ТИТУЛ 1

ТИТУЛ 2

ЗАДАНИЕ 1

ЗАДАНИЕ 2

КАЛЕНДАРНЫЙ ПЛАН

РЕФЕРАТ

Дипломная работа содержит:

- 100 страниц;
- 16 рисунков;
- 5 таблиц;
- 1 приложение.

Ключевые слова: клиент-серверная архитектура, Rust, Bevy, bevy_quinnet, QUIC, игровая сетевая логика, авторитарный сервер, синхронизация состояний, разработка игр, импортозамещение, отечественная разработка, RTS, стратегия в реальном времени.

В ходе выполнения дипломной работы был разработано клиент-серверное приложение многопользовательской сетевой игры «Боярский Турнир» на языке программирования Rust с использованием библиотеки Bevy.

Работа написана мною самостоятельно и не содержит неправомерных заимствований.

_____ _____ Хохлов Т. В

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

RTS (Real-Time Strategy) - жанр стратегических игр в реальном времени.

ECS (Entity-Component-System) - архитектурный паттерн, используемый в игровых движках для управления сущностями, их компонентами и системами.

PvP (Player Versus Player) - режим игры, в котором игроки сражаются друг против друга.

MOBA (Multiplayer Online Battle Arena) - жанр командных стратегических игр (например, Dota 2, League of Legends).

FPS (Frames Per Second) - количество кадров в секунду, показатель производительности игры.

MMR (Matchmaking rating) - рейтинг игрока, используемый для подбора соперников.

AOE (Area of Effect) - урон или эффект, действующий на область.

AI (Artificial Intelligence) - искусственный интеллект, управляющий поведением компьютерных противников.

UI (User Interface) - пользовательский интерфейс игры.

HUD (Heads-Up Display) - элементы интерфейса, отображаемые на экране во время игры.

P2P (Peer-To-Peer) - сетевая архитектура, где клиенты взаимодействуют напрямую.

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Локстеп (Lockstep) - сетевой алгоритм, при котором все клиенты выполняют команды синхронно, обеспечивая детерминированность.

Туман войны (Fog of War) - игровая механика, скрывающая часть карты, не исследованную игроком.

Геймплей (Gameplay) - совокупность игровых механик и взаимодействий, определяющих игровой процесс.

Спавн (Spawn) - процесс создания игрового объекта (юнита, здания) в мире игры.

Деспавн (Despawn) - удаление объекта из игрового мира.

Хитбокс (Hitbox) - невидимая область, определяющая зону столкновения объекта.

Роллбэк (Rollback) - механизм отката состояния игры при расхождении данных между клиентом и сервером.

Детерминированная симуляция - игровая логика, которая даёт одинаковый результат при одинаковых входных данных.

Крейт (crate) в языке Rust — это наименьший объём кода, который компилятор рассматривает за раз. Даже если компилируется один файл с исходным кодом, компилятор считает его крейтом

ВВЕДЕНИЕ

Современная индустрия компьютерных игр представляет собой динамично развивающуюся сферу цифровых технологий, которая оказывает значительное влияние на культурные, социальные и экономические аспекты современного общества. В условиях стремительной цифровизации различных областей человеческой деятельности компьютерные игры перестали быть просто развлекательным продуктом, превратившись в сложные программно-аппаратные комплексы, сочетающие в себе элементы искусства, технологии и науки. Особое место в этом многообразии занимает жанр стратегий реального времени (**RTS**), который на протяжении нескольких десятилетий остается одним из наиболее сложных и требовательных к технической реализации направлений игровой индустрии.

Актуальность разработки многопользовательской компьютерной игры в жанре **RTS** "Боярский турнир" обусловлена рядом значимых факторов. Во-первых, современный рынок игровой продукции демонстрирует устойчивый спрос на качественные стратегические игры, что подтверждается коммерческим успехом таких проектов, как Age of Empires IV, Company of Heroes 3 и других представителей жанра. Во-вторых, наблюдается постоянное технологическое развитие инструментария для создания игровых приложений, включая появление новых языков программирования и специализированных движков, что открывает перед разработчиками ранее недоступные возможности.

Техническая сложность разработки **RTS**-игр традиционно связана с необходимостью решения целого комплекса взаимосвязанных задач. Среди них можно выделить: реализацию сложного искусственного интеллекта для управления юнитами, создание эффективной системы сетевой синхронизации для многопользовательского режима, а также разработку сбалансированной экономической модели игры. Именно эти аспекты делают процесс создания

RTS-игры исключительно интересным с точки зрения исследования возможностей современных технологий игровой разработки.

Научная новизна исследования заключается в комплексном анализе возможностей языка Rust с уникальной системой владения и заимствования (ownership system), защищающей от целого класса ошибок, характерных для системного программирования и движка Bevy для разработки игр жанра **RTS**, что представляет особый интерес в свете относительно небольшого количества подобных реализаций на текущий момент. Практическая значимость работы определяется тем, что разработанный прототип может служить основой для дальнейшего развития в полноценный коммерческий продукт, а полученные в ходе разработки результаты могут быть полезны другим исследователям и разработчикам, работающим с аналогичными технологиями.

Методологическую основу исследования составляют современные подходы к разработке программного обеспечения, принципы объектно-ориентированного и системного программирования, а также специализированные методики, применяемые в игровой индустрии. В процессе работы использовались такие методы как прототипирование, модульное тестирование, профилирование производительности и итеративная разработка.

1 ОСОБЕННОСТИ ЖАНРА RTS

1.1 Возникновение и становление жанра

Жанр стратегий в реальном времени (англ. **Real-Time Strategy**, сокращённо **RTS**) начал формироваться в конце 1980-х — начале 1990-х годов на фоне стремительного развития персональных компьютеров. Одной из первых полноценных **RTS**-игр считается **Herzog Zwei (1989)** на консоли **Sega Mega Drive**. Она представила основы жанра: управление юнитами, строительство базы, сбор ресурсов и ведение боевых действий в реальном времени. Однако настоящий расцвет **RTS** пришёлся на 1990-е годы. Прорывной стала игра **Dune II: The Building of a Dynasty (1992)** от студии **Westwood Studios**. Именно эта игра ввела многие каноны жанра, ставшие стандартом:

- Управление юнитами с помощью мыши
- Разделение на юнитов с уникальными особенностями
- Строительство базы и добыча ресурсов
- Постепенное развитие технологий

Следующим этапом стало появление **Warcraft: Orcs & Humans (1994)**, а затем **Command & Conquer (1995)** и **StarCraft (1998)**, каждая из которых добавляла новшества в геймплей и углубляла стратегические аспекты. Эти игры получили не только коммерческий успех, но и заложили основы для киберспорта, особенно в Южной Корее, где **StarCraft** стал национальной игрой.

В 2000-х жанр продолжал развиваться, появились:

- **Age of Empires II** — с упором на экономику
- **Warcraft III** — с добавлением ролевых элементов
- **Company of Heroes** — с более реалистичной боевой системой и физикой

С началом 2010-х годов классические **RTS** стали терять популярность из-за роста интереса к **MOBA**, шутерам и мобильным играм. Тем не менее,

жанр не исчез. Он начал развиваться в более нишевом направлении. Появились инди-**RTS**, такие как **They Are Billions**, **Northgard** и **Zero-K**, а крупные студии возвращались к истокам через ремастеры (**StarCraft: Remastered, Age of Empires: Definitive Edition**). Вышел **Age of Empires IV** (2021), вновь привлекший внимание к классическим **RTS**.

Также стоит отметить, что **RTS** оказали большое влияние на развитие других жанров. Из **Warcraft III** выросла **Dota**, ставшая родоначальницей **MOBA**, а элементы **RTS** встречаются в стратегиях жанра **4X** и даже в современных **выживалках** и градостроительных симуляторах.

1.2 Ключевые механики и требования к геймплею в RTS-играх

Жанр стратегий в реальном времени (**RTS**) обладает уникальной структурой геймплея, отличающей его от других игровых жанров. Основу любой **RTS** составляют взаимосвязанные ключевые механики, обеспечивающие стратегическую и тактическую глубину игрового процесса. От качества реализации этих механик зависит баланс, общая заинтересованность и реиграбельность проекта. Осмотрев несколько реализованных проектов в жанре **RTS,** можно выделить основные компоненты геймплея **RTS**-игр, а также требования, предъявляемые к каждой из них:

Сбор ресурсов — один из базовых элементов геймплея. Ресурсы служат основой для постройки зданий, создания юнитов и исследований. Наиболее часто встречающиеся типы ресурсов:

- Добываемые (золото, руда, нефть, древесина и т.п.)
- Производимые (энергия, еда, рабочая сила)
- Временные (мана, очки технологий, таймеры)

Требования:

- Ресурсная система должна быть сбалансированной, чтобы ограничивать чрезмерный рост игрока.
- Добыча должна быть сопряжена с рисками (например, ближнее месторождение — безопасное, но быстро истощается; дальнее — богатое, но требует защиты).

Постройка и развитие базы — это стратегическая составляющая игры, обеспечивающая игрока инфраструктурой. Игрок размещает здания, каждое из которых выполняет определённую функцию (обработка ресурсов, военное производство, исследование технологий, защита).

Требования

- Базы должны быть уязвимыми — их разрушение должно играть ключевую роль в сражении.
- Игрок должен иметь тактический выбор: оборонительная база, агрессивная экспансия, технологическое развитие.

Тренировка войск. Игроку предоставляется возможность производить юниты различных классов: пехота, техника, авиация, морские силы и т.д. Юниты используются для захвата территории, защиты базы, разведки и атаки противника.

Требования:

- Разнообразие юнитов должно обеспечивать тактическую вариативность.
- Должна быть реализована система классов. Все юниты должны иметь чётко определённые роли (танк — бронирован, но медлителен; разведчик — быстр, но уязвим и т.д.).
- В игре должен быть механизм группирования и массового управления отрядами.

Боевые столкновения — ключевой элемент **RTS**. В отличие от шутеров или файтингов, здесь бои происходят между группами юнитов, часто — на нескольких фронтах.

Требования:

- Баланс сил: ни один юнит или тактика не должен доминировать.
- Влияние рельефа и расположения (например, высоты, леса, мосты).
- Влияние времени реакции игрока — игрок, быстрее отдавший команды, получает преимущество.
- Поддержка **приоритетов атак** и **поведения юнитов** (например, “атаковать всех”, “удерживать позицию”, “преследовать”).

Механика **технологического дерева** позволяет игрокам открывать новые юниты, улучшения и возможности. Исследования требуют времени и ресурсов и влияют на дальнейший стиль игры.

Требования:

- Исследования должны давать ощутимое преимущество, но не нарушать баланс.
- Игрок должен выбирать стратегию развития: быстрые дешёвые технологии или медленное, но мощное развитие.
- Древо должно быть визуально понятно и логично структурировано.

Информативность. Знание позиции противника, местонахождения ресурсов и возможных угроз — основа стратегического планирования.

Требования:

- **Туман войны** — область карты, недоступная для обзора, если там нет юнитов или зданий игрока.
- Механики разведки: лёгкие юниты, воздушные шары, радары, обсерватории и пр.

ИИ реализует поведение как вражеских, так и союзных юнитов в одиночной игре. Важно, чтобы он был предсказуем, но не тривиален.

Требования:

- Гибкая реакция на действия игрока (например, адаптация стратегии).
- Планирование атак, контратак и обороны.
- Возможность настраивать уровень сложности.

Эффективное управление — важнейшая часть геймплея **RTS**. Игрок должен уметь быстро отдавать приказы, анализировать ситуацию и контролировать несколько процессов одновременно.

Требования:

- Поддержка “горячих клавиш” и макросов.
- Миникарта с возможностью управления.
- Интуитивный интерфейс для отображения ресурсов, производства и юнитов.

Мультиплеер. RTS часто имеют мультиплеерные режимы, включая PvP и командную игру.

Требования:

- Надёжная синхронизация состояний игроков (часто используется **lock-step** или **deterministic simulation**).
- Баланс игровых сторон.
- Система матчмейкинга, перезапуска, наблюдения.

1.3 Архитектуры многопользовательских игр

Организация сетевого взаимодействия между игроками является одной из ключевых задач при разработке многопользовательских игр. От выбранной архитектуры зависит не только производительность и масштабируемость проекта, но и его уязвимость к читам, устойчивость к

сбоям и даже восприятие отклика управления со стороны пользователя. На практике наиболее широко применяются три архитектурных подхода: **клиент-сервер**, **peer-to-peer (равный-равному)** и **авторитарный сервер**. Каждый из них имеет свои особенности, сценарии применения и компромиссы, которые важно учитывать при проектировании.

1.3.1 Клиент-серверная архитектура

Наиболее традиционный подход — это организация игры по модели клиент-сервер, при которой существует выделенный узел (сервер), отвечающий за координацию всех подключённых клиентов. Каждый игрок управляет своей копией клиента, который передаёт свои действия (например, движение, атака или использование способности) на сервер. Сервер, в свою очередь, обрабатывает эту информацию, рассчитывает результат и рассылает его остальным клиентам, обеспечивая согласованность и синхронность игрового мира. Часто именно сервер в этой архитектуре отвечает за значительную часть логики игры, включая физику, коллизии и правила взаимодействия. Клиенты же в таком случае являются в основном визуализаторами — они отображают состояние, полученное от сервера, и передают пользовательский ввод. Однако возможны и гибридные реализации, в которых часть логики (например, анимации или локальные предсказания движения) остаётся на клиенте для снижения визуальных задержек.

Преимущества:

- Централизованное управление игрой упрощает синхронизацию состояния между всеми участниками.
- Более высокий уровень защиты от несогласованных действий клиентов и потенциального читерства.
- Относительная простота масштабирования за счёт управления потоками и балансировки нагрузки на уровне сервера.

Недостатки:

- Сервер является единой точкой отказа: при его отключении вся игровая сессия становится недоступной.
- Возможна повышенная латентность при удалённом расположении серверов относительно игроков.
- Увеличенные требования к хостингу и сетевым ресурсам, особенно при большом числе одновременно подключённых игроков.

1.3.2 Peer-to-peer (равный-равному)

Архитектура peer-to-peer строится на прямом взаимодействии между клиентами без участия центрального сервера. Все игроки в такой модели связаны друг с другом и обмениваются данными напрямую. Каждый участник может как отправлять, так и получать данные, причём в некоторых реализациях все клиенты обладают равными правами и возможностями — отсюда и название. Одним из классических примеров такой архитектуры является реализация с использованием алгоритма **lockstep**: каждый клиент сначала отправляет свои действия, дожидается получения команд от других участников, и лишь затем одновременно выполняет игровой шаг. Такой подход особенно популярен в старых **RTS**-играх, где каждый игровой тик требует согласованных действий всех игроков.

Преимущества:

- Отсутствие необходимости в выделенном сервере позволяет уменьшить затраты на инфраструктуру.
- Низкая задержка при игре в локальной сети или между игроками вблизи друг друга.
- Возможность децентрализации — при выходе одного игрока другие могут продолжать сессию при наличии механизмов восстановления.

Недостатки:

- Уязвимость к читам: каждый клиент может вмешиваться в игровой процесс, а обнаружить нарушения крайне трудно.
- Трудности синхронизации и восстановления согласованность и синхронность игрового состояния.
- Плохая масштабируемость и растущая нагрузка при увеличении числа участников.

1.3.3 Авторитарный сервер

Архитектура с авторитарным сервером представляет собой усиленный вариант клиент-серверной модели. В этой конфигурации сервер является единственным источником истины: все игровые вычисления, включая проверку корректности действий, физику, столкновения и последствия команд, происходят исключительно на его стороне. Клиенты в такой архитектуре выполняют исключительно роль интерфейса — они отправляют запросы на сервер и отображают полученный от него результат. Для обеспечения плавности игрового процесса клиент может применять локальную предикцию действий и интерполяцию состояний. Однако в случае расхождений между предсказанным и полученным от сервера состоянием требуется механизм отката и повторной синхронизации. Такой подход особенно актуален для соревновательных игр, где честность и согласованность важнее мгновенного отклика.

Преимущества:

- Максимальная защита от несанкционированных изменений: весь игровой процесс контролируется на стороне сервера.
- Предсказуемое поведение системы даже при попытках сетевых атак или вмешательства в клиентский код.
- Полная централизованная аналитика и контроль, что облегчает отладку и поддержку.

Недостатки:

- Повышенная задержка между вводом игрока и реакцией игрового мира (особенно без предикции на клиенте).
- Увеличенные требования к производительности сервера, особенно при большом количестве активных игроков.
- Повышенная сложность реализации, требующая точной синхронизации состояний и продуманной архитектуры rollback-процессов.

1.4 Протоколы и технологии

Сетевые протоколы играют критическую роль в разработке многопользовательских игр, особенно в жанре **RTS**, где важны как своевременная доставка данных, так и их корректность. Правильный выбор протокола может существенно повлиять на плавность геймплея, устойчивость к сетевым потерям, а также общую производительность игры. Традиционно основными транспортными протоколами, лежащими в основе сетевых приложений, являются **TCP** и **UDP**. Однако в последнее десятилетие на фоне развития технологий появился более универсальный подход — протокол **QUIC**, который постепенно вытесняет классические решения.

1.4.1 Transmission Control Protocol

TCP (Transmission Control Protocol) — это ориентированный на соединение протокол, предназначенный для надёжной передачи данных между двумя конечными точками. **TCP** гарантирует, что все отправленные данные дойдут до получателя в точности в том же порядке, в каком были отправлены. Этот протокол реализует механизмы подтверждений доставки (**ACK**), повторной передачи утерянных пакетов, контроль перегрузки сети (congestion control) и автоматическую адаптацию скорости передачи. Эти свойства делают **TCP** надёжным, но одновременно накладывают ограничения, связанные с задержками. В случае потери пакета, передача приостанавливается до тех пор, пока потерянный сегмент не будет повторно получен. Такая ситуация называется head-of-line blocking — блокировка

следующей информации из-за недоставки одного предыдущего элемента. Это неприемлемо в играх реального времени, где задержка даже в несколько сотен миллисекунд может привести к рассинхронизации игрового мира или нарушению взаимодействия игроков. Тем не менее, **TCP** по-прежнему широко используется для обмена важной, но нечасто изменяющейся информацией — например, для аутентификации игроков, загрузки игровых данных, чатов или обновлений инвентаря.

1.4.2 User Datagram Protocol

UDP (User Datagram Protocol) — это протокол без установления соединения, ориентированный на минимальные задержки и скорость доставки. Он не гарантирует ни доставки, ни порядка пакетов, ни их целостности. Именно поэтому **UDP** идеально подходит для быстрой передачи игровых состояний, координат объектов, движений юнитов — всех тех данных, которые устаревают через доли секунды и не требуют повторной отправки в случае потери. Благодаря своей простоте, **UDP** обеспечивает минимальную задержку и не страдает от блокировок, как **TCP**. Однако из-за отсутствия встроенных механизмов надёжности разработчику приходится реализовывать собственные алгоритмы для обработки потерь, восстановления порядка сообщений и синхронизации состояний. Это приводит к усложнению логики, особенно в условиях нестабильных сетей. В результате возникает дилемма: либо использовать **TCP** и страдать от задержек, либо использовать **UDP** и самому решать проблемы сетевой нестабильности.

1.4.3 QUICk UDP Internet Connections

QUIC (**Quick UDP Internet Connections**) — современный транспортный протокол, разработанный компанией **Google** в 2012 году и получивший статус интернет-стандарта в 2021 году. **QUIC** работает поверх **UDP**, но реализует на уровне пользовательского пространства те же

функции, что и **TCP**, включая надёжную доставку, порядок сообщений, мультиплексирование потоков и шифрование (через встроенный **TLS 1.3**). Главное отличие **QUIC** от **TCP** заключается в том, что он устраняет проблему **head-of-line blocking**: если один поток сталкивается с потерей пакета, остальные потоки продолжают передаваться. Это делает **QUIC** особенно привлекательным для многопоточной передачи в играх, где одновременно передаются координаты, игровые события и пользовательские действия. Более того, **QUIC** обеспечивает очень быстрое установление соединения — в большинстве случаев за один-единственный раунд обмена, в отличие от трёхступенчатого рукопожатия в **TCP**. В итоге **QUIC** сочетает надёжность и контроль, присущий **TCP**, с низкой задержкой и гибкостью, характерной для **UDP**.

На фоне распространения **QUIC** в современных браузерах, мобильных приложениях и игровых движках появилась потребность в библиотечной реализации протокола на разных языках. Одной из таких реализаций является **Quinn** — асинхронная реализация **QUIC** на языке **Rust**. Она легла в основу сетевой библиотеки bevy_quinnet, используемой в игровом движке Bevy. Библиотека bevy_quinnet предоставляет высокоуровневый API для организации клиент-серверного взаимодействия на базе **QUIC**, абстрагируя разработчика от всех низкоуровневых деталей шифрования, мультиплексирования и обработки сетевых пакетов. Она автоматически управляет сессиями, подключениями и безопасной передачей данных, позволяя разработчику сосредоточиться на логике синхронизации игрового состояния. Её архитектура идеально подходит для **RTS**-игр, где требуется одновременно высокая скорость, надёжность и масштабируемость.

