Санкт-Петербургский государственный университет

***МИРХАЕВ Ильнар Ильдарович***

**Выпускная квалификационная работа**

***«Разработка прототипа соревновательной многопользовательской игры»***

Уровень образования: бакалавриат

Направление *09.03.03 «Прикладная информатика»*

Основная образовательная программа СВ.5078

Профиль *«Прикладная информатика в области искусств и гуманитарных наук»*

**Научный руководитель:**

Канд. физ.-мат. наук, доцент, кафедра информационных систем в

искусстве и гуманитарных

науках, СПбГУ

Щербаков Павел Петрович

**Консультанты:**

Ст. преподаватель, кафедра информационных систем в искусстве и гуманитарных науках, СПбГУ,

Логдачева Елена Викторовна;

Старший 3D-художник,  
«TRACE studio»,

Коротких Алексей Евгеньевич

**Рецензент:**

генеральный директор, Общество с ограниченной ответственностью «БХВ-Петербург»,  
Шишигин Игорь Владимирович

Санкт-Петербург

2023

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ   
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ  
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ФАКУЛЬТЕТ ИСКУССТВ**

**Кафедра информационных систем в искусстве и гуманитарных науках**

УТВЕРЖДАЮ

Заведующий кафедрой

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ЗАДАНИЕ

по подготовке выпускной квалификационной работы студента Мирхаева Ильнара Ильдаровича

1. Тема работы: Разработка прототипа соревновательной многопользовательской игры
2. Срок сдачи студентом законченной работы: июнь 2023
3. Исходные данные к работе: оригинальная идея
4. План-график выполнения квалификационной работы:

|  |  |  |  |
| --- | --- | --- | --- |
| **Номера и содержание этапов работы** | **Плановая дата сдачи** | **Фактическая дата сдачи** | **Дата выдачи рецензии** |
| 1. Планирование архитектуры игры, изучение и выбор технологий для реализации | 10.2022 |  |  |
| 2. Разработка основных игровых механик | 10.2022 – 11.2022 |  |  |
| 3. Разработка сетевого взаимодействия и отладка прототипа | 11.2022 – 02.2023 |  |  |
| 4. Разработка архитектуры проекта | 02.2023 – 03.2023 |  |  |
| 5. Доработка игровых механик с поправкой на архитектуру и сетевую составляющую | 03.2023 – 04.2023 |  |  |
| 6. Финальная отладка прототипа игры | 04.2023 – 05.2023 |  |  |

Консультанты по работе: ст. преподаватель Логдачева Елена Викторовна, старший 3D-художник Коротких Алексей Евгеньевич

Руководитель от кафедры: канд. физ.-мат. наук, доцент, Щербаков Павел Петрович

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(должность, Фамилия Имя Отчество, подпись)

Задание принял к исполнению\_\_\_\_\_\_\_\_\_\_\_\_\_ Подпись студента\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (дата)

**АННОТАЦИЯ**

выпускной квалификационной работы

Мирхаева Ильнара Ильдаровича

«Разработка прототипа соревновательной многопользовательской игры»

Целью работы является разработка и отладка прототипа многопользовательской игры.

Данная ВКР состоит из шести глав, разбитых на параграфы. В первой главе проведен анализ паттернов и приемов построения игровой архитектуры. Вторая глава посвящена изучению типов сетевого взаимодействия, а третья – изучению существующих библиотек для реализации сетевого взаимодействия на движке Unity. В четвертой главе описан процесс разработки основных механик игры с использованием инструментов движка. В пятой главе описана реализация сетевого взаимодействия, и продемонстрированы отличия между двумя представленными решениями. Последняя глава описывает применение выбранных в первой главе подходов к построению архитектуры.

Автор работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

подпись (фамилия, имя, отчество)

Руководитель работы \_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

подпись (фамилия, имя, отчество)

**ANNOTATION**

Graduation thesis

Mirkhaev Ilnar Ildarovich

«Development of a prototype of a competitive multiplayer game»

The aim of the work is to develop and debug a prototype of a multiplayer game.

This thesis consists of six chapters divided into paragraphs. In the first chapter the author analyzes patterns and techniques for creating a game architecture. The second and third chapters are devoted to the study of networking types and existing libraries for implementing networking in Unity game engine. The fourth chapter describes the process of developing the main mechanics of the game using the tools of the engine. In the fifth chapter the author describes the implementation of networking and demonstrates the differences between the two presented solutions. The last chapter describes applied architectural patterns and the resulting game architecture.

Student \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

signature (full name)

Supervisor \_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

signature (full name)

**ОГЛАВЛЕНИЕ**

[Определения 8](#_Toc135403782)

[Введение 10](#_Toc135403783)

[1. Анализ подходов к организации архитектуры игры. 12](#_Toc135403784)

[1.1. Важность архитектуры при разработке игр 12](#_Toc135403785)

[1.2. ECS 12](#_Toc135403786)

[1.2.1. Общие сведения 12](#_Toc135403787)

[1.2.2. Преимущества и недостатки 13](#_Toc135403788)

[1.3. Паттерны MVC, MVP, MVVM 14](#_Toc135403789)

[1.3.1. Общие сведения 15](#_Toc135403790)

[1.3.2. Применение в разработке игр 15](#_Toc135403791)

[1.3.3. Выводы 15](#_Toc135403792)

[1.4. Inversion of Control и Dependency Injection 16](#_Toc135403793)

[1.4.1. Общие определения 16](#_Toc135403794)

[1.4.2. Принцип работы 16](#_Toc135403795)

[1.4.3. Преимущества и недостатки 17](#_Toc135403796)

[1.5. Выбор архитектуры 18](#_Toc135403797)

[2. Изучение сетевого взаимодействия в многопользовательских играх. 19](#_Toc135403798)

[2.1. Peer-to-Peer (P2P) 19](#_Toc135403799)

[2.2. Выделенный сервер 20](#_Toc135403800)

[2.3. Клиент-хост 21](#_Toc135403801)

[3. Изучение существующих средств для сетевого взаимодействия на движке Unity 22](#_Toc135403802)

[3.1. Netcode for GameObjects от Unity 22](#_Toc135403803)

[3.2. Fusion от Photon 24](#_Toc135403804)

[4. Разработка базовых механик игры 25](#_Toc135403805)

[4.1. Необходимые механики 25](#_Toc135403806)

[4.2. Выбор средств разработки 25](#_Toc135403807)

[4.3. Использование инструментов Unity 26](#_Toc135403808)

[4.4. Реализация механик 26](#_Toc135403809)

[4.4.1. Обработка ввода игрока 26](#_Toc135403810)

[4.4.2. Управление персонажем и камерой 28](#_Toc135403811)

[4.4.3. Стрельба и попадание 31](#_Toc135403812)

[5. Реализация сетевого взаимодействия 33](#_Toc135403813)

[5.1. Netcode for GameObjects 33](#_Toc135403814)

[5.1.1. Основные компоненты 33](#_Toc135403815)

[5.1.2. Подключение 34](#_Toc135403816)

[5.1.3. Возникшие проблемы 35](#_Toc135403817)

[5.2. Photon Fusion 36](#_Toc135403818)

[5.2.1. Основные компоненты Photon 36](#_Toc135403819)

[5.2.2. Обработка ввода 37](#_Toc135403820)

[5.2.3. Разработка механики здоровья и получения урона 40](#_Toc135403821)

[5.2.4. Выводы 41](#_Toc135403822)

[6. Организация архитектуры проекта в Unity 42](#_Toc135403823)

[6.1. Использование VContainer 42](#_Toc135403824)

[6.2. Способы внедрения зависимостей 43](#_Toc135403825)

[6.3. Конечный вид архитектуры 44](#_Toc135403826)

[Заключение 46](#_Toc135403827)

[Список использованных источников 47](#_Toc135403828)

# Определения

* Игровая механика – способ взаимодействия с игровыми объектами в рамках установленных ограничений игры, изменяющий её состояние, влияющее на принятие игроком последующих решений;
* Игровой движок – базовое программное обеспечение компьютерной игры;
* Unity – игровой движок, межплатформенная среда разработки компьютерных игр, разработанная американской компанией Unity Technologies;
* Сеттинг – среда, в которой происходит действие; место, время и условия действия;
* Шаблон проектирования или паттерн (англ. design pattern) в разработке программного обеспечения — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста;
* Библиотека – сборник подпрограмм или объектов, используемых для разработки программного обеспечения;
* Класс – модель для создания объектов определённого типа, описывающая их структуру и определяющая алгоритмы (функции или методы) для работы с этими объектами;
* Метод – это функция или процедура, принадлежащая какому-то классу или объекту;
* Интерфейс – структура программы/синтаксиса, определяющая отношение с объектами, объединенными только некоторым поведением;
* Компонент – отдельный набор функциональности, который можно добавить объекту, чтобы наделить его этой функциональностью;
* Фреймворк – программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта;

# Введение

На протяжении последних нескольких лет игровая индустрия является одной из наиболее прибыльных и быстрорастущих индустрий развлечений в мире, обгоняя по годовому доходу кинопрокат и музыкальный бизнес [1]. Для создания успешных видеоигр необходима команда из высококвалифицированных специалистов, в их числе игровые программисты и разработчики, в чьи обязанности входит разработка различных аспектов игры: игровые механики, интеграция графики и звукового сопровождения, программирование пользовательского интерфейса, сетевые коммуникации и т.д. Благодаря разработчикам игровые миры могут существовать и функционировать.

