.

3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

3.1 Сетевое взаимодействие

Взаимодействие между клиентами и сервером построено по классической клиент-серверной архитектуре с использованием протокола RUDP. Основой коммуникации является обмен строго типизированными сообщениями, где сервер обрабатывает все игровые события.

Серверная часть реализует основной игровой цикл, который выполняется с фиксированной частотой 20 раз в секунду. В каждом цикле сервер последовательно обрабатывает: получение входных данных от всех подключенных клиентов, валидацию этих данных на предмет возможных нарушений, расчет нового состояния игроков и рассылку обновлений всем подключенным клиентам.

Клиентская часть отвечает за сбор пользовательского ввода, его отправку на сервер, получение и применение обновлений игрового состояния.

Взаимодействие между клиентами и сервером организовано через систему строго типизированных сообщений, где ключевую роль играет атрибут [MessageHandler], обеспечивающий автоматическую маршрутизацию сетевых команд. Этот механизм позволяет связать конкретные типы сообщений с соответствующими методами обработки, создавая четкую и расширяемую структуру сетевого взаимодействия. Когда сервер или клиент получают пакет данных, система автоматически определяет, какой метод должен обработать это сообщение, основываясь на его идентификаторе, указанном в атрибуте. Особенностью реализации является поддержка различных режимов доставки: надежные сообщения для критически важных событий вроде финиша гонки используют механизм подтверждения получения, тогда как частые обновления позиций передаются по ненадежному каналу для минимизации задержек.

3.2 Игровой цикл и синхронизация

Игровой цикл представляет собой четко структурированный процесс, обеспечивающий синхронизированное выполнение всех событий гонки на сервере и клиентах. Центральным элементом системы является сервер, который управляет состоянием заезда через последовательность взаимосвязанных фаз. Процесс старта гонки инициируется после проверки готовности всех участников и включает синхронизированный обратный отсчет, в течение которого сервер последовательно рассылает клиентам сообщения с текущим временем до начала соревнования. Каждое такое сообщение содержит временные метки, позволяющие клиентам точно синхронизировать локальные таймеры, компенсируя возможные сетевые задержки. В момент старта сервер переводит всех участников в активное

состояние, разблокирует управление транспортными средствами и начинает фиксацию времени прохождения кругов.

Механизм подсчета кругов реализован через систему триггерных зон, размещенных вдоль трассы, с обязательным прохождением финишной линии для регистрации завершенного круга. Сервер постоянно отслеживает позиции всех игроков, проверяя факт пересечения финишной черты.

Определение победителя происходит при достижении первым из участников установленного количества кругов, после чего сервер инициирует процедуру завершения гонки. Информация о победителе рассылается всем клиентам в специальном сообщении. Параллельно активируется визуальный эффект, отмечающий завершение соревнования.

Система возрождения игроков обеспечивает восстановление транспортных средств после столкновения с бортом. Процедура возрождения включает сброс физических параметров транспортного средства, восстановление начальной ориентации и позиции. Клиенты получают уведомление о возрождении с указанием новой позиции, что позволяет плавно интегрировать игрока обратно в гонку без резких переходов.

3.3 Физика и управление транспортными средствами

Физика и управление транспортными средствами в проекте реализованы через систему компонентов на основе Rigidbody, что обеспечивает реалистичное поведение космических кораблей при сохранении высокой производительности. В классе PlayerMovement используется физический движок Unity для обработки ускорения, поворотов и столкновений. Основная тяга создается через AddForce в направлении transform.forward с учетом ввода игрока, при этом удерживание LeftShift активирует множитель ускорения sprintMultiplier. Поворот осуществляется через AddTorque к Rigidbody, что создает плавное вращение вокруг оси Y. Для эффекта наклона при поворотах используется камера-прокси (camProxy), которая плавно изменяет свой угол через Quaternion.Lerp в зависимости от ввода по горизонтали (A/D). Важной особенностью является ограничение максимальной скорости через rb.velocity.normalized * maxSpeed, что предотвращает неконтролируемое ускорение.

Обработка пользовательского ввода происходит в PlayerController, где массив bool[] inputs фиксирует состояние клавиш WASD и Shift. Эти данные упаковываются в сообщение и отправляются на сервер с частотой через ненадежные RUDP-пакеты для минимизации задержки. На серверной стороне PlayerMovement.SetInput применяет полученные данные к физической модели, обеспечивая детерминированное поведение на всех клиентах. Особое внимание уделено обработке столкновений - при обнаружении коллайдера с тегом "TrackObstacle" через OnCollisionEnter вызывается RaceManager.RespawnPlayer, который телепортирует игрока на стартовую

позицию и синхронизирует это состояние через надежное сообщение PlayerRespawned.

Синхронизация физики между клиентом и сервером реализована через двусторонний обмен сообщениями. Сервер регулярно рассылает текущие позиции, вращения и векторы скорости всех игроков через ненадежные сообщения. Клиенты применяют эти данные для плавной интерполяции. Для локального игрока используется гибридный подход - его ввод сначала отправляется на сервер, а затем применяется после получения подтвержденного состояния, что исключает рассинхронизацию. При возрождении сервер принудительно устанавливает нулевые значения velocity и angularVelocity через надежные сообщения, гарантируя идентичное состояние на всех клиентах. Такая архитектура обеспечивает баланс между отзывчивостью управления и точной синхронизацией игрового состояния.

3.4 Пользовательский интерфейс

Пользовательский интерфейс в гоночной игре построен вокруг класса UIManager, который динамически управляет отображением различных экранов в зависимости от состояния игры. Основной интерфейс включает три ключевых компонента: экран подключения с полем ввода имени, игровой HUD с информацией о гонке и экран завершения заезда. При старте игры активируется меню подключения, где игрок вводит свое имя перед соединением с сервером. После успешного подключения UIManager автоматически переключается на игровой интерфейс.

Экран финиша активируется при завершении гонки и демонстрирует имя победителя, полученное из данных сервера. Для создания драматического эффекта используется отдельная камера finishCamera, которая плавно перехватывает управление от основной игровой камеры через корутину SwitchCameraCoroutine.