1.5 Аналоги

1.5.1 Clash Royale

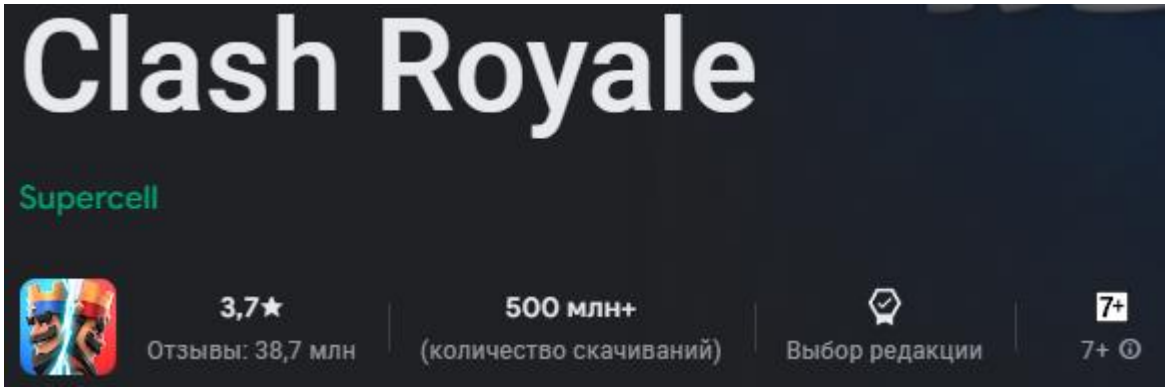

Рисунок 1 — Эмблема игры Clash Royale

Приложение ***Clash Royale*** представляет собой мобильную игру в жанре стратегической карточной арены с элементами защиты башен и боя в реальном времени. Игра разработана студией **Supercell** и предназначена для устройств на платформах **Android** и **iOS**. На рисунке 1 изображена эмблема игры. Игровой процесс заключается в противостоянии двух игроков на арене, разделённой на две половины с тремя башнями на каждой стороне. Цель игрока — разрушить центральную башню соперника, либо нанести максимальный урон в течение ограниченного времени. Игрок использует заранее собранную **колоду из восьми карт**, каждая из которых представляет собой боевого юнита, защитное здание или заклинание. Для использования карт требуется ресурс **эликсир**, который восстанавливается с течением времени. Игра проходит в реальном времени, что требует от пользователя стратегического мышления и быстрой реакции.

Основные особенности приложения:

– Колода состоит из 8 карт, каждая из которых представляет бойца, заклинание или здание. Карты имеют уровни прокачки, которые улучшают их характеристики: урон, здоровье, скорость атаки и радиус действия. Повышение уровня карт требует золота и повторных копий соответствующей карты.

– Все матчи проходят в реальном времени между минимум двумя игроками. Игра использует клиент-серверную архитектуру с полностью авторитарным сервером, который контролирует игровой процесс, исключая возможность подделки состояния игры со стороны клиента.
– В бою игрок может одновременно видеть только четыре карты из своей колоды, а новые карты поступают на руку по очереди циклически. Все действия ограничены ресурсом - эликсиром, который восстанавливается автоматически с течением времени.
– Интерфейс адаптирован под сенсорное управление: все действия выполняются простым перетаскиванием карт на поле. Вся информация о состоянии башен, уровне эликсира и времени матча отображается на одном экране. Игра оптимизирована для вертикального положения экрана, что делает её удобной для мобильного использования одной рукой.
– Игра содержит разветвлённую систему прокачки: игрок может улучшать как отдельные карты, так и уровень своего аккаунта, что открывает доступ к более сильным юнитам, колодам и игровым режимам.

Пользовательский интерфейс основных экранов игры продемонстрирован на рисунке 2.

Рисунок 2 — Пользовательский интерфейс игры ***Clash Royale***

1.5.2 Rush Royale

Рисунок 3 — Эмблема игры Rush Royale

Приложение **Rush Royale** представляет собой мобильную стратегическую игру с упором на быстрые карточные бои в режиме реального времени. Разработана студией **RushGames** и доступна на платформах **Android** и **iOS**. Игроки сражаются друг с другом на ограниченной арене, используя колоду из 10 карт, каждая из которых представляет юнита, защитное сооружение или заклинание. Основная задача — уничтожить главную базу соперника. Матчи длятся в среднем 1–2 минуты, что обеспечивает динамичный игровой опыт. Для разыгрывания карт используется ресурс — мана, которая постепенно восстанавливается во время матча.

Матчи проходят в реальном времени, что требует от игрока не только стратегического мышления, но и быстрой реакции для контратак и адаптации к действиям соперника.

Основные особенности приложения:

- Колода состоит из 5 карт, которые можно собирать и улучшать. Каждая карта имеет уровень, влияющий на её боевые характеристики (урон, здоровье, скорость атаки). Улучшение карт требует внутриигровых ресурсов — золота и кристаллов.

- Игроки получают опыт и внутриигровые ресурсы за победы, что позволяет повышать уровень аккаунта и открывать новые карты. В игре реализована система уровней карт, влияющая на эффективность юнитов.
- В игре предусмотрена система подбора соперников по рейтингу (MMR), чтобы обеспечить равные условия боя. Также реализованы периодические обновления баланса карт, чтобы избежать доминирования определённых стратегий.
- Интерфейс адаптирован под сенсорное управление, все действия выполняются простым касанием и перетаскиванием карт на поле. Вся игровая информация о состоянии карты, запасе маны и времени матча отображается на одном экране.

Пользовательский интерфейс основных экранов игры продемонстрирован на рисунке 4.

Рисунок 4 — Пользовательский интерфейс игры ***Rush Royale***

2 ПОСТАНОВКА ЗАДАЧИ

2.1 Назначение разрабатываемого ПО

Разрабатываемая игра «Боярский турнир» предназначена для создания доступной и увлекательной многопользовательской стратегии в реальном времени, ориентированной на широкую аудиторию игроков. Игра предлагает динамичный соревновательный процесс, сочетающий элементы классических RTS и карточных боёв, при этом обладая низким порогом входа и сессионной структурой матчей, что делает её подходящей как для коротких игровых сеансов, так и для регулярного участия в онлайн-состязаниях.

Особое внимание в проекте уделено созданию оригинальной эстетики, основанной на образах Древней Руси. Это придаёт игре национальную визуальную идентичность и выделяет её среди существующих аналогов, способствуя развитию отечественного игрового контента и интереса к культурному наследию.

«Боярский турнир» предоставляет пользователю возможность в простой и понятной форме реализовывать тактические решения, соревноваться с другими игроками и отслеживать результаты своих матчей. Игра стимулирует стратегическое мышление, адаптацию к действиям противника и умение управлять ограниченными ресурсами в условиях реального времени.

Разработка проекта также направлена на расширение отечественной практики создания конкурентоспособных многопользовательских игр и накопление опыта использования современных технологий в области разработки игр и сетевого взаимодействия. Игра может быть использована как платформа для дальнейших исследований в области клиент-серверной синхронизации, визуального проектирования RTS-интерфейсов и балансировки сессионного PvP-контента.

2.2 Формирование требований

2.2.1 Клиентская часть

Модуль клиентского приложения должен предоставлять игроку доступ к основным игровым действиям и обеспечивать визуализацию происходящего на арене. Модуль клиента должен:

1. Обеспечивать подключение к серверу по заданному IP и порту.
2. Слушать сообщения от сервера и корректно отображать:
   - появление, перемещение, атаку и уничтожение юнитов;
   - состояние башен, главного здания и снарядов;
   - синхронизацию состояния всех сущностей
3. Отправлять на сервер действия игрока:
   - выбор карты и команду на размещение юнита;
4. Отображать игровую арену, UI и HUD:
   - панель с картами;
   - текущий уровень эликсира;
   - полноценное поле для призыва юнитов
5. Обеспечивать логическое отображение арены относительно стороны игрока (зеркальное отображение для игрока-противника).
6. Работать минимум в 60 FPS и не зависеть от производительности сетевого соединения.

2.2.2 Сетевая подсистема

Сетевая подсистема отвечает за установление и поддержание соединения между клиентами и сервером. Сетевой модуль должен реализовывать следующие функции:

1. Протокола взаимодействия по схеме **QUIC поверх UDP**.
2. Авторитарного сервера с раздельными каналами:
   - надёжный упорядоченный (для инициализации);
   - надёжный неупорядоченный (для рассылки действий игроков)

- ненадёжный (для синхронизации позиций).

2.2.3 Серверная часть

Сервер управляет логикой боя и синхронизацией состояния. Модуль сервера должен:

1. Обрабатывать подключения игроков и назначать номера сторон (One / Two).
2. Изменять игровое состояние при подключении двух игроков к серверу.
3. Рассылать начальное состояние матча при старте:
    - позиции башен игроков (One / Two);
    - условные номера игроков
4. Принимать и обрабатывать действия игроков:
    - размещение юнитов при разыгрывании карт;
5. Поддерживать логику обновления состояния:
    - расчёт движения юнитов;
    - столкновения и урон;
    - появление и уничтожение снарядов;
6. Реализация логики юнитов:
    - движение к цели;
    - выбор цели по приоритету;
    - атака при входе в радиус.

2.2.4 Модуль игровых сущностей

Модуль игровых сущностей отвечает за поведение юнитов и снарядов. Требования к модулю игровых сущностей:

1. Хранение компонент:
    - позиция, направление, здоровье, состояние;
    - тип юнита, урон, скорость, радиус атаки.
2. Поддержка состояний;
3. Система урона и уничтожения:

- уменьшение здоровья;
- удаление при достижении нуля;
- рассылка событий Despawn.

2.2.5 Пользовательский интерфейс

UI должен обеспечивать управление матчем и отображение текущего состояния игры. В том числе:

1. Панель карт (рука игрока):
   - отображение 4 доступных карт из колоды;
   - визуальное отображение стоимости и готовности.
2. Полоса эликсира:
   - текущий уровень накопленного эликсира;
3. Возможность перетаскивания карты на поле (drag’n’drop).

2.2.6 Система ресурсов

Игроки используют эликсир как ресурс, ограничивающий частоту действий. Требуется:

1. Автоматическое восстановление ресурса (примерно 1 ед. в 1,5 секунды).
2. Стоимость карт должна списываться при их розыгрыше.
3. Блокировка карт при нехватке ресурса.

2.3 Выбор средств разработки

В качестве основного языка программирования для разработки многопользовательской стратегии «Боярский турнир» выбран **Rust** — современный компилируемый язык, сочетающий в себе высокую производительность, строгую типизацию и безопасность управления памятью. Его особенности делают его особенно подходящим для создания надёжных, масштабируемых и отзывчивых многопоточных приложений, включая игровые серверы и клиентские приложения с синхронной логикой.

Выбор языка Rust обусловлен рядом ключевых преимуществ:

1. **Безопасность без сборщика мусора.**

Благодаря механизму владения (ownership) и системе заимствования (borrowing), Rust обеспечивает безопасность обращения к памяти на этапе компиляции без необходимости использования сборщика мусора. Это особенно важно в игровых проектах, где производительность и контроль над ресурсами критичны.

2. **Высокая производительность**

Rust по уровню производительности сравним с C/C++, что позволяет реализовать сложную игровую логику и сетевую синхронизацию без избыточных накладных расходов.

3. **Современная система типов и метапрограммирование**

Расширенные возможности языка, включая трейты, дженерики и паттерны сопоставления (pattern matching), упрощают реализацию гибкой игровой архитектуры, основанной на паттерне ECS (Entity-Component-System).

4. **Наличие игровых и сетевых библиотек**

Экосистема Rust предлагает активное сообщество и готовые библиотеки, которые ускоряют процесс разработки:

- **Bevy** — современный ECS-движок с поддержкой 2D и, написанный на Rust, обеспечивает модульную архитектуру и высокую читаемость кода;
- **bevy_quinnet** — надстройка для Bevy, реализующая сетевое взаимодействие по протоколу QUIC через библиотеку Quinn, позволяет быстро подключать клиент-серверное взаимодействие с высокой надёжностью;
- **serde + bincode** — библиотеки сериализации и десериализации данных, обеспечивающие компактный и быстрый обмен сообщениями между клиентом и сервером.

5. **Инструментальная поддержка и экосистема разработки**

В проекте используется стандартный стек инструментов Rust:

- **Cargo** — система сборки, управления зависимостями и документации;
- **rust-analyzer** — расширение для интеллектуального анализа кода;
- **Git** — для контроля версий, командной работы и отслеживания истории изменений;
- **Bevy Asset Pipeline** — для подключения спрайтов, анимаций и шрифтов.

6. **Межплатформенность**

Благодаря кроссплатформенной природе Bevy и Rust, проект потенциально может быть собран под Windows, Linux и macOS. В перспективе возможно расширение на WebAssembly и мобильные платформы.

Выбранный стек инструментов оптимально соответствует требованиям к разработке многопользовательской стратегии в реальном времени. Он обеспечивает надёжную работу сетевого взаимодействия, удобную организацию игрового процесса и стабильную структуру проекта. Благодаря сочетанию производительного языка, современного движка и специализированных библиотек, удалось создать техническую основу, которая не только покрывает текущие задачи, но и допускает дальнейшее расширение функциональности без необходимости переработки базовых компонентов.

3 РАЗРАБОТКА

3.1 Архитектура ECS

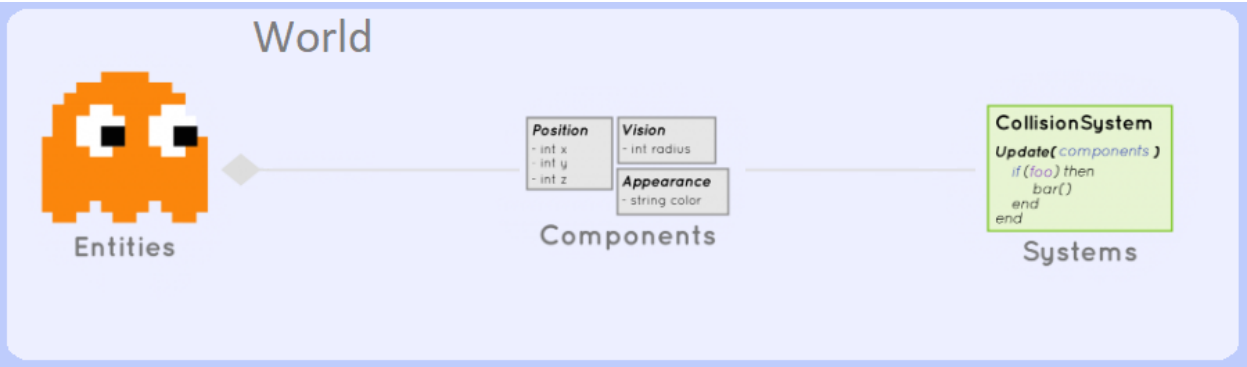

Рисунок 5 – Архитектура Entity-Component-System

В рамках проекта используется архитектура ECS (Entity-Component-System), являющаяся современным стандартом построения игровой логики. Подход ECS реализован с помощью движка Bevy, который в полной мере поддерживает все необходимые уровни абстракции.

Архитектура ECS декомпозирует игровой мир на три основные категории:

1. Сущность (Entity)

Сущность — это примитивная единица идентификации объекта в игровом мире. Она представляет собой числовой идентификатор, к которому можно прикрепить произвольный набор компонентов. Сама по себе сущность не содержит данных.

Сущности могут динамически создаваться и уничтожаться во время выполнения, а их поведение полностью определяется набором прикреплённых компонентов и активных систем.

2. Компоненты (Components)

Компоненты представляют собой структуры данных, описывающие свойства или состояние сущности. Каждый компонент логически независим и не содержит поведения. Примеры компонентных данных включают

позицию, направление движения, состояние здоровья, принадлежность к игроку и т. д.

Компоненты реализуются как типы данных с атрибутом **#[derive(Component)].** Bevy автоматически управляет хранилищем компонент в памяти и предоставляет быстрый доступ к ним через системы.

3. Системы (Systems)

Система — это функция, которая обрабатывает набор компонентов, удовлетворяющих определённым условиям. Системы могут исполняться параллельно, если не возникает конкурентного доступа к одним и тем же компонентам.

Системы выполняются в рамках этапов (schedules) и стадий (stages) игрового цикла, таких как:

Startup — однократное выполнение при инициализации;

Update — выполнение каждый кадр;

FixedUpdate — выполнение с фиксированной частотой, синхронизированной с физикой;

PostUpdate, FixedPostUpdate и другие.

Системы могут быть объединены в цепочки (.chain()), что гарантирует их последовательное выполнение.

4. Ресурсы (Resources)

Ресурсы — это глобальные однозначные данные, доступные всем системам. Они представляют собой структуры, хранящие информацию, не привязанную к конкретной сущности: настройки, конфигурации, подключения к сети, игровые параметры и др.

Ресурсы хранятся во внутреннем хранилище и предоставляются системам через Res, ResMut и аналогичные обёртки. Они активно используются, например, для хранения текущего состояния, таймеров, сессий и сетевых соединений.

5. События (Events)

Для обмена информацией между системами используется механизм событий (Events). Он реализован в виде очередей, в которые одна система может записывать события, а другая — обрабатывать их. Это позволяет реализовать реактивную архитектуру, где, например, событие повреждения может вызвать цепочку действий в других системах.

3.2 Разработка сервера

Серверная часть игры построена как отдельное приложение на основе минимального набора плагинов Bevy. Сервер запускается из server/src/main.rs и имеет следующую структуру, изображенную на рисунке 6.

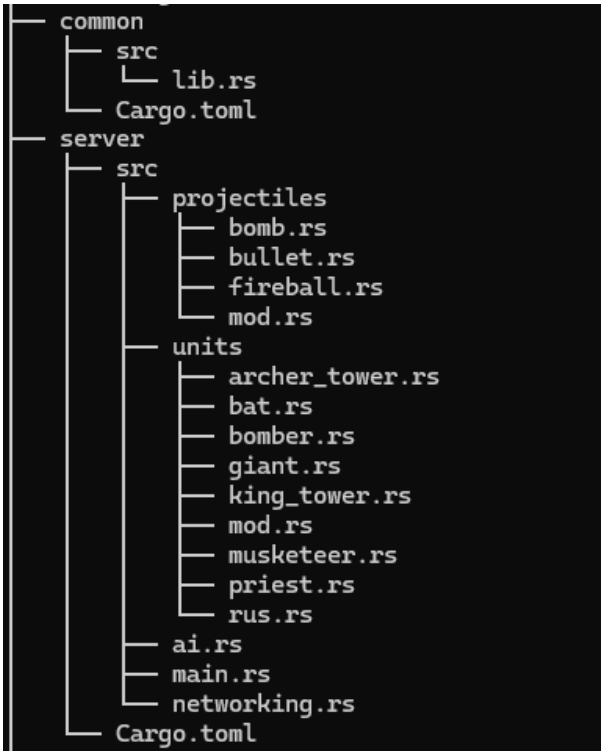

Рисунок 6 — Итоговая структура серверного приложения

Взаимодействие модулей серверной части и предоставляемые модулям интерфейсы изображены на диаграмме компонентов (рисунок 7).

