



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ \_\_\_\_\_ Робототехника и комплексная автоматизация \_\_\_\_\_

КАФЕДРА \_\_\_\_\_ Системы автоматизированного проектирования \_\_\_\_\_

## РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

**НА ТЕМУ:**

***Разработка и интеграция сетевых компонентов в  
шаблон многопользовательской игры на Unreal  
Engine 4***

Студент РК6-83Б  
(Группа)

\_\_\_\_\_  
(Подпись, дата) Д.В. Боженко  
(И.О.Фамилия)

Руководитель ВКР

\_\_\_\_\_  
(Подпись, дата) Ф.А. Витюков  
(И.О.Фамилия)

Нормоконтролер

\_\_\_\_\_  
(Подпись, дата) С.В. Грошев  
(И.О.Фамилия)

2023 г.

## АННОТАЦИЯ

В данной работе рассмотрены основные возможности современных многопользовательских игр. Описаны основные концепции сетевого программирования в Unreal Engine 4. Описаны все доступные виды авторизации в Epic Online Subsystem. Реализовано автоматическое зачисление очков, после каждого завершенного матча, глобальная таблица лидеров с разделением на лиги на уровне приложения, два вида лобби с управлением подключениями всех пользователей к уровню лобби.

Тип работы: выпускная квалификационная работа.

Тема работы: Разработка и интеграция сетевых компонентов в шаблон многопользовательской игры на Unreal Engine 4.

Объект исследований: Процесс разработки и внедрения сетевых компонентов.

# СОДЕРЖАНИЕ

АННОТАЦИЯ.....	2
ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ .....	4
ВВЕДЕНИЕ .....	6
1 Возможности современных многопользовательских игр .....	7
2 Сетевое программирование в Unreal Engine.....	10
2.1 Структура многопользовательской игры .....	10
2.2 Репликация .....	11
2.3 Важнейшие классы в Unreal Engine 4 .....	12
2.4 Remote procedure call (RPC) .....	14
2.5 Сетевые роли .....	16
2.6 Игровые сессии .....	17
3 Разработка таблицы лучших игроков на уровне приложения.....	19
3.1 Алгоритм начисления очков для глобальной таблицы лидеров .....	21
3.2 Проблема читаемости данных .....	24
3.3 Отображение заработанных очков в интерфейсе пользователя.....	25
4 Разработка трех видов лобби .....	34
4.1 Настройка игрового уровня .....	36
4.2 Управление подключениями пользователей.....	40
4.3 Круговая задержка (Round Trip Time) .....	42
4.4 Admin-style лобби .....	44
4.5 Admin-style + crowd-style лобби .....	46
5 Внутриигровой голосовой чат .....	50
ЗАКЛЮЧЕНИЕ.....	52
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	54
ПРИЛОЖЕНИЕ А .....	56

## ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

**ЯП** — язык программирования.

**Многопользовательская игра** — режим компьютерной игры, в котором играет более одного пользователя по сети Интернет.

**Движок** — это программный фреймворк, предназначенный в первую очередь для разработки графических приложений и обычно включающий соответствующие библиотеки и программы поддержки.

**UE 4** — движок Unreal Engine 4.

**Клиент** — машина, которая получает информацию об игровом мире через сервер и на которой происходит отрисовка игрового процесса.

**Сервер** — мощная вычислительная машина, через которую происходит обмен информацией об игровом мире без отрисовки графики и воспроизведения звуков.

**Удаленный игрок** — игрок, который находится на другой машине в пределах одной игровой сети.

**Локальный игрок** — игрок, который находится на локальной машине.

**LAN** — локальная вычислительная сеть, где все участники находятся, как правило, в пределах одной ограниченной территории.

**AActor** — один из основных классов в UE 4, являющийся базовым для всех остальных классов, представленных в игровом мире.

**Online Subsystem (далее OSS)** — кроссплатформенная система, позволяющая использовать современные возможности многопользовательских игр.

**Epic Online Services (далее EOS)** — OSS, предоставляемая компанией Epic Games.

**Epic Account Services (далее EAS)** — плагин, являющийся частью EOS, предоставляющий доступ к интерфейсу авторизации и многим другим интерфейсам.

**Epic Game Services** — плагин, являющийся частью EOS, предоставляющий доступ к интерфейсам, связанным с игровым процессом.

**Developer Portal** — интернет-ресурс Epic Games, предназначенный для создания и редактирования настроек приложения, использующего EOS.

**УЗ** — учетная запись.

**Клиент** — машина, которая получает информацию об игровом мире через сервер и на которой происходит отрисовка игрового процесса.

**Сервер** — мощная вычислительная машина, через которую происходит обмен информацией об игровом мире без отрисовки графики и воспроизведения звуков.

**RPC** — Remote procedure call — инструмент в языке программирования C++ в UE 4, используемый для обмена информацией между клиентами и сервером об игровом мире.

**Коллбэк** — обычная функция, ссылка на которую передается другой функции в качестве параметра.

**HUD** — часть визуального интерфейса игрока, которая отображается на переднем плане экрана пользователя.

## ВВЕДЕНИЕ

Рынок видеоигр стремительно развивается с каждым годом. На сегодняшний день рынок игр во всем мире является одним из самых больших сегментов мирового рынка цифрового контента, ежегодно генерируя многомиллиардные доходы и привлекая огромную аудиторию. Наибольшая доля в структуре российского рынка приходится на сегмент онлайн-игр. По данным *Mail.ru Group*, в 2019 году его объем увеличился на 9% и составил 56,7 млрд рублей (около \$1 млрд).

Среди всех жанров игр на данный момент самыми популярными являются *ММО* (массовые многопользовательские онлайн игры), которые использует *Real-Time Multiplayer*. *Real-Time Multiplayer* — это тот режим игры, в котором каждый пользователь получает и отправляет данные об игровом мире с выделенного игрового сервера несколько десятков раз за отведенный промежуток времени. Данное понятие называется тикрейт, т. е. какое количество запросов сервер может обрабатывать за установленные промежуток времени.

Целью данной практической работой является изучение одной из доступных подсистем для движка Unreal Engine 4 (UE 4), а именно предоставляемых ей программных интерфейсов, таких как матчмейкинг, лобби, таблицы лидеров и т. п., которые широко применяются в современных многопользовательских играх. На основе полученных знаний разработать вышеперечисленные сетевые компоненты и внедрить их в шаблон многопользовательской игры.

Данная цель является актуальной, так как изучение концепции создания многопользовательских игр является необходимым условием освоения рынка видеоигр, которые в свою очередь стремительно развиваются и набирают большую популярность в сфере информационных технологий.

# 1 Возможности современных многопользовательских игр

Современные многопользовательские игры включают в себя множество возможностей, которые они могут предоставить пользователям. Самыми распространенными из них являются авторизация, система достижений, матчмейкинг (система подбора игроков), создание лобби, таблицы лидеров, система голосового чата и античит-система.

Одним из самых популярных инструментов, который позволяет создавать графические 3D приложения с сетевым взаимодействием пользователей, является движок Unreal Engine 4. Движок позволяет разработчику сконцентрироваться на проектировании взаимодействий между объектами на сцене. При этом не приходится вручную прописывать отображение каждого полигона на сцене или производить физические или математические вычисления над объектами. Всю вышеперечисленную работу выполняет движок, позволяя разработчику сконцентрироваться на проектировании архитектуры своего приложения.

Unreal Engine предоставляет возможность разработки с помощью визуального программирования (Blue prints), а также с помощью реализации алгоритмов на языке программирования C++. Так как современные многопользовательские игры должны решать проблему масштабируемости, скорость работы приложения играет очень важную роль. Разработка на C++ с минимальным использованием Blue prints позволяет обеспечить максимальную скорость и производительность работы любого графического приложения.

В проектирование взаимодействий между объектами входят такие действия, как расположение отдельных объектов на сцене, создание классов, работа с ассетами, создание отдельных уровней и проектирование сетевой части приложения.

В UE 4 существует несколько Online Subsystem (OSS), которые предоставляют доступ к возможностям современных многопользовательских

игр, а именно Online Subsystem EOS, Online Subsystem Steam, Online Subsystem Oculus, Online Subsystem Google Play, а также Online Subsystem Null. Каждая из перечисленных подсистем предоставляет возможность использовать возможности современных многопользовательских игр, добавляя собственные интеграции.

Online Subsystem Epic Online Services (EOS) является хорошим выбором, так как предоставляет широкие возможности выбора интерфейсов для реализации возможностей многопользовательских игр, которые могут расширить функционал любого шаблона многопользовательской игры. Также EOS имеет подробную документацию, которую необходимо использовать при интеграции предоставленных интерфейсов в проект.

EOS подразделяется на два вида сервисов: EAS и EOS. Оба сервиса предоставляют большое количество интерфейсов, которые добавляют в игру возможности современных многопользовательских игр. Оба плагина могут использоваться как одновременно, так каждый по отдельности в независимости друг от друга.

EAS в основном предоставляет интерфейсы для авторизации пользователей с помощью учетной записи (УЗ) Epic Games и управления списком друзей. Epic Games Services предоставляют интерфейсы, которые связаны с управлением многопользовательского игрового процесса пользователей.

Для того, чтобы в приложении можно было использовать данные сервисы, необходимо провести предварительную настройку приложения на Developer Portal Epic Games, где нужно получить необходимые данные для инициализации приложения и разрешить использование двух вышеперечисленных сервисов.

Также современные многопользовательские игры предоставляют пользователю удобные виджеты и интерфейсы, с помощью которых он может эффективно управлять приложением. В Unreal Engine для создания виджетов используется такой инструмент, как Unreal Motion Graphics (UMG). С помощью интерфейса UMG разработчик может удобно создавать и размещать на экране



виджеты. Для управления созданными виджетами создаются C++ классы, которые привязываются к созданному файлу интерфейса. Далее в классе определяются переменные и функции, с помощью которых в дальнейшем в приложении выполняется взаимодействие пользователя с созданными виджетами.

## 2 Сетевое программирование в Unreal Engine

Соединение в многопользовательских играх происходит по клиент-серверной модели, когда несколько клиентов подключаются к выделенному серверу и через него передают друг другу информацию об игровом мире [2].

На этапе разработки многопользовательской игры, важно понимать, как и с помощью каких инструментов игрового движка реализовано сетевое взаимодействие между игроками, какие при этом создаются объекты классов, сколько копий каждого класса создается и на какой машине они находятся. Также важно понимать, что на каждом уровне могут использоваться свои классы, и необходимо для каждого отдельного уровня задавать правильные классы с своей разделенной логикой.

### 2.1 Структура многопользовательской игры

В UE 4 существует четыре основных мода многопользовательской игры [3]:

**Standalone** — автономная игра, где сам экземпляр игры является сервером. Игра, запущенная в таком режиме, не принимает никаких подключений от удаленных игроков.

**Client** — режим игры, в котором она имеет роль клиента, и работает только при подключении к игровому серверу.

**Listen-Server** — режим игры, в котором сервером становится один из клиентов и на котором размещена сетевая многопользовательская сессия. В таком режиме игра как принимает запросы от удаленных игроков, так и содержит своих локальных игроков. Такой режим многопользовательской игры хорошо подходит для развертывания кооперативных игр, где все игроки находятся в пределах LAN и сетевые взаимодействия осуществляются по P2P-сети.

Dedicated-Server — режим игры, в котором сервер расположен на отдельно выделенной машине. Экземпляр игры, запущенный на выделенном сервере, принимает запросы от удаленных игроков, но сам не содержит никаких локальных игроков. Следовательно, на таком экземпляре игры отсутствуют такие функции, ориентированные на игроков, как отрисовка графики, вывод звуков и пользовательский ввод. Данное решение является основным для большинства многопользовательских игр, где есть много игроков, так как выделенные сервера обладают большой вычислительной мощностью и обеспечивают безопасность от обмана.