4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

В процессе тестирования игры были выявлены и последовательно устранены различные критические ошибки как на стороне клиента, так и на сервере, что значительно улучшило стабильность и пользовательский опыт.

Одной из первых решенных проблем стала корректная обработка ситуаций с недоступностью сервера - теперь при попытке подключения к несуществующему серверу или при вводе некорректного адреса игра не завершается аварийно, а плавно возвращает игрока в меню ввода имени с соответствующим логом, что предотвращает любые неожиданные вылеты.

Особое внимание было уделено физике столкновений – в первоначальной версии при контакте с ограждениями трассы транспортные средства могли проходить сквозь них или беспорядочно телепортироваться. Для решения этой проблемы была полностью переработана система коллизий: все элементы трассы получили оптимизированные Mesh Collider, а к игровым объектам добавлены Rigidbody с реалистичными параметрами массы и сопротивления. Дополнительно реализован механизм респавна – при вылете за пределы трассы или застревании игрок автоматически возвращается на стартовую позицию, что сохраняет динамику гонки без необходимости перезапуска.

В ходе тестирования была обнаружена критическая ошибка синхронизации игрового процесса, проявлявшаяся в некорректном определении локального игрока. Суть проблемы заключалась в том, что при старте гонки все подключенные клиенты получали управление одним и тем же игровым объектом, в то время как другие машины оставались неподвижными. Это происходило из-за отсутствия четкого разделения между локальными и удаленными экземплярами игроков на клиентской стороне.

Для решения этой проблемы была разработана и внедрена система идентификации локального игрока с использованием флага isLocal. Теперь при инициализации игрового объекта происходит проверка: если экземпляр принадлежит текущему клиенту, активируется система ввода и физика, в противном случае – объект управляется только данными, полученными от сервера.

Большинство проблем возникало на этапе проектирования и были устранены в результате ручного тестирования. Финальная версия программного средства обеспечивает устойчивую работу, обработку исключений и корректное взаимодействие пользователя с игрой.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

При запуске игры перед пользователем открывается интерфейс подключения, изображенный на рисунке 5.1, где можно ввести своё имя перед началом гонки. Если имя не указано, система автоматически присвоит игроку имя "Guest". Для подключения к игровой сессии необходимо нажать кнопку "Подключиться".

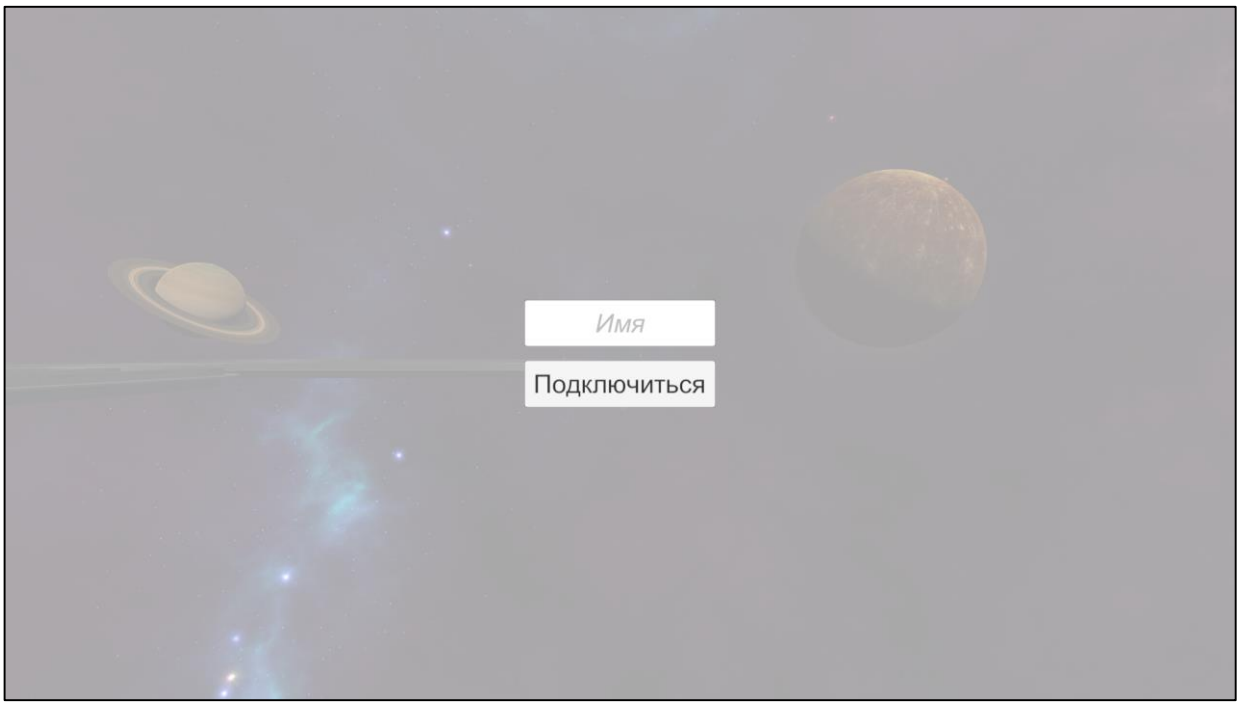

Рисунок 5.1 – Холст подключения к игре

На рисунке 5.2 представлено изображение, которое пользователь видит после подключении к гонке.