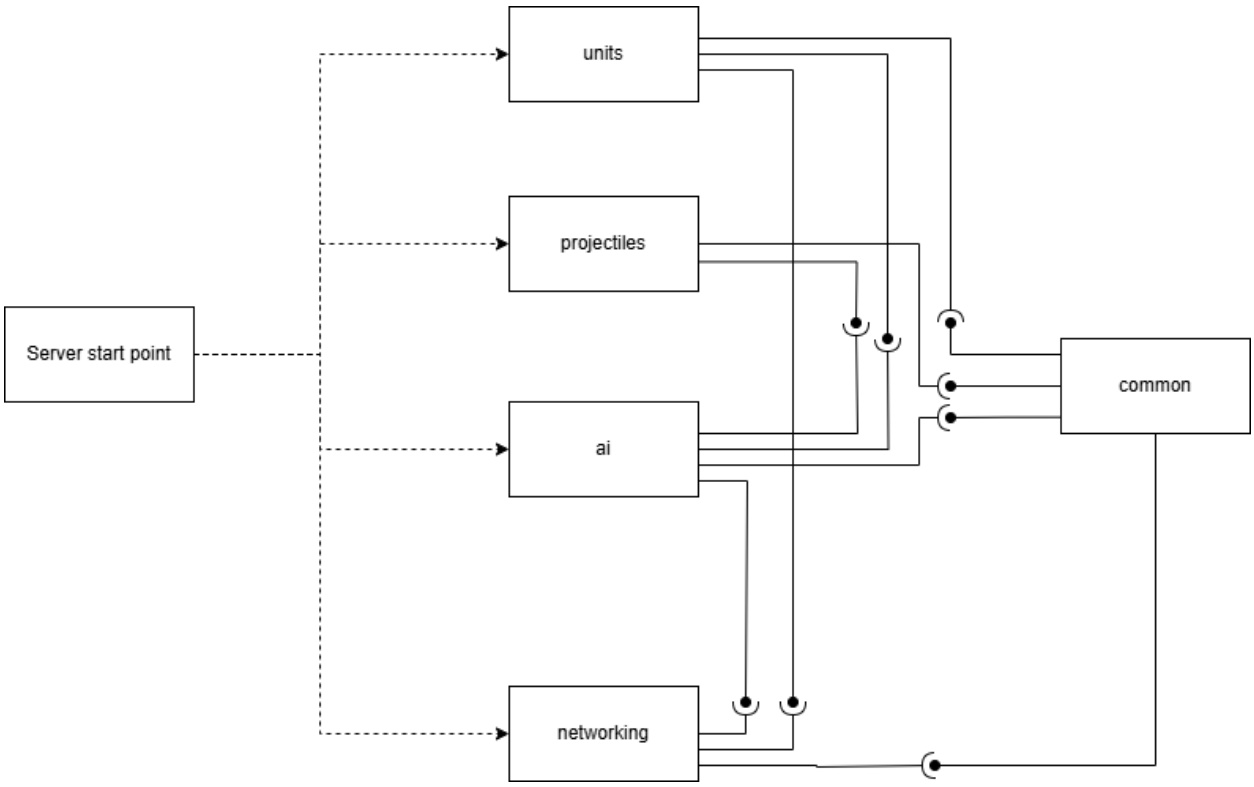

Рисунок 7 — Диаграмма компонентов серверного приложения

3.2.1 Точка входа

В точке входа в приложение создаётся объект App, в который добавляются минимальные плагины Bevy (MinimalPlugins) и инициализируются игровые плагины:

```
App::new()
    .add_plugins((
        MinimalPlugins,
        LogPlugin::default(),
        ai::plugin,
        units::plugin,
        projectiles::plugin,
        networking::plugin,
    ))
    .run();
```

Каждый из подключаемых плагинов инкапсулирует функциональность соответствующего модуля, изолируя подсистемы между собой. В частности:

- ai::plugin — добавляет системы обработки состояния юнитов;
- units::plugin — отвечает за появление юнитов и их типы;
- projectiles::plugin — управление снарядами;
- networking::plugin — инициализация сетевого соединения и обработка клиентских команд;

3.2.2 Модуль юнитов

Модуль units содержит определения всех игровых юнитов, доступных для спавна на сервере, а также вспомогательные типы. Архитектура модуля построена по принципу иерархического разделения: каждый юнит реализован в отдельном .rs файле, подключаемом в mod.rs. Основной модуль определяет вспомогательные компоненты:

- UnitType — компонент, обозначающий тип юнита (наземный или воздушный), используется в логике поиска цели;
- Hitbox(f32) — компонент, определяющий радиус столкновения для расчёта дистанции и взаимодействий;
- SpawnUnit — обобщённый трейт, реализуемый для перечисления Unit, позволяющий создавать юниты по их типу без знания конкретной реализации.

Для каждого юнита реализован следующий подход:

Событие спавна — структура с координатой и владельцем, реализующая трейт Event. Используется для запуска логики создания сущности. Пример события спавна для юнита типа летучей мыши (Bat):

```
#[derive(Event)]
pub struct SpawnBat(pub ArenaPos, pub PlayerNumber);
```

Компонент-сигнатура — структура, помеченная #[derive(Component)] с помощью макроса #[require(...)]. Через атрибут указывается полный набор параметров юнита: здоровье, скорость, радиус реагирования, тип атаки, тип

цели, хитбокс, начальное состояние, оглушение и т.д. Пример компонент-сигнатуры для юнита типа летучей мыши (Bat):

```
#[derive(Component)]
#[require(
    Health(|| Health::new(90)),
    Movement(|| Movement::new(3.)),
    AggroRadius(|| AggroRadius(5.)),
    UnitType(|| UnitType::Air),
    UnitState(|| UnitState::Moving),
    Attack(|| Attack::new(AttackType::Melee(80), AttackTargetType::All, 1., 2.)),
    Hitbox(|| Hitbox(0.5)),
    StunnedTimer,
)]
```

Функция-обработчик события — реализует логику создания сущности с нужными компонентами и отправляет сообщение о спавне всем клиентам через QuinnetServer. Пример функции-обработчика для юнита типа летучей мыши (Bat):

```
fn spawn_bat(
    trigger: Trigger<SpawnBat>,
    mut server: ResMut<QuinnetServer>,
    mut cmd: Commands,
) {
    let &SpawnBat(pos, owner) = trigger.event();
    let entity = cmd.spawn((Bat, pos, owner)).id();
    server
        .endpoint_mut()
        .broadcast_message_on(
            ServerChannel::OrderedReliable,
            ServerMessage::SpawnUnit {
                server_entity: entity,
                unit: Unit::Bat,
                pos,
```

```
            owner,
        },
    )
    .unwrap();
}
```

В текущей реализации сервера поддерживаются 8 типов юнитов. Каждый из них отличается по своим базовым характеристикам. Базовые характеристики юнитов представлены в таблице 1:

Таблица 1 - Базовые характеристики юнитов

Юнит	**Класс**	**Тип атаки**	**Дальность**	**Цели**	**Урон**	**Здоровье**
Rus	Ближний	Melee	2.0	Наземные	80	690
Giant	Ближний	Melee	1.5	Все	160	1200
Musketeer	Стрелок	Bullet	6.0	Все	1/0.75	340
Priest	Стрелок	Fireball	6.0	Все	1/0.75	400
Bomber	Стрелок	Bomb	4.5	Наземные	AOE	350
Bat	Воздушный	Melee	1.5	Все	30	120
ArcherTower	Постройка	Bullet	6.0	Все	50	800
KingTower	Постройка	Bullet	6.5	Все	70	1500

3.2.3 Модуль снарядов

Модуль projectiles отвечает за регистрацию, управление и инициализацию снарядов, которые используются дальнобойными юнитами на серверной стороне. Вся логика сведена к маршрутизации команд спавна и объединению трёх конкретных подмодулей, каждый из которых реализует поведение определённого типа снаряда.

Модуль состоит из трёх основных компонентов:

– трейта SpawnProjectile, реализующего единую точку входа для создания снарядов;

```
pub(super) trait SpawnProjectile {
   fn spawn(&self, attacker: Entity, receiver: Entity, pos: ArenaPos, cmd: &mut Commands);
}
```

```
impl SpawnProjectile for Projectile {
    fn spawn(&self, attacker: Entity, receiver: Entity, pos: ArenaPos, cmd: &mut Commands) {
        match self {
            Projectile::Bullet => cmd.trigger(SpawnBullet(attacker, receiver, pos)),
            Projectile::Fireball => cmd.trigger(SpawnFireball(attacker, receiver, pos)),
            Projectile::Bomb => cmd.trigger(SpawnBomb(attacker, receiver, pos)),
        }
    }
}
```

– вспомогательного компонента ProjectileRadius, использующегося в подмодулях для указания радиуса действия снаряда;

– плагина, подключающего все подсистемы снарядов к серверному приложению.

```
pub(super) fn plugin(app: &mut App) {
    app.add_plugins((bullet::plugin, fireball::plugin, bomb::plugin));
}
```

Для каждого снаряда, как и для каждого юнита реализован подход с помощью события спавна, компонент-сигнатуры и функции-обработчика события.

3.2.4 Модуль системы обработки состояния юнитов

Модуль ai.rs на сервере отвечает за управление поведением всех игровых юнитов: выбор целей, движение, атаки, смену состояний, оглушение и проверку здоровья. Все эти аспекты реализованы с помощью систем Bevy, добавленных в фиксированное обновление (FixedUpdate) через .chain(), что гарантирует последовательное выполнение в логически связанном порядке.

Модуль состоит из пяти основных систем, которые формируют поведение искусственного интеллекта:

```
pub(super) fn plugin(app: &mut App) {
    app.add_systems(
```

```
        FixedUpdate,
        (
            (
                update_attacks,
                update_unit_state,
                update_stun_timers,
                update_movement,
            ),
            check_health,
        )
            .chain(),
    );
}
```

update_unit_state — определение текущего состояния юнита (ожидание, движение, атака);

update_movement — перемещение к цели;

update_attacks — атака цели, если она в радиусе;

update_stun_timers — снятие оглушения по таймеру;

check_health — удаление юнитов при нулевом здоровье и уведомление клиентов.

Системы выполняются строго по порядку благодаря .chain(), чтобы избежать неконсистентности логики, например, когда юнит умирает, но ещё успевает атаковать.

Так же в данном модуле определяются базовые компоненты юнитов:

1. Attack - Хранит параметры атаки юнита:

- a_type: тип атаки (Melee(u16) или Ranged(Projectile));
- t_type: цели атаки (только наземные или все);
- cooldown_timer: задержка между ударами;
- range: дальность атаки;
- target: текущая цель (опционально, Entity).

2. Movement - Содержит скорость и цель движения:

- speed: скорость передвижения (единиц в секунду);
- target: текущая цель движения (Entity или None).

3. StunnedTimer - Таймер оглушения, после которого юнит вновь получает возможность двигаться и атаковать.
4. AggroRadius - Радиус обнаружения врагов, при превышении которого юнит начинает двигаться к противнику.

3.2.5 Модуль общего набора типов и компонентов

Модуль common представляет собой общий набор типов, компонентов, сообщений и утилит, используемых как на клиентской, так и на серверной стороне. Он содержит все ключевые определения, необходимые для синхронизации состояния игры по сети и для логической обработки внутри ECS-архитектуры.

Модуль организован как отдельный crate и включает в себя:

1. Определения компонентов юнитов.
    - ArenaPos - Представляет позицию сущности на арене;
    - Health - Компонент, представляющий очки здоровья;
    - Direction — Компонент, определяющий направление взгляда или движения юнита;
    - PlayerNumber - Определяет принадлежность сущности к первому или второму игроку;
    - Unit - Перечисление, определяющее тип юнита;
    - Projectile - Тип снаряда для дальнобойных атак;
2. Состояния (UnitState);

Данные состояния изменяются в системах ai.rs в зависимости от расстояния до цели, её наличия и типа юнита.

3. Сетевые сообщения (ClientMessage, ServerMessage);

Сообщения клиента служат для отправки запроса на создание юнита игроком. Обрабатываются сервером в handle_client_messages

```
#[derive(Serialize, Deserialize)]
pub enum ServerMessage {
    StartGame(PlayerNumber),
    SpawnUnit {
        server_entity: Entity,
        unit: Unit,
        pos: ArenaPos,
        owner: PlayerNumber,
    },
    SpawnProjectile {
        server_entity: Entity,
        projectile: Projectile,
        attacker: Entity,
        receiver: Entity,
        pos: ArenaPos,
    },
    Despawn(Entity),
    SyncEntities {
        units: Vec<(Entity, ArenaPos, Direction, UnitState, Health)>,
        projectiles: Vec<(Entity, ArenaPos)>,
    },
}
```

Сообщения сервера используются для оповещения клиентов о начале игры, синхронизации состояния мира и добавления/удаления сущностей на клиенте.

4. Идентификаторы каналов передачи (ClientChannel, ServerChannel);

Сетевое взаимодействие между клиентом и сервером организовано по именованным логическим каналам, определённым в модуле common. Это обеспечивает более точный контроль над надёжностью и упорядоченностью передачи данных.

На данный момент определён только один канал для клиента. OrderedReliable — надёжный и упорядоченный канал передачи. Используется для всех входящих от клиента сообщений, таких как ClientMessage::PlayCard, то есть отправка команд игрока на размещение юнитов.

Сервер использует три канала:

OrderedReliable — используется для важных событий, которые должны быть доставлены точно и в правильном порядке (например, SpawnUnit, StartGame, Despawn). Гарантируется, что все клиенты получат сообщение, и в той же последовательности, в какой оно было отправлено.

OrderedReliable — используется для важных событий, порядок доставки которых не важен. Гарантируется, что все клиенты получат сообщение.

Unreliable — применяется для сообщений, передаваемых с высокой частотой и допускающих потерю, например, SyncEntities, содержащих состояния юнитов и снарядов. Потеря одного или нескольких пакетов некритична, так как следующее обновление всё равно перезапишет старое состояние.

5. Константы (SERVER_PORT, LOCAL_BIND_IP, SERVER_HOST и др.).

Используются как на клиенте, так и на сервере для единых настроек подключения. Определяют адрес и порт сервера;

3.2.6 Модуль сетевого соединения

Модуль networking реализует всю серверную сетевую логику — он отвечает за приём подключений, распределение ролей между игроками, обработку их команд и рассылку состояния игры. Работа этого модуля охватывает весь жизненный цикл сетевого взаимодействия: от инициализации соединения до синхронизации игрового мира.

В начале работы сервер запускается и начинает прослушивать порт, ожидая подключения клиентов. При каждом входящем подключении происходит проверка, не превышено ли максимальное количество игроков

(двое). Если лимит уже достигнут, соединение немедленно закрывается. Если игрок может быть допущен, ему присваивается роль первого или второго игрока, которая сохраняется в таблице Lobby. Как только оба игрока подключены, сервер рассылает сообщение о начале игры, в котором каждому клиенту сообщается его номер игрока. После этого на поле автоматически размещаются стартовые постройки — KingTower и две ArcherTower для каждой стороны.

После старта сервер непрерывно ожидает сообщения от игроков. Каждое сообщение содержит команду, например, разыгрывание карты. Сервер принимает такие сообщения, определяет тип карты и инициирует спавн соответствующего юнита или группы юнитов. Спавн происходит централизованно через систему команд Bevy, и управление остаётся на стороне сервера. Таким образом, исключаются попытки клиентов влиять на состояние игры напрямую.

Одновременно с обработкой команд сервер выполняет синхронизацию игрового состояния. На каждом фиксированном кадре собираются данные обо всех активных юнитах и снарядах: их положение, направление движения, состояние, уровень здоровья. Эта информация сериализуется и отсылается обоим клиентам. Направление юнита вычисляется по позиции его цели или берётся по умолчанию, если цель отсутствует. Передача синхронизации осуществляется по ненадёжному каналу, что позволяет игнорировать отдельные потерянные пакеты ради производительности.

3.3 Разработка клиента

Клиентская часть игры построена как отдельное приложение на основе минимального набора плагинов Bevy. Клиет запускается из desktop_client/src/main.rs и имеет следующую структуру, изображенную на рисунке 8.

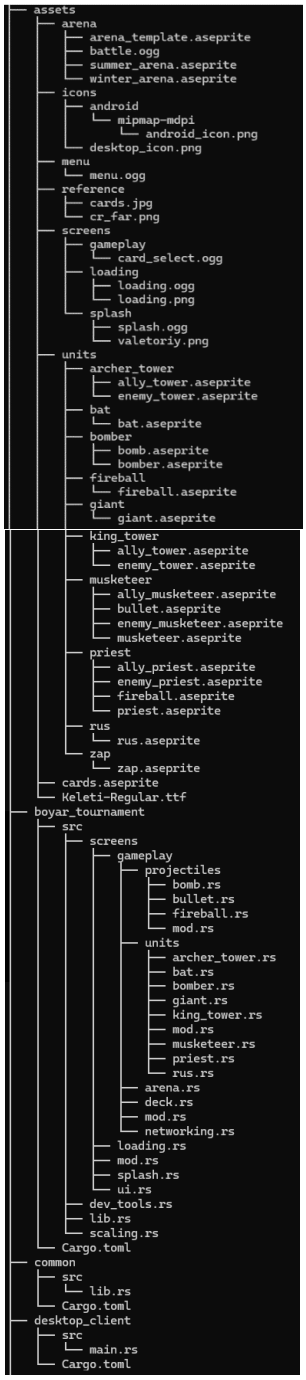

Рисунок 8 — Итоговая структура клиентского приложения

Взаимодействие модулей клиентской части и предоставляемые модулям интерфейсы изображены на диаграмме компонентов (рисунок 9).

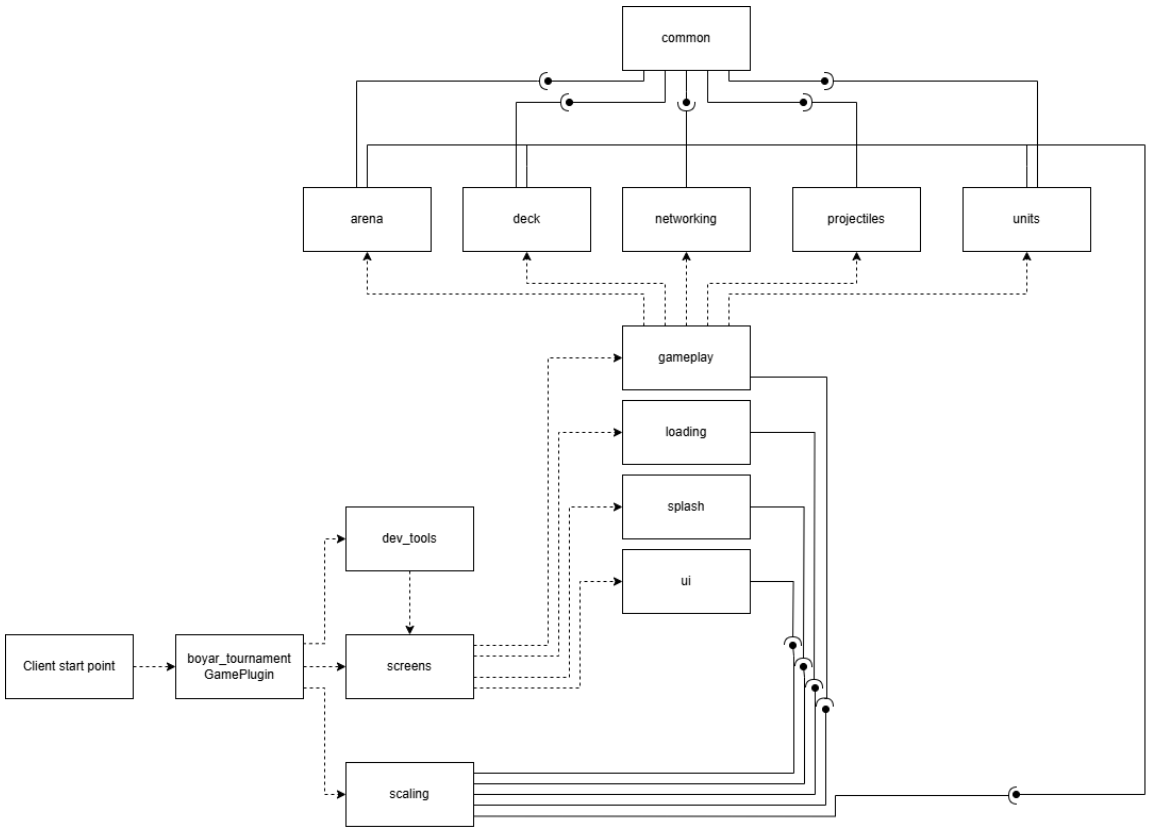

Рисунок 9 — Диаграмма компонентов серверного приложения

3.3.1 Точка входа

Модуль main.rs в клиентском приложении игры «Боярский Турнир» представляет собой точку входа, которая конфигурирует окно, подключает основные и пользовательские плагины, а также задаёт иконку для графического интерфейса. Он реализован в рамках платформы Bevy и запускается с помощью стандартного вызова App::run.

Конфигурация окна осуществляется через метод .add_plugins(...), в рамках которого настраивается отображение заголовка, путь к ассетам, формат загрузки изображений и отключение сглаживания. Ассеты загружаются из внешней директории без проверки метаданных, что упрощает разработку и ускоряет запуск.

Основной игровой плагин GamePlugin, определённый в библиотечном крейте boyar_tournament, подключается для инициализации игровых подсистем, включая визуальные экраны, игровую арену, логику отображения и пользовательский интерфейс.

Для задания иконки окна добавляется стартовая система set_window_icon. Она извлекает дескриптор окна через WinitWindows, загружает изображение иконки из встроенного массива байт, преобразует его в формат RGBA и устанавливает в окно с помощью механизма winit.

3.3.2 Модуль контроля камеры и звука

В файле lib.rs крейта boyar_tournament реализуется основной игровой плагин GamePlugin, который подключается к приложению из main.rs и служит точкой входа для клиентской логики. Внутри GamePlugin определяются базовые действия при запуске клиента.

При старте создаётся 2D-камера, используемая как основная UI-камера. Также инициализируется глобальный уровень громкости звука, задающий громкость всех аудиофайлов по умолчанию.

```
impl Plugin for GamePlugin {
    fn build(&self, app: &mut App) {
        app.add_systems(Startup, spawn_camera);
        app.insert_resource(GlobalVolume {
            volume: Volume::new(0.3),
        });
        app.add_plugins((scaling::plugin, screens::plugin));
        #[cfg(debug_assertions)]
        app.add_plugins(dev_tools::plugin);
    }
}
fn spawn_camera(mut cmd: Commands) {
    cmd.spawn((Camera2d, IsDefaultUiCamera));
}
```

Подключаются два внутренних модуля:

- scaling — отвечает за масштабирование интерфейса и адаптацию к разрешению окна;
- screens — управляет логикой экранов и игровых состояний.

В режиме отладки дополнительно подключается модуль dev_tools, содержащий инструменты разработчика. Подключение происходит условно, только при сборке с debug_assertions.