## 2.2 Репликация

Репликация — это синхронизация информации об игровом мире между сторонами. Другими словами, репликация — это механизм, который создает множество копий одного и того же объекта. Объект и его копии хранятся на разных машинах (на клиенте и на сервере), за счет чего между игроками, находящимися на разных игровых машинах, происходит синхронизация информации об одном и том же объекте. Реплицировать можно переменные, события и объекты. Передача измеренной информации об объекте может осуществляться от сервера к одному клиенту, от любого клиента к серверу, и от сервера ко всем клиентам.

В UE 4 для реализации репликации объектов используется возможность класса `AActor`, так как все объекты, представленные в игровом мире, являются сущностями этого класса [4].

Важно понимать, что игрок видит перемещение другого игрока у себя на локальной машине именно за счет репликации. Это происходит за счет того, что при репликации объекта, его копии хранятся на локальных машинах других игроков, а также на сервере. При перемещении игрок посылает серверу свои координаты, тот, в свою очередь, анализируя эти координаты, передает их

копиям этого игрока, которые расположены на остальных клиентах. Следовательно, когда игрок видит перемещение другого игрока у себя на локальной машине, он видит перемещение копии этого игрока, которая полностью управляется сервером.

Реплицирование переменных в UE 4 также осуществляется особым образом [5]. Переменную, которую надо реплицировать помечается с помощью макроса `UPROPERTY(Replicated)`. Важно понимать сам механизм реплицирования. Когда переменная, находящаяся на сервере, изменяет свое значение, то ее значение также будет изменено на всех клиентах. При этом также возможно указать функцию, которая будет выполняться при изменении реплицированной переменной через флаг `ReplicatedUsing`. Важно заметить, что данная функция будет выполняться только при изменении реплицированной переменной и только на клиентах, и никогда не будет выполняться на сервере.

## **2.3 Важнейшие классы в Unreal Engine 4**

Для базовой реализации простой многопользовательской игры в UE 4 существует несколько базовых классов [6]:

`AGameMode` — самый важный класс, который отвечает за правила игры. Важно знать, что экземпляр такого класса находится только на сервере. Чтобы избежать нечестной игры, все действия, связанные с игровой логикой, запрашиваются через сущность этого класса. Важно понимать, что попытка получить доступ к сущности класса `AGameMode` с клиента будет безуспешной.

`AGameState` — класс, который содержит в себе информацию о текущем состоянии игры, например, о количестве подключенных к сессии игроков. Сущность данного класса располагается на сервере, а также его копия располагается на каждом из клиентов. Таким образом, сущность данного класса самая важная, которая необходима для передачи общей информации между сервером и клиентами.

APlayerState — класс, который содержит в себе всю текущую информацию об игроке, подключенном к игровой сессии. Сущность данного класса находится на каждом клиенте, а также копия сущности класса APlayerState каждого клиента находится на сервере. Следовательно, сервер знает о сущности APlayerState каждого клиента, а клиент знает о существовании только собственного класса APlayerState.

APlayerController — класс, который остается за игроком на протяжении всей игровой сессии. С помощью APlayerController клиента можно легко управлять интерфейсом игрока, когда необходимо освежить новую информацию о состоянии объектов игрового мира, например, изменение очков, изменение здоровья и другое. Распределение по клиентам и серверу такое же, как у APlayerState.

ANUD — класс, который существуют только на клиенте, который является владельцем данного класса. ANUD используется для управления виджетами клиентами, управлением данным классом можно осуществлять с клиента, либо же с сервера с помощью RPC.

APawn — класс, который представляет из себя объект на сцене, которым управляет игрок либо же сервер. Данный класс является производным от класса AActor, поэтому возможно его реплицирование. Каждый клиент и сервер знает о существовании о каждом объекте класса APawn. Ниже представлена схема репликации классов (Рисунок 1).

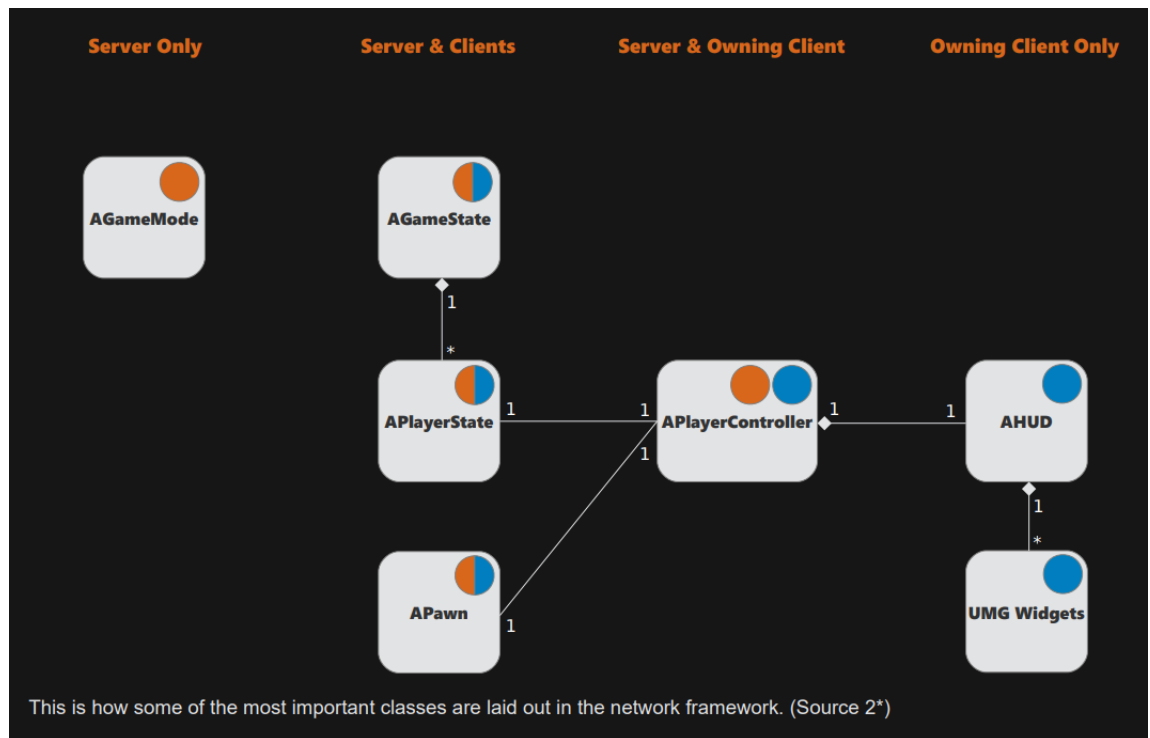


Рисунок 1 — Схема репликации основных классов UE

## 2.4 Remote procedure call (RPC)

RPC — это особые функции в UE 4, которые вызываются на одной машине, а выполняются на другой. RPC помечаются через макрос `UFUNCTION()`. Всего существует три вида RPC:

**Client** — данный тип RPC, как правило, вызывается с сервера. Модификатор **Client** говорит о том, что данная функция будет выполнена только на том клиенте, который владеет данной RPC-функцией. Данный вид RPC хорошо подходит для обновления виджета в HUD клиента, когда на сервере произошло изменение реплицированной переменной (Рисунок 2).

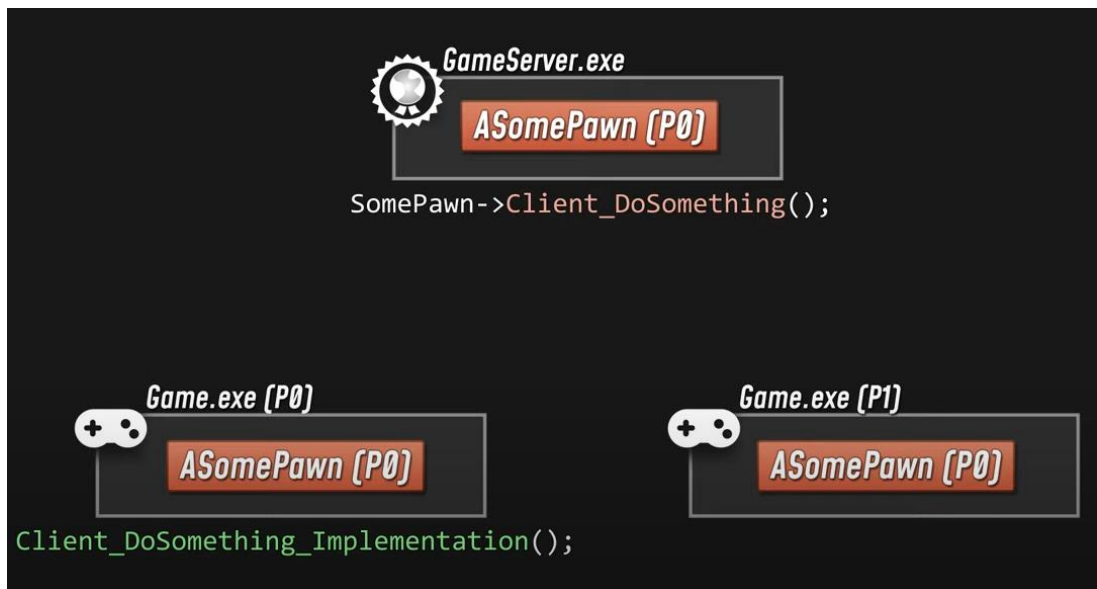


Рисунок 2 — Схема вызова и выполнения Client RPC

Server — данный тип RPC вызывается с клиента, а выполнение функции производится только на сервере (Рисунок 3).

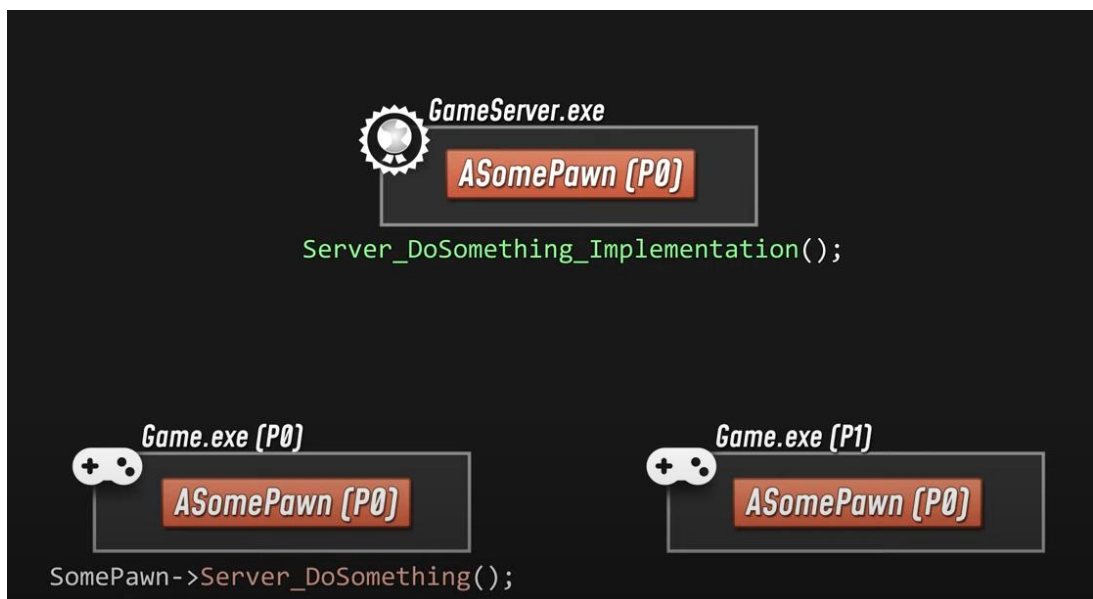


Рисунок 3 — Схема вызова и выполнения Server RPC

NetMulticast — данный тип RPC всегда вызывается с сервера и будет выполнен как на всех клиентах, так и на сервере. Такой тип RPC хорошо подходит, например, для проигрывания анимации после смерти игрока (Рисунок 4).