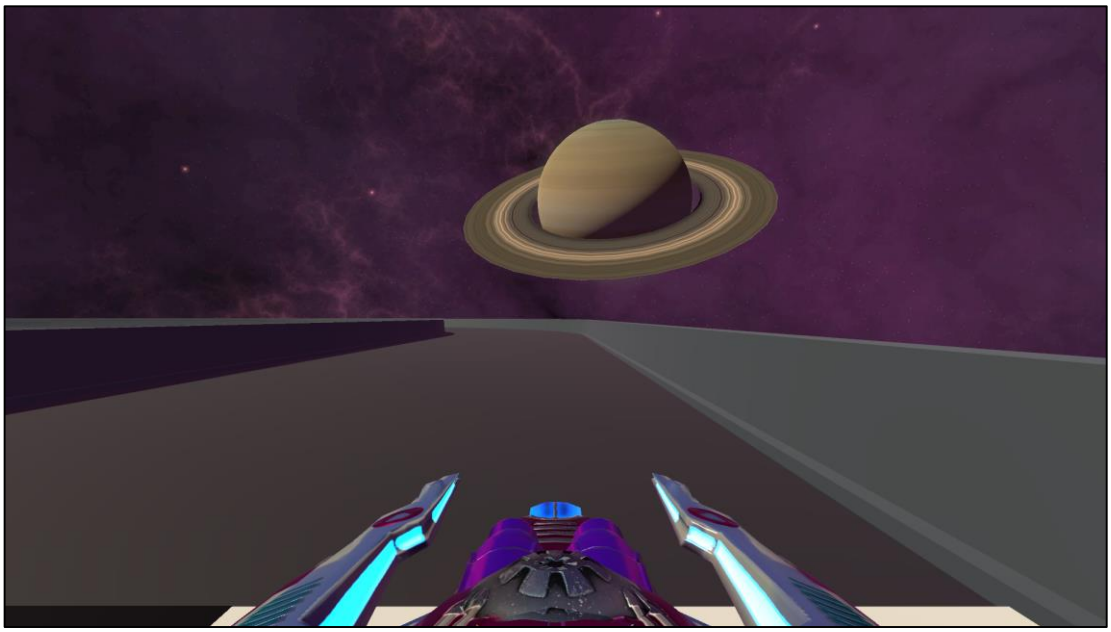

Рисунок 5.2 – Ожидание начала гонки

Перед стартом гонки система ожидает подключения как минимум двух игроков. В этот период управление транспортным средством заблокировано, что предотвращает преждевременное начало движения. Как только второй участник присоединяется к игре, появляется возможность управления машиной с помощью классической схемы: клавиши WASD отвечают за движение и повороты, а LeftShift активирует временное ускорение.

На рисунке 5.3 изображен произвольный момент гонки.

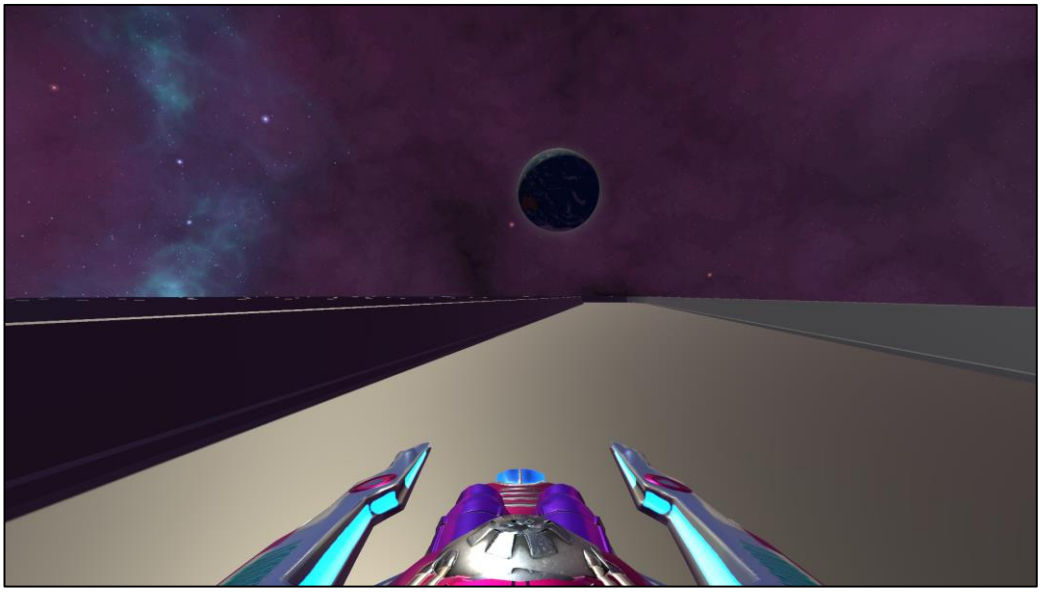

Рисунок 5.3 – Произвольный момент гонки

В процессе гонки камера следует за транспортным средством игрока, предоставляя оптимальный обзор трассы и окружающей обстановки. Физика

движения реализована с учётом реалистичного поведения автомобиля, включая особенности управления при заносах и столкновениях. Система автоматического возрождения возвращает игрока на трассу в случае вылета за пределы гоночной зоны, минимизируя простои и сохраняя динамику соревнования.

После пересечения финишной черты первым игроком, у остальных участников гонки отобразится холст с именем победителя, представленный на рисунке 5.4.

Рисунок 5.4 – Холост с отображением имени победителя

ЗАКЛЮЧЕНИЕ

В современном мире многопользовательские онлайн-игры занимают важное место в индустрии развлечений, предоставляя игрокам возможность соревноваться и взаимодействовать в реальном времени. В рамках данного проекта была разработана сетевая гоночная игра с использованием игрового движка Unity и технологии RUDP для передачи данных.

Разработанное приложение предоставляет пользователям полноценный гоночный симулятор с возможностью онлайн-соревнований. Были успешно реализованы все основные функции, включая систему подключения игроков, синхронизацию движения транспортных средств в реальном времени, физику столкновений и систему определения победителя. Особое внимание было уделено созданию интерфейса, позволяющего легко подключиться к игре и начать соревнование.

Ключевые особенности реализованного решения включают:

– систему лобби с ожиданием подключения второго игрока;

– реалистичную физику движения и столкновений на основе Rigidbody;

– синхронизацию позиций всех участников через RUDP-протокол;

– механизмы возрождения при вылете с трассы;

– визуализацию результатов гонки с определением победителя.

Среди ограничений текущей версии можно отметить:

– отсутствие системы учетных записей и статистики;

– ограниченный набор трасс и машин;

– базовую систему сопряжения без возможности выбора конкретного соперника.