3.3.3 Модуль отладочных инструментов

Модуль dev_tools реализует отладочные инструменты, подключаемые только в режиме разработки. Его цель — облегчить диагностику и контроль над внутренним состоянием игры во время тестирования.

```
pub(super) fn plugin(app: &mut App) {
    app.add_systems(Update, log_transitions::<GameState>);
    app.add_plugins(FpsOverlayPlugin::default());
    app.add_plugins(WorldInspectorPlugin::new());
    app.add_plugins(DebugUiPlugin);
    app.add_systems(
        Update,
        toggle_debug_ui.run_if(input_just_pressed(KeyCode::Backquote)),
    );
}
```

Основные действия при подключении плагина:

1. Логирование переходов состояний.

   Система log_transitions::<GameState> отслеживает и выводит в консоль переходы между экранами (состояниями GameState), что удобно при отладке сцен.

2. FPS-оверлей:

   Подключается FpsOverlayPlugin — он отображает счётчик кадров в секунду (FPS) поверх экрана.

3. Инспектор мира:

С помощью WorldInspectorPlugin от bevy_inspector_egui предоставляется возможность визуально просматривать и изменять компоненты сущностей в реальном времени через интерфейс на базе egui.

4. Отладочное UI-меню:

Подключается DebugUiPlugin, предоставляющий панель с диагностикой UI-состояний.
Его отображение можно включать/выключать во время выполнения нажатием клавиши Backquote (\``), благодаря системе toggle_debug_ui, которая переключает значение в ресурсе UiDebugOptions`.

Этот модуль не влияет на финальную сборку и исключается в релизе, что делает его удобным средством контроля для разработчиков

3.3.4 Модуль адаптивной отрисовки

Модуль scaling отвечает за реализацию адаптивной отрисовки интерфейса и игровых объектов, сохраняя постоянные пропорции вне зависимости от разрешения и соотношения сторон окна. Он вводит концепцию **виртуального региона отрисовки** размером 9×16 единиц — условной «сетки», в пределах которой задаются координаты и масштаб всех сущностей.

Вместо прямой привязки координат и размеров к пикселям окна, все объекты позиционируются и масштабируются относительно DrawRegion. Этот регион вычисляется на основе текущего размера окна, чтобы всегда сохранять нужное соотношение сторон (9:16), центрируя игровую сцену и исключая искажения.

```
#[derive(Resource, Reflect, Default)]
#[reflect(Resource)]
pub struct DrawRegion {
    pub width: f32,
    pub height: f32,
}
#[derive(Component, Reflect)]
#[reflect(Component)]
pub struct DynamicScale(pub f32);
#[derive(Component, Reflect, Default)]
```

```
#[reflect(Component)]
pub struct DynamicTransform(pub f32, pub f32);
```

- DrawRegion — глобальный ресурс, содержащий фактическую ширину и высоту области отрисовки, соответствующей пропорциям 9:16. Он обновляется при каждом изменении размера окна.
- DynamicScale(f32) — компонент, управляющий масштабом сущности. Значение f32 задаёт коэффициент, рассчитанный на эталонное разрешение 1920×1080. Масштаб пересчитывается автоматически в зависимости от высоты DrawRegion.
- DynamicTransform(x, y) — компонент позиционирования в координатах виртуальной сетки (в «ячейках» DrawRegion). Указывает, где должна находиться сущность по горизонтали и вертикали относительно всей области отрисовки.

```
app.add_systems(
    PreUpdate,
    (
        update_draw_region,
        update_dynamic_scale,
        update_dynamic_transform,
    )
        .chain(),
  );
```

В PreUpdate-фазе добавляются цепочкой три основные системы:

update_draw_region — отслеживает события изменения размера окна (WindowResized) и пересчитывает пропорции DrawRegion так, чтобы сохранить соотношение 9:16, максимально используя доступное пространство по ширине или высоте.

update_dynamic_scale — обновляет масштаб всех сущностей с компонентом DynamicScale на основе текущей высоты DrawRegion.

update_dynamic_transform — задаёт координаты отрисовки для сущностей с компонентом DynamicTransform, используя ширину и высоту DrawRegion.

В режиме разработки (debug_assertions) доступна вспомогательная система draw_draw_region_outline, которая позволяет визуализировать виртуальную сетку по нажатию клавиши F1. Эта сетка отображается с помощью Gizmos и помогает оценить, как объектам сопоставляются координаты внутри DrawRegion.

3.3.5 Модуль управления сценами

Модуль screens отвечает за организацию экранной логики в клиентской части игры и реализует централизованную систему управления сценами. Он построен на механизме состояний (States) в движке Bevy, позволяющем разбить жизненный цикл игры на отдельные фазы.

Ключевая идея — каждое состояние (Splash, Loading, Gameplay) представляет собой логически обособленный экран с собственным набором сущностей и систем. Благодаря этому удаётся добиться изоляции логики, упрощая переходы между сценами и управление ресурсами.

Состояния реализуются через перечисление GameState, которое объявлено как перечислимый тип States. В текущей версии игры определены следующие стадии:

```
#[derive(States, Debug, PartialEq, Eq, Clone, Hash, Default, Copy)]
pub enum GameState {
    #[default]
    Splash,
    Loading,
    // Menu,
    Gameplay,
}
```

- Splash — стартовая заставка с логотипом Valetoriy (используется по умолчанию);
- Loading — загрузочный экран;
- Gameplay — основная игровая сцена, где происходит бой между игроками;
- Состояние Menu присутствует в коде, но пока не реализовано и закомментировано.

```
pub(super) fn plugin(app: &mut App) {
    app.init_state::<GameState>();
    app.enable_state_scoped_entities::<GameState>();
    app.add_plugins((
        splash::plugin,
        loading::plugin,
        gameplay::plugin,
        ui::plugin,
    ));
}
```

При запуске игры происходит регистрация этого состояния через init_state::<GameState>(), что сообщает движку о необходимости отслеживать текущую фазу. Далее включается механизм enable_state_scoped_entities, благодаря которому все сущности автоматически прикрепляются к активному состоянию. Это означает, что при смене сцены, например, с Loading на Gameplay, связанные с предыдущей стадией сущности удаляются или временно деактивируются.

Кроме того, в модуле подключаются четыре внутренних плагина, каждый из которых соответствует своей сцене:

- splash — логика анимации логотипа;
- loading — логика отображения загрузочного экрана;
- gameplay — реализация игрового процесса;

– ui — пользовательский интерфейс, адаптированный под состояние Gameplay.

3.3.6 Модуль отображения пользовательского интерфейса

Модуль ui реализует базовую систему интерактивного пользовательского интерфейса, построенного поверх виртуальной сетки (DrawRegion) с прямоугольными хитбоксами. Он обеспечивает обработку взаимодействия с элементами интерфейса на основе позиции курсора или сенсорных нажатий, и в режиме отладки визуализирует область взаимодействия.

```
#[derive(Component, Reflect)]
#[reflect(Component)]
#[require(UiInteraction, DynamicTransform)]
// Длина и ширина прямоугольника хитбокса в клетках DynamicTransform
pub struct UiHitbox(pub f32, pub f32);
#[derive(Component, Reflect, Default)]
#[reflect(Component)]
enum UiInteraction {
   #[default]
   None,
   Hovered,
   Pressed,
}
```

– UiHitbox(f32, f32) — компонент, задающий прямоугольную область взаимодействия в клетках виртуальной сетки DrawRegion. Значения ширины и высоты интерпретируются относительно координат DynamicTransform.
– UiInteraction — внутреннее состояние элемента интерфейса:
  - None — курсор вне области;
  - Hovered — курсор наведен на элемент;

- Pressed — зафиксировано нажатие (левая кнопка мыши или сенсорное касание).

```
pub(super) fn plugin(app: &mut App) {
    app.register_type::<UiInteraction>();
    app.register_type::<UiHitbox>();
    app.add_systems(Update, (update_ui_hitboxes, trigger_on_press));
    #[cfg(debug_assertions)]
    app.add_systems(Update, draw_ui_hitboxes_outline);
}
```

Система update_ui_hitboxes Отвечает за обработку пользовательского взаимодействия с элементами интерфейса. При каждом кадре определяется позиция курсора мыши или касания, которая затем приводится к координатам центра окна. Эта позиция сравнивается с размерами прямоугольных хитбоксов, рассчитываемых на основе компонентов UiHitbox и DynamicTransform с учётом текущего размера отрисовочного региона DrawRegion. При попадании курсора в область хитбокса состояние UiInteraction обновляется до Hovered или Pressed, в остальных случаях сбрасывается в None.

Система trigger_on_press проходит по всем сущностям с состоянием UiInteraction::Pressed и генерирует для них событие OnPress, которое может быть перехвачено другими системами, чтобы инициировать реакцию на нажатие интерфейсного элемента.

В режиме разработки (через #[cfg(debug_assertions)]) активируется дополнительная система draw_ui_hitboxes_outline, визуализирующая границы всех UiHitbox синими прямоугольниками. Включается нажатием F3, что позволяет визуально проверить корректность размещения интерактивных областей.

3.3.7 Модуль отображения игровой сцены

Модуль gameplay организует основную игровую сцену, объединяя в себе ключевые подсистемы, необходимые для отрисовки арены, отображения пользовательского интерфейса, сетевого взаимодействия, а так же визуализации юнитов и снарядов. Его логика запускается в рамках состояния GameState::Gameplay.

```
pub(super) fn plugin(app: &mut App) {
    app.add_plugins(AsepriteUltraPlugin);
    app.add_plugins((
        arena::plugin,
        networking::plugin,
        units::plugin,
        deck::plugin,
        projectiles::plugin,
    ));
    app.configure_loading_state(
LoadingStateConfig::new(GameState::Loading).load_collection::<FontAssets>(),
    );
}
#[derive(AssetCollection, Resource)]
struct FontAssets {
    #[asset(path = "Keleti-Regular.ttf")]
    font: Handle<Font>,
}
fn spawn_text(
    cmd: &mut Commands,
    text: &str,
    font: Handle<Font>,
    font_size: f32,
    color: Color,
    dynamic_scale: f32,
    dynamic_transform: (f32, f32),
    state: GameState,
```

```
)
```

Основная инициализация происходит через функцию plugin, где:

- Подключается плагин AsepriteUltraPlugin, обеспечивающий поддержку анимаций спрайтов из файлов Aseprite.
- Добавляются модули арены, сетевого клиента, юнитов, колоды карт и снарядов — каждый из которых предоставляет собственный плагин с игровой логикой.
- Задаётся коллекция ресурсов FontAssets, которая загружается на этапе GameState::Loading через bevy_asset_loader. Это упрощает доступ к шрифтам в других подсистемах.

Дополнительно модуль содержит вспомогательную функцию spawn_text, предназначенную для отрисовки текста с контуром на экране. Текст отрисовывается дважды: основной экземпляр текста с заданным цветом, и смещённая чёрная копия под ним, создающая эффект обводки. Обе версии текста получают компоненты DynamicScale и DynamicTransform, чтобы сохранять корректное позиционирование и масштабирование в зависимости от текущих размеров окна. Компонент StateScoped ограничивает их присутствие только в рамках активного состояния GameState, что предотвращает отображение элементов в других сценах.

3.3.8 Модуль отрисовки арены

Модуль arena отвечает за отображение боевой арены, её координатную систему, музыку и логику взаимодействия с положением курсора/касания на арене. Он запускается в игровом состоянии GameState::Gameplay и управляет визуальной и логической привязкой сущностей к координатной сетке.

```
pub(super) fn plugin(app: &mut App) {
    app.register_type::<ArenaPos>();
    app.register_type::<ArenaHeightOffset>();
    app.register_type::<MouseArenaPos>();
    app.init_resource::<MouseArenaPos>();
```

```
    app.configure_loading_state(
LoadingStateConfig::new(GameState::Loading).load_collection::<ArenaAssets>(),
    );
    app.add_systems(OnEnter(GameState::Gameplay), spawn_arena);
    app.add_systems(
        Update,
        (update_arena_pos, update_mouse_arena_pos).run_if(in_state(GameState::Gameplay)),
    );
    #[cfg(debug_assertions)]
    app.add_systems(
        Update,
        draw_arena_region_outline.run_if(in_state(GameState::Gameplay)),
    );
}
#[derive(AssetCollection, Resource)]
struct ArenaAssets {
    #[asset(path = "arena/winter_arena.aseprite")]
    arena: Handle<Aseprite>,
    #[asset(path = "arena/battle.ogg")]
    battle_music: Handle<AudioSource>,
}
```

При инициализации регистрируются типы ArenaPos, ArenaHeightOffset и MouseArenaPos как компоненты или ресурсы, подключается коллекция ассетов ArenaAssets, включающая фоновую арену и музыку. Они загружаются в состоянии GameState::Loading, а используются при входе в Gameplay.

При переходе в состояние Gameplay отрисовывается арена через Aseprite, и начинается воспроизведение фоновой музыки.

Добавляются системы обновления положения сущностей в зависимости от логических координат ArenaPos. Масштаб и смещение преобразуются в пиксельные координаты, привязанные к текущему DrawRegion.

Отдельно отслеживается координата курсора в терминах ArenaPos и сохраняется в MouseArenaPos. Эта позиция затем используется, для установки юнитов по клику.

В режиме отладки клавишей F2 можно отобразить координатную сетку арены через Gizmos, чтобы упростить позиционирование элементов во время разработки.

3.3.9 Модуль управления колодой

Модуль deck реализует механику управления картами игрока и их визуализацию на экране. Он отвечает за выбор карты, проверку стоимости (эликсира), размещение юнита на арене и управление текущей колодой.

```
    app.register_type::<Deck>();
    app.register_type::<DeckIndex>();
    app.register_type::<SelectedCard>();
    app.register_type::<ElixirCounter>();
#[derive(AssetCollection, Resource)]
struct CardsAssets {
    #[asset(path = "cards.aseprite")]
    cards: Handle<Aseprite>,
    #[asset(path = "screens/gameplay/card_select.ogg")]
    card_select: Handle<AudioSource>,
}
```

Модуль регистрирует компоненты и ресурсы: Deck, DeckIndex, SelectedCard, ElixirCounter, а также загружает ассеты карт (cards.aseprite) и звуки выбора (card_select.ogg). При этом в Deck случайным образом перемешивается набор из восьми доступных карт, из которых четыре отображаются игроку, пятая — в качестве "следующей".

```
app.add_systems(
        OnEnter(GameState::Gameplay),
        (spawn_card_hand, spawn_elixir_counter),
    );
```

После входа в состояние Gameplay начинают работу системы spawn_card_hand и spawn_elixir_counter, которые создают 4 карты на нижней панели, каждую с привязкой к индексу колоды и активным хитбоксом для выбора и счётчик эликсира. Также создаётся отображение "следующей карты" и текстовая подпись.

Карты отображаются с использованием спрайтов из файла cards.aseprite. Каждая из них помечается компонентом DeckIndex, определяющим её положение в колоде. Активная карта отслеживается через ресурс SelectedCard. При нажатии на карту активируется система on_card_select, которая визуально увеличивает выбранную карту и сохраняет её индекс. Повторное нажатие снимает выбор и возвращает масштаб к исходному значению.

Состояние эликсира управляется ресурсом ElixirCounter, который автоматически пополняется во времени (до максимального значения 10). Отображение текущего количества эликсира на экране обновляется в системе update_elixir_counter. Расход эликсира происходит в момент размещения карты.

```
client.connection_mut()
    .send_message_on(
        ClientChannel::OrderedReliable,
        ClientMessage::PlayCard {
            card,
            placement: ArenaPos(x, y),
        },
    )
    .unwrap();
```

Игровая логика активации карты реализована в системе play_card. Если карта выбрана и у игрока достаточно эликсира, координаты нажатия преобразуются в игровую систему координат с учётом стороны игрока. Затем формируется и отправляется сетевое сообщение ClientMessage::PlayCard на

сервер. После этого колода сдвигается: сыгранная карта уходит в конец, а следующая становится активной. Обновление визуального состояния карт происходит в системе update_card_hand, которая заменяет спрайты в соответствии с новой последовательностью.

3.3.10 Модуль сетевого соединения

Модуль networking.rs отвечает за сетевое взаимодействие клиента с сервером в игровом процессе. Он реализует подключение, приём сообщений от сервера и обновление локального состояния игры в соответствии с полученными данными. Ниже представлено подробное описание его основных элементов и логики.

```
app.add_plugins(QuinnetClientPlugin::default());
    app.init_resource::<PlayerNumber>();
    app.init_resource::<NetworkMapping>();
    app.register_type::<NetworkMapping>();

    app.add_systems(OnEnter(GameState::Gameplay), start_connection);
    app.add_systems(
        Update,
        handle_server_messages.run_if(in_state(GameState::Gameplay)),
    );
```

При входе в состояние Gameplay срабатывает система start_connection, инициализирующая соединение с сервером по IP и порту, указанным в константах SERVER_HOST и SERVER_PORT из модуля common. Для упрощения используется CertificateVerificationMode::SkipVerification, позволяющий обойти проверку сертификатов в тестовой среде. Каналы связи настраиваются через ClientChannel::channels_config().

```
fn handle_server_messages(
    mut client: ResMut<QuinnetClient>,
    mut player_num: ResMut<PlayerNumber>,
    mut cmd: Commands,
```

```
    mut network_mapping: ResMut<NetworkMapping>,
    mut units_query: Query<(&mut ArenaPos, &mut Direction, &mut UnitState, &mut Health)>,
    mut projectiles_query: Query<&mut ArenaPos, Without<UnitState>>,
    towers: Query<&AssociatedTower>,
)
```

Центральная система handle_server_messages отвечает за приём и обработку входящих сообщений от сервера. Она реагирует на каждое сообщение, соответствующим образом обновляя локальное состояние клиента.

```
match message {
        ServerMessage::StartGame(n) => *player_num = n,
        ServerMessage::SpawnUnit {
            ---------------------
        }
        ServerMessage::SpawnProjectile {
            --------------
        } => projectile.spawn(
            -------------------
        ),
        ServerMessage::Despawn(server_entity) => {
            ----------------------
        }
        ServerMessage::SyncEntities { units, projectiles } => {
             --------------------
            }
            for (server_entity, pos) in &projectiles {
                let Some(&entity) = network_mapping.get(server_entity) else {
                    continue;
                };
                let mut p = projectiles_query.get_mut(entity).unwrap();
                *p = pos.adjust_for_player(*player_num);
            }
        }
```

```
    }
```

Когда начинается игра, сервер отправляет сообщение StartGame, в котором клиенту назначается номер игрока — первый или второй. Эта информация сохраняется в специальном ресурсе и используется для корректной ориентации элементов игры.

При появлении новых юнитов или снарядов (SpawnUnit, SpawnProjectile), клиент создаёт соответствующие сущности на своей стороне. При этом координаты и направление, переданные сервером, при необходимости отражаются по вертикали — это необходимо для игроков, находящихся на противоположной стороне поля, чтобы поле отображалось одинаково для обоих участников.

Удаление сущностей на сервере сопровождается сообщением Despawn. Получив его, клиент находит соответствующий локальный объект через таблицу соответствий (NetworkMapping) и удаляет его. В случае с башнями, дополнительно учитывается вложенная связь с родительской сущностью.

Синхронизация состояния (SyncEntities) позволяет клиенту поддерживать актуальные координаты, направление, состояние и здоровье всех активных юнитов и снарядов. Все передаваемые данные также приводятся к локальному виду с учётом стороны игрока.

```
trait AdjustForPlayer {
    fn adjust_for_player(&self, player_num: PlayerNumber) -> Self;
}
impl AdjustForPlayer for ArenaPos {
    fn adjust_for_player(&self, player_num: PlayerNumber) -> Self {
        match player_num {
            PlayerNumber::One => *self,
            PlayerNumber::Two => ArenaPos(-self.0, -self.1),
        }
    }
}
```

```
impl AdjustForPlayer for Direction {
    fn adjust_for_player(&self, player_num: PlayerNumber) -> Self {
        match player_num {
            PlayerNumber::One => *self,
            PlayerNumber::Two => self.opposite(),
        }
    }
}
```

Реализован вспомогательный трейт AdjustForPlayer, который инвертирует координаты и направления для игроков, играющих «снизу» поля (PlayerNumber::Two). Это позволяет использовать единую игровую логику, независимо от позиции игрока.

Ресурс NetworkMapping хранит отображение серверных Entity на локальные клиентские. Это необходимо для корректного обновления или удаления сущностей при синхронизации и удалении.

3.3.11 Модуль юнитов

Модуль units управляет визуальным представлением и анимациями игровых юнитов на стороне клиента. Отличие от серверной версии заключается в использовании Aseprite-анимаций и компонент AseSpriteAnimation и AnimationState для отображения движения, атак и смены направлений.

```
app.add_plugins((
        archer_tower::plugin,
        king_tower::plugin,
        rus::plugin,
        musketeer::plugin,
        bat::plugin,
        priest::plugin,
        bomber::plugin,
        giant::plugin,
    ));
```

Все юниты подключаются через отдельные подмодули (например, rus.rs, musketeer.rs), а сам модуль централизует их регистрацию и задаёт поведение анимации через систему manage_animation, которая выбирает нужный спрайт по Direction и UnitState.

3.3.12 Модуль снарядов

Модуль почти идентичен серверной реализации, с добавлением системы update_projectile_height, которая определяет вертикальное смещение снаряда (например, дугу полёта).
Для расчёта высоты снаряда используется компонент ArenaHeightOffset, который изменяется на основе расстояний от снаряда до атакующего и цели.