В качестве темы выпускной квалификационной работы была выбрана разработка прототипа многопользовательской соревновательной игры. В целях разработки используется игровой движок Unity от компании Unity Technologies. Прототип помогает сформировать четкое видение будущего проекта на ранних стадиях разработки и, в отличие от финальной сборки, необязательно должен быть идеальным в исполнении, поэтому его производство менее затратное.

В игре предполагается состязание игроков друг с другом с целью одержать верх над противником, будь то другой игрок или вражеская команда. Сеттинг – Средневековье с элементами фэнтези. В прототипе должен быть реализован персонаж-лучник.

Целью ВКР является разработка и отладка игрового прототипа многопользовательской игры. Для достижения данной цели были поставлены следующие задачи:

1. Анализ подходов к организации архитектуры игры
2. Изучение типов сетевого взаимодействия в многопользовательских играх
3. Изучение средств для разработки сетевого взаимодействия на игровом движке Unity
4. Разработка основных механик игры
5. Реализация сетевого взаимодействия в игре
6. Организация архитектуры проекта в Unity

По окончанию работы подразумевается, что разработанный прототип можно будет протестировать с разных устройств, подключившись к одной игровой комнате. В рамках выполнения поставленных задач изучены и применены на практике инструменты движка Unity, архитектурные паттерны и принципы программирования, сетевое взаимодействие в играх и инструменты для его реализации. Полученных знаний и навыков будет достаточно, чтобы считаться начинающим игровым разработчиком на Unity.

# Анализ подходов к организации архитектуры игры.

## Важность архитектуры при разработке игр

Архитектура играет важную роль в определении структуры, функциональности и эффективности игры. Хорошо спроектированная архитектура помогает организовать и структурировать все компоненты игры, что делает разработку и поддержку проекта более удобными.

Также она облегчает масштабирование игры в будущем. Когда игра развивается и добавляются новые функции и механики, правильная архитектура позволяет легко внедрять эти изменения без необходимости переписывать большую часть кода.

Несмотря на все преимущества наличия архитектуры, к ее выбору необходимо подходить с умом, так как некоторые архитектуры могут быть сложными для понимания и внедрения, и на работу с ними может уйти больше ресурсов, чем на разработку без каких-либо архитектурных решений.

## ECS

### Общие сведения

ECS (Entity Component System) – это паттерн проектирования, который расшифровывается как «Сущность-Компонент-Система», в основном используется при создании игр. Он основан на сущностях, состоящих из компонентов, и системах, которые работают с компонентами сущностей.

«Сущности» (Entity) являются объектами общего назначения, по сути контейнерами, не обладающими свойствами и выступающими в качестве хранилищ для компонентов. «Компоненты» (Component) – это блоки данных, определяющие всевозможные свойства сущностей. Все эти данные, сгруппированные в контейнеры, обрабатываются логикой, существующей в виде «Систем» - классов с определенными методами.

Примерный процесс работы ECS можно описать так: каждый кадр системы обращаются к некому игровому пространству, где хранятся сущности, и запрашивают только те, которые имеют компоненты, с которыми работает эта система (Рис. 1).

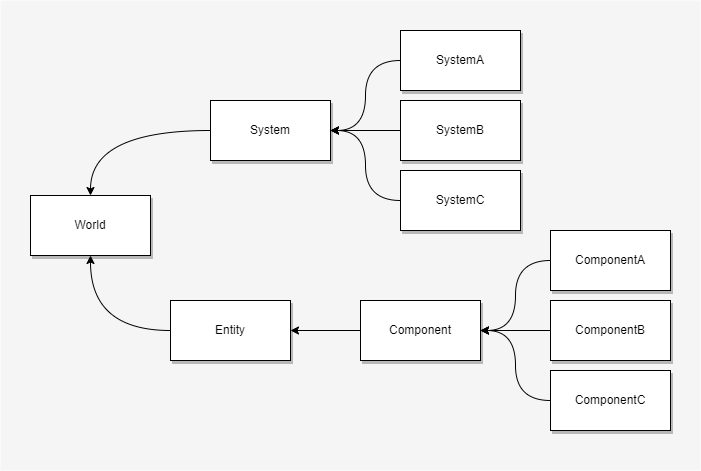


Рисунок - Условная схема связей ECS

К примеру, система, отвечающая за воздействие сил гравитации на объекты, попытается получить сущности, которые имеют компонент гравитации.

### Преимущества и недостатки

Данный паттерн не зависит от какого-либо движка и может быть реализован множеством способов. Все сущности, компоненты и системы должны где-то храниться и каким-то образом инициализироваться – все это уже является особенностями реализации ECS-решения для конкретного движка.

У такой архитектуры множество преимуществ:

* **Композиция против наследования.** В случае множественного наследования классов может наследоваться много ненужной функциональности. В случае ECS она появляется или исчезает при добавлении или удалении компонента, так как сущности – это композиции из компонентов, своего рода конструктор из блоков.
* **Разделение логики и данных.** Возможность менять логику (менять системы, удалять или добавлять компоненты), не ломая данные. Т.е. можно в любой момент отключить группу систем, отвечающих за определенную функциональность, все остальное продолжит работать, и это не затронет данные.
* **Упрощается игровой цикл.** Игровой цикл разбивается на системы, обращение к которым происходит из одной точки в коде. Данные обрабатываются «потоком» в системе, независимо от движка.

Пожалуй, из минусов подхода можно выделить следующие:

* **Высокий порог вхождения.** Нужно учиться думать, проектировать и писать код по-другому. Хоть идею подхода достаточно просто описать, для того чтобы научиться его правильно использовать, требуется много практики. Также многие паттерны проектирования в ECS реализуются совсем по-другому.
* **Порядок вызова систем влияет на работу всей игры.** Обычно, системы зависимы друг от друга, порядок их выполнения задается списком, и они выполняются в этом порядке. Если случайно поменять порядок, то все станет работать не так, как ожидалось.

## Паттерны MVC, MVP, MVVM

В семействе паттернов MVx обычно выделяют три основных паттерна: Model-View-Controller (MVC), Model-View-Presenter (MVP), Model-View-ViewModel (MVVM). MVC – это фундаментальный паттерн, который нашел применение во многих технологиях и дал развитие новым. Он был придуман как архитектурное решение, которое позволяло бы отделить графический интерфейс от бизнес-логики. Таким образом эти паттерны состоят из трех частей, которые и дают им название.

### Общие сведения

Под моделью (Model), обычно понимается часть, представляющая данные и функциональную бизнес-логику приложения. Она отвечает за получение, обработку и хранение данных. Модель должна быть полностью независима от остальных частей продукта.

Представление (View) отображает данные пользователю и обрабатывает пользовательский ввод. Оно получает данные от модели и отображает их пользователю в удобном формате. Представление не содержит бизнес-логики и не изменяет данные.

Контроллер (Controller), презентер (Presenter) или вью-модель (ViewModel) в свою очередь управляют взаимодействием между моделью и представлением.

### Применение в разработке игр

Если рассматривать реализацию данных паттернов на движке Unity, то в роли представления будут выступать компоненты MonoBehaviour, так как они крепятся к игровым объектам и «живут» непосредственно в игровом мире и могут отображать данные и реагировать на ввод игрока. В качестве модели необходимо реализовать свой класс с данными и логикой. Посредником между моделью и представлением будет выступать обычный класс, который в зависимости от выбранного паттерна будет по-разному коммуницировать с моделью и представлением.

### Выводы

MVx паттерны создавались для решения задачи взаимодействия пользовательского интерфейса с моделью, и в рамках разработки приложений они подходят лишь для того, чтобы организовать интерфейсный слой приложения. Но помимо этого, в разработке игр они также подходят для отделения всей визуальной составляющей от игровой логики, будь то интерфейс или игровые объекты.

## Inversion of Control и Dependency Injection

### Общие определения

Инверсия управления (Inversion of Control, IoC) – это принцип программирования, используемый для уменьшения связности кода. Его суть в том, что каждый компонент системы должен быть как можно более изолированным от других, не полагаясь в своей работе на детали конкретной реализации других компонентов.

IoC-контейнер – это какая-либо библиотека или фреймворк, который позволяет упростить и автоматизировать написание кода с использованием данного подхода.

Очень часто термины IoC и DI используются вместе. DI или внедрение зависимостей (Dependency Injection) – это одна из реализаций принципа IoC, при которой объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

### Принцип работы

Принцип работы DI-фреймворка следующий: есть объект, который использует некоторый интерфейс (зависит от него), и есть несколько классов-реализаций этого интерфейса. Объекту неважно, какой класс использовать, он просто нуждается в какой-нибудь реализации. Т.н. контейнер, или инжектор, знает о нуждающемся объекте и передает ему зависимость, используемую в данной конфигурации.

Рассмотрим на абстрактном примере. Автомобилю для работы необходим двигатель. В контейнере указывается вид двигателя, который используется в конкретном случае. Контейнер знает о зависимостях, необходимых машине, и передает их реализации (Рис. 2).