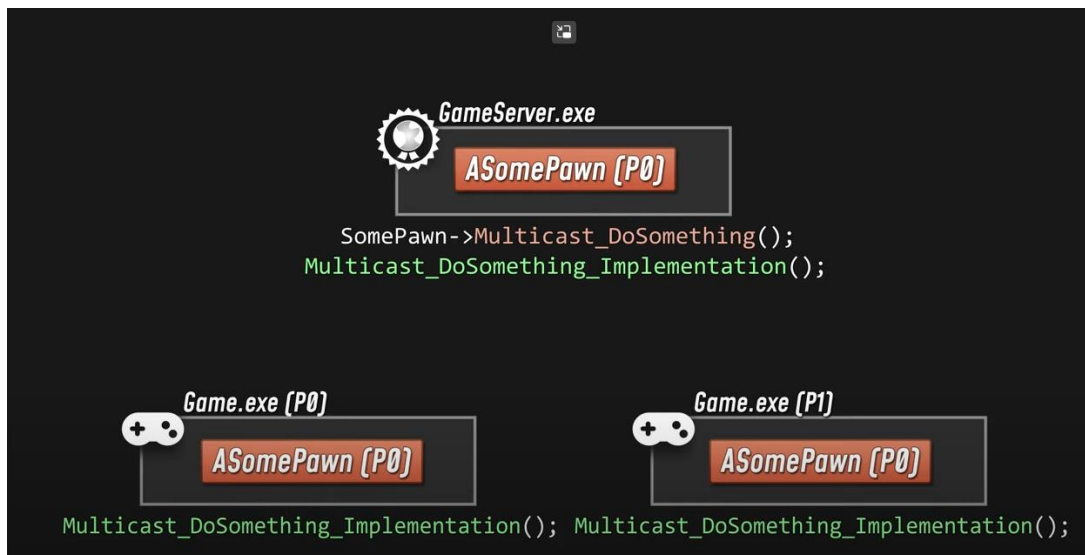


Рисунок 4 — Схема вызова и выполнения NetMulticast RPC

## 2.5 Сетевые роли

В UE 4 существует 4 основных сетевых роли, которые может иметь AActor: ROLE\_AutonomousProху — сетевая роль, которая показывает, что данный AActor находится на клиенте и управляется живым игроком.

ROLE\_SimulatedProху — сетевая роль, которая показывает, что данный AActor находится на клиенте и управляется сервером.

ROLE\_Athority — сетевая роль, которая определяет, что данный AActor или его копия находится на сервере.

ROLE\_None — сетевая роль, которая дается AActor в случае, если он не обладает ни одной ролью из вышеперечисленных.

Ниже представлена таблица, с помощью которой можно понять, какую роль получает AActor в зависимости от этого, на какой машине находится он или его копия (Рисунок 5).



	<i>GameServer.exe</i> NM_DedicatedServer	<i>Game.exe [P0]</i> NM_Client	<i>Game.exe [P1]</i> NM_Client
<i>AGameModeBase</i>	<i>AUTHORITY</i>	[not replicated]	[not replicated]
<i>APlayerController [P0]</i>	<i>AUTHORITY</i>	<i>AutonomousProxy</i>	[not replicated]
<i>APlayerController [P1]</i>	<i>AUTHORITY</i>	[not replicated]	<i>AutonomousProxy</i>
<i>APawn [P0]</i>	<i>AUTHORITY</i>	<i>AutonomousProxy</i>	<i>SimulatedProxy</i>
<i>APawn [P1]</i>	<i>AUTHORITY</i>	<i>SimulatedProxy</i>	<i>AutonomousProxy</i>

Рисунок 5 — Схема распределения сетевых ролей на примере двух игроков

## 2.6 Игровые сессии

Сессии, также, как и сетевое программирование, является базовым аспектом, с помощью которого строятся многопользовательские игры. С помощью сетевого программирования прорабатывается процесс взаимодействия персонажей друг с другом и с игровым миром в целом. Сессии служат для того, чтобы контролировать подключение игроков к выделенному серверу или хосту.

Сессия — это большая структура данных, которая содержит в себе множество полей, настроек и методов, с помощью которых можно задавать правила подключения игроков к серверу. Базовыми настройками игровой сессии является:

Максимальное количество игроков, которые могут одновременно подключиться к игре.

1. Доступность игры из сети Интернет.
2. Возможность использовать лобби.
3. Возможность подключаться к сессии после начала игрового процесса.

Сущность сессии всегда находится на сервере. Клиенты, которые хотят подключиться к северу, сперва должны подключиться к игровой сессии. Если подключение клиента успешно прошло проверку, ему разрешается присоединиться к игровому серверу. Таким образом роль сессии заключается в том, что она является промежуточным узлом подключения между клиентом и сервером, который проверяет подключение каждого клиента и управляет им.

### 3 Разработка таблицы лучших игроков на уровне приложения

Для дальнейшей разработки сетевой части приложения о реализации взаимодействий между пользователями были реализованы следующие классы [7]:

- **ULab4GameInstance** — класс, производный от класса **UGameInstance**. Данный класс реализован для авторизации пользователя, создания сессии, уничтожения сессии, поиска существующих комнат, присоединения к списку найденных комнат, а также для взаимодействия с глобальной таблицей лидеров в EOS. Причина, по которой данные взаимодействия осуществляются именно в этом классе состоит в том, что сущность данного класса создается после инициализации приложения и имеет такое же время жизни, как и время работы приложения. Это дает возможность обращаться к реализованным функциям на всех уровнях.
- **APersonProjectile** — класс, производный от класса **AActor**. Сущность данного класса представляет собой снаряд, который игроки запускают друг в друга во время основного игрового процесса. Данный класс был создан для того, чтобы при попадании снаряда определять, кто является его владельцем и в кого попал снаряд. Таким образом можно регистрировать факт попадания снаряда и в дальнейшем подсчитывать очки пользователей.
- **ALab4Character** — класс, производный от класса **ACharacter**. Данный класс создан для того, чтобы управлять персонажем пользователя и его состоянием (текущий уровень здоровья, количество заработанных очков, имя пользователя). Данный класс содержит много реплицированных переменных, RPC, а также функции переопределения нажатия клавиш, которые пользователь нажимает для управления своим персонажем.
- **ALab4GameMode** — класс, производный от класса **AGameMode**. Данный класс определяет правила игрового процесса, которые происходят на

главном уровне, такие, как текущее состояние матча и возрождение игрока. Также в классе определены другие классы, которые AGameMode будет располагать на сцене по умолчанию, и настройки основного уровня (необходимое количество очков для победы, время запуска игрового уровня, время ожидания всех игроков перед началом матча и т.д.). Данные настройки всегда должны храниться, в сущности, этого класса во избежание обмана со стороны клиентов. Чтобы клиенты могли получить настройки, перечисленные выше, они после своей инициализации должны сделать ряд запросов и записать полученные значения у себя в памяти.

- ALab4HUD — класс, производный от класса AHUD. Сущность данного класса создается для каждого пользователя на игровом уровне и хранит у себя в памяти переменные виджетов, которые соответствуют виджетам, представленным на экране игрока во время игрового процесса (количество очков пользователя, уровень здоровья, информационные сообщения о ликвидации).
- ALab4PlayerController — класс, производный от класса APlayerController. Данный класс создан для управления виджетами, которые хранятся в сущности класса ALab4HUD. Также данный класс отвечает за правила отрисовки данных виджетов.
- ALab4PlayerState — класс, производный от класса APlayerState. Данный класс используется в Unreal Engine для того, чтобы хранить всю необходимую информацию о пользователе. Сущность данного класса реплицируется движком по умолчанию, что сразу делает его очень удобным в использовании. В данном классе также переопределены методы начисления очков и функция изменения имени пользователя.
- URankedLeaderbord — класс, производный от класса UUserWidget. Данный класс создан для управления таблицей лидеров на уровне приложения. Основной логикой в таблице является выполнение запросов, когда пользователь нажимает на кнопки с лигами игроков, и ему выводится список пользователей, которые соответствуют заданному интервалу лиги.

### 3.1 Алгоритм начисления очков для глобальной таблицы лидеров

Для преобразования внутриигровых очков каждого игрока в очки, которые будут представлены в глобальной таблице лидеров, необходимо применить определенный алгоритм [8]. Для многопользовательских игр, где игра происходит в режиме “каждый сам за себя” и “команда на команду”, необходимо анализировать таблицу игроков с внутриигровыми очками и в соответствии с заработанными очками определить, какое значение отправить в глобальную таблицу лидеров. Для реализации вышеописанного алгоритма, можно использовать нормализацию внутриигровых очков и дальнейшую нормировку данных, которые будут отправлены в глобальную таблицу лидеров. Нормализация — это один из способов предобработки информации, при котором входные данные приводятся к заданному диапазону, например, к диапазону  $[0; 1]$  или  $[-1; 1]$ .

Для получения нормализующей функции, которая работает с нормализованными внутриигровыми очками, нужно определить дискретный набор данных (Рисунок 6).

-1	-1
-0,9	-0,72
-0,8	-0,6
-0,7	-0,48
-0,6	-0,4
-0,5	-0,32
-0,4	-0,24
-0,3	-0,16
-0,2	-0,08
-0,1	-0,04
0	0
0,1	0,04
0,2	0,08
0,3	0,16
0,4	0,24
0,5	0,32
0,6	0,4
0,7	0,48
0,8	0,6
0,9	0,72
1	1

Рисунок 6 — Дискретный набор данных для кривой начисления очков

Далее по представленному набору данных с помощью интерполяции можно получить аналитическое выражение необходимой функции. Полученное аналитическое выражение функции имеет вид  $y = 0.4849x^3 - 1e^{-14}x^2 + 0.4674x + 3e^{-14}$ . График данной функции представлен ниже (Рисунок 7).

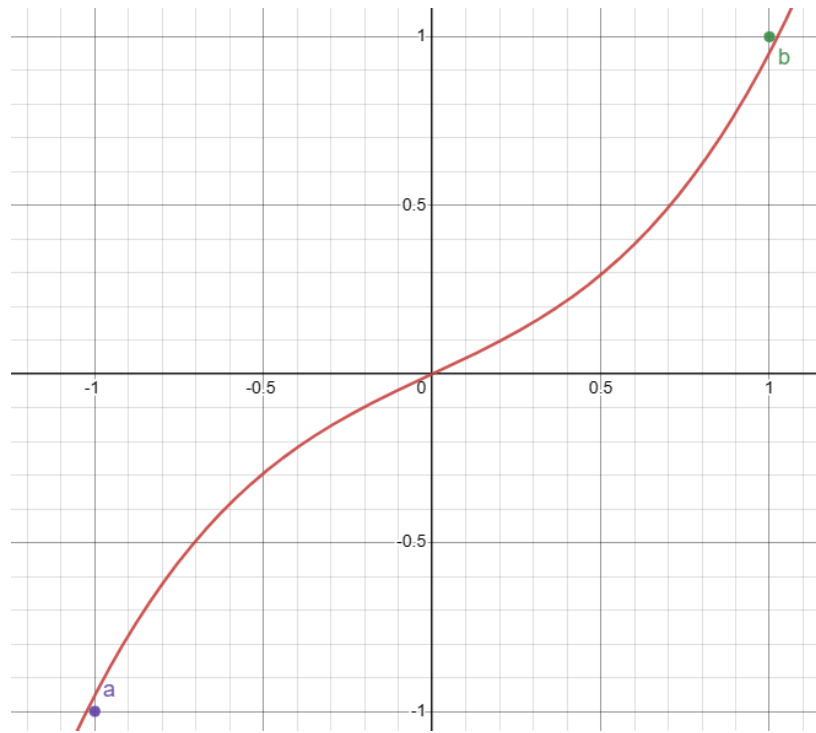


Рисунок 7 — График преобразующей линии тренда  $[-1;1]$

Как видно на графике, при  $x = 1$  и при  $x = -1$  значение функции не равняется граничным значениям 1 и  $-1$  соответственно. Это произошло вследствие того, что полученное аналитическое выражение полинома является аппроксимацией. Для этого необходимо найти значения аргумента, при которых функция принимает необходимые граничные значения и скорректировать интервал принимаемых значений. Скорректированный интервал будет иметь вид  $[-1.03; 1.03]$ .