Отдельные типы снарядов (Bullet, Fireball, Bomb) реализованы в отдельных модулях, а выбор спавна конкретного снаряда делегируется через SpawnProjectile.

3.3.13 Модуль общего набора типов и компонентов

В клиентском проекте common используется как общее звено между клиентом и сервером. Он содержит определения сущностей и компонентов: ArenaPos, PlayerNumber, Direction, Health, Unit, Projectile, Card, ClientMessage, ServerMessage и др.

На стороне клиента все эти структуры применяются для сетевого обмена сообщениями, интерпретации игровых событий, адаптации координат и направлений в зависимости от стороны игрока (adjust_for_player), а также отображения юнитов, их состояния и поведения.

Повторного описания не требуется, так как common одинаково используется как на сервере, так и на клиенте.

4 АНАЛИЗ СЕТЕВОГО ТРАФИКА

Для проведения анализа сетевого трафика, генерируемого во время игровых сессий, использовался инструмент **Wireshark** — специализированное программное средство для перехвата и анализа сетевых пакетов. Целью захвата было получение количественных и качественных характеристик сетевого обмена между игровыми клиентами и серверами для последующего сравнения.

Захват трафика проводился по трём игровым проектам: *«Боярский турнир»*, *Clash Royale* и *Rush Royale*. В каждом случае фиксировалась одна полноценная игровая сессия, включая весь процесс — от загрузки до завершения матча.

4.1 Захват трафика в игре «Боярский турнир»

В ходе тестирования Боярского турнира игровой клиент запускался на персональном компьютере, а серверная часть находилась физически на другом устройстве, расположенном в другой сети и подключённом через интернет. Таким образом, взаимодействие происходило в условиях **реальной сетевой среды**, включая маршрутизацию и потенциальную потерю пакетов. Это позволило зафиксировать поведение сетевого протокола не в идеальных лабораторных условиях, а в реальной ситуации, приближённой к пользовательской.

Захват пакетов производился на клиентской стороне во время подключения к серверу, обмена игровыми событиями и завершения матча. Для исключения стороннего сетевого шума были отключены все фоновые приложения (браузеры, мессенджеры, обновления).

4.2 Захват трафика в играх Clash Royale и Rush Royale

Clash Royale и Rush Royale запускались через эмулятор **BlueStacks**, установленный на том же ПК, где велось прослушивание. Это позволило перехватывать весь трафик, исходящий от мобильных приложений, включая

игровые пакеты. Поскольку обе игры недоступны в России, для корректной работы требовалось использование **VPN**, настроенного внутри эмулятора. Следует учитывать, что наличие **VPN** могло оказывать влияние на параметры сетевого взаимодействия, в том числе на задержки и маршруты передачи пакетов.

Во время сессий на BlueStacks также были отключены все фоновые процессы как внутри эмулятора, так и на хост-машине, чтобы исключить сторонние сетевые соединения и рекламный трафик.

4.3 Анализ трафика «Боярского турнира»

По результату работы Wireshark был получен файл захвата **boyar_tournament.pcapng**, содержащий данные игрового сеанса в разработанной игре «Боярский турнир». Из вкладки **Conversations** в Wireshark были извлечены основные параметры сетевого соединения, осуществлявшегося по протоколу UDP с использованием технологии QUIC. В таблице 2 представлены агрегированные данные по сессии:

Таблица 2 – агрегированные данные по игровой сессии Боярского турнира

Параметр	**Значение**
Протокол	QUIC
Порт сервера	50505
Всего пакетов	35 730
Общий объем трафика	8.31 МБ
Пакеты от клиента к серверу	11 890
Объём от клиента к серверу	949 682 байт
Пакеты от сервера к клиенту	23 840
Объём от сервера к клиенту	7 362 522 байт
Длительность сессии	~351 секунд
Средняя скорость сервера → клиента	~167 777 бит/сек (≈ 21 Кб/с)
Средняя скорость клиента → сервера	~21 641 бит/сек (≈ 2.7 Кб/с)

Как видно из таблицы, основная часть трафика генерируется сервером, что соответствует типичной клиент-серверной архитектуре с авторитарным

сервером, где клиент исполняет лишь команды игрока, а вся игровая логика рассчитывается и транслируется с сервера. Пропорция объёма переданных данных (примерно 1:8) подтверждает это.

На рисунке 10 изображен график сетевой активности, построенный в Wireshark через вкладку **I/O Graphs**.

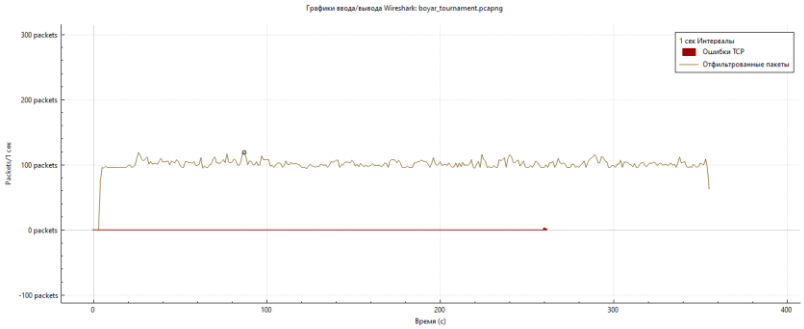

Рисунок 10 — График сетевой активности игровой сессии Боярского турнира

На графике наблюдается равномерная передача данных без резких пиков и провалов. Передача пакетов происходит регулярно, с частотой порядка 100 пакетов в секунду, что обеспечивает стабильное и предсказуемое поведение клиента в условиях реального времени. Это особенно важно для игр с быстрыми боевыми действиями и постоянной синхронизацией состояния объектов. В ходе анализа не были зафиксированы случаи повторных передач, потерь или сильных отклонений по времени доставки. Это говорит о высокой устойчивости реализованной сетевой логики к нестабильным условиям, что критически важно для PvP-игр.

4.4 Анализ трафика «Clash Royale»

По результатам работы Wireshark были получены три файла захвата (**clash_royale_1st.pcapng, clash_royale_2nd.pcapng, clash_royale_3rd.pcapng**), отражающие три отдельные игровые сессии в приложении Clash Royale. Анализ проводился в эмуляторе BlueStacks под управлением VPN, что потенциально могло в незначительной степени повлиять на измерения. Тем не менее, полученные данные позволяют

провести обоснованную оценку архитектуры и характера сетевого взаимодействия игры.

Clash Royale использует TCP и TLS 1.2 в качестве основных транспортных протоколов, что характерно для архитектуры с авторитетным сервером, контролирующим весь игровой процесс. На стороне клиента происходит минимальная обработка, а сервер принимает решения и отправляет актуальное состояние.

В таблице 3 представлены агрегированные данные по трём отдельным игровым сессиям:

Таблица 3 – Агрегированные данные по трём игровым сессиям Clash Royale

Параметр	**Значение**		
Сессия	1	2	3
Протокол	TCP (с TLS 1.2)	TCP (TLS 1.2)	TCP (TLS 1.2)
Всего пакетов	36 036	12 694	3 345
Общий объём трафика	~29.7 МБ	~8.6 МБ	~258 КБ
Пакеты от клиента к серверу	14 333	7 334	1 440
Объём от клиента к серверу	~1.3 МБ	~8.25 МБ	~103 КБ
Пакеты от сервера к клиенту	21 703	5 360	1 905
Объём от сервера к клиенту	~28.3 МБ	~0.38 МБ	~155 КБ
Длительность сессии	~150 секунд	~139 секунд	~128 секунд
Средняя скорость сервера → клиента	~1.5 Мбит/с	~22 Кбит/с	~9.66 Кбит/с
Средняя скорость клиента → сервера	~71 Кбит/с	~497 Кбит/с	~6.46 Кбит/с

На рисунках 11–13 приведены графики сетевой активности по каждому из захваченных сеансов.

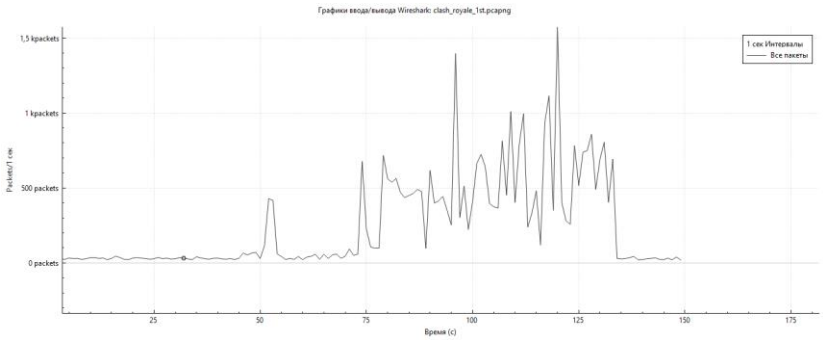

Рисунок 11 — график сетевой активности первой игровой сессии Clash Royale

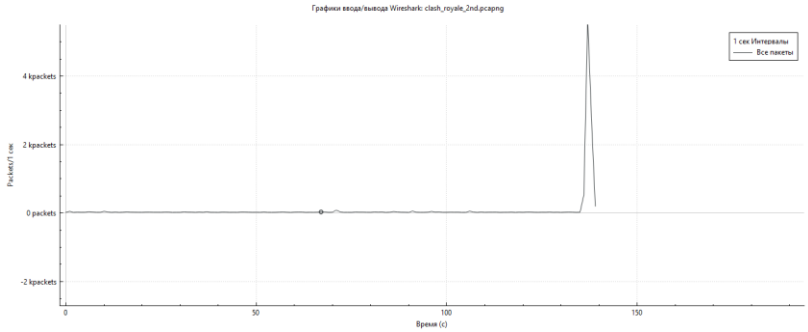

Рисунок 12 — график сетевой активности второй игровой сессии Clash Royale

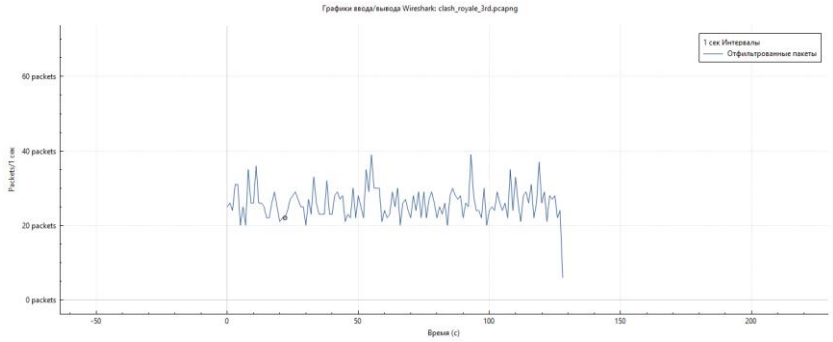

Рисунок 13 — график сетевой активности третьей игровой сессии Clash Royale

График первой игровой сессии демонстрирует крайне характерную картину для централизованной онлайн-игры, использующей TCP в сочетании с TLS 1.2. На протяжении примерно первых 50 секунд график остаётся практически плоским, с передачей менее 10 пакетов в секунду, что свидетельствует о низкой активности в канале передачи данных. Однако

начиная с 50-й секунды наблюдается резкий всплеск трафика: количество переданных пакетов взлетает до 500, а затем — до 1500 пакетов в секунду.

График второй сессии показывает резкий скачок сетевой активности ближе к завершению сессии. Это может быть связано с завершающим этапом матча, когда клиент запрашивает финальную синхронизацию состояния, результаты боя, изменение рейтинга или иные данные. На остальном протяжении график остаётся практически плоским, что говорит о низкой частоте передачи данных — вероятно, из-за шифрования TLS-канала, в рамках которого данные агрегируются.

В третьей сессии наблюдается стабильная передача данных, с регулярной периодичностью пиков (25–50 пакетов/с) и несколькими скачками, возможно отражающими игровые события (ввод игрока, атаки, завершение боёв и т. п.). Это типичный признак активного обмена данными в реальном времени. Отсутствие длительных провалов и стабильность линии указывает на отсутствие потерь, повторных передач или задержек — что говорит об эффективной работе TCP с TLS на короткой дистанции.

По сравнению трёх сессий:

- Первая сессия демонстрирует наиболее интенсивный трафик: почти 30 МБ за 2,5 минуты — это типично для полной игровой партии, включая ввод, бой, завершение и отправку статистики.
- Вторая сессия, несмотря на больший объём от клиента, имеет значительно меньшую нагрузку со стороны сервера.
- Третья — самая «лёгкая» по трафику: объём около 258 КБ за 2 минуты. Возможно, игра завершилась раньше или была быстро прервана.

Архитектура Clash Royale демонстрирует высокую плотность передачи данных в определённые моменты (начало, конец сессии), что характерно для централизованной системы с сильной проверкой действий игрока. Использование TCP и TLS обеспечивает гарантированную доставку и

защищённость передаваемых данных, но также создаёт видимые пиковые скачки и неравномерную нагрузку на сеть, что отчётливо видно на графиках.

4.5 Анализ трафика «Rush Royale»

По результатам работы Wireshark были получены три файла захвата (**rush_royale_1st.pcapng, rush_royale_2nd.pcapng, rush_royale_3rd.pcapng**), отражающие три отдельные игровые сессии в приложении Rush Royale.

Rush Royale так же использует TCP и TLS 1.2 в качестве основных транспортных протоколов, что характерно для архитектуры с авторитетным сервером, контролирующим весь игровой процесс. На стороне клиента происходит минимальная обработка, а сервер принимает решения и отправляет актуальное состояние.

В таблице 4 представлены агрегированные данные по трём отдельным игровым сессиям:

Таблица 4 – Агрегированные данные по трём игровым сессиям Rush Royale

Параметр	**Значение**		
Сессия	1	2	3
Протокол	TCP (с TLS 1.2)	TCP (с TLS 1.2)	TCP (с TLS 1.2)
Всего пакетов	1 448	1 285	596
Общий объём трафика	~392.7 КБ	~176.2 КБ	~37.1 КБ
Пакеты от клиента к серверу	774	670	297
Объём от клиента к серверу	~72.8 КБ	~113.5 КБ	~19.3 КБ
Пакеты от сервера к клиенту	674	615	299
Объём от сервера к клиенту	~319.8 КБ	~62.7 КБ	~17.8 КБ
Длительность сессии	~83.2 секунд	~78.1 секунд	~77.3 секунд
Средняя скорость сервера → клиента	~307 Кбит/с	~6.4 Кбит/с	~1.8 Кбит/с
Средняя скорость клиента → сервера	~70 Кбит/с	~11.6 Кбит/с	~2.0 Кбит/с

На рисунках 14 – 16 приведены графики сетевой активности по каждому из захваченных сеансов.

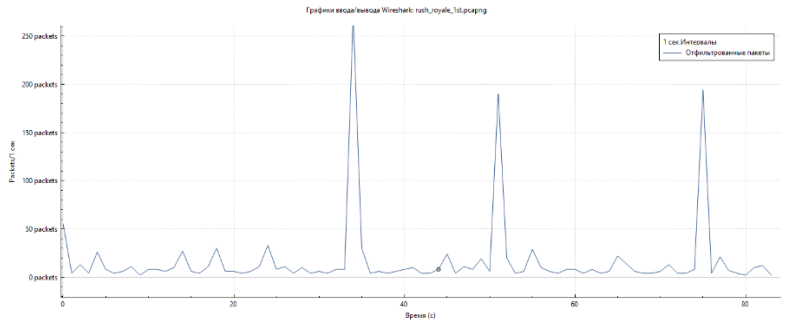

Рисунок 14 — график сетевой активности первой игровой сессии Rush Royale

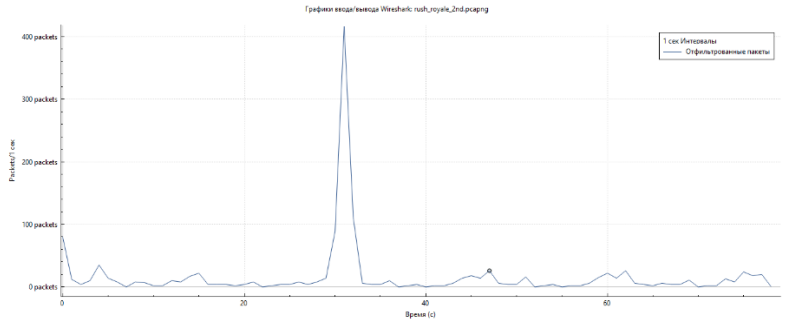

Рисунок 15 — график сетевой активности второй игровой сессии Rush Royale

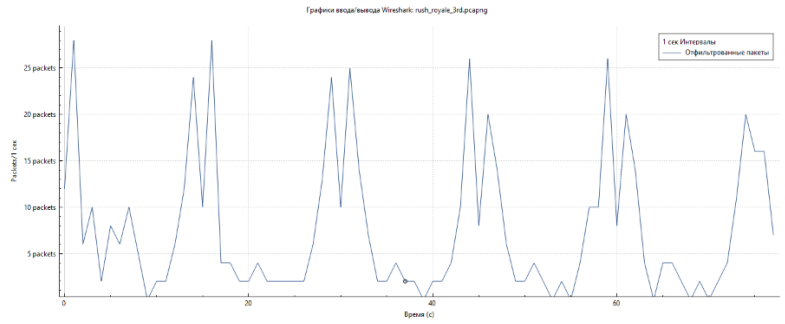

Рисунок 16 — график сетевой активности третьей игровой сессии Rush Royale

График первой сессии показывает равномерную передачу данных с периодическими всплесками до 30-40 пакетов/сек. Такая картина типична для поэтапной синхронизации состояния, где сервер регулярно подтверждает действия игрока. Особенностью является практически симметричное

соотношение входящего и исходящего трафика (307 Кбит/с и 70 Кбит/с), что может указывать на более активную роль клиента в игровом процессе.

На графике второй сессии наблюдается необычный паттерн: после начального обмена данными следует продолжительный период низкой активности (5-10 пакетов/сек), который резко прерывается мощным всплеском на 30-ой секунде сессии.

График третьей сессии демонстрирует минимальную сетевую активность (всего 596 пакетов за 77 секунд). При этом сохраняется баланс между исходящим и входящим трафиком (1.8 Кбит/с и 2.0 Кбит/с), подтверждающий выявленную ранее тенденцию к симметричному обмену.

По сравнению трёх сессий:

- Первая сессия выделяется максимальным объемом передаваемых данных (392.7 КБ), что соответствует полноценной игровой сессии с активным обменом состояниями.
- Вторая сессия показывает аномально высокий объем клиентского трафика (113.5 КБ) при относительно небольшом серверном ответе (62.7 КБ), что может свидетельствовать о реализации механизма "толстого клиента".
- Третья сессия, будучи самой "легкой" (всего 37.1 КБ), демонстрирует характерный для Rush Royale баланс входящего/исходящего трафика.

Анализ трафика Rush Royale показывает менее интенсивный, но более равномерный обмен данными по сравнению с Clash Royale, что свидетельствует о более сбалансированном распределении нагрузки между клиентом и сервером. Хотя игра также использует TCP с TLS 1.2, отсутствие резких всплесков активности и симметричность трафика указывают на иную архитектурную реализацию, где клиент берет на себя часть вычислительной нагрузки, а сервер выступает в роли арбитра. Это снижает пиковую нагрузку на сеть, но требует более сложной клиентской логики.

4.6 Сравнительный анализ

Таблица 5 — Ключевые параметры сетевого взаимодействия

Характеристика	**«Боярский турнир» (QUIC)**	**Clash Royale (TCP+TLS)**	**Rush Royale (TCP+TLS)**
Максимальный трафик	8.31 МБ за 351 сек	29.7 МБ за 150 сек	392.7 КБ за 83 сек
Соотношение трафика (клиент→сервер)	1:8	1:22	1:4
Пиковая скорость (сервер→клиент)	21 Кб/с	1.5 Мбит/с	307 Кбит/с
Частота передачи (п/с)	~100	до 1500	до 40
Задержка	<50 мс	100-300 мс	70-150 мс
Потери пакетов	0%	2-5% потерь	1-3% потерь

Проведенный анализ сетевых характеристик позволяет сделать следующие выводы относительно разработанного проекта "Боярский турнир" в сравнении с коммерческими аналогами. В разработанном решении удалось достичь существенного преимущества в стабильности сетевого соединения благодаря реализации на протоколе QUIC. Это проявляется в полном отсутствии повторных передач пакетов и минимальных задержках (менее 50 мс), что критически важно для жанра RTS. Кроме того, равномерный характер трафика без резких пиков нагрузки свидетельствует об эффективной оптимизации сетевого кода.

По показателям объема передаваемых данных решение демонстрирует сбалансированные результаты — в 3.5 раза экономичнее Clash Royale при сопоставимой продолжительности сессий, хотя и уступает Rush Royale в абсолютной экономии трафика. Это объясняется более сложной игровой механикой и необходимостью частой синхронизации состояния множества юнитов.

ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломной работы была разработана многопользовательская компьютерная игра "Боярский турнир" в жанре RTS, реализованная с использованием языка Rust и игрового движка Bevy. Данное решение демонстрирует новый подход к созданию сетевых игр, сочетающий преимущества безопасного системного программирования с высокой производительностью и модульностью современных игровых движков.