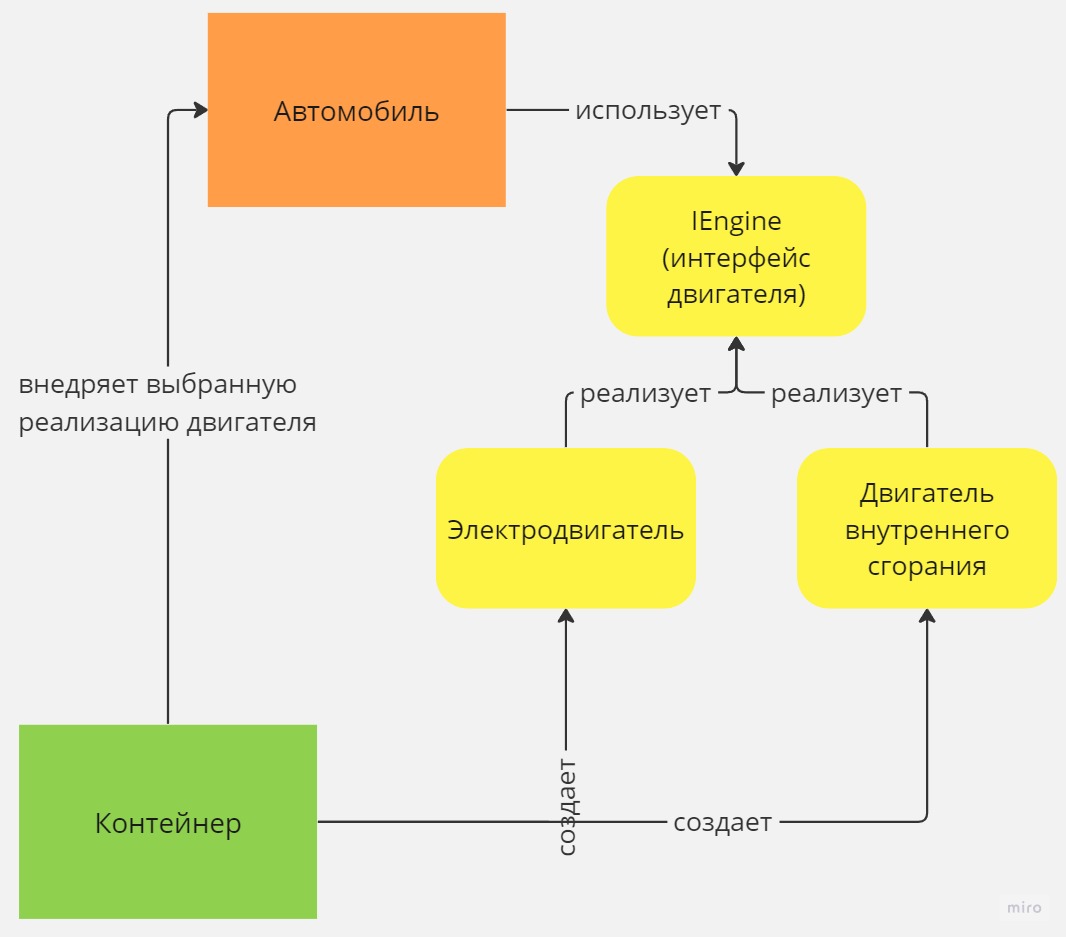


Рисунок - Схема работы DI

### Преимущества и недостатки

Из плюсов использования инверсии управления и внедрения зависимостей можно выделить следующее:

* Каждый модуль ответственен за ту задачу, для которой он был создан;
* Модулям не нужно знать, как работают другие системы и модули, вместо этого они работают «по контракту», из-за чего становятся более гибкими, тестируемыми и легкими в поддержании;
* При замене используемого модуля не возникает побочных эффектов;
* Разработчики могут работать над разными классами, использующими друг друга, нуждаясь только в знании интерфейса-контракта, через который классы должны «общаться». Так, например, пишутся сторонние плагины для программ.

Минусов меньше, и они незначительны:

* Разрабатываемая система становится зависимой от используемого фреймворка, т.к. каждый фреймворк чем-то отличается от других;
* Использование рефлексии. Рефлексия – процесс, во время которого программа может отслеживать и модифицировать собственную структуру и поведение во время выполнения. Хоть этот процесс очень медленный, по сравнению с выполнением остального кода, в DI он необходим только при запуске приложения или системы.

## Выбор архитектуры

Были проанализированы основные паттерны и подходы к организации архитектуры. Так как на данный момент создаваемый прототип не запланирован для выпуска в магазины, ожидаемая функциональность небольшая. Поэтому можно ограничиться использованием принципа IoC и его реализацией путем DI, использование ECS или MVx замедлило бы разработку прототипа на данном этапе.

# Изучение сетевого взаимодействия в многопользовательских играх.

Для понимания того, как устроены многопользовательские игры, необходимо разобраться в том, как игроки, или клиенты, взаимодействуют друг с другом и сервером, если таковой имеется.

## Peer-to-Peer (P2P)

Одноранговая сеть (англ. peer-to-peer – равный к равному) основана на равноправии участников. Обычно в такой сети отсутствует выделенный сервер, а каждый узел (peer) как является клиентом, так и выполняет функции сервера. Такая организация позволяет сохранять работоспособность сети при любом количестве и любом сочетании доступных узлов. Участниками сети являются все узлы (Рис. 3).

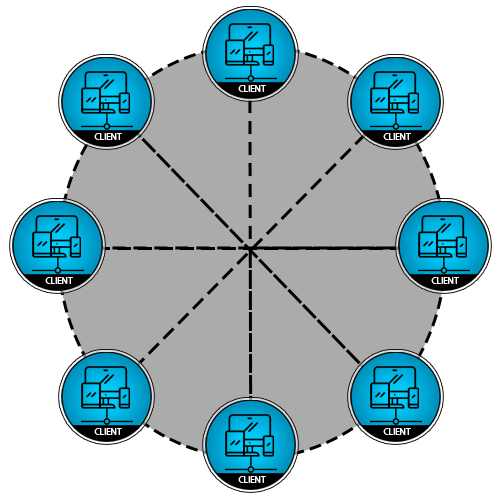


Рисунок - Схема связей в P2P сети [4]

Одноранговая сеть может быть реализована по-разному:

* Пара устройств могут быть связаны напрямую, например, по USB или LAN-проводу;
* Постоянная инфраструктура, связывающая несколько компьютеров в небольшом помещении по проводам;
* Сеть большого масштаба, в которой специальные протоколы и приложения устанавливают прямые связи между пользователями через Интернет.

## Выделенный сервер

Для крупномасштабных игр с большим количеством игроков обычно используют множество серверов, распределенных по миру. Сервер является авторитетным источником событий в многопользовательской видеоигре. Он передает данные о внутреннем состоянии игры, чтобы подключенные клиенты могли поддерживать точную версию игрового мира для отображения игрокам. Устройство, предоставляющее сервисы формата «клиент-сервер» в режиме сервера, называется хостом (Рис. 4).

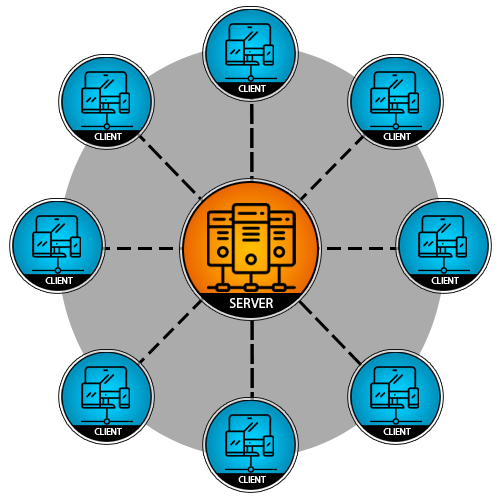


Рисунок - Схема связей в сети с отдельным сервером [4]

Этот вариант лучше P2P тем, что единственно верным источником информации является сервер, в то время как в одноранговой сети все клиенты являются равноправными, из-за чего синхронизация игрового мира крайне сложна.

Но для содержания отдельного сервера требуется много ресурсов, что является критическим недостатком для небольших команд или одиночных разработчиков игр.

## Клиент-хост

Другим популярным решением является использование одного из компьютеров в сети в качестве сервера. Клиент, который дополнительно выполняет функции сервера, называется клиент-хостом. Клиент посылает команды клиент-хосту, тот выполняет их на своем компьютере и посылает результат обратно этому клиенту и всем остальным.

У данного метода имеются свои недостатки:

* Клиент-хосту приходиться общаться с удаленными игроками через домашнее Интернет-соединение;
* Снижается производительность, так как компьютер, на котором запущен сервер, также создает изображение игры;
* Клиент-хост имеет преимущество перед другими клиентами, так как у него отсутствует задержка соединения;
* Сервер и игра «вылетают», когда клиент-хост покидает игру.

С другой стороны, данный способ соединения достаточно удобен и прост в реализации, так как нет необходимости заводить отдельные сервера – каждый может «хостить» игры на своем компьютере.

В силу вышесказанного в данной работе будет реализован именно этот тип соединения.

# Изучение существующих средств для сетевого взаимодействия на движке Unity

За время существования игрового движка Unity появилось множество готовых решений для реализации сетевого взаимодействия между игроками. Рассмотрим самые популярные из них.

## Netcode for GameObjects от Unity

Компания Unity недавно выпустила собственную библиотеку для сетевого взаимодействия под названием Netcode for GameObjects (NGO). NGO заменяет своего устаревшего предшественника, другую библиотеку от Unity – UNet.