Так как функция должна принимать на вход параметр, принадлежащий интервалу  $[-1.03; 1.03]$ , необходимо привести внутриигровые очки каждого игрока к данному интервалу. Для этого необходимо воспользоваться формулой общего вида, которая приводит данные к произвольному диапазону  $[a; b]$  и выражается следующим образом:

$$\hat{x} = a + \frac{(x - x_{min})(b - a)}{x_{max} - x_{min}}, \quad (1)$$

где  $a$  — левая граница требуемого диапазона,  $b$  — правая граница требуемого диапазона,  $x_{min}$  — минимальное число внутриигровых очков, которое заработали все игроки,  $x_{max}$  — максимальное число внутриигровых очков, которое заработали все игроки,  $x$  — число внутриигровых очков, которое заработал текущий игрок.

Тогда алгоритм начисления очков заключается в том, что игрок, набравший наибольшее количество внутриигровых очков, получает максимальное нормализованное количество очков 1.00, игрок, набравший минимальное количество внутриигровых очков, получает минимальное нормализованное количество очков  $-1.00$ . Игрок, занявший среднюю позицию во внутриигровой таблице, получает 0 внутриигровых очков за заверченный раунд.

### 3.2 Проблема читаемости данных

Использование нормализованных данных, где точность составляет до 7 знаков после запятой достаточно удобно для хранения, так как маловероятно, что число переполнит разрядную сетку. С другой стороны, возникает проблема, которая заключается в том, что у пользователя будут трудности с чтением и анализом отображаемых глобальных очков. Для решения данной проблемы необходимо выполнить нормировку нормализованных значений и получить результаты в определенных понятных единицах.

Нормировка — это корректировка нормализованных значений в соответствии с выбранным алгоритмом с целью сделать их более читаемыми [9].

Чтобы нормировать подсчитанное количество внутриигровых очков каждого игрока, было принято решение умножать высчитанные нормализованные очки на коэффициент равный 25 и округлять полученные результаты до ближайшего целого числа. Округление результата необходимо, так



как в EOS очки в глобальных таблицах лидеров хранятся только в целочисленных значениях. Округление до ближайшего целого числа осуществляется с помощью метод `FloorToInt` класса `FMath`: `FMath::FloorToInt(float)`.

В итоге после нормировки пользователь получает данные в более понятных для него единицах. Игрок, получивший максимальное количество очков, получает +25 очков за матч, минимальное количество очков — -25. Данные числа кратны 5, их удобно считать и анализировать, однако теперь они будут занимать больше места при хранении в EOS.

### **3.3 Отображение заработанных очков в интерфейсе пользователя**

Для того чтобы начать использовать таблицы лидеров в своем приложении, необходимо зарегистрировать в системе EOS на Developer Portal новую таблицу лидеров и статистику, которую эта новая таблица будет отслеживать [10].

В качестве статистики разработчик имеет возможность отслеживать различные игровые данные игроков: количество собранных предметов, время прохождения определенного уровня, общее количество поражений и побед или просто общее количество раз совершения игроком какого-либо определенного действия. Ниже представлен способ создания новой отслеживаемой статистики на Developer Portal (Рисунок 8).

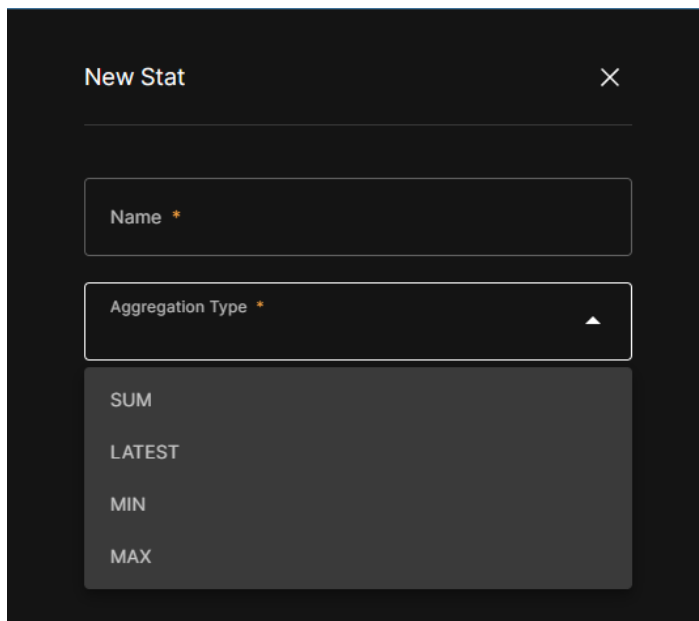


Рисунок 8 — Создание отслеживаемой игровой статистики на Developer Portal

Как видно, для создания статистики необходимо задать ее имя и тип. Всего в системе EOS существует 4 типа статистики [11]:

1. Тип SUM — позволяет суммировать каждый результат, который отправляет игрок. Игрок, набравший наибольшее количество очков, занимает первое место.
2. Тип MIN — позволяет зарегистрировать наименьший результат игрока. Игрок, заработавший наименьшее количество очков, занимает первое место.
3. Тип MAX — позволяет зарегистрировать наибольший результат игрока. Игрок, набравший наибольшее количество очков, занимает первое место.
4. Тип LATEST — позволяет зарегистрировать самый последний по дате результат игрока. Игрок, набравший наибольшее количество очков, занимает первую позицию.

Для создания глобальной таблицы лидеров отлично подходит тип SUM, так как необходимо для игрока суммировать его заработанные очки в конце

каждого матча и сразу отсортировать глобальную таблицу лидеров по данному значению.

Для создания глобальной таблицы лидеров также необходимо задать ее имя, статистику, которая она будет отслеживать, и время жизни таблицы (для удобства можно поставить бессрочный период жизни). Создание глобальной таблицы лидеров представлено ниже (Рисунок 9).

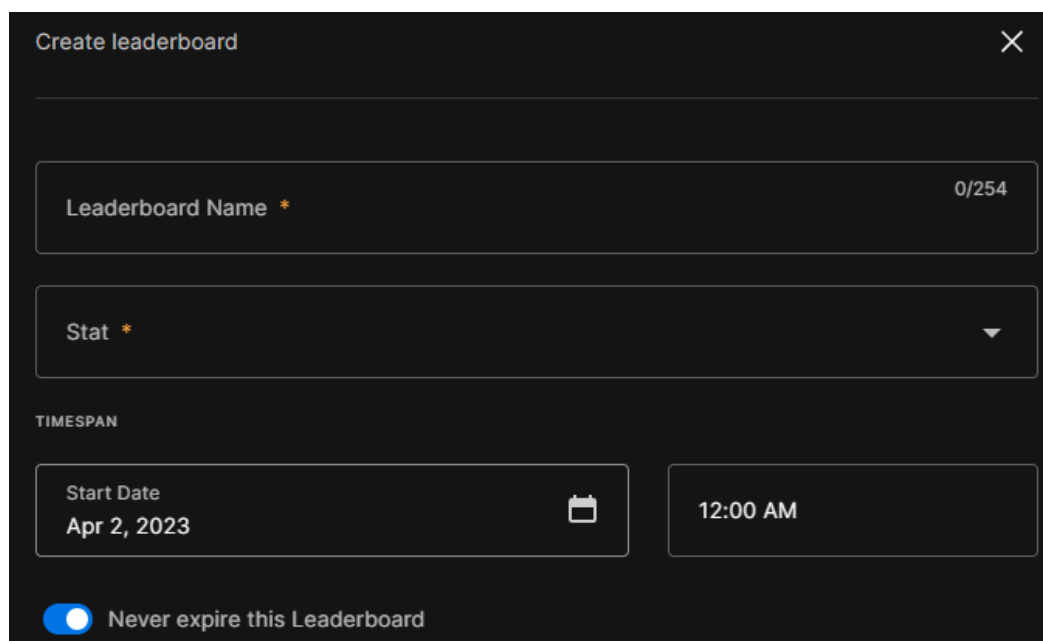
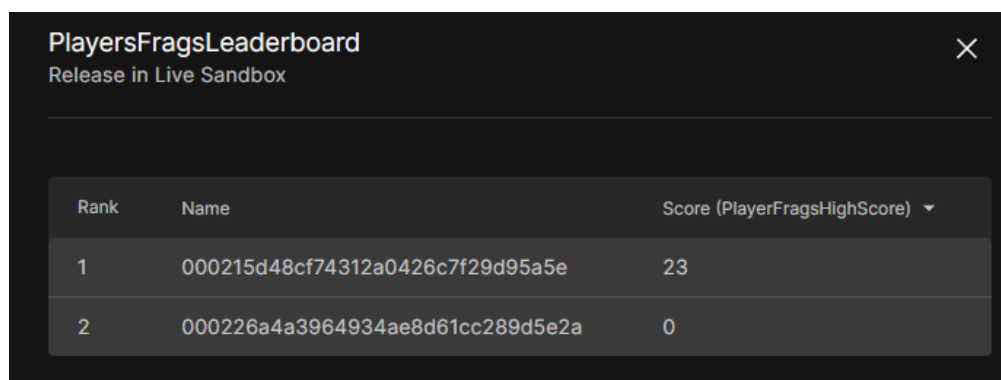


Рисунок 9 — Создание глобальной таблицы лидеров на Developer Portal

На Developer Portal можно получить доступ к созданной таблице лидеров и увидеть все записи игроков, данные которых записывались в созданную глобальную таблицу. Данная таблица доступна только владельцу приложения и является необходимой лишь во время разработки (Рисунок 10).



Rank	Name	Score (PlayerFragHighScore)
1	000215d48cf74312a0426c7f29d95a5e	23
2	000226a4a3964934ae8d61cc289d5e2a	0

Рисунок 10 — Глобальная таблица лидеров на Developer Portal

Необходимо разработать отдельную таблицу с делениями на лиги на уровне приложения, чтобы каждый пользователь имел доступ к данной таблице и имел возможность просматривать ее содержимое.

Для реализации данной задачи были созданы два виджета средствами UMG: виджет глобальной таблицы игроков и виджет записи игрока, которая добавляется в таблицу друг за другом.

Таблица представляет собой виджет, состоящий из кнопок для управления лигами и их контейнера, в который можно добавлять бесконечно много записей и прокручивать с помощью колеса мыши (Рисунок 11).