Перспективы развития проекта включают:

– добавление системы регистрации и личного кабинета;

– расширение количества одновременных участников гонки;

– реализацию магазина с возможностью кастомизации машин;

– разработку редактора трасс;

– введение системы рейтингов и достижений.

Проведенное тестирование подтвердило стабильность работы приложения в различных сетевых условиях. Игра демонстрирует плавную синхронизацию движения, а система возрождения корректно обрабатывает различные сценарии столкновений и вылетов с трассы.

Таким образом, разработанное решение представляет собой полноценную сетевую гоночную игру, которая может служить основой для создания более сложного игрового продукта с расширенным функционалом и улучшенной графикой. Проект наглядно демонстрирует возможности Unity для разработки многопользовательских онлайн-игр и практическое применение технологий сетевой синхронизации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] ProHoster [Электронный ресурс]. – Решим доступа: https://prohoster.info/blog/administrirovanie/realizacziya-reliable-udp-protokola-dlya-net.

[2] Unity [Электронный ресурс]. – Решим доступа: https://unity.com/ru.

[3] Fandom [Электронный ресурс]. – Решим доступа: https://wipeout.fandom.com/wiki/Wipeout_(series).

[4] Fandom [Электронный ресурс]. – Решим доступа: https://starwars.fandom.com/wiki/Star_Wars:_Episode_I_Racer.

[5] Fandom [Электронный ресурс]. – Решим доступа: https://redout.games/redout/.

[6] Unity Documentation [Электронный ресурс]. – Решим доступа: https://docs.unity3d.com/ru/530/Manual/AssetStore.

[7] Федоров, А. С. Программирование на C#: практическое руководство / А. С. Федоров. – М.: БХВ-Петербург, 2022. – 400 с.

[8] Ричардсон, Д. Создание игр на Unity: от идеи до реализации / Д. Ричардсон. – М.: Вильямс, 2021. – 320 с.

[9] Хейл, Дж. Физика движения и столкновений в Unity / Дж. Хейл. – СПб.: Питер, 2020. – 350 с.

[10] Паттерсон, К. Разработка 3D-игр на Unity: полное руководство / К. Паттерсон. – М.: Вильямс, 2021. – 400 с.