NGO обладает достаточно подробной документацией, позволяет легко передавать данные об игровых объектах и мире всем игрокам. Благодаря этому можно сосредоточиться на создании своей игры, а не на низкоуровневых протоколах и сетевых инфраструктурах [4].

Вместе с документацией Unity предоставляет примеры игр, реализованных при помощи NGO (Рис. 5). Проекты с кодом размещены в открытом доступе на сайте GitHub.



Рисунок - Boss Room, учебный проект от Unity, разработанный на NGO [5]

Помимо NGO, Unity предлагает разработчикам дополнительные сервисы для создания многопользовательских игр. Среди них:

* **Relay**. Упрощает присоединение клиентов к хосту при помощи кода доступа. Игроки обращаются к публичному удаленному релей-серверу (relay server), который затем соединяет их с хостом. Это избавляет от проблем со сменой сети и IP-адреса, преобразованием IP-адресов (механизм NAT), файрволами между игроками. Также благодаря посреднику, которым выступает релей-сервер, игрокам не нужно знать IP-адреса друг друга, что повышает безопасность и конфиденциальность.
* **Lobby**. Работает похожим образом Relay, основное отличие в том, что Lobby использует публичные или приватные комнаты (лобби), в которых можно собрать игроков вместе перед игровой сессией. Скорее всего похожая технология используется в популярной игре Among Us, которая, кстати, разработана на Unity (Рис. 6).



Рисунок - Лобби перед игровой сессией. Among Us (2018)

Использование NGO вместе с сервисами Unity представляется достаточно удобным, так как не нужно реализовывать различные протоколы преобразования сетевых адресов.

## Fusion от Photon

Компания Photon предоставляет несколько похожих решений.   
PUN (Photon Unity Networking) – когда-то самое популярное решение для Unity, является устаревшим. Также считается, что у PUN проблемы с производительностью, расширяемостью и функциональностью.

На смену PUN пришел Fusion, новая библиотека с повышенной производительностью [6], а также некоторыми полезными функциями:

* **Использование релей-сервера по умолчанию**. Photon бесплатно предоставляет релей-сервер для вашего приложения, который допускает до 20 одновременных игроков;
* **Прогнозирование на стороне клиента**. Дает игрокам мгновенный ответ на их действия, не лишая сервера полномочий, даже в условиях высокой задержки и потери сети;
* **Интерполяция снэпшотов**. Снэпшот – это статический срез состояния игровой системы в определенный момент времени. Fusion позволяет плавно переходить между ними даже при плохом соединении.
* **Компенсация задержки**. Делает синхронизацию игры между игроками более точной. Избавляет от типичной проблемы, когда игрок целится в противника, который на самом деле уже находится на другом месте.

В целом решение от Photon похоже на NGO, но в Fusion встроены функции, которые обычно разработчикам приходится разрабатывать самим, из-за чего решение кажется более преимущественным.

# Разработка базовых механик игры

## Необходимые механики

Из идеи игры можно выделить следующие основные механики, которые должны быть реализованы в первую очередь:

* Обработка ввода игрока;
* Управление персонажем, вращение камеры;
* Атака персонажа, а именно стрельба из лука;
* Попадание снарядов в объекты окружения или в игрока.

## Выбор средств разработки

Для реализации игровых механик в Unity можно программировать на языке C# (c-sharp, «си-шарп») или использовать визуальное программирование.

Визуальное программирование – это способ создания программ или алгоритмов, использующий визуальные элементы интерфейса, такие как иконки и блоки, вместо текста кода (Рис. 7).

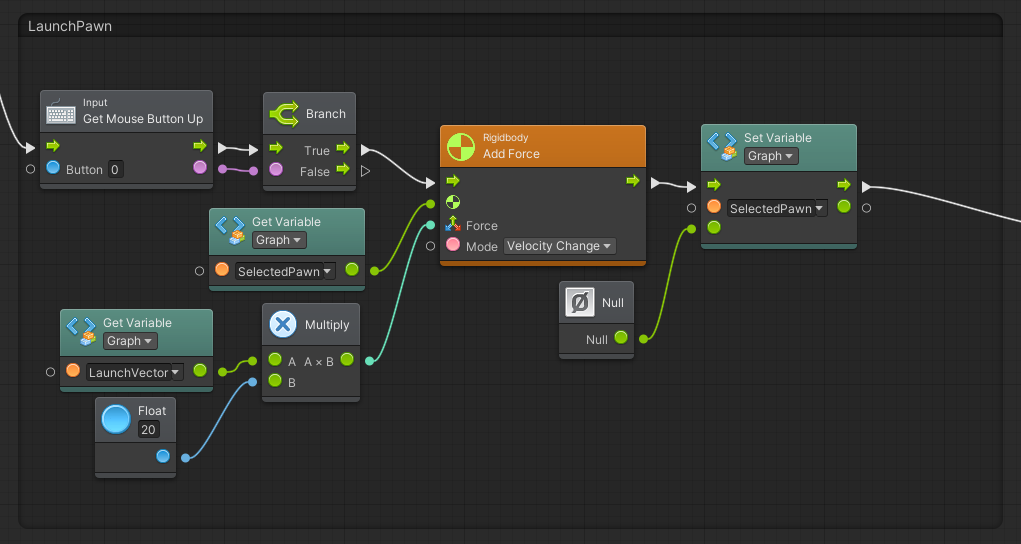


Рисунок - Визуальное программирование в Unity

Визуальное программирование обычно используется на начальных этапах разработки, но в долгосрочной перспективе классическое программирование показывает свои преимущества. К тому же данная работа подразумевает демонстрацию навыков написания качественного кода и проектирования гибкой архитектуры, поэтому для разработки было выбрано программирование на языке C#.

## Использование инструментов Unity

В игровых движках существуют готовые инструменты, которые можно гибко использовать в своих проектах. В Unity существует специальный менеджер пакетов (Package Manager), который позволяет легко добавить необходимые инструменты в проект.

В данном проекте используются следующие инструменты:

* **Cinemachine**. Очень гибкий и простой в использовании набор инструментов для работы с игровой камерой. Используется для создания вида от 1-го или 3-го лица, кат-сцен и игровых трейлеров.
* **Input System**. Новая система обработки ввода игрока упрощает управление в игре. Благодаря ей, можно легко обрабатывать разные устройства ввода, а также переназначать кнопки, как пожелает игрок, прямо как в крупных играх.

## Реализация механик

### Обработка ввода игрока

Используемая Input System позволяет создать Input Actions (дословно «действия ввода»). Они состоят из трех колонок: Action Maps, Actions и Action Properties.

Action Maps («карты действий») – это смысловые блоки или списки, контекстуально объединяющие назначения кнопок и действия. Например, для управления персонажем, машиной, вертолетом понадобились бы три разные «карты действий».

Actions («действия») – это непосредственно список назначений кнопок для действий, которые задает сам пользователь. В них можно задать множество назначений кнопок для одного действия.

Action Properties («свойства действий») позволяют указать тип действия и получаемого значения, а также дополнительную обработку получаемых данных.

На данный момент в проекте существует единственная «карта действий» игрока, в которой описаны действия для управления персонажем: ходьба, вращение камеры, бег, прыжок, стрельба. Каждое действие сопровождается двумя назначениями кнопок: одно для клавиатуры и мыши, второе – для геймпадов или контроллеров. Для действий ходьбы и поворота необходимо получать значение в виде двухмерного вектора, так как нужно знать направление движения и вращения. А для остальных действий достаточно знать, что соответствующая кнопка была нажата или отпущена (Рис. 2).

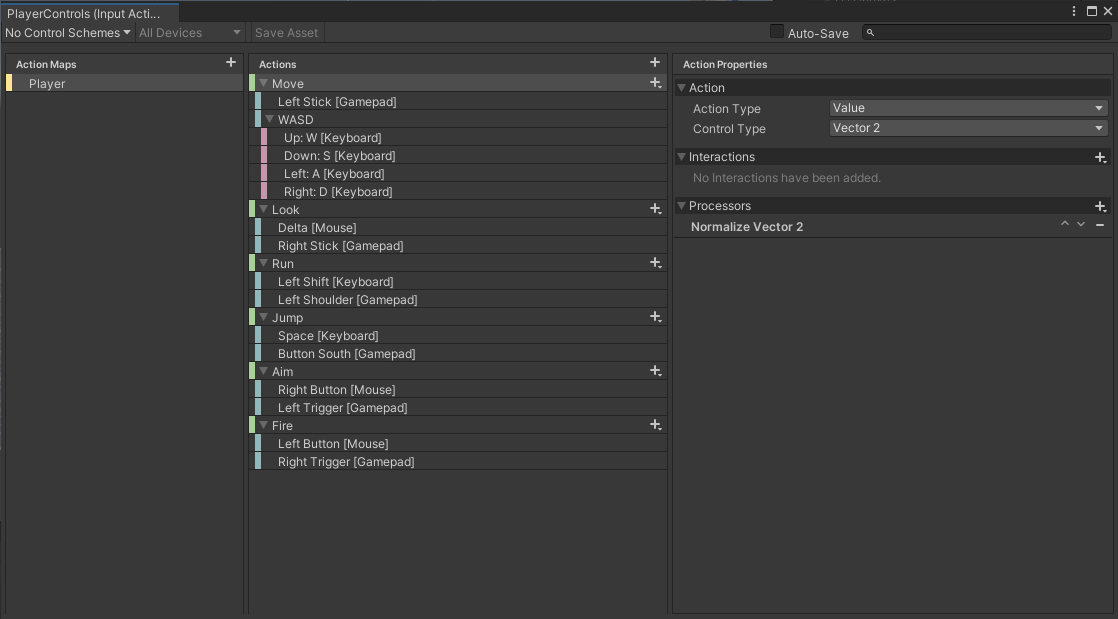


Рисунок - Окно настройки Input Actions

Имея настроенные действия, можно обращаться к ним через код при помощи компонента Player Input (Рис. 9), который должен быть прикреплен к какому-либо объекту.