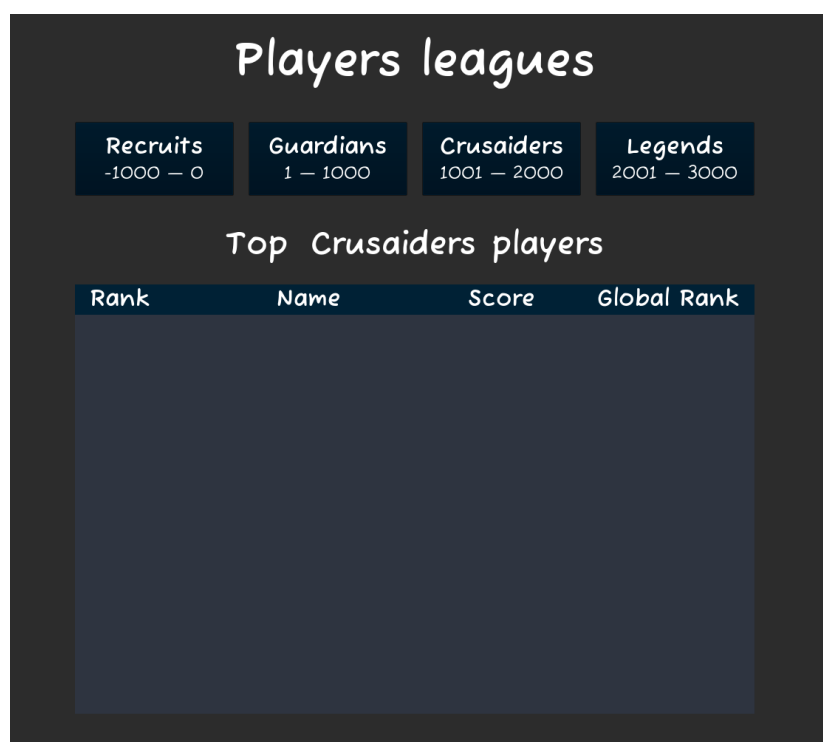


Рисунок 11 — Виджет глобальной таблицы лидеров



## Листинг 1 — Получение информации об игроках из глобальной таблицы лидеров

```
void ULab4GameInstance::QueryGlobalRanks(const int32 LeftBoundry,
const int32 RightBoundry)
{
    LeaderboardsPtr = LeaderboardsPtr == nullptr ?
OnlineSubsystem->GetLeaderboardsInterface() : LeaderboardsPtr;

    if (!LeaderboardsPtr.IsValid())
    {
        return;
    }

    FOnlineLeaderboardReadRef ReadRef =
MakeShared<FOnlineLeaderboardRead, ESPMode::ThreadSafe>();
    ReadRef->LeaderboardName = RankedLeaderboardName;
    ReadRef->ColumnMetadata.Add(FColumnMetadata(FName(TEXT("PlayerFragHighScore")), EOnlineKeyValuePairDataType::Int32));
    ReadRef->SortedColumn = FName(TEXT("Score"));
    QueryGlobalRanksDelegateHandle = LeaderboardsPtr->AddOnLeaderboardReadCompleteDelegate_Handle(
        FOnLeaderboardReadComplete::FDelegate::CreateUObject(
            this,
            &ULab4GameInstance::HandleQueryGlobalRanksResult,
            ReadRef
        )
    );

    if (!LeaderboardsPtr->ReadLeaderboardsAroundRank(
        0,
        100,
        ReadRef
    ))
    {
        LeaderboardsPtr->ClearOnLeaderboardReadCompleteDelegate_Handle(QueryGlobalRanksDelegateHandle);
        QueryGlobalRanksDelegateHandle.Reset();
    }
}
```

Проанализировав листинг 1, можно увидеть, что запрос в глобальную таблицу лидеров выполняется с помощью программного интерфейса `LeaderboardsInterface`. Далее в методе создается переменная `FOnlineLeaderboardReadRef ReadRef` ссылочного типа, которая будет содержать в себе запрашиваемые данные о таблице лидеров. Чтобы сделать запрос с помощью метода `OnlineLeaderboardInterface::ReadLeaderboardsAroundRank` необходимо указать имя глобальной таблицы, имя статистики, которую

отслеживает данная таблица, а также имя поля, в которое будет помещен результат запроса и по которому будет производиться сортировка.

Для обработки запроса данных глобальной таблицы лидеров создается коллбэк `ULab4GameInstance::HandleQueryGlobalRanksResult(const bool bWasSuccessful, FOnlineLeaderboardReadRef LeaderboardReadRef)`. В нем происходит фильтрация пользователей на лиги, в соответствии с их очками в глобальной таблице. Также для каждого пользователя создается виджет, который далее помещается в виджет глобальной таблицы на уровне приложения.

В результате каждый пользователь имеет возможность на уровне приложения просматривать глобальную таблицу лидеров с разделением игроков на лиги (Рисунок 12).

Для того, чтобы таблица лидеров заполнялась автоматически после каждого заверченного матча, необходимо в соответствии с вышеописанным алгоритмом начисления глобальных очков вычислить, какое количество очков заработал каждый игрок, выполнить их нормировку для лучшей читаемости и затем после нормировки отправить результат в EOS и отразить результат подсчетов в HUD игрока.

Для отображения результатов конца матча, где указан выигравший игрок и заработанное количество очков, также необходимо создать виджет, который будет автоматически появляться по завершении игры на экране игрока. Пример такого виджета представлен ниже (Рисунок 13).

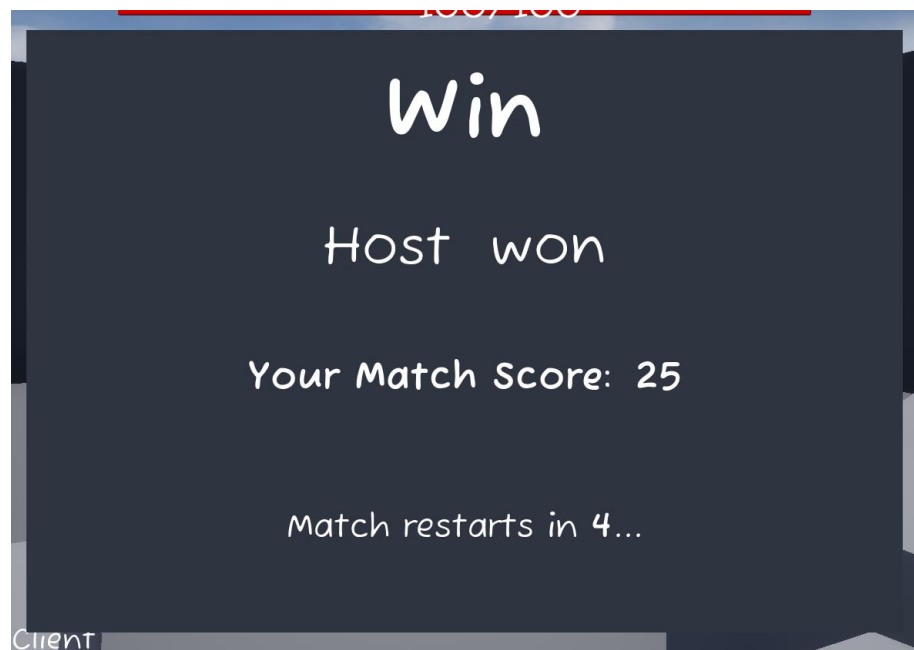


Рисунок 13 — Виджет выигравшего игрока с информацией о завершившемся матче

Управление таким виджетом также осуществляется с помощью C++. Сперва на сервере необходимо подсчитать, какое количество очков заработал каждый игрок, затем с помощью RPC, которое будет вызываться с сервера на клиент, необходимо показать данный виджет. В данном случае использование RPC типа Client оправдано, так как управление всеми виджетами во время игрового процесса осуществляется с помощью классов ANUD и APlayerController, которые доступны только клиенту, который владеет экземплярами данных классов. Ниже представлены два метода, с помощью которых выполняется управление виджетом (листинг 2).



## Листинг 2 — Методы управления виджетом с результатами матча

```
void
ALab4PlayerController::ClientGameOverToggle_Implementation(ALab4Pl
ayerState* WinnerPlayerState, float NormalizedPlayerScore)
{
    Lab4HUD = Lab4HUD == nullptr ? Cast<ALab4HUD>(GetHUD()) :
    Lab4HUD;

    if (Lab4HUD && WinnerPlayerState)
    {
        UE_LOG(LogTemp, Error, TEXT("Show UI winner"))
        Lab4HUD->ShowGameOverWidget(WinnerPlayerState,
        NormalizedPlayerScore);
        SetRestartCountdownTimer();
    }
}

void ALab4HUD::ShowGameOverWidget(const ALab4PlayerState*
WinnerPlayerState, float NormalizedPlayerScore)
{
    APlayerController* PlayerController =
    GetOwningPlayerController();

    if (PlayerController && GameOverWidgetClass)
    {
        GameOverWidget =
        CreateWidget<UMyUserWidget>(PlayerController,
        GameOverWidgetClass);
    }

    if (GameOverWidget && WinnerPlayerState)
    {
        GameOverWidget->SetWinnerText(WinnerPlayerState,
        NormalizedPlayerScore);
        GameOverWidget->AddToViewport();
    }
}
```

## 4 Разработка трех видов лобби

Лобби — это отдельная комната, куда может подключиться ограниченное количество игроков перед началом матча. Как правило, для лобби выделяется отдельный игровой уровень и отдельный класс `AGameMode`, так как для лобби на стороне сервера существуют свои правила, отличные от правил, определенных на игровом уровне. Есть два варианта лобби по реализации игрового процесса [12]:

- Пользователь имеет возможность управлять персонажем и перемещаться по карте во время ожидания остальных игроков (далее `GameAndUI`).
- На экран пользователя добавляется множество виджетов с информацией о текущем уровне, пользователю доступно только управления данными виджетами (далее `UIOnly`).

Первый тип характерен для проектов, где пользователю необходимо время, чтобы ознакомиться с принципами взаимодействия с игровым миром в предстоящем матче на следующем уровне.

Второй тип характерен для остальных проектов, где на экран пользователя необходимо вывести больше информации о текущем состоянии уровня, например, текущее количество подключившихся игроков, имя каждого игрока и его статус готовности начать предстоящий матч.

Также существует два типа лобби по стилю управления: `admin-style` лобби и `crowd-style` лобби.

`Admin-style` лобби — это вид лобби, в котором игрок, находящийся на машине, на которой запущена сессия, может самостоятельно запустить матч, нажав, например, на кнопку, расположенную в его HUD. Такой вид лобби может подойти только для игры типа `Listen-Server`, так как в таком случае сервер, расположенный на машине одного из пользователей, имеет графическое

отображение и правилами игры можно управлять с помощью графического интерфейса.

Crowd-style лобби — это вид лобби, в котором система автоматически начинает игру, как только количество подключившихся в лобби игроков достигло заданного количества и все игроки подтвердили свой статус готовности начать матч, нажав на кнопку подтверждения статуса, расположенную в HUD. Такой вид лобби может использовать в многопользовательской игре как типа Dedicated-Server, так и типа Listen-Server, так как для управления правилами игры в таком стиле не нужно никакого графического интерфейса. Большинство современных многопользовательских игр используют именно crowd-style лобби, потому что в них используются отдельные сервера.

Для проектирования шаблона многопользовательской игры важно учитывать возможность использования альтернативных решений на том или ином уровне. Для этого необходимо реализовать все вышеперечисленные виды уровня лобби.

Для реализации всех вышеперечисленных видов лобби были созданы следующие классы:

1. AEmptyLobbyGameMode — класс, производный от класса AGameMode. Данный класс определяет правила уровня лобби и дает возможность проверять статусами всех игроков и возможность управлять загрузкой следующего игрового уровня.
2. AEmptyLobbyPlayerController — класс, производный от класса APlayerController. Позволяет управлять действиями и вводи игрока, а также элементами экрана пользователя на уровне лобби.
3. AEmptyLobbyPlayerState — класс, производный от класса APlayerState. Данный класс содержит в себе реплицированную переменную bool bIsReady, которая отражает статус готовности игрока. Данный класс был создан, так как массив всех игроков в структуре AGameModeBase::GameState->PlayerArray является структурой TArray<APlayerState>.