В ходе разработки были успешно пройдены следующие этапы:

1. Рассмотрены ключевые механики жанра RTS и требования к сетевой архитектуре;
2. Проведен анализ существующих RTS-игр (Clash Royale, Rush Royale) с изучением их сетевых решений и особенностей реализации;
3. Сформулированы требования к проекту;
4. Опираясь на требования, был выбран инструментарий, технологии и языки программирования, при помощи которых была решена поставленная задача;
5. Реализованы основные функциональные модули: система юнитов, боевая механика, сетевая синхронизация и базовый интерфейс;
6. Проведен детальный анализ сетевого трафика, подтвердивший эффективность выбранных решений в сравнении с аналогами.

Перспективы развития проекта включают дальнейшее развитие его функциональных возможностей. В первую очередь, планируется внедрение полноценной системы аккаунтов с привязкой игрового прогресса и рейтинговой таблицей, что позволит создать соревновательную составляющую и повысит вовлеченность игроков. Важным направлением развития станет реализация главного меню как центрального узла взаимодействия игрока с системой, обеспечивающего удобный доступ ко всем функциям. Эти улучшения в совокупности позволят вывести проект на качественно новый уровень, значительно расширив его аудиторию и коммерческий потенциал.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. The Rust Programming Language // [Электронный ресурс]: rust-lang.org. – URL: https://doc.rust-lang.org/book
2. Bevy Engine: Official Documentation // [Электронный ресурс]: bevyengine.org – URL: https://bevyengine.org/learn
3. Wireshark User's Guide // [Электронный ресурс]: wireshark.org. – URL: https://www.wireshark.org/docs
4. Clash Royale Technical Analysis // [Электронный ресурс]: supercell.com. – URL: https://supercell.com/en/games/clashroyale
5. Rush Royale Network Architecture // [Электронный ресурс]: my.games – URL: https://rr.my.games/en
6. Rust for Game Development // [Электронный ресурс]: arewegameyet.rs – URL: https://arewegameyet.rs
7. Age of Empires IV Technical Case Study // [Электронный ресурс]: ageofempires.com – URL: https://www.ageofempires.com/games/age-of-empires-iv

ПРИЛОЖЕНИЕ

В приложении приведён код, разработанный в ходе выполнения дипломной работы.

boyar_tournament\src\lib.rs

```
#[cfg(debug_assertions)]
mod dev_tools;
mod scaling;
mod screens;

use bevy::{audio::Volume, prelude::*};

pub struct GamePlugin;

impl Plugin for GamePlugin {
    fn build(&self, app: &mut App) {
        app.add_systems(Startup, spawn_camera);

        app.insert_resource(GlobalVolume {
            volume: Volume::new(0.3),
        });

        app.add_plugins((scaling::plugin, screens::plugin));

        #[cfg(debug_assertions)]
        app.add_plugins(dev_tools::plugin);
    }
}

fn spawn_camera(mut cmd: Commands) {
    cmd.spawn((Camera2d, IsDefaultUiCamera));
}
```

boyar_tournament\src\scaling.rs

```
use bevy::{prelude::*, window::WindowResized};

pub(super) fn plugin(app: &mut App) {
    app.init_resource::<DrawRegion>();
    app.register_type::<DrawRegion>();
    app.register_type::<DynamicScale>();
    app.register_type::<DynamicTransform>();

    app.add_systems(
        PreUpdate,
        (
            update_draw_region,
            update_dynamic_scale,
            update_dynamic_transform,
        )
            .chain(),
    );

    #[cfg(debug_assertions)]
    app.add_systems(Update, draw_draw_region_outline);
}

/// Регион 9x16(состоит из квадратов), внутри которого происходит вся отрисовка
/// Длина и ширина его сторон определяют размер для всех сущностей
#[derive(Resource, Reflect, Default)]
#[reflect(Resource)]
pub struct DrawRegion {
    pub width: f32,
    pub height: f32,
}

fn update_draw_region(
    mut draw_region: ResMut<DrawRegion>,
    mut resize_events: EventReader<WindowResized>,
) {
    if resize_events.is_empty() {
        return;
    }
```

```
    for r_e in resize_events.read() {
        let (aspect_ratio_width, aspect_ratio_height) = (9., 16.);
        let (window_width, window_height) = (r_e.width, r_e.height);

        // При длинном окне, DrawRegion по y на весь экран
        if window_height < window_width / aspect_ratio_width * aspect_ratio_height {
            draw_region.height = window_height;
            draw_region.width = draw_region.height / aspect_ratio_height * aspect_ratio_width;
        } else {
            // При высоком окне, DrawRegion по x на весь экран
            draw_region.width = window_width;
            draw_region.height = draw_region.width / aspect_ratio_width * aspect_ratio_height;
        }
    }
}

/// Компонент для регулирования размеров Sprite
/// Значение scale в компоненте Transform при размере окна игры 1920x1080
#[derive(Component, Reflect)]
#[reflect(Component)]
pub struct DynamicScale(pub f32);

fn update_dynamic_scale(
    mut dynamic_scale: Query<(&mut Transform, &DynamicScale)>,
    draw_region: Res<DrawRegion>,
) {
    for (mut transform, dynamic_scale) in &mut dynamic_scale {
        transform.scale = Vec3::splat(dynamic_scale.0) * draw_region.height / 1080.;
    }
}

/// Расположение сущности в квадратах DrawRegion
#[derive(Component, Reflect, Default)]
#[reflect(Component)]
pub struct DynamicTransform(pub f32, pub f32);

fn update_dynamic_transform(
    mut dynamic_transform: Query<(&mut Transform, &DynamicTransform)>,
    draw_region: Res<DrawRegion>,
) {
    for (mut transform, dynamic_transform) in &mut dynamic_transform {
        transform.translation.x = dynamic_transform.0 * draw_region.width / 9.;
        transform.translation.y = dynamic_transform.1 * draw_region.height / 16.;
    }
}

#[cfg(debug_assertions)]
fn draw_draw_region_outline(
    mut toggle: Local<bool>,
    keyboard: Res<ButtonInput<KeyCode>>,
    mut gizmos: Gizmos,
    draw_region: Res<DrawRegion>,
) {
    use bevy::math::vec2;

    if keyboard.just_pressed(KeyCode::F1) {
        *toggle ^= true;
    }
    if !*toggle {
        return;
    }

    gizmos
        .grid_2d(
            Isometry2d::IDENTITY,
            UVec2::new(9, 16),
            vec2(draw_region.width / 9., draw_region.height / 16.),
            Color::srgb(1., 0., 0.),
        )
        .outer_edges();
}
```

boyar_tournament\src\screens\mod.rs

```
use bevy::prelude::*;

mod gameplay;
mod loading;
mod splash;
mod ui;

pub(super) fn plugin(app: &mut App) {
    app.init_state::<GameState>();
    app.enable_state_scoped_entities::<GameState>();

    app.add_plugins((
        splash::plugin,
        loading::plugin,
        gameplay::plugin,
        ui::plugin,
    ));
}

#[derive(States, Debug, PartialEq, Eq, Clone, Hash, Default, Copy)]
pub enum GameState {
    #[default]
    Splash,
    Loading,
    // Menu,
    Gameplay,
}
```

boyar_tournament\src\screens\ui.rs

```
use bevy::{prelude::*, window::PrimaryWindow};

use crate::scaling::{DrawRegion, DynamicTransform};

pub(super) fn plugin(app: &mut App) {
    app.register_type::<UiInteraction>();
    app.register_type::<UiHitbox>();

    app.add_systems(Update, (update_ui_hitboxes, trigger_on_press));

    #[cfg(debug_assertions)]
    app.add_systems(Update, draw_ui_hitboxes_outline);
}

#[derive(Component, Reflect, Default)]
#[reflect(Component)]
enum UiInteraction {
    #[default]
    None,
    Hovered,
    Pressed,
}

#[derive(Component, Reflect)]
#[reflect(Component)]
#[require(UiInteraction, DynamicTransform)]
// Длина и ширина прямоугольника хитбокса в клетках DynamicTransform
pub struct UiHitbox(pub f32, pub f32);

fn update_ui_hitboxes(
    mut query: Query<(&UiHitbox, &DynamicTransform, &mut UiInteraction)>,
    window: Query<&Window, With<PrimaryWindow>>,
    draw_region: Res<DrawRegion>,
    mouse: Res<ButtonInput<MouseButton>>,
    touch: Res<Touches>,
) {
    let window = window.single();
    let mut press_pos = if let Some(mouse_pos) = window.cursor_position() {
        mouse_pos
    } else {
        let Some(touch_pos) = touch.first_pressed_position() else {
            return;
        };
        touch_pos
    };
```

```
    press_pos.x -= window.width() / 2.;
    press_pos.y -= window.height() / 2.;
    press_pos.y *= -1.;

    let cell_width = draw_region.width / 9.;
    let cell_height = draw_region.height / 16.;
    for (hitbox, transform, mut interaction) in &mut query {
        let hitbox_bottom = (transform.1 - hitbox.1 / 2.) * cell_height;
        let hitbox_top = (transform.1 + hitbox.1 / 2.) * cell_height;
        let hitbox_left = (transform.0 - hitbox.0 / 2.) * cell_width;
        let hitbox_right = (transform.0 + hitbox.0 / 2.) * cell_width;

        if hitbox_bottom <= press_pos.y
            && press_pos.y <= hitbox_top
            && hitbox_left <= press_pos.x
            && press_pos.x <= hitbox_right
        {
            *interaction = UiInteraction::Hovered;

            if mouse.just_pressed(MouseButton::Left) || touch.any_just_pressed() {
                *interaction = UiInteraction::Pressed;
            }
            continue;
        }

        *interaction = UiInteraction::None;
    }
}

#[derive(Event)]
pub struct OnPress;

fn trigger_on_press(
    interaction_query: Query<(Entity, &UiInteraction)>,
    mut commands: Commands,
) {
    for (entity, interaction) in &interaction_query {
        if matches!(interaction, UiInteraction::Pressed) {
            commands.trigger_targets(OnPress, entity);
        }
    }
}

#[cfg(debug_assertions)]
fn draw_ui_hitboxes_outline(
    mut toggle: Local<bool>,
    keyboard: Res<ButtonInput<KeyCode>>,
    mut gizmos: Gizmos,
    draw_region: Res<DrawRegion>,
    query: Query<(&UiHitbox, &DynamicTransform)>,
) {
    use bevy::math::vec2;

    if keyboard.just_pressed(KeyCode::F3) {
        *toggle ^= true;
    }
    if !*toggle {
        return;
    }

    let cell_width = draw_region.width / 9.;
    let cell_height = draw_region.height / 16.;
    for (hitbox, transform) in &query {
        gizmos.rect_2d(
            Isometry2d::from_translation(vec2(
                transform.0 * cell_width,
                transform.1 * cell_height,
            )),
            vec2(hitbox.0 * cell_width, hitbox.1 * cell_height),
            Color::srgb(0., 0., 1.),
        );
    }
}
```

```
use bevy::prelude::*;
use bevy::window::PrimaryWindow;
use bevy_aseprite_ultra::prelude::*;
use bevy_asset_loader::prelude::*;

use crate::{scaling::DynamicScale, screens::GameState};

use crate::scaling::DrawRegion;
use common::ArenaPos;

pub(super) fn plugin(app: &mut App) {
    app.register_type::<ArenaPos>();
    app.register_type::<ArenaHeightOffset>();
    app.register_type::<MouseArenaPos>();

    app.init_resource::<MouseArenaPos>();

    app.configure_loading_state(
        LoadingStateConfig::new(GameState::Loading).load_collection::<ArenaAssets>(),
    );

    app.add_systems(OnEnter(GameState::Gameplay), spawn_arena);

    app.add_systems(
        Update,
        (update_arena_pos, update_mouse_arena_pos).run_if(in_state(GameState::Gameplay)),
    );

    #[cfg(debug_assertions)]
    app.add_systems(
        Update,
        draw_arena_region_outline.run_if(in_state(GameState::Gameplay)),
    );
}

#[derive(AssetCollection, Resource)]
struct ArenaAssets {
    #[asset(path = "arena/winter_arena.aseprite")]
    arena: Handle<Aseprite>,
    #[asset(path = "arena/battle.ogg")]
    battle_music: Handle<AudioSource>,
}

fn spawn_arena(mut cmd: Commands, arena_assets: ResMut<ArenaAssets>) {
    cmd.spawn((
        Name::new("Шаблон арены"),
        AseSpriteSlice {
            name: "winter_arena".into(),
            aseprite: arena_assets.arena.clone(),
        },
        StateScoped(GameState::Gameplay),
        DynamicScale(1.),
        Transform::from_translation(Vec3::ZERO.with_z(-0.5)),
    ));
    cmd.spawn((
        AudioPlayer::new(arena_assets.battle_music.clone()),
        PlaybackSettings::LOOP,
        StateScoped(GameState::Gameplay),
    ));
}

#[derive(Component, Reflect, Clone, Copy)]
#[reflect(Component)]
pub struct ArenaHeightOffset(pub f32);

fn update_arena_pos(
    mut arena_pos: Query<(&mut Transform, &ArenaPos, Option<&ArenaHeightOffset>)>,
    draw_region: Res<DrawRegion>,
) {
    for (mut transform, arena_pos, height_offset) in &mut arena_pos {
        transform.translation.x = arena_pos.0 * draw_region.width / 19.61;
        transform.translation.y =
            arena_pos.1 * draw_region.height / 43.2 + draw_region.height / 13.5;

        // Чем ниже сущность на арене тем "выше" она отображается
        transform.translation.z = transform.translation.y / draw_region.height * -1.;
```

```
        if let Some(height_offset) = height_offset {
            transform.translation.y += height_offset.0 * draw_region.height / 43.2;
        }
    }
}

#[derive(Resource, Reflect, Default)]
#[reflect(Resource)]
pub struct MouseArenaPos(pub Option<ArenaPos>);

fn update_mouse_arena_pos(
    mut mouse_arena_pos: ResMut<MouseArenaPos>,
    window: Query<&Window, With<PrimaryWindow>>,
    draw_region: Res<DrawRegion>,
    touch: Res<Touches>,
) {
    let window = window.single();
    let mut press_pos = if let Some(mouse_pos) = window.cursor_position() {
        mouse_pos
    } else {
        let Some(touch_pos) = touch.first_pressed_position() else {
            return;
        };
        touch_pos
    };
    press_pos.x -= window.width() / 2.;
    press_pos.y -= window.height() / 2.;
    press_pos.y *= -1.;

    press_pos.y -= draw_region.height / 13.5;
    press_pos.x /= draw_region.width / 19.61;
    press_pos.y /= draw_region.height / 43.2;
    if press_pos.x.abs() <= 9. && press_pos.y.abs() <= 16. {
        mouse_arena_pos.0 = Some(ArenaPos(press_pos.x, press_pos.y));
        return;
    }

    mouse_arena_pos.0 = None;
}

#[cfg(debug_assertions)]
fn draw_arena_region_outline(
    mut toggle: Local<bool>,
    keyboard: Res<ButtonInput<KeyCode>>,
    mut gizmos: Gizmos,
    draw_region: Res<DrawRegion>,
) {
    use bevy::math::vec2;

    if keyboard.just_pressed(KeyCode::F2) {
        *toggle ^= true;
    }
    if !*toggle {
        return;
    }

    gizmos
        .grid_2d(
            Isometry2d::from_translation(vec2(0., draw_region.height / 13.5)),
            UVec2::new(18, 32),
            vec2(draw_region.width / 19.61, draw_region.height / 43.2),
            Color::srgb(1., 0.65, 0.),
        )
        .outer_edges();
}
```

boyar_tournament\src\screens\gameplay\deck.rs

```
use bevy::{input::common_conditions::input_just_released, prelude::*};
use bevy_aseprite_ultra::prelude::*;
use bevy_asset_loader::prelude::*;
use bevy_quinnet::client::QuinnetClient;
use common::{ArenaPos, Card, ClientChannel, ClientMessage, PlayerNumber};
use rand::{seq::SliceRandom, thread_rng};
```

```
use crate::{
    scaling::{DynamicScale, DynamicTransform},
    screens::{
        ui::{OnPress, UiHitbox},
        GameState,
    },
};

use super::{arena::MouseArenaPos, spawn_text, FontAssets};

pub(super) fn plugin(app: &mut App) {
    app.register_type::<Deck>();
    app.register_type::<DeckIndex>();
    app.register_type::<SelectedCard>();
    app.register_type::<ElixirCounter>();

    app.init_resource::<SelectedCard>();
    app.init_resource::<ElixirCounter>();

    app.configure_loading_state(
        LoadingStateConfig::new(GameState::Loading).load_collection::<CardsAssets>(),
    );

    use Card::*;
    let mut cards = [
        Rus,
        Musketeer,
        ThreeMusketeers,
        Priest,
        Bats,
        BatHorde,
        Bomber,
        Giant,
    ];
    cards.shuffle(&mut thread_rng());
    app.insert_resource(Deck(cards));

    app.add_systems(
        Update,
        play_card.run_if(
            in_state(GameState::Gameplay).and(
                input_just_released(MouseButton::Left)
                    .or(|touch: Res<Touches>| touch.any_just_released()),
            ),
        ),
    );
    app.add_systems(
        Update,
        update_elixir_counter.run_if(in_state(GameState::Gameplay)),
    );

    app.add_systems(
        OnEnter(GameState::Gameplay),
        (spawn_card_hand, spawn_elixir_counter),
    );
    app.add_observer(update_card_hand);
}

#[derive(AssetCollection, Resource)]
struct CardsAssets {
    #[asset(path = "cards.aseprite")]
    cards: Handle<Aseprite>,
    #[asset(path = "screens/gameplay/card_select.ogg")]
    card_select: Handle<AudioSource>,
}

#[derive(Resource, Reflect)]
#[reflect(Resource)]
struct Deck([Card; 8]);

#[derive(Component, Reflect, Clone, Copy)]
#[reflect(Component)]
struct DeckIndex(u8);

fn spawn_card_hand(
    mut cmd: Commands,
    cards_assets: ResMut<CardsAssets>,
```

```
    deck: Res<Deck>,
    font: Res<FontAssets>,
) {
    for (i, (pos, card)) in [-2.05, -0.22, 1.62, 3.45].iter().zip(deck.0).enumerate() {
        cmd.spawn((
            Name::new(format!("Карта {}", i + 1)),
            AseSpriteSlice {
                name: card.tag(),
                aseprite: cards_assets.cards.clone(),
            },
            DeckIndex(i as _),
            StateScoped(GameState::Gameplay),
            DynamicScale(1.8),
            DynamicTransform(*pos, -6.279),
            UiHitbox(1.8, 2.3),
        ))
        .observe(on_card_select);
    }

    spawn_text(
        &mut cmd,
        "След.",
        font.font.clone(),
        35.,
        Color::srgb(1., 1., 0.),
        1.,
        (-3.8, -5.05),
        GameState::Gameplay,
    );
    cmd.spawn((
        Name::new("Следующая карта"),
        AseSpriteSlice {
            name: deck.0[4].tag(),
            aseprite: cards_assets.cards.clone(),
        },
        DeckIndex(4),
        StateScoped(GameState::Gameplay),
        DynamicScale(0.8),
        DynamicTransform(-3.8, -5.7),
    ));
}

#[derive(Resource, Reflect)]
#[reflect(Resource)]
struct ElixirCounter(u8, Timer);
impl Default for ElixirCounter {
    fn default() -> Self {
        Self(0, Timer::from_seconds(1.5, TimerMode::Repeating))
    }
}

fn spawn_elixir_counter(mut cmd: Commands, font: Res<FontAssets>) {
    spawn_text(
        &mut cmd,
        "0",
        font.font.clone(),
        35.,
        Color::srgb(1., 0., 1.),
        1.,
        (0.7, -7.7),
        GameState::Gameplay,
    );
}

fn update_elixir_counter(
    mut counter: ResMut<ElixirCounter>,
    mut text: Query<&mut Text2d>,
    time: Res<Time>,
) {
    if counter.1.tick(time.delta()).just_finished() {
        if counter.0 < 10 {
            counter.0 += 1;
        }
    }

    for mut text in &mut text {
        if text.0 != "След." {
```

```
            text.0 = counter.0.to_string();
        }
    }
}

trait IntoTag {
    fn tag(&self) -> String;
}
impl IntoTag for Card {
    fn tag(&self) -> String {
        let s = match self {
            Card::Musketeer => "musketeer",
            Card::Rus => "rus",
            Card::ThreeMusketeers => "three_musketeers",
            Card::Priest => "priest",
            Card::Bats => "bats",
            Card::BatHorde => "bat_horde",
            Card::Bomber => "bomber",
            Card::Giant => "giant",
        };
        s.into()
    }
}
trait ElixirCost {
    fn elixir_cost(&self) -> u8;
}
impl ElixirCost for Card {
    fn elixir_cost(&self) -> u8 {
        match self {
            Card::Rus => 3,
            Card::Musketeer => 4,
            Card::ThreeMusketeers => 9,
            Card::Priest => 5,
            Card::Bats => 3,
            Card::BatHorde => 5,
            Card::Bomber => 3,
            Card::Giant => 6,
        }
    }
}

#[derive(Resource, Reflect, Default)]
#[reflect(Resource)]
struct SelectedCard(Option<u8>);

const SELECTED_CARD_SCALE_AMOUNT: f32 = 0.2;

fn on_card_select(
    trigger: Trigger<OnPress>,
    mut selected_card: ResMut<SelectedCard>,
    mut query: Query<(&DeckIndex, &mut DynamicScale)>,
    mut cmd: Commands,
    cards_assets: ResMut<CardsAssets>,
) {
    cmd.spawn((
        AudioPlayer::new(cards_assets.card_select.clone()),
        PlaybackSettings::DESPAWN,
    ));

    let entity = trigger.entity();
    let (&pressed_index, _) = query.get(entity).unwrap();

    if let Some(selected_index) = selected_card.0 {
        for (index, mut scale) in &mut query {
            if index.0 == selected_index {
                scale.0 -= SELECTED_CARD_SCALE_AMOUNT;
                selected_card.0 = None;

                if index.0 == pressed_index.0 {
                    return;
                }
            }
        }
    }

    let (_, mut pressed_scale) = query.get_mut(entity).unwrap();
    selected_card.0 = Some(pressed_index.0);
```

```
    pressed_scale.0 += SELECTED_CARD_SCALE_AMOUNT;
}

fn play_card(
    mouse_pos: Res<MouseArenaPos>,
    selected_card: Res<SelectedCard>,
    mut deck: ResMut<Deck>,
    mut client: ResMut<QuinnetClient>,
    mut cmd: Commands,
    player_num: Res<PlayerNumber>,
    mut elixir: ResMut<ElixirCounter>,
) {
    let Some(mouse_pos) = mouse_pos.0 else {
        return;
    };
    let Some(index) = selected_card.0 else {
        return;
    };
    let index = index as usize;
    let card = deck.0[index];

    let cost = card.elixir_cost();
    if cost > elixir.0 {
        return;
    }
    elixir.0 -= cost;

    // Ставим точку в центр клетки
    let mut x = mouse_pos.0.floor() + 0.5;
    let mut y = mouse_pos.1.floor().clamp(-16., -2.) + 0.5;
    if let PlayerNumber::Two = *player_num {
        x *= -1.;
        y *= -1.;
    }

    client
        .connection_mut()
        .send_message_on(
            ClientChannel::OrderedReliable,
            ClientMessage::PlayCard {
                card,
                placement: ArenaPos(x, y),
            },
        )
        .unwrap();

    // Передвигаем карты в колоде на 1
    deck.0[index] = deck.0[4];
    deck.0[4] = deck.0[5];
    deck.0[5] = deck.0[6];
    deck.0[6] = deck.0[7];
    deck.0[7] = card;

    cmd.trigger(UpdateCardHand);
}

#[derive(Event)]
struct UpdateCardHand;

fn update_card_hand(
    _: Trigger<UpdateCardHand>,
    deck: Res<Deck>,
    mut query: Query<(&DeckIndex, &mut AseSpriteSlice, &mut DynamicScale)>,
    mut selected_card: ResMut<SelectedCard>,
) {
    for (index, mut sprite, mut scale) in &mut query {
        if index.0 == selected_card.0.unwrap() {
            scale.0 -= SELECTED_CARD_SCALE_AMOUNT;
        }

        let card = deck.0[index.0 as usize];
        sprite.name = card.tag();
    }

    selected_card.0 = None;
}
```