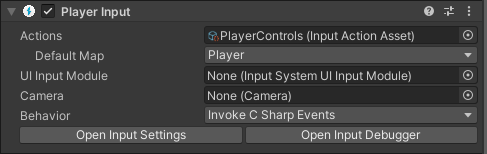


Рисунок - Компонент Player Input

При нажатии на кнопку будет срабатывать соответствующее ей действие, а в коде будет вызываться определенное событие. Каждое действие обладает тремя событиями: started (действие началось), performed (действие исполняется), canceled (действие закончилось). Каждое событие вызывается только в тех кадрах, в которых действие выполнялось. Также можно в любой момент считать значение с любого действия.

На события действий можно подписывать классы, которые используют ввод игрока. В данном случае это классы PlayerController, отвечающий за передвижение игрока, и PlayerAttack, отвечающий за атаку.

### Управление персонажем и камерой

В классе PlayerController обрабатывается передвижение, а именно считываются значения X и Y полученного двухмерного вектора и присваиваются вектору движения персонажа. Так как в Unity ось Y в трехмерном пространстве обозначает вертикальную ось, то значение Y двухмерного вектора ввода нужно присваивать значению Z вектора движения, поскольку оно означает горизонтальное направление (Рис. 10).

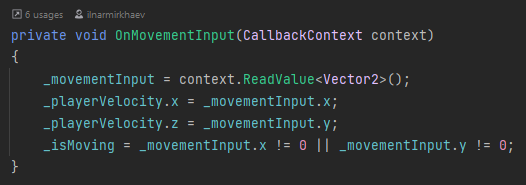


Рисунок - Метод обработки движения

Каждый кадр выполняется метод Update, в котором вызываются методы для обработки гравитации, выполнения передвижения, вращения и анимации персонажа (Рис. 11).

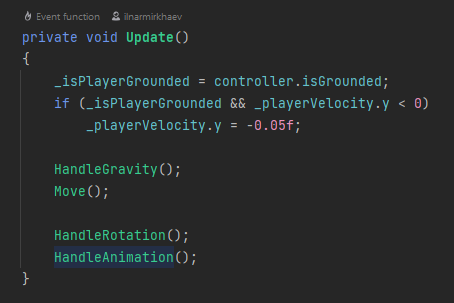


Рисунок - Метод Update в классе PlayerController

Теперь персонаж двигается и анимируется (Рис. 12).

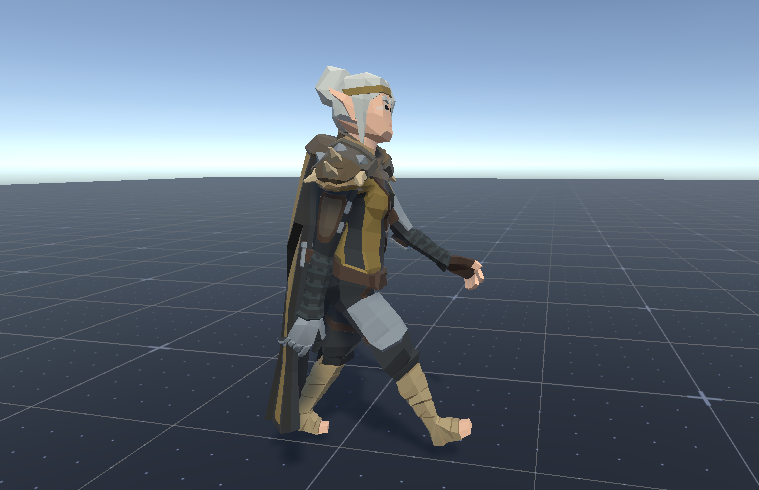


Рисунок - Кадр с анимацией походки персонажа

Вращение персонажа основывается на том, куда сейчас направлена камера. Управление камерой выполняется компонентами пакета Cinemachine. Главные компоненты – это Cinemachine Brain (т.н. мозг) и CinemachineVirtualCamera. Первый непосредственно управляет обычной камерой Unity, переписывая ее значения позиции, вращения и тд. Данные для управления он получает от второго компонента, «виртуальной камеры». Она предоставляет много возможностей для перехвата камеры. Например, можно указать за каким объектом следовать (относительно которого будет меняться позиция камеры), а за каким наблюдать (на него камера будет «смотреть»). А также можно задать тип следования и вращения (Рис. 13).

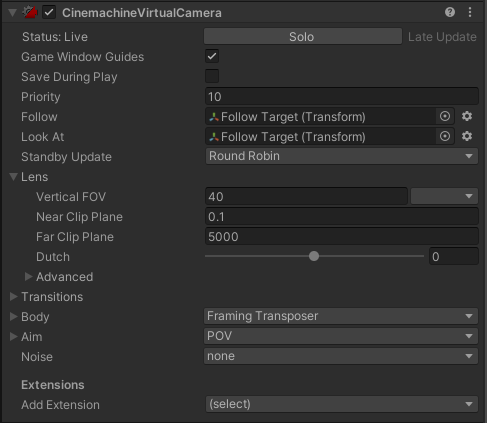


Рисунок - Компонент CinemachineVirtualCamera

Ниже представлено, как выглядит вид от первого лица, сделанный при помощи Cinemachine (Рис. 14).



Рисунок - Вид от первого лица с видимой анимацией персонажа

### Стрельба и попадание

Стрельба из лука реализована похожим образом. Класс PlayerAttack реагирует на нажатия кнопки атаки. Когда кнопка нажата, лук начинает «заряжаться» (хоть пока это и не отражено визуально), когда отпущена – оружие выстреливает, персонаж выпускает стрелу (Рис. 15).

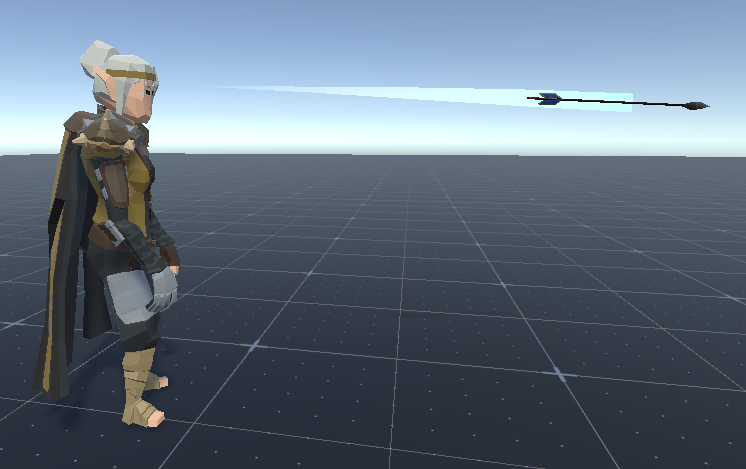


Рисунок - Стрельба

Стрела является отдельным объектом со своей логикой и физикой. Компонент Rigidbody (дословно «твердое тело») наделяет объекты физическим поведением, благодаря ему не нужно писать код для передвижения, как в случае с персонажем, достаточно задать объекту начальную силу движения, как и происходит со снарядами в жизни. Для лучшей визуализации стреле был добавлен компонент Trail Renderer, который отрисовывает путь движения объекта.

Обычно для обработки столкновений используются компоненты Collider (коллайдер), которые при столкновении вызывают соответствующие встроенные методы. Но в ходе разработки было выяснено, что такой подход плохо работает с быстродвижущимися объектами, такими как стрелы. Поэтому необходимо было отдельно написать логику столкновения в классе Arrow.

Помимо встроенного метода Update, который срабатывает каждый кадр, существует метод FixedUpdate, вызываемый в каждое обновление (т.н. тик) физической системы Unity. В нем вручную была написана проверка столкновения: каждый тик между прошлым и нынешним положением наконечника стрелы проводится прямая, если она пересекла какой-нибудь коллайдер, стрела останавливается в точке пересечения. Для удобства отладки был написан вспомогательный инструмент, который рисует цветное перекрестие в заданной точке (Рис. 16).

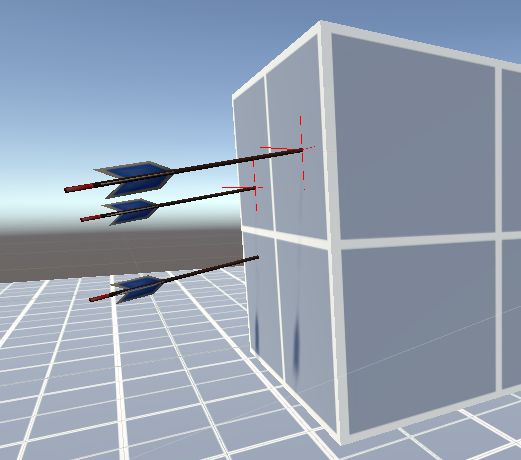


Рисунок - Столкновение стрел с другим объектом

# Реализация сетевого взаимодействия

## Netcode for GameObjects

Сначала для разработки сетевого взаимодействия использовалась библиотека NGO.