4. `AEmptyLobbyHUD` — класс, производный от класса `AHUD`. Данный класс содержит в себе переменные, производные от `UUserWidget`, которые связаны с виджетами, представленными на экране игрока, и является связующим звеном между `AEmptyLobbyPlayerController` и `UStatusControllWidget`.
5. `UStatusControllWidget` — класс, производный от класса `UUserWidget`. Данный класс создан для управления всеми объектами, находящимися на экране на уровне лобби.
6. `UStatusGridWidget` — класс, производный от класса `UUserWidget`. Данный класс создан для управления таблицей пользователей, которая отображает текущих подключившихся игроков, их пинг и статус готовности.
7. `ALobbyGameMode` — класс, производный от класса `AGameMode`. Данный класс создан для лобби типа `GameAndUI`, который создан для выбора пешки, управляемой игроком, и управления подключениями всех пользователей для автоматического старта заданного требуемого количества игроков.

#### **4.1 Настройка игрового уровня**

Для того, чтобы создать лобби, для начала необходимо создать отдельный игровой уровень и провести его настройки (Рисунок 14).

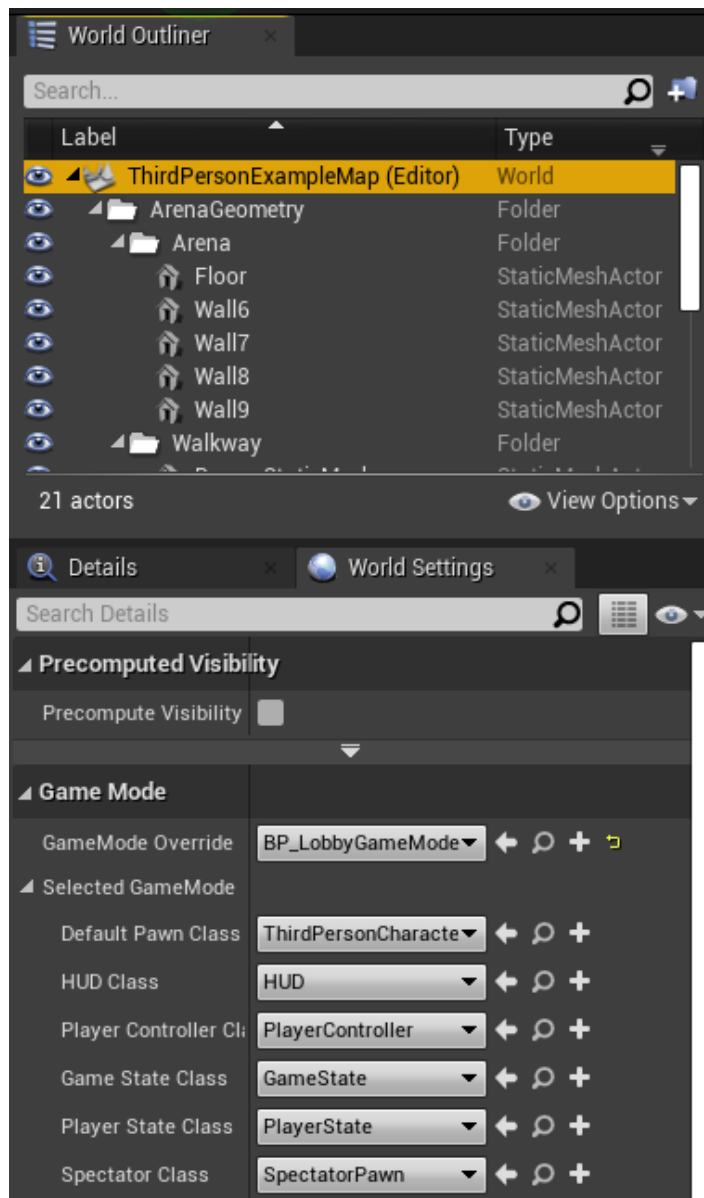


Рисунок 14 — Настройка GameAndUI лобби как отдельного игрового уровня

Отдельный игровой уровень представляет собой карту (файл с расширением. umap) [13]. Как видно на рисунке, представленном выше, карта имеет список геометрии, которая будет отображаться на сцене, и свои настройки в разделе World Outliner. Данный вариант настройки уровня характерен для GameAndUI лобби, так как на уровне присутствует геометрия и в качестве Default Pawn Class задан класс персонажа, который имеет меш для отображения на сцене. Вариант настройки уровня с UIOnly лобби представлен ниже (Рисунок 15).

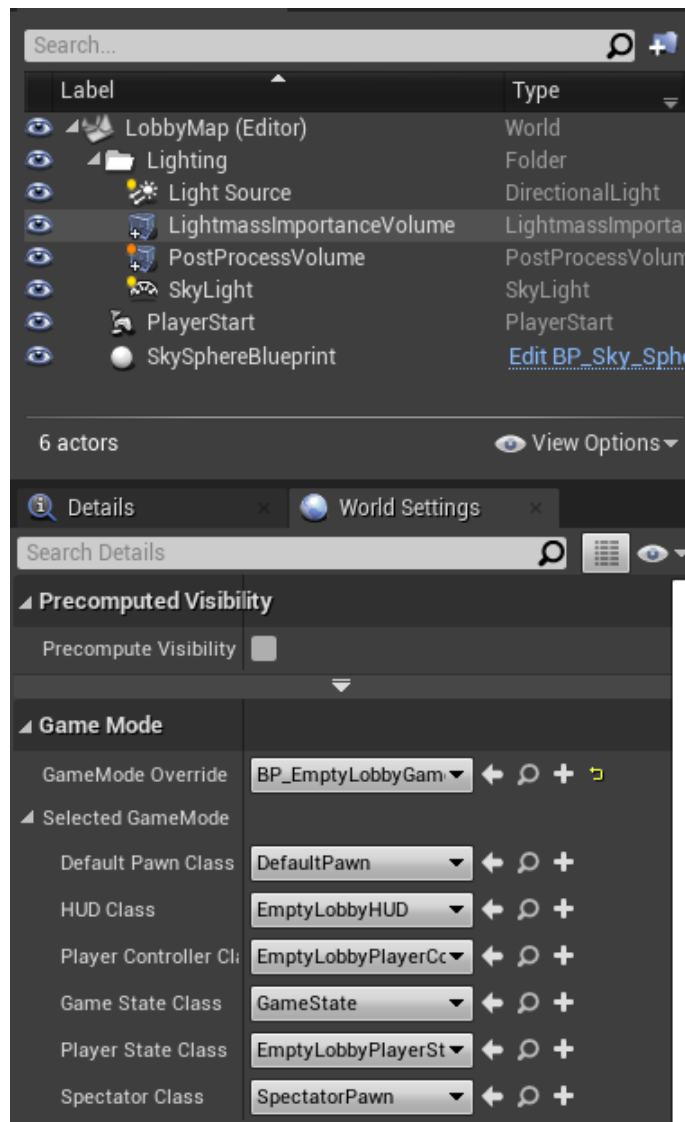


Рисунок 15 — Настройка UIOnly лобби как отдельного игрового уровня

В варианте лобби UIOnly видно, что на уровне нет геометрии и в качестве Default Pawn Class задана пешка по умолчанию.

Самыми важными настройками для карты является настройка GameMode, которая определяет, какой экземпляр класса AGameMode будет задаваться для уровня по умолчанию. Также в разделе Selected GameMode присутствует перечисление классов, экземпляры которых будут созданы на уровне по умолчанию. В итоге для уровня необходимо задать класс ALobbyGameMode, производный от класса GameMode, который будет определять правила подключения игроков к лобби. Также для пункта Default Pawn Class необходимо

указать пешку, которую сервер будет создавать в игровом уровне на месте компонента NetworkPlayerStart (Рисунок 16).

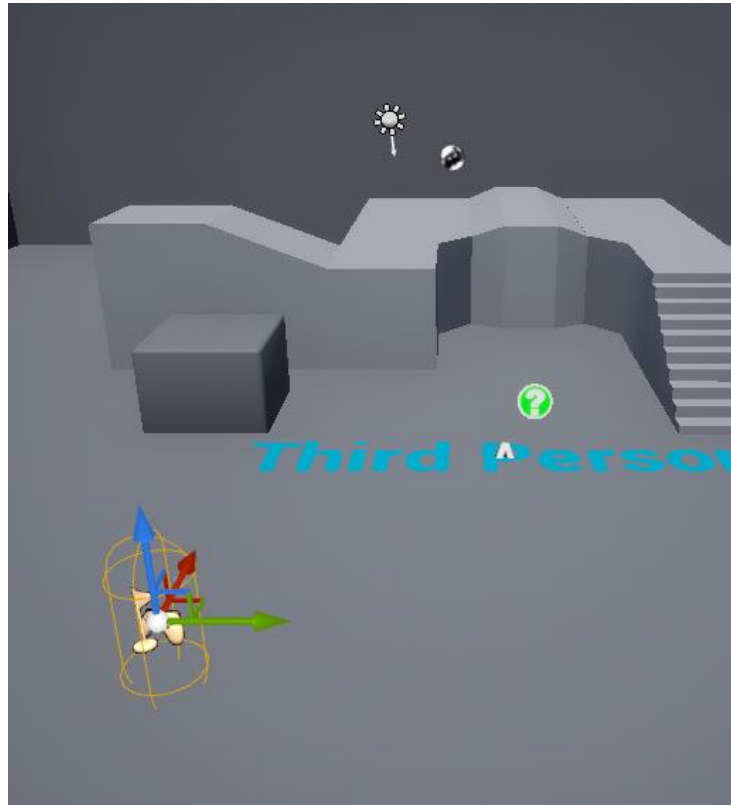


Рисунок 16 — Компонент NetworkPlayerStart на уровне лобби

Для GameAndUI лобби для управления пешкой задан режим управления FInputModeGameOnly — режим управления по умолчанию, в котором игроки могут управлять виджетами и своим персонажем. Для UIOnly лобби необходимо предоставить режим доступа управления только с помощью курсора мыши, для этого необходимо выполнять ряд действий в функции virtual APlayerController::ReceivedPlayer (листинг 3).

Листинг 3 — Задание способа ввода управления пешкой

```
void AEmptyLobbyPlayerController::ReceivedPlayer()
{
    Super::ReceivedPlayer();
    FInputModeUIOnly InputModeUIOnly;
    InputModeUIOnly.SetLockMouseToViewportBehavior(
        EMouseLockMode::DoNotLock);
    SetInputMode(InputModeUIOnly);
    bShowMouseCursor = true;
}
```

Задание классов по умолчанию также можно задать в конструкторе самого класса `ALobbyGameMode` (листинг 4).

#### Листинг 4 — Задание классов по умолчанию в конструкторе класса

```
ALobbyGameMode::ALobbyGameMode()
{
    static ConstructorHelpers::FClassFinder<APawn>
    PlayerPawnBPClass(
        TEXT("/Game/ThirdPersonCPP/Blueprints/ThirdPersonCharacter")
    );
    if (PlayerPawnBPClass.Class != nullptr)
    {
        DefaultPawnClass = PlayerPawnBPClass.Class;
    }
}
```

## 4.2 Управление подключениями пользователей

Управление подключениями пользователей, которые подключаются к уровню лобби, осуществляется с помощью функции `virtual void AGameMode::PostLogin(APlayerController* NewPlayer)` [14]. Данная функция, как видно, из ее определения, является виртуальной, и выполняется каждый раз на сервере, когда подключился новый игрок. Данная функция также принимает в качестве параметра указатель `APlayerController` подключившегося игрока. С помощью доступа к экземпляру класса `APlayerController` можно с помощью `RPC` типа `Client` вывести на экран каждого пользователя имя игрока, который только что подключился к уровню лобби (Рисунок 17).