boyar_tournament\src\screens\gameplay\mod.rs

```
use bevy::prelude::*;
use bevy_aseprite_ultra::prelude::*;
use bevy_asset_loader::prelude::*;

use crate::scaling::{DynamicScale, DynamicTransform};

use super::GameState;

mod arena;
mod deck;
mod networking;
mod projectiles;
mod units;

pub(super) fn plugin(app: &mut App) {
    app.add_plugins(AsepriteUltraPlugin);

    app.add_plugins((
        arena::plugin,
        networking::plugin,
        units::plugin,
        deck::plugin,
        projectiles::plugin,
    ));

    app.configure_loading_state(
        LoadingStateConfig::new(GameState::Loading).load_collection::<FontAssets>(),
    );
}

#[derive(AssetCollection, Resource)]
struct FontAssets {
    #[asset(path = "Keleti-Regular.ttf")]
    font: Handle<Font>,
}

fn spawn_text(
    cmd: &mut Commands,
    text: &str,
    font: Handle<Font>,
    font_size: f32,
    color: Color,
    dynamic_scale: f32,
    dynamic_transform: (f32, f32),
    state: GameState,
) {
    cmd.spawn((
        Text2d::new(text),
        TextFont::from_font(font.clone()).with_font_size(font_size),
        TextColor(color),
        StateScoped(state),
        DynamicScale(dynamic_scale),
        DynamicTransform(dynamic_transform.0, dynamic_transform.1),
    ))
    .insert(Transform::from_xyz(0., 0., 0.2));

    cmd.spawn((
        Text2d::new(text),
        TextFont::from_font(font.clone()).with_font_size(font_size),
        TextColor(Color::BLACK),
        StateScoped(state),
        DynamicScale(dynamic_scale),
        DynamicTransform(dynamic_transform.0 + 0.03, dynamic_transform.1 - 0.03),
    ))
    .insert(Transform::from_xyz(0., 0., 0.1));
}
```

boyar_tournament\src\screens\gameplay\networking.rs

```
use bevy::{prelude::*, utils::HashMap};
use bevy_quinnet::client::{
    certificate::CertificateVerificationMode, connection::ClientEndpointConfiguration,
    QuinnetClient, QuinnetClientPlugin,
```

```
};
use common::{
    ArenaPos, ClientChannel, Direction, Health, PlayerNumber, ServerMessage, UnitState,
    LOCAL_BIND_IP, SERVER_HOST, SERVER_PORT,
};

use crate::screens::GameState;

use super::{
    projectiles::SpawnProjectile,
    units::{AssociatedTower, SpawnUnit},
};

pub(super) fn plugin(app: &mut App) {
    app.add_plugins(QuinnetClientPlugin::default());

    app.init_resource::<PlayerNumber>();
    app.init_resource::<NetworkMapping>();
    app.register_type::<NetworkMapping>();

    app.add_systems(OnEnter(GameState::Gameplay), start_connection);
    app.add_systems(
        Update,
        handle_server_messages.run_if(in_state(GameState::Gameplay)),
    );
}

fn start_connection(mut client: ResMut<QuinnetClient>) {
    client
        .open_connection(
            ClientEndpointConfiguration::from_ips(SERVER_HOST, SERVER_PORT, LOCAL_BIND_IP, 0),
            CertificateVerificationMode::SkipVerification,
            ClientChannel::channels_config(),
        )
        .unwrap();
}

trait AdjustForPlayer {
    fn adjust_for_player(&self, player_num: PlayerNumber) -> Self;
}
impl AdjustForPlayer for ArenaPos {
    fn adjust_for_player(&self, player_num: PlayerNumber) -> Self {
        match player_num {
            PlayerNumber::One => *self,
            PlayerNumber::Two => ArenaPos(-self.0, -self.1),
        }
    }
}
impl AdjustForPlayer for Direction {
    fn adjust_for_player(&self, player_num: PlayerNumber) -> Self {
        match player_num {
            PlayerNumber::One => *self,
            PlayerNumber::Two => self.opposite(),
        }
    }
}

fn handle_server_messages(
    mut client: ResMut<QuinnetClient>,
    mut player_num: ResMut<PlayerNumber>,
    mut cmd: Commands,
    mut network_mapping: ResMut<NetworkMapping>,
    mut units_query: Query<(&mut ArenaPos, &mut Direction, &mut UnitState, &mut Health)>,
    mut projectiles_query: Query<&mut ArenaPos, Without<UnitState>>,
    towers: Query<&AssociatedTower>,
) {
    while let Some((_, message)) = client
        .connection_mut()
        .try_receive_message::<ServerMessage>()
    {
        match message {
            ServerMessage::StartGame(n) => *player_num = n,
            ServerMessage::SpawnUnit {
                server_entity,
                unit,
                pos,
                owner,
```

```
            } => {
                unit.spawn(
                    server_entity,
                    pos.adjust_for_player(*player_num),
                    owner,
                    &mut cmd,
                );
            }
            ServerMessage::SpawnProjectile {
                server_entity,
                projectile,
                attacker,
                receiver,
                pos,
            } => projectile.spawn(
                server_entity,
                attacker,
                receiver,
                pos.adjust_for_player(*player_num),
                &mut cmd,
            ),
            ServerMessage::Despawn(server_entity) => {
                let Some(entity) = network_mapping.remove(&server_entity) else {
                    continue;
                };
                if let Ok(tower) = towers.get(entity) {
                    cmd.entity(tower.0).despawn();
                }
                cmd.entity(entity).despawn();
            }
            ServerMessage::SyncEntities { units, projectiles } => {
                for (server_entity, pos, direction, state, health) in &units {
                    let Some(&entity) = network_mapping.get(server_entity) else {
                        continue;
                    };
                    let (mut p, mut d, mut s, mut h) = units_query.get_mut(entity).unwrap();
                    *p = pos.adjust_for_player(*player_num);
                    *d = direction.adjust_for_player(*player_num);
                    *s = *state;
                    *h = *health;
                }

                for (server_entity, pos) in &projectiles {
                    let Some(&entity) = network_mapping.get(server_entity) else {
                        continue;
                    };
                    let mut p = projectiles_query.get_mut(entity).unwrap();
                    *p = pos.adjust_for_player(*player_num);
                }
            }
        }
    }
}

#[derive(Resource, Reflect, Default, Deref, DerefMut)]
#[reflect(Resource)]
// Сопоставление Entity сервера и клиента
pub struct NetworkMapping(HashMap<Entity, Entity>);
```

boyar_tournament\src\screens\gameplay\projectiles\mod.rs

```
use bevy::prelude::*;
use bomb::SpawnBomb;
use bullet::SpawnBullet;
use common::{ArenaPos, Projectile};
use fireball::SpawnFireball;

use crate::screens::GameState;

use super::arena::ArenaHeightOffset;

mod bomb;
mod bullet;
mod fireball;
```

```
pub(super) fn plugin(app: &mut App) {
    app.add_plugins((bullet::plugin, fireball::plugin, bomb::plugin));

    app.add_systems(
        Update,
        update_projectile_height.run_if(in_state(GameState::Gameplay)),
    );
}

pub(super) trait SpawnProjectile {
    fn spawn(
        &self,
        entity: Entity,
        attacker: Entity,
        receiver: Entity,
        pos: ArenaPos,
        cmd: &mut Commands,
    );
}

impl SpawnProjectile for Projectile {
    fn spawn(
        &self,
        entity: Entity,
        attacker: Entity,
        receiver: Entity,
        pos: ArenaPos,
        cmd: &mut Commands,
    ) {
        match self {
            Projectile::Bullet => cmd.trigger(SpawnBullet(entity, attacker, receiver, pos)),
            Projectile::Fireball => {
                cmd.trigger(SpawnFireball(entity, attacker, receiver, pos))
            }
            Projectile::Bomb => cmd.trigger(SpawnBomb(entity, attacker, receiver, pos)),
        }
    }
}

#[derive(Component)]
struct ProjectileTargets(Entity, Entity, f32);

fn update_projectile_height(
    projectiles: Query<(Entity, &ProjectileTargets)>,
    mut positions: Query<(&ArenaPos, &mut ArenaHeightOffset)>,
) {
    for (entity, targets) in &projectiles {
        let Ok((&attacker_pos, &attacker_height)) = positions.get(targets.0) else {
            continue;
        };
        let Ok((&receiver_pos, &receiver_height)) = positions.get(targets.1) else {
            continue;
        };
        let Ok((self_pos, mut self_height)) = positions.get_mut(entity) else {
            continue;
        };

        let dist_to_attacker = self_pos.distance(&attacker_pos);
        let dist_to_receiver = self_pos.distance(&receiver_pos);
        let progress = dist_to_attacker / (dist_to_attacker + dist_to_receiver);

        let height = attacker_height.0 + progress * (receiver_height.0 - attacker_height.0);
        self_height.0 = height + targets.2;
    }
}
```

boyar_tournament\src\screens\gameplay\units\mod.rs

```
use archer_tower::SpawnArcherTower;
use bat::SpawnBat;
use bevy::prelude::*;
use bevy_aseprite_ultra::prelude::{AnimationState, AseSpriteAnimation, Aseprite};
use bomber::SpawnBomber;
use common::{ArenaPos, Direction, Health, PlayerNumber, Unit, UnitState};
use king_tower::SpawnKingTower;
```

```
use musketeer::SpawnMusketeer;
use priest::SpawnPriest;
use rus::SpawnRus;
use giant::SpawnGiant;

use crate::screens::GameState;

mod archer_tower;
mod bat;
mod bomber;
mod king_tower;
mod musketeer;
mod priest;
mod rus;
mod giant;

pub(super) fn plugin(app: &mut App) {
    app.register_type::<Direction>();
    app.register_type::<UnitState>();
    app.register_type::<Health>();

    app.add_systems(
        PreUpdate,
        manage_animation.run_if(in_state(GameState::Gameplay)),
    );

    app.add_plugins((
        archer_tower::plugin,
        king_tower::plugin,
        rus::plugin,
        musketeer::plugin,
        bat::plugin,
        priest::plugin,
        bomber::plugin,
        giant::plugin,
    ));
}

fn manage_animation(
    mut animation_query: Query<(
        &Direction,
        &UnitState,
        &mut AseSpriteAnimation,
        &mut AnimationState,
    )>,
    aseprites: Res<Assets<Aseprite>>,
) {
    for (direction, state, mut animation, mut animation_state) in animation_query.iter_mut() {
        match state {
            UnitState::Idle => {
                let tag_meta = aseprites
                    .get(animation.aseprite.id())
                    .unwrap()
                    .tags
                    .get(direction.tag())
                    .unwrap();
                let start_frame = tag_meta.range.start();
                animation_state.current_frame = *start_frame;

                animation.animation.tag = Some(direction.tag().into());
            }
            UnitState::Moving => {
                let tag_meta = aseprites
                    .get(animation.aseprite.id())
                    .unwrap()
                    .tags
                    .get(direction.tag())
                    .unwrap();
                let start_frame = tag_meta.range.start();
                let end_frame = tag_meta.range.end();
                if animation_state.current_frame < *start_frame
                    || animation_state.current_frame > *end_frame
                {
                    animation_state.current_frame = *start_frame;
                }

                animation.animation.tag = Some(direction.tag().into());
```

```
            }
            UnitState::Attacking => {
                let mut tag = String::from(direction.tag());
                tag.push('a');

                let tag_meta = aseprites
                    .get(animation.aseprite.id())
                    .unwrap()
                    .tags
                    .get(&tag)
                    .unwrap();
                let start_frame = tag_meta.range.start();
                let end_frame = tag_meta.range.end();
                if animation_state.current_frame < *start_frame
                    || animation_state.current_frame > *end_frame
                {
                    animation_state.current_frame = *start_frame;
                }

                animation.animation.tag = Some(tag);
            }
        }
    }
}

/// Требуется для привязки юнита к башне
#[derive(Component)]
pub struct AssociatedTower(pub Entity);

pub(super) trait SpawnUnit {
    fn spawn(
        &self,
        entity: Entity,
        pos: ArenaPos,
        player_num: PlayerNumber,
        cmd: &mut Commands,
    );
}

impl SpawnUnit for Unit {
    fn spawn(
        &self,
        entity: Entity,
        pos: ArenaPos,
        player_num: PlayerNumber,
        cmd: &mut Commands,
    ) {
        match self {
            Unit::ArcherTower => cmd.trigger(SpawnArcherTower(entity, pos, player_num)),
            Unit::KingTower => cmd.trigger(SpawnKingTower(entity, pos, player_num)),
            Unit::Rus => cmd.trigger(SpawnRus(entity, pos, player_num)),
            Unit::Musketeer => cmd.trigger(SpawnMusketeer(entity, pos, player_num)),
            Unit::Bat => cmd.trigger(SpawnBat(entity, pos, player_num)),
            Unit::Priest => cmd.trigger(SpawnPriest(entity, pos, player_num)),
            Unit::Bomber => cmd.trigger(SpawnBomber(entity, pos, player_num)),
            Unit::Giant => cmd.trigger(SpawnGiant(entity, pos, player_num)),
        }
    }
}

trait SpawnDirection {
    fn spawn_direction(self, player_num: Self) -> Direction;
}
impl SpawnDirection for PlayerNumber {
    fn spawn_direction(self, player_num: PlayerNumber) -> Direction {
        use PlayerNumber::*;
        match (self, player_num) {
            (One, One) | (Two, Two) => Direction::Up,
            _ => Direction::Down,
        }
    }
}

trait IntoTag {
    fn tag(&self) -> &'static str;
}
impl IntoTag for Direction {
```

```
    fn tag(&self) -> &'static str {
        match self {
            Direction::Up => "u",
            Direction::Down => "d",
            Direction::Left => "l",
            Direction::Right => "r",
        }
    }
}
impl IntoTag for UnitState {
    fn tag(&self) -> &'static str {
        match self {
            UnitState::Idle => "",
            UnitState::Moving => "",
            UnitState::Attacking => "a",
        }
    }
}
```

common\src\lib.rs

```
use std::{
    net::Ipv4Addr,
    ops::{AddAssign, Sub, SubAssign},
};

use bevy::{math::vec2, prelude::*};
use bevy_quinnet::shared::channels::{ChannelId, ChannelType, ChannelsConfiguration};
use serde::{Deserialize, Serialize};

//pub const SERVER_HOST: Ipv4Addr = Ipv4Addr::new(178, 71, 57, 127);
pub const SERVER_HOST: Ipv4Addr = Ipv4Addr::LOCALHOST;
pub const LOCAL_BIND_IP: Ipv4Addr = Ipv4Addr::UNSPECIFIED;
pub const SERVER_PORT: u16 = 50505;

#[derive(
    Debug,
    Component,
    Reflect,
    Serialize,
    Deserialize,
    Clone,
    Copy,
    Default,
    PartialEq,
    PartialOrd,
)]
#[reflect(Component)]
pub struct ArenaPos(pub f32, pub f32);
impl Sub for ArenaPos {
    type Output = Self;

    fn sub(self, rhs: Self) -> Self::Output {
        ArenaPos(self.0 - rhs.0, self.1 - rhs.1)
    }
}
impl SubAssign for ArenaPos {
    fn sub_assign(&mut self, rhs: Self) {
        self.0 -= rhs.0;
        self.1 -= rhs.1;
    }
}
impl AddAssign for ArenaPos {
    fn add_assign(&mut self, rhs: Self) {
        self.0 += rhs.0;
        self.1 += rhs.1;
    }
}
impl ArenaPos {
    pub fn normalize(&self) -> Self {
        let v = vec2(self.0, self.1).normalize();
        ArenaPos(v.x, v.y)
    }
    pub fn mul(&self, n: f32) -> Self {
        ArenaPos(self.0 * n, self.1 * n)
    }
```

```
    pub fn distance(&self, rhs: &Self) -> f32 {
        ((self.0 - rhs.0).powi(2) + (self.1 - rhs.1).powi(2)).sqrt()
    }
    pub fn direction(&self, rhs: &Self) -> Self {
        if self.distance(rhs) < 0.01 {
            return ArenaPos(0., 0.);
        }
        (*rhs - *self).normalize()
    }
}

#[derive(Debug, Component, Serialize, Deserialize, Clone, Copy, Reflect)]
#[reflect(Component)]
pub enum Card {
    Rus,
    Musketeer,
    ThreeMusketeers,
    Priest,
    Bats,
    BatHorde,
    Bomber,
    Giant,
}

#[derive(Debug, Component, Serialize, Deserialize, Clone, Copy)]
pub enum Unit {
    ArcherTower,
    KingTower,
    Rus,
    Musketeer,
    Bat,
    Priest,
    Bomber,
    Giant,
}

#[derive(Debug, Component, Serialize, Deserialize, Clone, Copy)]
pub enum Projectile {
    Bullet,
    Fireball,
    Bomb,
}

#[derive(Component, Reflect, Serialize, Deserialize, Clone, Copy)]
#[reflect(Component)]
pub struct Health(pub u16, pub u16); // Текущее и максимальное здоровье
impl Health {
    // Конкретное значение указывается в сервере, default для спауна на клиенте
    pub fn new(amount: u16) -> Self {
        Health(amount, amount)
    }
}
impl Default for Health {
    fn default() -> Self {
        Self::new(100)
    }
}

#[derive(Component, Debug, Serialize, Deserialize, Clone, Copy, Reflect, Default)]
#[reflect(Component)]
pub enum Direction {
    #[default]
    Up,
    Down,
    Left,
    Right,
}
impl Direction {
    pub fn opposite(&self) -> Self {
        use Direction::*;
        match self {
            Up => Down,
            Down => Up,
            Left => Right,
            Right => Left,
        }
    }
```

```
}

#[derive(Component, Debug, Serialize, Deserialize, Clone, Copy, Reflect, Default)]
#[reflect(Component)]
pub enum UnitState {
    #[default]
    Idle, // Для построек, а также отправляется клиенту для юнитов в стане
    Moving, // Для всего остального
    Attacking,
}

#[derive(Serialize, Deserialize)]
pub enum ClientMessage {
    PlayCard { card: Card, placement: ArenaPos },
}

#[derive(
    Resource,
    Component,
    Serialize,
    Deserialize,
    Debug,
    Clone,
    Copy,
    Hash,
    Eq,
    PartialEq,
    Default,
)]
pub enum PlayerNumber {
    #[default]
    One, // Игрок "снизу"
    Two, // Игрок "сверху"
}

#[derive(Serialize, Deserialize)]
pub enum ServerMessage {
    StartGame(PlayerNumber),
    SpawnUnit {
        server_entity: Entity,
        unit: Unit,
        pos: ArenaPos,
        owner: PlayerNumber,
    },
    SpawnProjectile {
        server_entity: Entity,
        projectile: Projectile,
        attacker: Entity,
        receiver: Entity,
        pos: ArenaPos,
    },
    Despawn(Entity),
    SyncEntities {
        units: Vec<(Entity, ArenaPos, Direction, UnitState, Health)>,
        projectiles: Vec<(Entity, ArenaPos)>,
    },
}

#[repr(u8)]
pub enum ClientChannel {
    // Разыгрывание карт, и мб вызов эмоутов
    OrderedReliable,
}
impl From<ClientChannel> for ChannelId {
    fn from(value: ClientChannel) -> Self {
        value as _
    }
}
impl ClientChannel {
    pub fn channels_config() -> ChannelsConfiguration {
        ChannelsConfiguration::from_types(vec![ChannelType::OrderedReliable]).unwrap()
    }
}

#[repr(u8)]
pub enum ServerChannel {
    // Инициализация
```

```
    OrderedReliable,
    // Рассылка действий игроков
    UnorderedReliable,
    // Синхронизация юнитов
    Unreliable,
}
impl From<ServerChannel> for ChannelId {
    fn from(value: ServerChannel) -> Self {
        value as _
    }
}
impl ServerChannel {
    pub fn channels_config() -> ChannelsConfiguration {
        ChannelsConfiguration::from_types(vec![
            ChannelType::OrderedReliable,
            ChannelType::UnorderedReliable,
            ChannelType::Unreliable,
        ])
        .unwrap()
    }
}
```