Приложение А
(обязательное)
Исходный код программы

Client:

```
public enum ServerToClientId : ushort
{
    playerSpawned = 1,
    playerMovement,
    PlayerRespawned,
    RaceFinished,
    RaceStart,
    Countdown,
    CountdownUpdate,
    RaceStateUpdate,
}

public enum ClientToServerId : ushort
{
    name = 1,
    input,
}

public class NetworkManager : MonoBehaviour
{
    private static NetworkManager _singleton;

    public static NetworkManager Singleton
    {
        get => _singleton;
        private set
        {
            if (_singleton == null)
                _singleton = value;
            else if (_singleton != value)
            {
                Debug.Log($"{nameof(NetworkManager)}: instance already exists,
destroing duplicate!");
                Destroy(value);
            }
        }
    }

    public Client Client { get; private set; }

    [SerializeField] private string ip;
    [SerializeField] private ushort port;

    private void Awake()
    {
        Singleton = this;
    }

    private void Start()
    {
        Client = new Client();

        Client.Connected += DidConnect;
        Client.ConnectionFailed += FailedToConnect;
        Client.Disconnected += DidDisconnect;
    }
```

```
    private void FixedUpdate()
    {
        Client.Tick();
    }

    private void OnApplicationQuit()
    {
        Client.Disconnect();
    }

    public void Connect()
    {
        Client.Connect($"{ip}:{port}");
    }

    private void DidConnect(object sender, EventArgs e)
    {
        UIManager.Singleton.SendName();
    }

    private void FailedToConnect(object sender, EventArgs e)
    {
        UIManager.Singleton.BackToMain();
    }

    private void PlayerLeft(object sender, ClientDisconnectedEventArgs e)
    {
        if (Player.list.TryGetValue(e.Id, out Player player))
            Destroy(player.gameObject);
    }

    private void DidDisconnect(object sender, EventArgs e)
    {
        UIManager.Singleton.BackToMain();
        foreach (Player player in Player.list.Values)
        {
            Destroy(player.gameObject);
        }
    }
}

public class UIManager : MonoBehaviour
{
    private static UIManager _singleton;

    public static UIManager Singleton
    {
        get => _singleton;
        private set
        {
            if (_singleton == null)
                _singleton = value;
            else if (_singleton != value)
            {
                Debug.Log($"{nameof(UIManager)}: instance already exists, destroing
duplicate!");
                Destroy(value);
            }
        }
    }

    private void Awake()
    {
        Singleton = this;
```

```
        connectUI.SetActive(true);
        finishUI.SetActive(false);
        raceUI.SetActive(false);
    }



    public void ConnectClicked()
    {
        usernameField.interactable = false;
        connectUI.SetActive(false);

        NetworkManager.Singleton.Connect();
    }

    public void SendName()
    {
        Message message = Message.Create(MessageSendMode.reliable,
(ushort)ClientToServerId.name);
        message.AddString(usernameField.text);
        NetworkManager.Singleton.Client.Send(message);
    }

    public void ShowWinner(string winnerName)
    {
        StartCoroutine(SwitchCameraCoroutine());

        connectUI.SetActive(false);
        raceUI.SetActive(false);

        winnerText.text = $"{winnerName} wins the race!";
        finishUI.SetActive(true);
        finishUI.transform.SetAsLastSibling();
    }

    private IEnumerator SwitchCameraCoroutine()
    {
        finishCamera.gameObject.SetActive(true);

        float elapsed = 0f;
        while (elapsed < 5f)
        {
            elapsed += Time.deltaTime;
            yield return null;
        }
    }

    public void UpdateCountdown(int seconds)
    {
        countdownText.text = seconds.ToString();
        countdownText.gameObject.SetActive(true);

        if (seconds <= 0)
            countdownText.gameObject.SetActive(false);
    }


    [MessageHandler((ushort)ServerToClientId.RaceStateUpdate)]
    private static void HandleRaceState(Message message)
    {
        string state = message.GetString();
        UIManager.Singleton.ShowRaceMessage(state);

        if (state == "GO!")
        {
```

```
            var localPlayer = Player.list.Values.FirstOrDefault(p => p.IsLocal);
            if (localPlayer != null)
            {
                var controller = localPlayer.GetComponent<PlayerController>();
                if (controller != null)
                {
                    controller.EnableInput();
                    Debug.Log("Movement enabled for local player");
                }
            }
        }
    }
}

public class Player : MonoBehaviour
{
    public static Dictionary<ushort, Player> list = new Dictionary<ushort,
Player>();

    public ushort Id { get; private set; }
    public bool IsLocal { get; private set; }
    private string username;
    [SerializeField] private Transform camTransform;

    private void Move(Vector3 newPosition, Vector3 forward)
    {
        Rigidbody rb = GetComponent<Rigidbody>();
        if (rb != null)
        {
            rb.MovePosition(newPosition);
            rb.MoveRotation(Quaternion.LookRotation(forward));
        }
        else
        {
            transform.position = newPosition;
            transform.forward = forward;
        }

        if (!IsLocal)
        {
            camTransform.forward = forward;
        }
    }

    private void OnDestroy()
    {
        list.Remove(Id);
    }

    public static void Spawn(ushort id, string username, Vector3 position)
    {
        Player player;

        if (id == NetworkManager.Singleton.Client.Id)
        {
            player = Instantiate(GameLogic.Singleton.LocalPlayerPrefab, position,
Quaternion.identity).GetComponent<Player>();
            player.IsLocal = true;
        }
        else
        {
            player = Instantiate(GameLogic.Singleton.PlayerPrefab, position,
Quaternion.identity).GetComponent<Player>();
            player.IsLocal = false;
        }
```

```
        player.transform.rotation = Quaternion.Euler(0f, -90f, 0f);


        player.name = $"PLayer {id} ({(string.IsNullOrEmpty(username) ? "Guest" :
username)})";

        player.Id = id;
        player.username = username;

        list.Add(id, player);

    }

    #region Messages
    [MessageHandler((ushort)ServerToClientId.playerSpawned)]
    private static void SpawnPlayer(Message message)
    {
        ushort id = message.GetUShort();
        string username = message.GetString();
        Vector3 position = message.GetVector3();

        Spawn(id, username, position);
    }

    [MessageHandler((ushort)ServerToClientId.playerMovement)]
    private static void PlayerMovement(Message message)
    {
        ushort id = message.GetUShort();

        if (list.TryGetValue(id, out Player player))
        {
            Vector3 position = message.GetVector3();
            Vector3 forward = message.GetVector3();

            Rigidbody rb = player.GetComponent<Rigidbody>();
            if (rb != null)
            {
                rb.MovePosition(position);
                rb.MoveRotation(Quaternion.LookRotation(forward));
                rb.velocity = message.GetVector3();
                rb.angularVelocity = message.GetVector3();
            }
            else
            {
                player.transform.position = position;
                player.transform.forward = forward;
            }
        }
        else
        {
            Debug.LogWarning($"Player {id} not found in local list");
        }
    }

    [MessageHandler((ushort)ServerToClientId.PlayerRespawned)]
    private static void HandleRespawn(Message message)
    {
        ushort playerId = message.GetUShort();
        Vector3 position = message.GetVector3();
        Quaternion rotation = message.GetQuaternion();

        if (Player.list.TryGetValue(playerId, out Player player))
        {
            player.transform.SetPositionAndRotation(position, rotation);
```

```
            Rigidbody rb = player.GetComponent<Rigidbody>();
            if (rb != null)
            {
                rb.velocity = Vector3.zero;
                rb.angularVelocity = Vector3.zero;
            }
        }
    }

    [MessageHandler((ushort)ServerToClientId.RaceFinished)]
    private static void HandleRaceFinish(Message message)
    {
        ushort winnerId = message.GetUShort();
        string winnerName = list[winnerId].username;

        Debug.Log($"Race finished! Winner: {winnerName}");
        UIManager.Singleton.ShowWinner(winnerName);
    }

    [MessageHandler((ushort)ServerToClientId.CountdownUpdate)]
    private static void HandleCountdown(Message message)
    {
        int secondsLeft = message.GetInt();
        UIManager.Singleton.UpdateCountdown(secondsLeft);
    }

    #endregion
}

public class PlayerController : MonoBehaviour
{
    [SerializeField] private Transform camTransform;
    private bool canInput = false;


    private bool[] inputs;

    private void Start()
    {
        inputs = new bool[5];
    }

    private void Update()
    {
        inputs[0] = Input.GetKey(KeyCode.W);
        inputs[1] = Input.GetKey(KeyCode.S);
        inputs[2] = Input.GetKey(KeyCode.A);
        inputs[3] = Input.GetKey(KeyCode.D);
        inputs[4] = Input.GetKey(KeyCode.LeftShift);
    }

    private void FixedUpdate()
    {
        if (!canInput)
            return;

        bool hasInput = inputs.Any(i => i);
        if (hasInput)
            SendInput();
    }

    public void EnableInput()
    {
        canInput = true;
```

```
    }

    #region Messages

    private void SendInput()
    {
        Message message = Message.Create(MessageSendMode.unreliable,
(ushort)ClientToServerId.input);
        message.AddBools(inputs, false);
        message.AddVector3(camTransform.forward);
        NetworkManager.Singleton.Client.Send(message);
    }

    #endregion
}
```