Рисунок 17 — Отслеживание подключающихся игроков к уровню лобби

Для того, чтобы отследить общее количество подключившихся игроков, необходимо воспользоваться классом `AGameState`. Экземпляр данного класса доступен на сервере и содержит в себе поле `PlayerArray` — структура данных типа `TArray`, которая содержит в себе список всех игроков, подключенных к игре на момент обращения к члену класса.

Каждый раз, когда новый пользователь подключается к лобби, происходит сравнение общего количества игроков в поле `PlayerArray` с заранее заданным количеством игроков. Когда количество игроков совпадает необходимо выполнять команду `ServerTravel(FString("/Game/Maps/GamePlayMap?listen"))`, заранее задав свойство класса `AGameMode` `bUseSeamlessTravel = true`. Функция `ServerTravel` выполняется на сервере и выполняет перемещение сервера на другой уровень или карту. Также, важно знать, что все подключенные клиенты также последуют за сервером и они переподключатся к новому уровню. Это происходит из-за того, что сервер неявно вызывает у каждого подключенного клиента функцию `ClientTravel(FString("/Game/Maps/GamePlayMap?listen"))`.

Общее количество игроков, которое необходимо для подключения задается с помощью настроек сессии, которые находятся в структуре `FOnlineSessionSettings` в члене класса `NumPublicConnections`.

Свойство `bUseSeamlessTravel` необходимо для того, что уже ранее подключенные к серверу клиенты переподключились к новому уровню, сохраняя при этом подключение к текущему серверу. Для использования такого типа переподключения необходимо создать `TransitionMap` — пустой уровень, который является промежуточным между старым уровнем карты и новым игровым уровнем, к которому будут подключены все игроки, находящиеся в лобби.

### 4.3 Круговая задержка (Round Trip Time)

Для того, чтобы все клиенты имели достаточное количество времени для подключения к уровню, где будет происходить игра, необходимо перед началом матча добавить промежуточное состояние матча с таймером, в котором каждый игрок имеет возможность осмотреть карту с помощью элементов управления и подождать других игроков для подключения (Рисунок 18).



Рисунок 18 — Ожидание подключения всех игроков на игровой уровень

Как видно на рисунке, представленном выше, в HUD игрока используется таймер. Есть несколько проблем, которые могут возникнуть во время использования таймера:

1. Клиенты должны использовать время загрузки уровня, которое определяется на сервере, так как каждый клиент использует разное время загрузки игрового уровня.
2. При запросе серверного времени клиент может получить неправильное время, вызванное присутствием межсетевых задержек, возникающее при выполнении запросов от сервера к клиенту и от клиента к серверу.

Проблемы, представленные выше, описывают проблему круговых задержек в сети, которые необходимо решить с помощью множественных вызовов RPC (Рисунок 19).

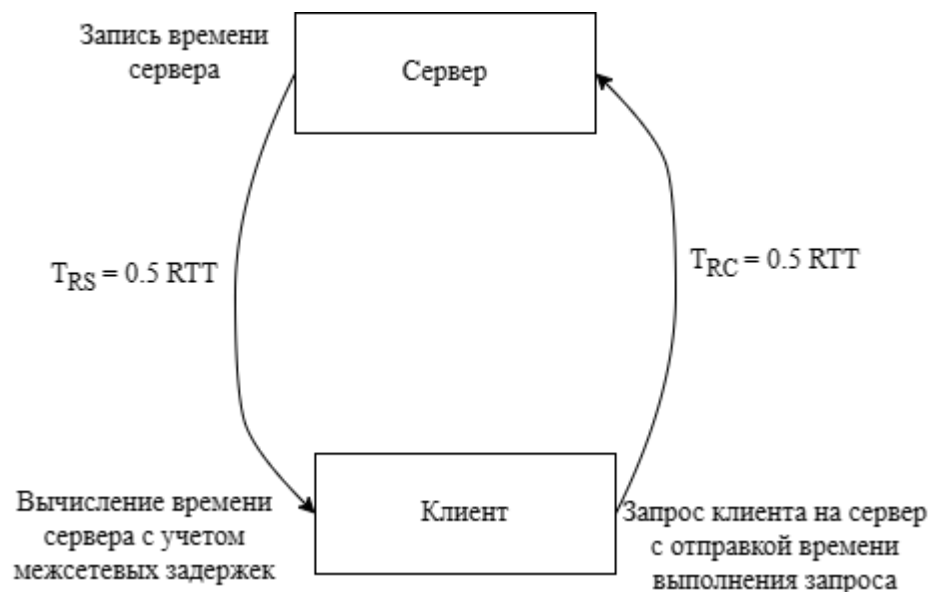


Рисунок 19 — Схема межсетевых задержек при выполнении клиент-серверных запросов

Для того, чтобы найти разницу во времени, с которой игровой уровень был загружен на сервере и на клиенте необходимо выполнить последовательный вызов RPC типа Client и Server, вызов которых представлен выше на схеме:

1. Отправить запрос типа Server RPC с клиента `ServerRequestServerTime(GetWorld->GetTimeSeconds)`, где в качестве

единственного аргумента вызова передается время, когда клиент сделал запрос.

2. Отправить запрос типа Client RPC ClientResponseServerTime (GetWorld->GetTimeSeconds, ClientRequestTime), где в качестве аргументов вызова передаются время, когда сервер получил запрос, и время, когда клиент сделал запрос.
3. Рассчитать время круговой задержки (далее RTT), сложив время, которое потребовалось серверу, чтобы получить запрос от клиента ( $T_{RC}$ ), и время, которое потребовалось клиенту, чтобы получить ответ от сервера ( $T_{RC}$ ).
4. Оценить реальное текущее время сервера на клиенте с учетом межсетевых задержек, прибавляя к текущему времени сервера половину RTT.
5. Вычислить на клиенте разницу во времени, с которой игровой уровень был запущен на сервере и на клиенте.

Оценив разницу времени загрузки игрового уровня на сервере и на клиенте, можно инициализировать таймер в HUD клиента. В итоге в HUD каждого клиента в промежуточное состояние матча будет отображаться таймер, который синхронизирован с серверным временем загрузки игрового уровня.

#### **4.4 Admin-style лобби**

Для реализации admin-style лобби необходимо в HUD игрока, который является владельцем сессии, добавить соответствующие элементы управления в виде дополнительного графического интерфейса.

Игроку, который является владельцем сессии, добавляется в его HUD, кнопка, по нажатию которой происходит перемещение сервера на другой уровень. У других пользователей, которые являются клиентами, не должно быть соответствующего элемента управления лобби.

Чтобы однозначно определить пользователя, который является владельцем лобби, можно вызвать функцию `UGamePlayStatics::GetPlayerController(this, 0)` в экземпляре класса `ALobbyGameMode`, где 0 — это индекс элемента в массиве всех экземпляров класса `APlayerController`. Так как владелец лобби подключается к уровню лобби самым первым, ему всегда будет соответствовать индекс 0.

Получив доступ к экземпляру `APlayerController` владельца лобби, можно добавить в его HUD ранее созданный с помощью средств UMG виджет, в котором будет содержаться кнопка для начала игрового процесса и соответствующее информационное сообщение (Рисунок 20).

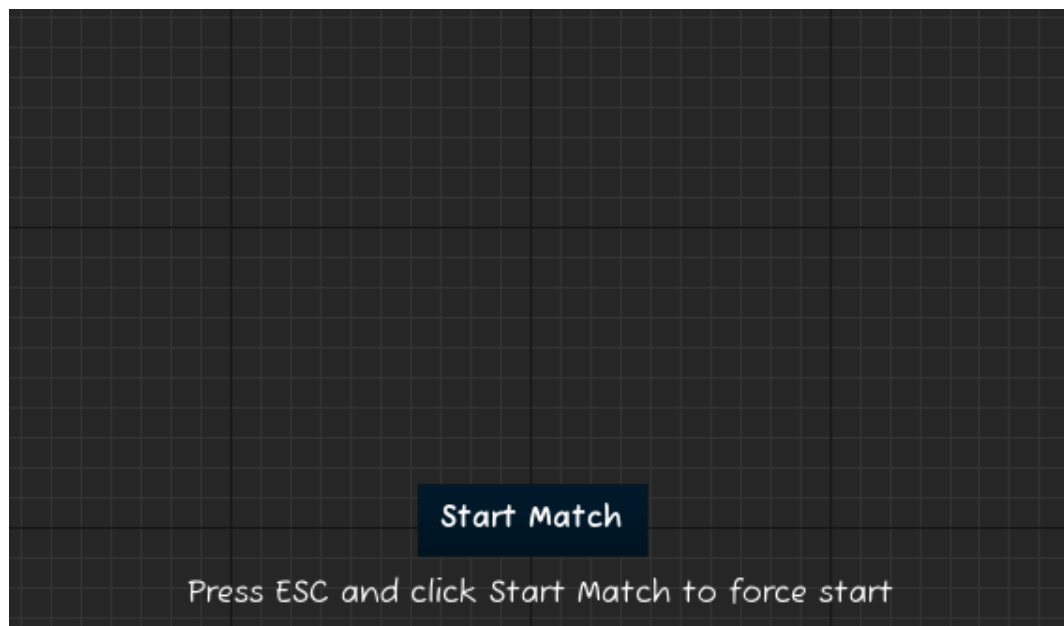


Рисунок 20 — Создание виджета для admin-style лобби

При перемещении сервера на конечный игровой уровень из экрана игрока необходимо удалить элемент управления матчем. В итоге в HUD пользователя, являющегося владельцем лобби, при первой загрузке соответствующего уровня появляется элемент управления лобби, с помощью которого возможно заранее, не дожидаясь заданного количества игроков, запустить матч (Рисунок 21).

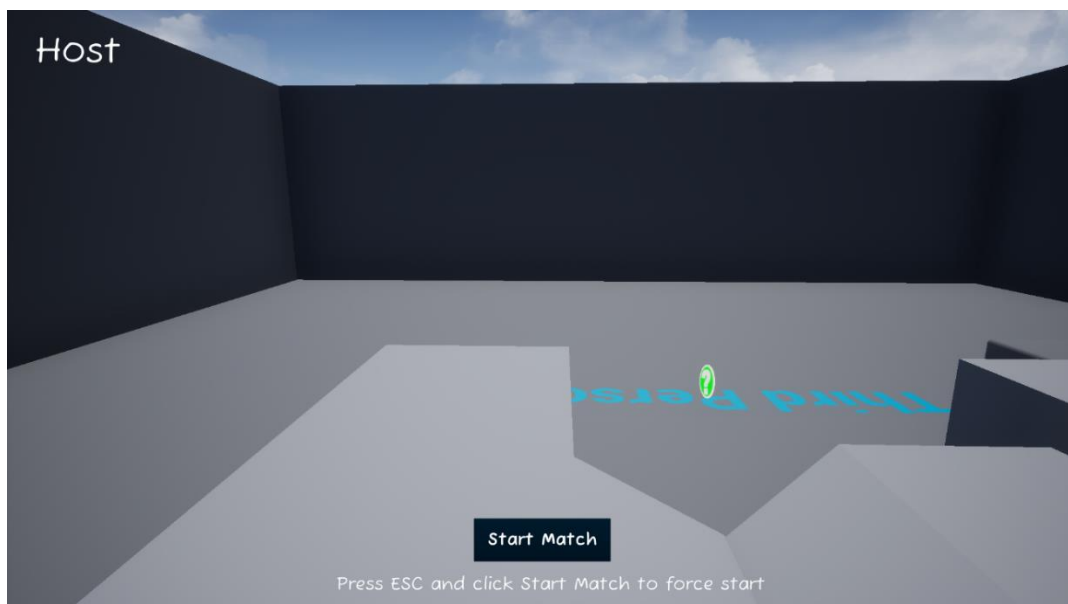


Рисунок 21 — Экран игрока, владеющего GameAndUI лобби

#### 4.5 Admin-style + crowd-style лобби

Для реализации admin-style лобби и crowd-style лобби необходимо в HUD каждого игрока добавить кнопки для изменения статуса готовности, а также необходимо добавить таблицу со всеми подключившимися игроками, где отображается имя каждого игрока и его статус готовности запустить матч.