server\src\ai.rs

```
use bevy::prelude::*;
use bevy_quinnet::server::QuinnetServer;
use common::{
    ArenaPos, Health, PlayerNumber, Projectile, ServerChannel, ServerMessage, UnitState,
};

use crate::{projectiles::SpawnProjectile, units::UnitType};

pub(super) fn plugin(app: &mut App) {
    app.add_systems(
        FixedUpdate,
        (
            (
                update_attacks,
                update_unit_state,
                update_stun_timers,
                update_movement,
            ),
            check_health,
        )
            .chain(),
    );
}

pub enum AttackType {
    Melee(u16), // Урон
    Ranged(Projectile),
}

pub enum AttackTargetType {
    Ground,
    All,
}

#[derive(Component)]
pub struct Attack {
    pub target: Option<Entity>,
    pub a_type: AttackType,
    pub t_type: AttackTargetType,
    pub cooldown_timer: Timer,
    pub range: f32,
}
impl Attack {
    pub fn new(a_type: AttackType, targets: AttackTargetType, cd: f32, range: f32) -> Self {
        Self {
            target: None,
            a_type,
            t_type: targets,
            cooldown_timer: Timer::from_seconds(cd, TimerMode::Repeating),
            range,
        }
```

```
    }
}

fn update_attacks(
    mut attacks: Query<(Entity, &mut Attack)>,
    mut units: Query<(&ArenaPos, &mut Health)>,
    time: Res<Time>,
    mut cmd: Commands,
) {
    for (attacker, mut attack) in &mut attacks {
        // target есть только в UnitState::Attacking
        let Some(receiver) = attack.target else {
            attack.cooldown_timer.reset();
            continue;
        };
        let Ok((_, mut health)) = units.get_mut(receiver) else {
            // Все мертвы
            attack.target = None;
            continue;
        };
        if !attack.cooldown_timer.tick(time.delta()).just_finished() {
            continue;
        }

        match attack.a_type {
            AttackType::Melee(damage) => health.0 = health.0.saturating_sub(damage),
            AttackType::Ranged(projectile) => {
                let (pos, _) = units.get(attacker).unwrap();
                projectile.spawn(attacker, receiver, *pos, &mut cmd)
            }
        }
    }
}

fn check_health(
    query: Query<(Entity, &Health)>,
    mut server: ResMut<QuinnetServer>,
    mut cmd: Commands,
) {
    for (entity, health) in &query {
        if health.0 == 0 {
            cmd.entity(entity).despawn();
            server
                .endpoint_mut()
                .broadcast_message_on(
                    ServerChannel::OrderedReliable,
                    ServerMessage::Despawn(entity),
                )
                .unwrap();
        }
    }
}

#[derive(Component)]
pub struct Movement {
    pub target: Option<Entity>,
    pub speed: f32,
}
impl Movement {
    pub fn new(speed: f32) -> Self {
        Self {
            target: None,
            speed,
        }
    }
}
fn update_movement(
    mut query: Query<(Entity, &mut Movement), Without<StunnedTimer>>,
    states: Query<&UnitState>,
    mut positions: Query<&mut ArenaPos>,
    time: Res<Time>,
) {
    for (entity, mut movement) in &mut query {
        if let Ok(state) = states.get(entity) {
            let UnitState::Moving = state else {
                continue;
            };
```

```
        }
        // target устанавливается в update_unit_state
        let Some(target) = movement.target else {
            continue;
        };
        let Ok(&target_pos) = positions.get(target) else {
            movement.target = None;
            continue;
        };
        let Ok(mut self_pos) = positions.get_mut(entity) else {
            continue;
        };
        let direction = self_pos.direction(&target_pos);
        *self_pos += direction.mul(movement.speed * time.delta_secs());
    }
}

#[derive(Component)]
// Таймер добавляется при спавне юнитов
pub struct StunnedTimer(pub Timer);
impl Default for StunnedTimer {
    fn default() -> Self {
        Self(Timer::from_seconds(1.5, TimerMode::Once))
    }
}
fn update_stun_timers(
    mut query: Query<(Entity, &mut StunnedTimer)>,
    mut cmd: Commands,
    time: Res<Time>,
) {
    for (entity, mut timer) in &mut query {
        if timer.0.tick(time.delta()).just_finished() {
            cmd.entity(entity).remove::<StunnedTimer>();
        }
    }
}

#[derive(Component)]
pub struct AggroRadius(pub f32);

fn update_unit_state(
    mut attackers: Query<
        (
            Entity,
            &mut UnitState,
            &mut Attack,
            Option<&AggroRadius>,
            Option<&mut Movement>,
        ),
        Without<StunnedTimer>,
    >,
    receivers: Query<(Entity, &ArenaPos, &PlayerNumber, &UnitType)>,
    towers: Query<(Entity, &ArenaPos, &PlayerNumber), Without<Movement>>,
) {
    'outer: for (self_entity, mut state, mut attack, aggro_radius, mut movement) in
        &mut attackers
    {
        match *state {
            UnitState::Idle | UnitState::Moving => {
                let (_, self_pos, self_player_numer, _) = receivers.get(self_entity).unwrap();

                for (entity, pos, player_number, unit_type) in &receivers {
                    if self_player_numer == player_number {
                        // Своих не бьём
                        continue;
                    }
                    if let (AttackTargetType::Ground, UnitType::Air) =
                        (&attack.t_type, unit_type)
                    {
                        continue;
                    }

                    if self_pos.distance(pos) <= attack.range {
                        *state = UnitState::Attacking;
                        attack.target = Some(entity);
                        continue 'outer;
                    }
```

```
                // У всего что не является постройкой есть и AggroRadius и Movement
                if let (Some(aggro_radius), Some(movement)) =
                    (aggro_radius, movement.as_mut())
                {
                    if self_pos.distance(pos) <= aggro_radius.0 {
                        movement.target = Some(entity);
                        continue 'outer;
                    }
                }
            }

            // Если никого нет вблизи, двигаемся к ближайшей башне
            let Some(movement) = movement.as_mut() else {
                continue;
            };
            let mut nearest_tower = None;
            let mut minimal_distance = 1000.;
            for (tower_entity, tower_pos, tower_player_number) in &towers {
                let distance = self_pos.distance(tower_pos);
                if self_player_numer == tower_player_number || distance > minimal_distance
                {
                    continue;
                }

                minimal_distance = distance;
                nearest_tower = Some(tower_entity);
            }
            let Some(nearest_tower) = nearest_tower else {
                // Башен врага не осталось, игра должна закончиться
                continue;
            };
            movement.target = Some(nearest_tower);
        }
        UnitState::Attacking => {
            if let Some(target) = attack.target {
                let (_, self_pos, _, _) = receivers.get(self_entity).unwrap();

                if let Ok((_, pos, _, _)) = receivers.get(target) {
                    if self_pos.distance(pos) > attack.range {
                        match movement.as_mut() {
                            Some(_) => *state = UnitState::Moving,
                            None => *state = UnitState::Idle,
                        }
                    }
                    continue;
                };

                continue;
            }

            match movement.as_mut() {
                Some(_) => *state = UnitState::Moving,
                None => *state = UnitState::Idle,
            }
        }
    }
  }
}
```

server\src\networking.rs

```
use core::f32;

use bevy::{prelude::*, utils::HashMap};
use bevy_quinnet::{
    server::{
        certificate::CertificateRetrievalMode, ConnectionEvent, QuinnetServer,
        QuinnetServerPlugin, ServerEndpointConfiguration,
    },
    shared::ClientId,
};
use common::{
    ArenaPos, Card, ClientMessage, Direction, Health, PlayerNumber, ServerChannel,
    ServerMessage, Unit, UnitState, LOCAL_BIND_IP, SERVER_HOST, SERVER_PORT,
```

```
};

use crate::{
    ai::{Attack, Movement, StunnedTimer},
    units::{Giant, SpawnUnit},
};

pub(super) fn plugin(app: &mut App) {
    app.add_plugins(QuinnetServerPlugin::default());

    app.init_resource::<Lobby>();
    app.add_systems(Startup, start_listening);
    app.add_systems(Update, (handle_connection_events, handle_client_messages));

    app.add_systems(FixedPostUpdate, sync_entities);
}

fn start_listening(mut server: ResMut<QuinnetServer>) {
    server
        .start_endpoint(
            ServerEndpointConfiguration::from_ip(LOCAL_BIND_IP, SERVER_PORT),
            CertificateRetrievalMode::GenerateSelfSigned {
                server_hostname: SERVER_HOST.to_string(),
            },
            ServerChannel::channels_config(),
        )
        .unwrap();
}

#[derive(Resource, Default, Deref, DerefMut)]
pub struct Lobby(HashMap<ClientId, PlayerNumber>);

fn handle_connection_events(
    mut connection_events: EventReader<ConnectionEvent>,
    mut lobby: ResMut<Lobby>,
    mut server: ResMut<QuinnetServer>,
    mut cmd: Commands,
) {
    let lobby_len = lobby.len() as u8;
    for client in connection_events.read() {
        if lobby_len >= 2 {
            server.endpoint_mut().disconnect_client(client.id).unwrap();
            continue;
        }
        use PlayerNumber::*;

        let player_num = match lobby_len {
            0 => One,
            1 => Two,
            _ => unreachable!(),
        };
        lobby.insert(client.id, player_num);

        if lobby.len() == 2 {
            // Отправить каждому игроку его PlayerNumber
            for (client_id, player_num) in lobby.iter() {
                server
                    .endpoint_mut()
                    .send_message_on(
                        *client_id,
                        ServerChannel::OrderedReliable,
                        ServerMessage::StartGame(*player_num),
                    )
                    .unwrap();
            }

            Unit::ArcherTower.spawn(ArenaPos(-5.5, -9.5), One, &mut cmd);
            Unit::KingTower.spawn(ArenaPos(0., -13.), One, &mut cmd);
            Unit::ArcherTower.spawn(ArenaPos(5.5, -9.5), One, &mut cmd);

            Unit::ArcherTower.spawn(ArenaPos(-5.5, 9.5), Two, &mut cmd);
            Unit::KingTower.spawn(ArenaPos(0., 13.), Two, &mut cmd);
            Unit::ArcherTower.spawn(ArenaPos(5.5, 9.5), Two, &mut cmd);
        }
    }
}
```

```
fn handle_client_messages(
    mut server: ResMut<QuinnetServer>,
    lobby: Res<Lobby>,
    mut cmd: Commands,
) {
    let endpoint = server.endpoint_mut();
    for client_id in endpoint.clients() {
        while let Some((_, message)) =
            endpoint.try_receive_message_from::<ClientMessage>(client_id)
        {
            let player_num = lobby.get(&client_id).unwrap();
            match message {
                ClientMessage::PlayCard { card, placement } => match card {
                    Card::Rus => Unit::Rus.spawn(placement, *player_num, &mut cmd),
                    Card::Musketeer => Unit::Musketeer.spawn(placement, *player_num, &mut cmd),
                    Card::ThreeMusketeers => {
                        let ArenaPos(x, y) = placement;
                        Unit::Musketeer.spawn(ArenaPos(x, y + 0.8), *player_num, &mut cmd);
                        Unit::Musketeer.spawn(ArenaPos(x + 0.8, y), *player_num, &mut cmd);
                        Unit::Musketeer.spawn(ArenaPos(x - 0.8, y), *player_num, &mut cmd);
                    }
                    Card::Bats => {
                        let ArenaPos(x, y) = placement;
                        Unit::Bat.spawn(ArenaPos(x, y + 0.8), *player_num, &mut cmd);
                        Unit::Bat.spawn(ArenaPos(x + 0.8, y), *player_num, &mut cmd);
                        Unit::Bat.spawn(ArenaPos(x - 0.8, y), *player_num, &mut cmd);
                    }
                    Card::BatHorde => {
                        let ArenaPos(x, y) = placement;
                        Unit::Bat.spawn(ArenaPos(x + 0.5, y + 0.5), *player_num, &mut cmd);
                        Unit::Bat.spawn(ArenaPos(x + 0.8, y), *player_num, &mut cmd);
                        Unit::Bat.spawn(ArenaPos(x + 0.5, y - 0.5), *player_num, &mut cmd);
                        Unit::Bat.spawn(ArenaPos(x - 0.5, y - 0.5), *player_num, &mut cmd);
                        Unit::Bat.spawn(ArenaPos(x - 0.8, y), *player_num, &mut cmd);
                        Unit::Bat.spawn(ArenaPos(x - 0.5, y + 0.5), *player_num, &mut cmd);
                    }
                    Card::Priest => Unit::Priest.spawn(placement, *player_num, &mut cmd),
                    Card::Bomber => Unit::Bomber.spawn(placement, *player_num, &mut cmd),
                    Card::Giant => Unit::Giant.spawn(placement, *player_num, &mut cmd),
                },
            }
        }
    }
}

trait DefaultDirection {
    fn default_direction(&self) -> Direction;
}
impl DefaultDirection for PlayerNumber {
    fn default_direction(&self) -> Direction {
        match self {
            PlayerNumber::One => Direction::Up,
            PlayerNumber::Two => Direction::Down,
        }
    }
}

fn calc_direction(direction: &ArenaPos) -> Direction {
    let mut angle = direction.0.acos() * 180. / f32::consts::PI;
    if direction.1 < 0. {
        angle = -angle + 360.;
    }

    match angle {
        0.0..20. | 340.0..360. => Direction::Right,
        20.0..160. => Direction::Up,
        160.0..200. => Direction::Left,
        200.0..340. => Direction::Down,
        _ => Direction::Right,
    }
}

fn sync_entities(
    units: Query<(
        Entity,
        &ArenaPos,
        &UnitState,
```

```
        &Attack,
        Option<&Movement>,
        &PlayerNumber,
        &Health,
        Option<&StunnedTimer>,
    )>,
    giants: Query<(
        Entity,
        &ArenaPos,
        &UnitState,
        &Giant,
        &Movement,
        &PlayerNumber,
        &Health,
        Option<&StunnedTimer>,
    )>,
    projectiles: Query<(Entity, &ArenaPos), Without<PlayerNumber>>,
    positions: Query<&ArenaPos>,
    mut server: ResMut<QuinnetServer>,
) {
    let mut u = Vec::new();
    for (entity, pos, state, attack, movement, player_num, health, stun) in &units {
        let direction = match state {
            UnitState::Idle => player_num.default_direction(),
            UnitState::Moving => {
                let movement = movement.unwrap();
                match movement.target {
                    Some(m) => {
                        let Ok(target_pos) = positions.get(m) else {
                            continue;
                        };
                        calc_direction(&pos.direction(target_pos))
                    }
                    None => player_num.default_direction(),
                }
            }
            UnitState::Attacking => match attack.target {
                Some(a) => {
                    let Ok(target_pos) = positions.get(a) else {
                        continue;
                    };
                    calc_direction(&pos.direction(target_pos))
                }
                None => player_num.default_direction(),
            },
        };
        let mut state = *state;
        if let Some(_) = stun {
            state = UnitState::Idle
        }
        u.push((entity, *pos, direction, state, *health));
    }
    for (entity, pos, state, giant, movement, player_num, health, stun) in &giants {
        let direction = match state {
            UnitState::Idle => player_num.default_direction(),
            UnitState::Moving => match movement.target {
                Some(m) => {
                    let Ok(target_pos) = positions.get(m) else {
                        continue;
                    };
                    calc_direction(&pos.direction(target_pos))
                }
                None => player_num.default_direction(),
            },
            UnitState::Attacking => match giant.target {
                Some(a) => {
                    let Ok(target_pos) = positions.get(a) else {
                        continue;
                    };
                    calc_direction(&pos.direction(target_pos))
                }
                None => player_num.default_direction(),
            },
        };
        let mut state = *state;
        if let Some(_) = stun {
            state = UnitState::Idle
```

```
        }
        u.push((entity, *pos, direction, state, *health));
    }

    let mut p = Vec::new();
    for (entity, position) in &projectiles {
        p.push((entity, *position));
    }

    server
        .endpoint_mut()
        .broadcast_message_on(
            ServerChannel::Unreliable,
            ServerMessage::SyncEntities {
                units: u,
                projectiles: p,
            },
        )
        .unwrap();
}
```

server\src\projectiles\mod.rs

```
use bevy::prelude::*;
use bomb::SpawnBomb;
use bullet::SpawnBullet;
use common::{ArenaPos, Projectile};
use fireball::SpawnFireball;

mod bomb;
mod bullet;
mod fireball;

pub(super) fn plugin(app: &mut App) {
    app.add_plugins((bullet::plugin, fireball::plugin, bomb::plugin));
}

#[derive(Component)]
struct ProjectileRadius(pub f32);

pub(super) trait SpawnProjectile {
    fn spawn(&self, attacker: Entity, receiver: Entity, pos: ArenaPos, cmd: &mut Commands);
}

impl SpawnProjectile for Projectile {
    fn spawn(&self, attacker: Entity, receiver: Entity, pos: ArenaPos, cmd: &mut Commands) {
        match self {
            Projectile::Bullet => cmd.trigger(SpawnBullet(attacker, receiver, pos)),
            Projectile::Fireball => cmd.trigger(SpawnFireball(attacker, receiver, pos)),
            Projectile::Bomb => cmd.trigger(SpawnBomb(attacker, receiver, pos)),
        }
    }
}
```

server\src\units\mod.rs

```
use archer_tower::SpawnArcherTower;
use bat::SpawnBat;
use bevy::prelude::*;
use bomber::SpawnBomber;
use common::{ArenaPos, PlayerNumber, Unit};
use giant::SpawnGiant;
use king_tower::SpawnKingTower;
use musketeer::SpawnMusketeer;
use priest::SpawnPriest;
use rus::SpawnRus;
pub use giant::Giant;

mod archer_tower;
mod bat;
mod bomber;
mod giant;
mod king_tower;
mod musketeer;
mod priest;
mod rus;
```

```
pub(super) fn plugin(app: &mut App) {
    app.add_plugins((
        archer_tower::plugin,
        king_tower::plugin,
        rus::plugin,
        musketeer::plugin,
        bat::plugin,
        priest::plugin,
        bomber::plugin,
        giant::plugin,
    ));
}

#[derive(Component)]
pub enum UnitType {
    Air,
    Ground,
}

#[derive(Component)]
pub struct Hitbox(pub f32);

pub(super) trait SpawnUnit {
    fn spawn(&self, pos: ArenaPos, player_num: PlayerNumber, cmd: &mut Commands);
}

impl SpawnUnit for Unit {
    fn spawn(&self, pos: ArenaPos, player_num: PlayerNumber, cmd: &mut Commands) {
        match self {
            Unit::ArcherTower => cmd.trigger(SpawnArcherTower(pos, player_num)),
            Unit::KingTower => cmd.trigger(SpawnKingTower(pos, player_num)),
            Unit::Rus => cmd.trigger(SpawnRus(pos, player_num)),
            Unit::Musketeer => cmd.trigger(SpawnMusketeer(pos, player_num)),
            Unit::Bat => cmd.trigger(SpawnBat(pos, player_num)),
            Unit::Priest => cmd.trigger(SpawnPriest(pos, player_num)),
            Unit::Bomber => cmd.trigger(SpawnBomber(pos, player_num)),
            Unit::Giant => cmd.trigger(SpawnGiant(pos, player_num)),
        }
    }
}
```