Server:

```
public enum ServerToClientId : ushort
{
    playerSpawned = 1,
    playerMovement,
    PlayerRespawned,
    RaceFinished,
    RaceStart,
    Countdown,
    CountdownUpdate,
    RaceStateUpdate,
}

public enum ClientToServerId : ushort
{
    name = 1,
    input,
}

public class NetworkManager : MonoBehaviour
{
    private static NetworkManager _singleton;

    public static NetworkManager Singleton
    {
        get => _singleton;
        private set
        {
            if (_singleton== null)
                _singleton = value;
            else if (_singleton != value)
            {
                Debug.Log($"{nameof(NetworkManager)}: instance already exists,
destroing duplicate!");
                Destroy(value);
            }
        }
    }

    public Server Server { get; private set; }

    [SerializeField] private ushort port;
    [SerializeField] private ushort maxClientCount;

    private void Awake()
    {
        Singleton = this;
```

```
    }

    private void Start()
    {

        Server = new Server();
        Server.Start(port, maxClientCount);

        Server.ClientDisconnected += PlayerLeft;
    }

    private void FixedUpdate()
    {
        Server.Tick();
    }

    private void OnApplicationQuit()
    {
        Server.Stop();
    }

    private void PlayerLeft(object sender, ClientDisconnectedEventArgs e)
    {
        if (Player.list.TryGetValue(e.Id, out Player player))
            Destroy(player.gameObject);
    }
}

public class Player : MonoBehaviour
{
    public static Dictionary<ushort, Player> list = new Dictionary<ushort,
Player>();

    public ushort Id { get; private set; }
    public string Username {  get; private set; }

    public PlayerMovement Movement => movement;

    [SerializeField] private PlayerMovement movement;


    private void OnDestroy()
    {
        list.Remove(Id);
    }

    public static void Spawn(ushort id, string username)
    {
        foreach (Player otherPlayer in list.Values)
            otherPlayer.SpawnedMessage(id);
        Player player = Instantiate(GameLogic.Singleton.PlayerPrefab, new
Vector3(0f, 1f, 0f), Quaternion.identity).GetComponent<Player>();
        player.name = $"Player {id} ({(string.IsNullOrEmpty(username) ? "Guest" :
username)})";
        player.Id = id;
        player.Username = string.IsNullOrEmpty(username) ? "Guest" : username;

        if (player.Id % 2 == 0)
            player.transform.position = new Vector3(-40f, -0.7f, 138f);
        else
            player.transform.position = new Vector3(-40f, -0.7f, 132f);

        player.transform.rotation = Quaternion.Euler(0f, -90f, 0f);

        Rigidbody rb = player.GetComponent<Rigidbody>();
```

```
        if (rb == null) rb = player.gameObject.AddComponent<Rigidbody>();
        rb.isKinematic = false;
        rb.interpolation = RigidbodyInterpolation.Interpolate;

        list.Add(id, player);
        player.SpawnedMessage();
        RaceManager.Singleton?.OnPlayerConnect(player);
    }

    #region Messages
    private void SpawnedMessage()
    {

NetworkManager.Singleton.Server.SendToAll(AddSpawnData(Message.Create(MessageSendMod
e.reliable, (ushort)ServerToClientId.playerSpawned)));
    }

    private void SpawnedMessage(ushort toClientId)
    {

NetworkManager.Singleton.Server.Send(AddSpawnData(Message.Create(MessageSendMode.rel
iable, (ushort)ServerToClientId.playerSpawned)), toClientId);
    }

    private Message AddSpawnData(Message message)
    {
        message.AddUShort(Id);
        message.AddString(Username);
        message.AddVector3(transform.position);
        return message;
    }

    [MessageHandler((ushort)ClientToServerId.name)]
    private static void Name(ushort fromClientId, Message message)
    {
        Spawn(fromClientId, message.GetString());
    }

    [MessageHandler((ushort)ClientToServerId.input)]
    private static void Input(ushort fromClientId, Message message)
    {
        if (list.TryGetValue(fromClientId, out Player player))
            player.Movement.SetInput(message.GetBools(5), message.GetVector3());
    }

    [MessageHandler((ushort)ServerToClientId.playerMovement)]
    private static void HandlePlayerMovement(Message message)
    {
        ushort id = message.GetUShort();
        if (Player.list.TryGetValue(id, out Player player))
        {
            Vector3 position = message.GetVector3();
            Vector3 forward = message.GetVector3();

            Rigidbody rb = player.GetComponent<Rigidbody>();
            if (rb != null)
            {
                rb.MovePosition(position);
                rb.MoveRotation(Quaternion.LookRotation(forward));
            }
            else
            {
                player.transform.position = position;
                player.transform.forward = forward;
            }
```

```
        }
    }

    #endregion
}

[RequireComponent(typeof(Rigidbody))]
public class PlayerMovement : MonoBehaviour
{
    private float currentTilt;
    private bool[] inputs;
    private Vector3 physicsVelocity;
    private bool canMove = false;

    private void Start()
    {
        inputs = new bool[5];
        rb.maxAngularVelocity = 5f;
    }

    private void FixedUpdate()
    {
        if (!enabled) return;
        Vector2 inputDirection = GetInputDirection();
        ProcessMovement(inputDirection);
        ProcessRotation(inputDirection);
        ApplyTiltEffect(inputDirection);
        LimitVelocity();
        SendMovement();
    }

    public void EnableMovement()
    {
        canMove = true;
    }

    private Vector2 GetInputDirection()
    {
        Vector2 direction = Vector2.zero;
        if (inputs[0]) direction.y += 1;
        if (inputs[1]) direction.y -= 1;
        if (inputs[2]) direction.x -= 1;
        if (inputs[3]) direction.x += 1;
        return direction;
    }

    private void ProcessMovement(Vector2 inputDirection)
    {
        float thrust = inputDirection.y * thrustForce;
        if (inputs[4]) thrust *= sprintMultiplier;

        Vector3 force = transform.forward * thrust;
        rb.AddForce(force, ForceMode.Acceleration);

        if (inputDirection.x != 0)
        {
            float driftForce = thrustForce * 0.3f * inputDirection.x;
            rb.AddForce(transform.right * driftForce, ForceMode.Acceleration);
        }
    }

    private void ProcessRotation(Vector2 inputDirection)