### Основные компоненты

В NGO существует несколько основных компонентов: Network Manager, Network Object, Network Behaviour.

Network Manager, или сетевой менеджер, обеспечивает возможность создавать игровые сессии и присоединяться к ним. Также в нем можно указать объект игрока, который будет использоваться при подключении к игровой сессии, и сетевые объекты, чьи данные должны синхронизироваться по сети. На данный момент к сетевым объектам в проекте относятся игрок и стрелы (Рис. 17).

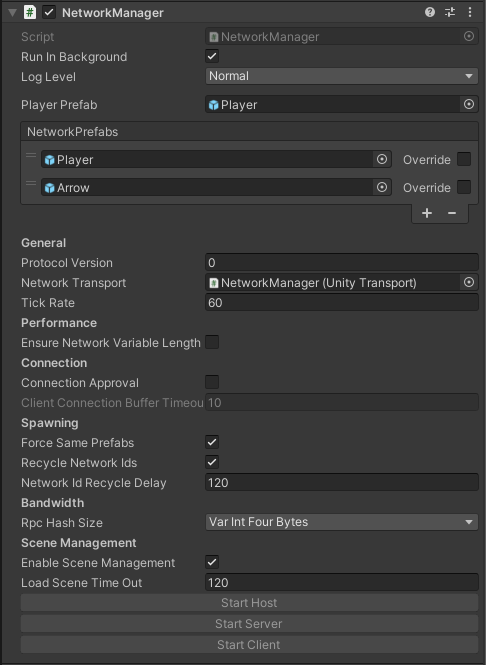


Рисунок - Компонент Network Manager

Компонент Network Object (сетевой объект) лишь указывает на то, что объект участвует в сетевом взаимодействии.

Network Behaviour в чистом виде не является компонентом, это базовый класс для написания кода поведения сетевого объекта, как обычный MonoBehaviour в Unity. Network Behaviour не несет в себе никакого смысла без компонента Network Object, поэтому движок автоматически добавляет второй при добавлении первого.

Классы PlayerAttack, PlayerController и Arrow являются классами сетевого поведения. Первые два сообщают сети о действиях игрока и возвращают результаты этих действий (ходьба, стрельба), а третий – о состоянии стрелы: летит ли она, уже остановилась или вылетела за границы мира.

### Подключение

Сетевой менеджер позволяет одним нажатием создать игровую сессию или подключиться к заданному адресу, внизу у компонента есть кнопки. Но для удобства следует дополнительно вынести кнопки соединения в пользовательский интерфейс, что и было сделано. Кнопка «Host» подключает игрока как хоста, то есть одновременно делает его сервером и клиентом, кнопка «Client» делает игрока клиентом (Рис. 18).

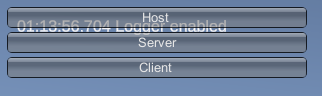


Рисунок - Кнопки создания игровой сессии и подключения

По умолчанию в менеджере задан IP-адрес 127.0.0.1, указывающий на компьютер пользователя, поэтому после нажатия клиент будет пытаться подключиться к самому себе. На самом деле это возможно, если запустить несколько клиентов игры на одном компьютере, что нас полностью устраивает (Рис. 19).

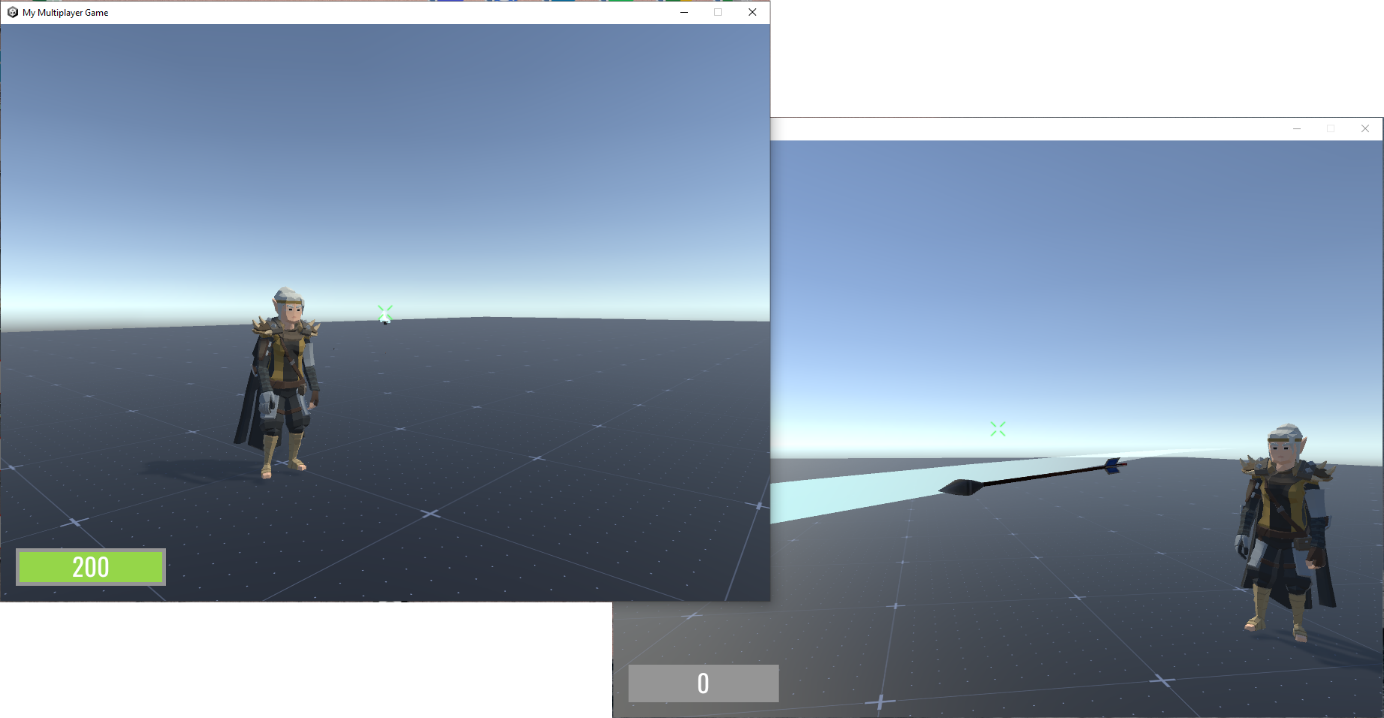


Рисунок - Запуск двух игровых клиентов

По предыдущей иллюстрации видно, что все необходимые данные синхронизируются, а именно положение и вращение игрока, и положение стрел.

### Возникшие проблемы

При дальнейшем рассмотрении было обнаружено, что даже при запуске двух клиентов на одном компьютере появлялись заметные сетевые задержки. Также разработанный фрагмент выглядел неудовлетворительно из-за отсутствия прогнозирования на стороне клиента и моментального ответа игры на действия игрока. Выражалось это в том, что между нажатием на кнопки управления и началом движения персонажа проходило заметное время.

Эти недостатки делают библиотеку NGO сложной в использовании при прототипировании игры. Поэтому было принято решение перенести проект на Photon Fusion, так как эта библиотека предоставляет «из коробки» решения для этих проблем.

## Photon Fusion

### Основные компоненты Photon

В Photon Fusion существует два основных компонента: Network Runner и Network Object.

Network Runner – ядро Fusion. Он отвечает за соединение, обмен данных и симуляцию состояния игрового мира, который должен быть одинаковым для всех игроков.

Network Object назначает объекту сетевой идентификатор и позволяет участвовать в синхронизированной симуляции. Также позволяет указать, должен ли объект оставаться в мире, если игрок-владелец покинул игровую комнату.

Наследуя классы Simulation Behaviour и Network Behaviour, можно описать сетевую логику поведения объекта. Первый позволяет контролировать каждую итерацию (тик) симуляции, второй является его наследником и может дополнительно содержать свойства состояния игры, которые необходимо синхронизировать, например, здоровье игрока.

Также Fusion предоставляет встроенные компоненты для синхронизации объектов разных типов:

* Network Transform – для объектов, чьи позиция и вращение не зависят от физической системы;
* Network Rigidbody – для «твердых тел», объектов, зависящих от физики;
* Network Character Controller – для объектов, которыми управляет игрок, например, персонаж.

С этими компонентами отображение объектов поддерживается плавным, так как в Fusion используется современная технология интерполяции снэпшотов, которая компенсирует неидеальные сетевые условия, такие как задержка, потеря пакетов и другие.

К примеру, с помощью Network Character Controller можно быстро настроить игрового персонажа. Достаточно описать логику движения и указать модель персонажа в поле Interpolation Target, и Fusion позаботится о том, чтобы объект плавно двигался между кадрами (Рис. 20).