    {
        float rotationTorque = inputDirection.x * rotationSpeed;
        rb.AddTorque(transform.up * rotationTorque, ForceMode.Acceleration);
```

```
    }

    private void ApplyTiltEffect(Vector2 inputDirection)
    {
        float targetTiltZ = -inputDirection.x * tiltAngle;
        currentTilt = Mathf.Lerp(currentTilt, targetTiltZ, tiltSpeed *
Time.fixedDeltaTime);

        float targetTiltX = -inputDirection.y * tiltAngle * 0.5f;

        camProxy.localRotation = Quaternion.Lerp(
            camProxy.localRotation,
            Quaternion.Euler(targetTiltX, 0, currentTilt),
            tiltSpeed * Time.fixedDeltaTime
        );
    }

    private void LimitVelocity()
    {
        if (rb.velocity.magnitude > maxSpeed)
        {
            rb.velocity = rb.velocity.normalized * maxSpeed;
        }
    }

    public void SetInput(bool[] inputs, Vector3 forward)
    {
        this.inputs = inputs;
        camProxy.forward = forward;
    }

    private void SendMovement()
    {
        Message message = Message.Create(MessageSendMode.unreliable,
(ushort)ServerToClientId.playerMovement);
        message.AddUShort(player.Id);
        message.AddVector3(transform.position);
        message.AddVector3(transform.forward);
        message.AddVector3(rb.velocity);
        message.AddVector3(rb.angularVelocity);
        NetworkManager.Singleton.Server.SendToAll(message);
    }

    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.CompareTag("TrackObstacle"))
        {
            Debug.Log($"[Collision] Player {player.Id} collided with
{collision.gameObject.name} at position: {transform.position}");
            RaceManager.Singleton.RespawnPlayer(player);
        }
    }
}

public class RaceManager : MonoBehaviour
{
    private static RaceManager _singleton;
    public static RaceManager Singleton
    {
        get => _singleton;
        private set
        {
            if (_singleton == null)
                _singleton = value;
            else if (_singleton != value)
```

```
            {
                Debug.Log($"{nameof(RaceManager)}: instance already exists,
destroying duplicate!");
                Destroy(value);
            }
        }
    }

    private Dictionary<ushort, int> playerLaps = new Dictionary<ushort, int>();
    private int totalLaps = 1;
    private bool raceFinished;
    private ushort winnerId;

    private int requiredPlayers = 2;
    private float countdownTime = 5f;

    private bool raceStarted;
    private Coroutine countdownCoroutine;
    private readonly HashSet<ushort> readyPlayers = new HashSet<ushort>();

    private void Awake()
    {
        Singleton = this;
    }

    public void RespawnPlayer(Player player)
    {
        Vector3 respawnPosition = player.Id % 2 == 0
            ? new Vector3(-40f, -0.7f, 138f)
            : new Vector3(-40f, -0.7f, 132f);

        Quaternion respawnRotation = Quaternion.Euler(0f, -90f, 0f);

        player.transform.SetPositionAndRotation(respawnPosition, respawnRotation);

        Rigidbody rb = player.GetComponent<Rigidbody>();
        if (rb != null)
        {
            rb.velocity = Vector3.zero;
            rb.angularVelocity = Vector3.zero;
        }

        Message message = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.PlayerRespawned);
        message.AddUShort(player.Id);
        message.AddVector3(respawnPosition);
        message.AddQuaternion(respawnRotation);
        NetworkManager.Singleton.Server.SendToAll(message);
    }

    public void OnPlayerCrossedFinishLine(Player player)
    {

        if (!playerLaps.ContainsKey(player.Id))
        {
            playerLaps[player.Id] = 0;
        }

        playerLaps[player.Id]++;
        Debug.Log($"Player {player.Id} ({player.Username}) completed lap
{playerLaps[player.Id]}/{totalLaps}");

        if (playerLaps[player.Id] >= totalLaps)
        {
            raceFinished = true;
```

```
            winnerId = player.Id;
            string winnerName = Player.list[winnerId].Username;
            Debug.Log($"Player {winnerId} ({winnerName}) won the race!");
            SendRaceFinished(winnerId);
        }
    }

    public void OnPlayerConnect(Player player)
    {
        if (raceStarted) return;

        readyPlayers.Add(player.Id);
        Debug.Log($"Player {player.Id} connected. Ready:
{readyPlayers.Count}/{requiredPlayers}");

        if (readyPlayers.Count >= requiredPlayers)
        {
            StartRaceCountdown();
        }
    }

    private void StartRaceCountdown()
    {
        if (countdownCoroutine != null)
            StopCoroutine(countdownCoroutine);

        countdownCoroutine = StartCoroutine(CountdownRoutine());
    }

    private IEnumerator CountdownRoutine()
    {
        foreach (var player in Player.list.Values)
        {
            player.Movement.enabled = false;
        }

        SendRaceStateUpdate("WAITING");
        yield return new WaitForSeconds(1f);

        for (int i = 3; i > 0; i--)
        {
            SendCountdownUpdate(i);
            SendRaceStateUpdate($"STARTING IN {i}");
            yield return new WaitForSeconds(1f);
        }

        SendRaceStateUpdate("GO!");
        raceStarted = true;

        foreach (var player in Player.list.Values)
        {
            player.Movement.enabled = true;
        }

        Debug.Log("Race started! Players can move now");
    }

    private void StartRace()
    {
        SendRaceStateUpdate("GO!");

        foreach (var player in Player.list.Values)
        {
            player.Movement.EnableMovement();
        }
```

```
    }

    private void SendCountdownUpdate(int secondsLeft)
    {
        Message msg = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.CountdownUpdate);
        msg.AddInt(secondsLeft);
        NetworkManager.Singleton.Server.SendToAll(msg);
    }

    private void SendRaceStateUpdate(string message)
    {
        Message msg = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.RaceStateUpdate);
        msg.AddString(message);
        NetworkManager.Singleton.Server.SendToAll(msg);
    }

    private void SendRaceFinished(ushort winnerId)
    {
        Message msg = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.RaceFinished);
        msg.AddUShort(winnerId);
        NetworkManager.Singleton.Server.SendToAll(msg);
    }
}
```

MyNetwork:

```
namespace MyNetwork
{
    public enum MessageSendMode : byte
    {
        Unreliable,
        Reliable
    }

    public class Message
    {
        public const int MaxSize = 1250;
        public MessageSendMode SendMode { get; private set; }
        public byte[] Data { get; private set; } = new byte[MaxSize];
        public int Position { get; set; }

        public Message(MessageSendMode mode)
        {
            SendMode = mode;
        }

        public void Add(byte value) => Data[Position++] = value;
        public byte GetByte() => Data[Position++];

        public void Add(string value)
        {
            byte[] bytes = Encoding.UTF8.GetBytes(value);
            Add((ushort)bytes.Length);
            foreach (byte b in bytes)
            {
                Add(b);
            }
        }

        public string GetString()
        {
            ushort length = GetUShort();
```

```
        byte[] bytes = new byte[length];
        for (int i = 0; i < length; i++)
        {
            bytes[i] = GetByte();
        }
        return Encoding.UTF8.GetString(bytes);
    }

    public void Add(ushort value)
    {
        Add((byte)(value >> 8));
        Add((byte)value);
    }

    public ushort GetUShort()
    {
        return (ushort)((GetByte() << 8) | GetByte());
    }
}

public class UdpConnection
{
    private UdpClient socket;
    private IPEndPoint remoteEndPoint;

    public UdpConnection(string ip, int port)
    {
        remoteEndPoint = new IPEndPoint(IPAddress.Parse(ip), port);
        socket = new UdpClient();
    }

    public void Send(Message message)
    {
        socket.Send(message.Data, message.Position, remoteEndPoint);
    }

    public Message Receive()
    {
        var data = socket.Receive(ref remoteEndPoint);
        return new Message(MessageSendMode.Unreliable);
    }
}

public class Client
{
    private UdpConnection connection;

    public void Connect(string ip, int port)
    {
        connection = new UdpConnection(ip, port);
    }

    public void Send(Message message)
    {
        connection.Send(message);
    }

    public Message Receive()
    {
        return connection.Receive();
    }
}

public class Server
{
```

```
        private UdpClient socket;
        private Dictionary<IPEndPoint, ClientInfo> clients = new
Dictionary<IPEndPoint, ClientInfo>();

        public void Start(int port)
        {
            socket = new UdpClient(port);
            new Thread(Listen).Start();
        }

        private void Listen()
        {
            while (true)
            {
                IPEndPoint clientEndPoint = null;
                var data = socket.Receive(ref clientEndPoint);
                var message = new Message(MessageSendMode.Unreliable);

                if (!clients.ContainsKey(clientEndPoint))
                    clients[clientEndPoint] = new ClientInfo(clientEndPoint);

                HandleMessage(message, clientEndPoint);
            }
        }

        private void HandleMessage(Message message, IPEndPoint client)
        {
            byte messageType = message.GetByte();

            switch (messageType)
            {
                case 0:
                    OnClientConnected(client);
                    break;

                case 1:
                    ushort messageId = message.GetUShort();
                    OnMessageReceived(client, messageId, message);
                    break;

                case 2:
                    OnClientDisconnected(client);
                    break;

                default:
                    Console.WriteLine($"Unknown message type: {messageType}");
                    break;
            }
        }

        private void OnClientConnected(IPEndPoint client)
        {
            var clientInfo = clients[client];
            Console.WriteLine($"Client connected: {client}, ID: {clientInfo.Id}");
            var response = new Message(MessageSendMode.Reliable);
            response.Add((byte)0);
            response.Add(clientInfo.Id);
            socket.Send(response.Data, response.Position, client);
        }

        private void OnMessageReceived(IPEndPoint client, ushort messageId, Message
message)
        {
            if (!clients.TryGetValue(client, out var clientInfo))
                return;
```

```
            clientInfo.UpdateActivity();

            Console.WriteLine($"Received message {messageId} from client
{clientInfo.Id}");
            switch (messageId)
            {
                case 1:
                    HandlePingMessage(clientInfo, message);
                    break;

                case 2:
                    HandleChatMessage(clientInfo, message);
                    break;
            }
        }

        private void HandlePingMessage(ClientInfo client, Message message)
        {
            var pong = new Message(MessageSendMode.Unreliable);
            pong.Add((byte)1);
            pong.Add((ushort)2);
            socket.Send(pong.Data, pong.Position, client.EndPoint);
        }

        private void HandleChatMessage(ClientInfo sender, Message message)
        {
            string text = message.GetString();
            Console.WriteLine($"Chat message from {sender.Id}: {text}");

            var broadcast = new Message(MessageSendMode.Reliable);
            broadcast.Add((byte)1);
            broadcast.Add((ushort)3);
            broadcast.Add(sender.Id);
            broadcast.Add(text);

            foreach (var client in clients.Values)
            {
                if (client.Id != sender.Id)
                {
                    socket.Send(broadcast.Data, broadcast.Position,
client.EndPoint);
                }
            }
        }

        private void OnClientDisconnected(IPEndPoint client)
        {
            if (clients.TryGetValue(client, out var clientInfo))
            {
                Console.WriteLine($"Client disconnected: {clientInfo.Id}");
                clients.Remove(client);
            }
        }

        public void SendToAll(Message message)
        {
            foreach (var client in clients.Keys)
            {
                socket.Send(message.Data, message.Position, client);
            }
        }
    }

    public class ClientInfo
```

```
    {
        public IPEndPoint EndPoint { get; }
        public string Name { get; set; }
        public DateTime LastActivity { get; set; }
        public ushort Id { get; }
        public short Ping { get; set; }

        private static ushort lastId = 0;

        public ClientInfo(IPEndPoint endPoint)
        {
            EndPoint = endPoint;
            LastActivity = DateTime.Now;
            Id = ++lastId;
        }

        public void UpdateActivity()
        {
            LastActivity = DateTime.Now;
        }

        public bool IsTimedOut(TimeSpan timeout)
        {
            return (DateTime.Now - LastActivity) > timeout;
        }
    }
}
```