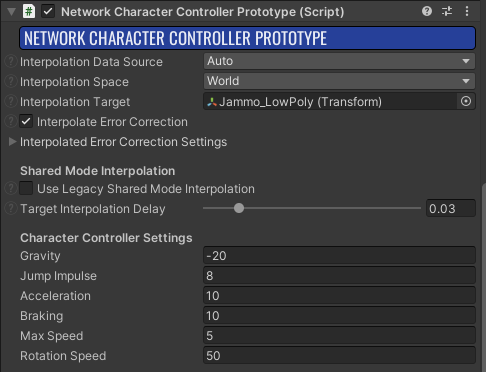


Рисунок 20 - Реализация Network Character Controller

### Обработка ввода

Fusion разделяет обработку ввода на два отдельных этапа:

1. Данные о вводе игрока собираются на локальной машине и помещаются в структуру данных. Это происходит только на клиентах и ​​хосте один раз в каждый тик. Данные отправляются на сервер, а также используются локально для немедленного предсказания на стороне клиента.
2. В методе FixedUpdateNetwork внутри Network Behaviour эти данные можно обработать, чтобы изменить состояние игры (продвинуть симуляцию). Это можно сделать как на клиенте (из любого сетевого объекта, для которого есть полномочия ввода), так и на хосте или сервере. На клиентах это может происходить несколько раз для одного и того же тика (кадра) для повторной симуляции.

Первый шаг можно рассматривать как обычный механизм обработки ввода в Unity, который записывает действия игрока и сохраняет их для последующего использования, а второй шаг – это несвязанный код для применения этого ввода для изменения сетевого состояния, что положительно влияет на общую архитектуру игры.

Для этого необходимо описать свою структуру данных, которая будет передаваться по сети (Рис. 21).

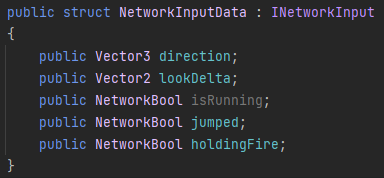


Рисунок - Структура данных ввода

Запись данных ввода необходимо описать в методе OnInput класса, реализующий интерфейс INetworkRunnerCallbacks. Для этого можно использовать новую систему ввода, которая была описана выше (Рис. 22).

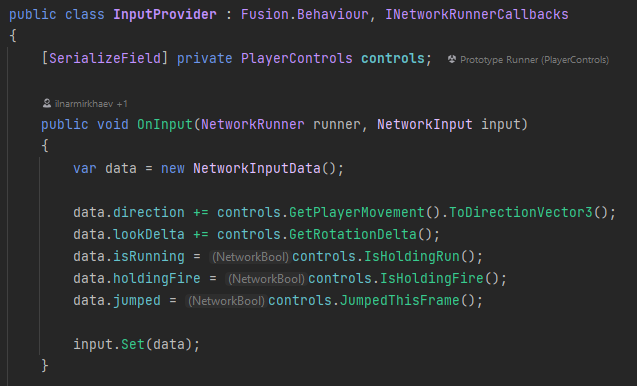


Рисунок - Запись данных ввода

Обработать данные и применить их для изменения состояния объекта можно, вызвав метод GetInput с выходным параметром в виде описанной выше структуры (Рис. 23).

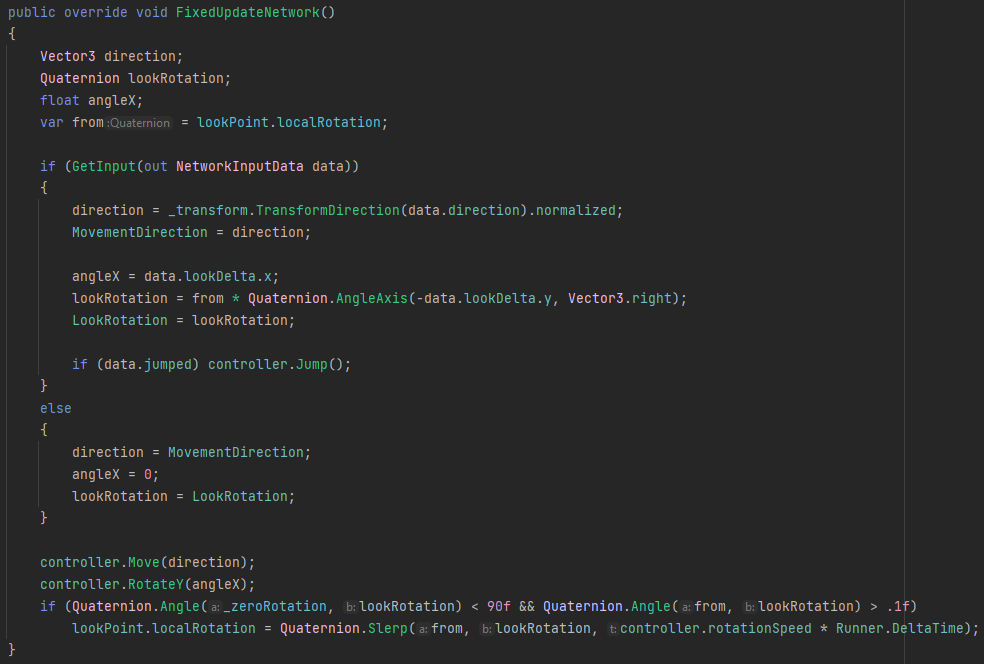


Рисунок - Обработка ввода и изменение позиции и вращения персонажа

### Разработка механики здоровья и получения урона

Здоровье игрока необходимо синхронизировать среди всех клиентов. Для этого используем компонент NetworkBehaviour со свойством HitPoints и атрибутом [Networked]. При изменении значения этого свойства будет вызываться метод указанный в значении OnChanged атрибута, в данном случае HpChanged (Рис. 24).

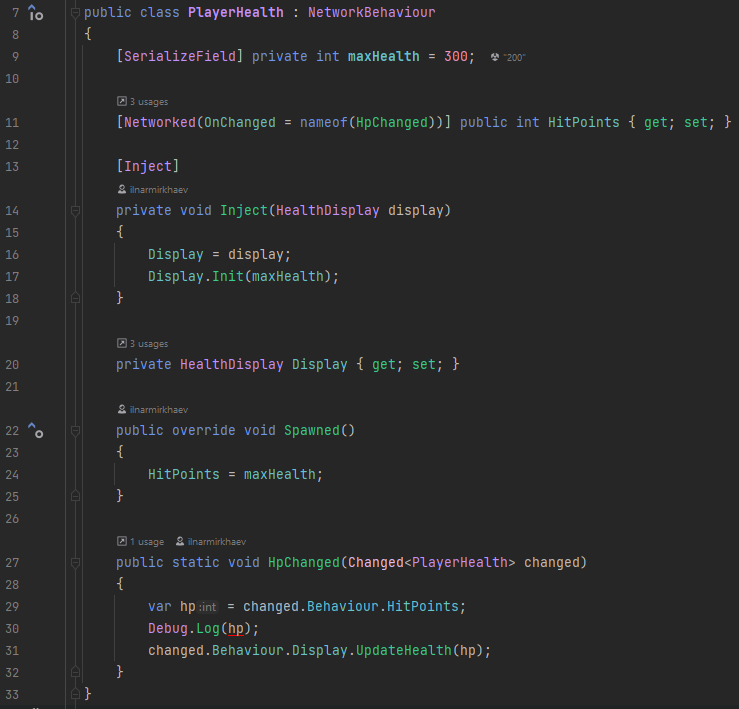


Рисунок - Класс PlayerHealth

При столкновении стрел с объектами происходит проверка на наличие компонента PlayerHealth, и наносится урон в случае положительного исхода. Изменение здоровья игрока отражается в интерфейсе на шкале здоровья в нижнем углу экрана. Эта шкала уникальна для каждого игрока, не синхронизируется по сети и служит лишь для представления данных (Рис. 25).



Рисунок - Изменение шкалы здоровья

### Выводы

Переход на Photon Fusion положительно повлиял на развитие проекта по нескольким причинам:

* Благодаря интерфейсу INetworkRunnerCallbacks получилось отделить обработку ввода от его применения, помимо этого можно будет задать логику для различных сценариев при подключении к игре;
* Скорость синхронизации игрового мира между клиентами и хостом значительно возросла, игра стала более плавной;
* Релей-сервер позволяет создавать комнаты и подключаться к ним только по названию, теперь можно подключиться друг к другу, находясь в разных точках.

По итогу Fusion предлагает множество встроенных инструментов для решения типичных проблем и задач, возникающих при разработке многопользовательских игр, чтобы разработчики могли сфокусироваться на реализации необходимых механик. Эта библиотека отлично подходит для протипирования игры и дальнейшей разработки.

# Организация архитектуры проекта в Unity

В качестве используемого DI-фреймворка был выбран VContainer в силу того, что он прост в освоении и использовании, а также в 5-10 раз быстрее других популярных фреймворков [7].

## Использование VContainer

Для его использования необходимо создать класс-наследник класса LifetimeScope и переопределить метод Configure. В нем можно зарегистрировать все использующие и используемые классы, компоненты, обработчики событий и другое (Рис. 26).

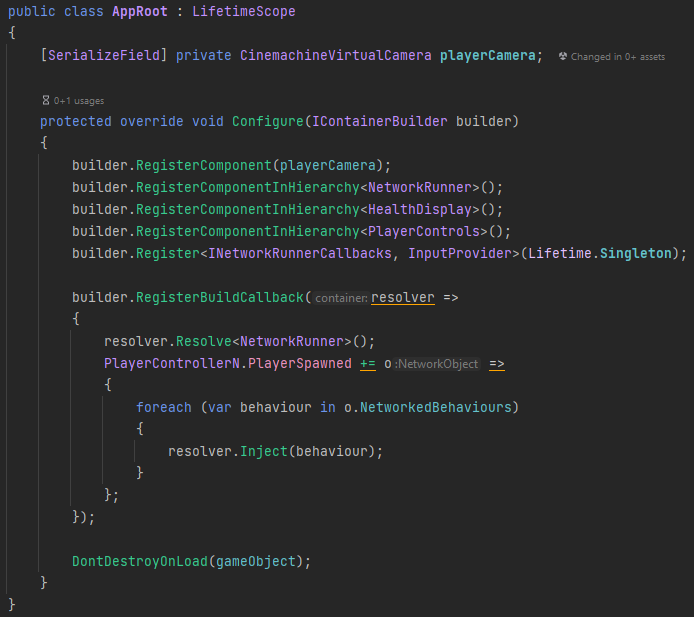


Рисунок - Класс AppRoot

Каждый такой класс называется «контекстом» и имеет свой «контейнер», хранящий зарегистрированные ранее сущности. У каждого контекста может быть один родительский контекст и несколько дочерних контекстов. Так, например, в большом проекте может быть главный контекст, который существует на протяжении всего жизненного цикла игры и предоставляет сервисы, необходимые в нескольких точках приложения, и дополнительные контексты, использующиеся в разных конкретных случаях (напр., отдельные контексты для главного меню игры и игровых уровней).

## Способы внедрения зависимостей

Есть несколько способов «достать» из контейнера нужную сущность:

* Внедрение в конструктор класса
* Внедрение в метод
* Внедрение в поле или свойство
* Внедрение в MonoBehaviour
* Использование контейнера напрямую

Для внедрения в конструктор необходимо зарегистрировать сам класс и те сущности, которые используются в конструкторе. Для внедрения в метод, поле или свойство необходимо отметить их атрибутом [Inject].

Лучше всего использовать внедрение в конструктор, так как оно является самым простым, гарантирует, что все зависимости класса были решены (иначе бы он не создался), и делает их прозрачными. Также благодаря этому можно понять, когда у класса слишком много ответственностей, так как размер конструктора будет соответственно большим.

Но у компонентов, подобных MonoBehaviour, нет конструктора, поэтому будем использовать внедрение в метод, внутри которого будем сохранять зависимости (Рис. 27).

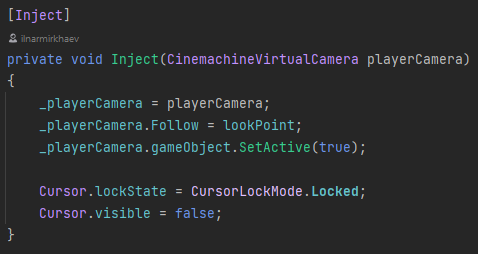


Рисунок - Внедрение камеры Cinemachine в компонент контроллер персонажа

Для этого после появления объекта персонажа в игре будем вызывать специальное событие, на которое подписан класс-контекст. Контейнер (resolver) пройдет по всем компонентам NetworkBehaviour этого объекта и внедрит зависимости во все члены классов, помеченные атрибутом [Inject] (Рис. 26).

## Конечный вид архитектуры

На данный момент архитектуру проекта можно представить в виде следующей диаграммы (Рис. 28):

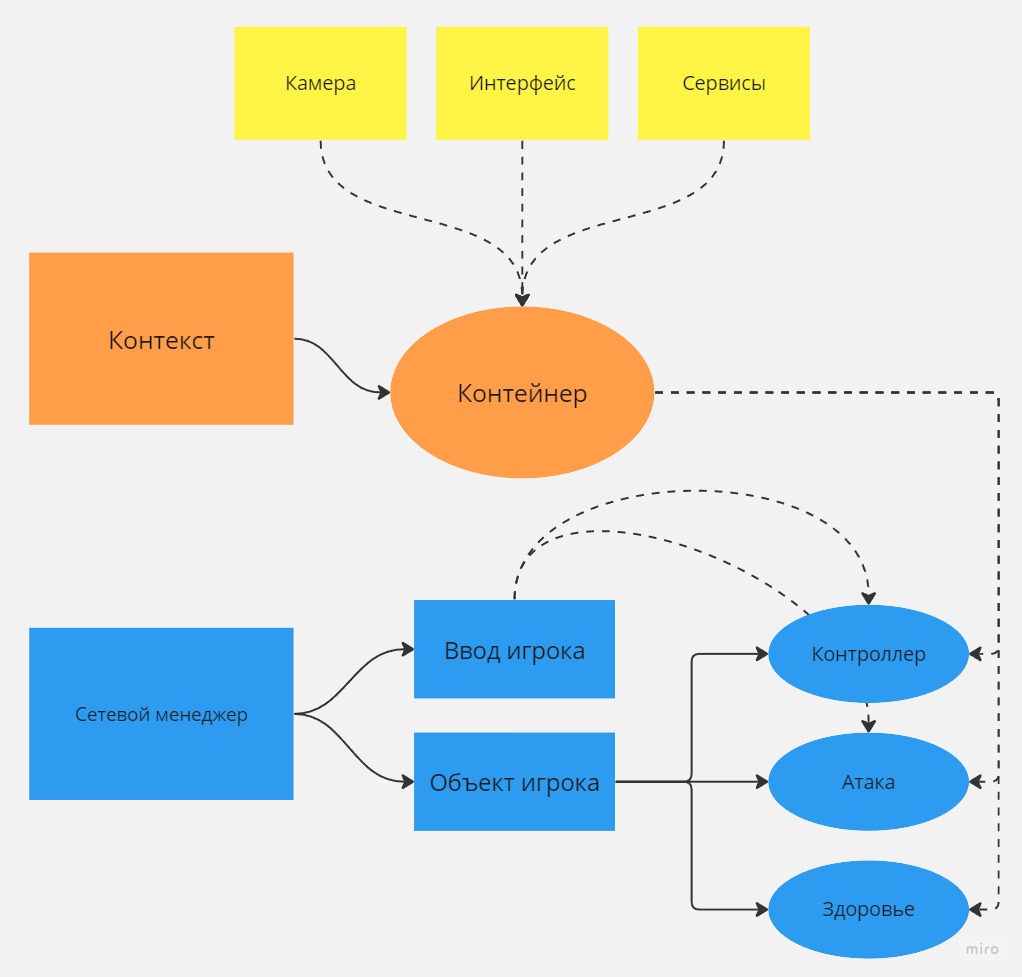


Рисунок - Диаграмма архитектуры проекта

Локально существует контекст и контейнер, хранящий в себе так же локальные объекты, которые не синхронизируются по сети. Так же существует сетевой мэнеджер (Network Runner), отвечающий за обработку ввода игрока и изменение системы в ответ на полученный ввод. Объект игрока содержит компоненты, которые участвуют в сетевом взаимодействии, но также влияющие на локальные объекты, такие как камера, пользовательский интерфейс, эффекты и т.д. Пунктирными линиями отмечены слабые связи в коде: ввод игрока, который записывается и обрабатывается в разных точках; компоненты игрока, которые получают зависимости из контейнера.

Таким образом была создана удобная и расширяемая архитектура, что упростит дальнейшую разработку проекта, особенно при добавлении новых механик или развития существующих.

# Заключение

В рамках данной работы была достигнута основная цель, заключающаяся в разработке и отладке прототипа многопользовательской соревновательной игры.

Были освоены основные архитектурные приемы и паттерны. При помощи современных инструментов игрового движка Unity были разработаны основные игровые механики с поправкой на сетевую составляющую. Были проанализированы и протестированы на практике два инструмента для реализации сетевого взаимодействия в Unity, был аргументирован выбор конечного инструмента. Также была спроектирована и разработана гибкая и качественная архитектура, отвечающая требованиям проекта.

При создании проекта был получен опыт построения игровой архитектуры и разработки сетевой составляющей многопользовательской игры.

Таким образом выбор используемых технологий и архитектуры позволил получить игровой прототип, в который можно поиграть с разных устройств. Также проект в дальнейшем можно будет поддерживать и развивать. Можно разработать новые механики и персонажей, заняться добавлением контента, визуальной и звуковой составляющей.

# Список использованных источников

1. Are You Not Entertained? [Электронный ресурс], <https://www.statista.com/chart/22392/global-revenue-of-selected-entertainment-industry-sectors/>
2. Game Programming Patterns, Роберт Нистром [Электронный ресурс], <https://gameprogrammingpatterns.com/>
3. Расширяемая и удобная в сопровождении архитектура игр на Unity, 2020 [Электронный ресурс], <https://habr.com/ru/company/otus/blog/530056/>
4. Официальные обучающие материалы Unity [Электронный ресурс], <https://docs-multiplayer.unity3d.com/>, <https://docs.unity.com/>, <https://docs.unity3d.com/>
5. Учебный проект Boss Room от Unity [Электронный ресурс], <https://github.com/Unity-Technologies/com.unity.multiplayer.samples.coop#testing-multiplayer>
6. Документация и обучающие материалы Photon Fusion [Электронный ресурс], <https://doc.photonengine.com/fusion/current/getting-started/fusion-intro>
7. Документация VContainer [Электронный ресурс], <https://vcontainer.hadashikick.jp/>