Управление статусом готовности можно осуществить с помощью двух кнопок Ready и Cancel. Для того, чтобы пользователь случайно не смог нажать на элемент управления готовностью, необходимо также добавить обработку статуса длительного нажатия и полосу прокрутки, которая показывает, какое количество времени осталось производить нажатие (Рисунок 22).

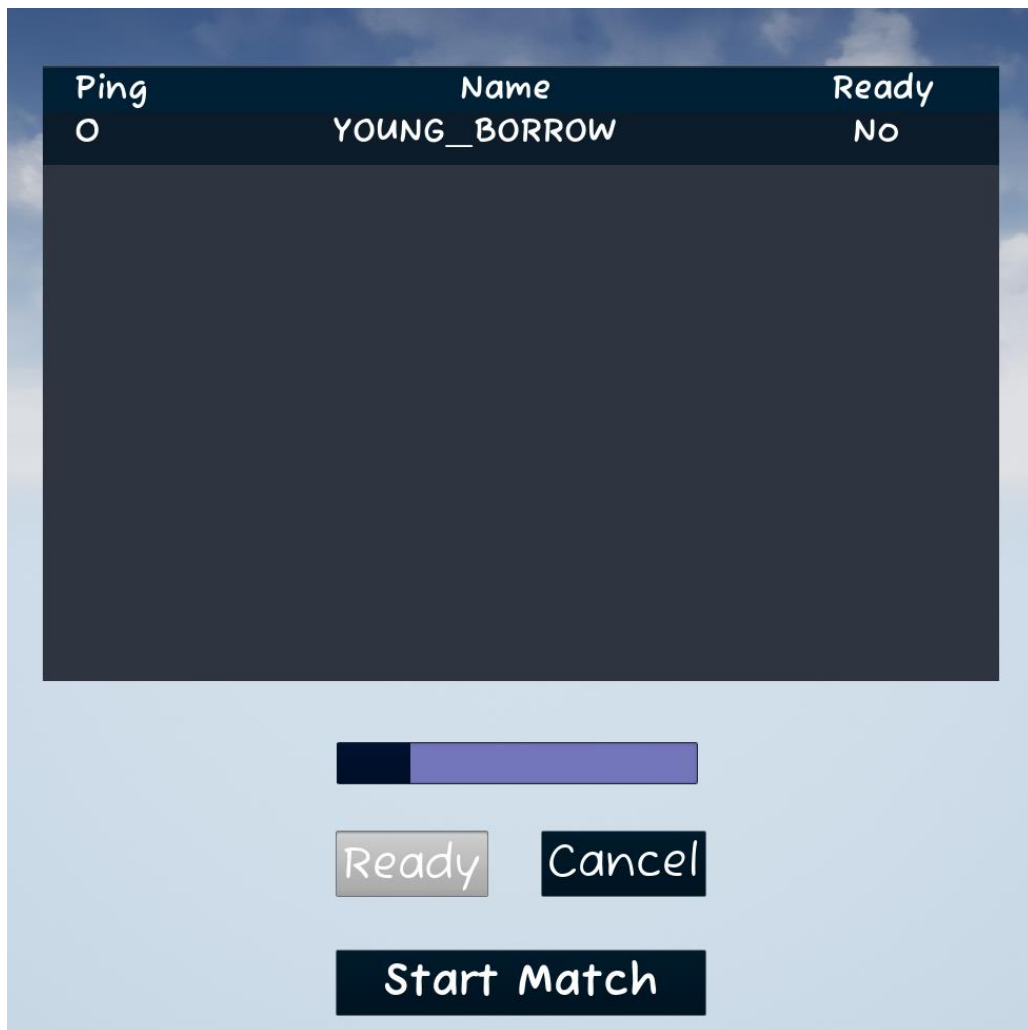


Рисунок 22 — Виджет пользователя для admin-style + crowd-style лобби

Кнопку “Start Match” также необходимо отображать только в HUD того игрока, кто является хостом сессии. Для определения сетевой роли игрока, владеющего HUD, можно использовать условие с проверкой сетевой роли (листинг 5).

Листинг 5 — Определение сетевой роли игрока, владеющего виджетом

```
EmptyLobbyOwningController =
Cast<AEmptyLobbyPlayerController>(GetOwningPlayer());
if (EmptyLobbyOwningController &&
    EmptyLobbyOwningController->GetLocalRole() ==
    ENetRole::ROLE_AutonomousProxy &&
    EmptyLobbyOwningController->GetRemoteRole() ==
    ENetRole::ROLE_Authority)
{
    StartMatchButton->SetVisibility(ESlateVisibility::Hidden);
}
```

Чтобы обеспечить покадровое обновление полосы прокрутки, представленной выше на рисунке, необходимо расширить возможности класса, отвечающего за управление данным виджетом. Так как каждый виджет, производный от класса `UUserWidget` не является потомком класса `AActor`, в котором реализована виртуальная функция `void Tick(float DeltaSeconds)`, необходимо выполнить множественное наследование и самостоятельно реализовать в классе виджета функцию, альтернативную функции `void Tick(float)` (листинг 6).

Листинг 6 — Расширение функционала класса виджета, производного от `UObject`

```
class LAB4_API UStatusControll : public UUserWidget, public
FTickableGameObject
{
    protected:
    virtual bool Initialize() override;
    virtual void Tick(float DeltaTime) override;
    virtual bool IsTickable() const override;
};
void UStatusControll::Tick(float DeltaTime)
{
    if (bIsProgressbarVisible)
    {
        SetPorgressbarPercantage();
    }
}
bool UStatusControll::IsTickable() const
{
    return true;
}
TStatId UStatusControll::GetStatId() const
{
    return TStatId();
}
```

После добавления и реализации функций, представленных выше, в теле функции `void UStatusControll::Tick(float DeltaTime)` необходимо выполнять код, который должен повторяться каждый тик.

Для начала загрузки нового игрового уровня система каждый раз проверяет статус готовности каждого игрока, когда хотя бы один из игроков изменил свой статус готовности с помощью элементов управления в своем HUD.



Данная логика всегда выполняется на сервере с помощью итерирования по массиву всех игроков.

Также на экран всех пользователей необходимо добавить информационное сообщение, чтобы всем пользователям было ясно, что причиной отложенного старта не является какая-либо непредвиденная внутренняя ошибка. Длительность видимости такого сообщения должна быть не больше, чем длительность нажатия кнопки для изменения статуса, чтобы на экране игрока не возникало постоянного визуального шума (Рисунок 23).

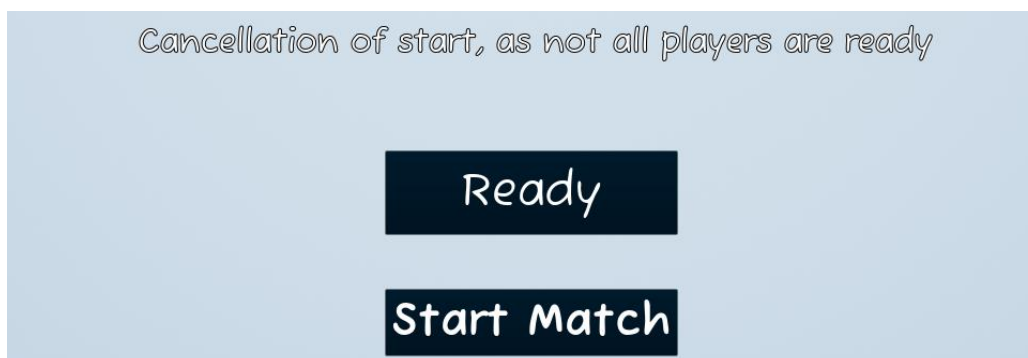


Рисунок 23 — Информационное сообщение об отмене старта предстоящего матча

## 5 Внутриигровой голосовой чат

Внутриигровой голосовой чат — еще одна из возможностей современных многопользовательских игр [15]. Голосовой чат предоставляет пользователям графического приложения возможность просто, без использования стороннего программного обеспечения, осуществлять общение. В Unreal Engine 4 в EOS также существуют программные интерфейсы, которые предоставляет низкоуровневые инструменты для внедрения голосовой связи в графическое приложение.

Для того, чтобы предоставить пользователю возможность производить общение, необходимо выполнять ряд следующих действий:

1. Интегрировать программный интерфейс `IVoiceChat` в проект;
2. Создать пользователя типа `IVoiceCharUser` для каждого игрока сразу после успешного процесса авторизации;
3. Связать пользователя с системой EOS с помощью метода `IVoiceCharUser::Login`;
4. Присоединить авторизованного пользователя к голосовому каналу лобби с помощью метода `IVoiceCharUser::JoinChanel`;
5. При выходе пользователя из системы пользователю необходимо покинуть голосовой канал лобби с помощью метода `IVoiceCharUser::LeaveChannel`;
6. После выхода пользователя из канала удалить сущность класса `IVoiceCharUser` с помощью метода `IVoiceCharUser::Uninitialize`.

После реализации вышеперечисленных шагов пользователь приложения может передавать свои голосовые данные и слышать других пользователей. Важно заметить, что передача голосовых данных будет происходить постоянно. Следовательно, пользователям необходимо предоставить возможность отправлять свои голосовые данные только тогда, как у них возникнет соответствующая необходимость. Для реализации данной возможности

необходимо добавить настройку, при которой пользователи смогут коммуницировать друг другом, например, по нажатию кнопки.

В файле /Config/DefaultEngine.ini необходимо добавить секции, которые сообщают, что приложение будет использовать программный интерфейс голосового чата (листинг 7).

Листинг 7 — Инициализация настроек для использования интерфейса

ГОЛОСОВОГО ЧАТА

```
[OnlineSubsystem]
DefaultPlatformService=EOS
bHasVoiceEnabled=true

[Voice]
bEnabled=true
```

Также для передачи голосовых данных по нажатию кнопки необходимо в файлах /Config/DefaultGame.ini и /Config/DefaultInput.ini привязать имя функции, которая будет выполнять по нажатию на определенную клавишу (листинг 8).

Листинг 8 — Добавление настроек для общения пользователей по нажатию клавиши

```
# DefaultInput.ini
[/Script/Engine.InputSettings]
+ActionMappings=(ActionName="PushToTalk",Key=T,bShift=False,bCtrl=False,bAlt=False,bCmd=False)

[/Script/Engine.PlayerInput]
+DebugExecBindings=(Key=T,Command="ToggleSpeaking true | OnRelease ToggleSpeaking false")

# DefaultGame.ini
[/Script/Engine.GameSession]
bRequiresPushToTalk=true
```

В результате у пользователя появляется возможность взаимодействовать с другими игроками посредством голосового чата с регулировкой по нажатию клавиши.

## ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были решены следующие задачи:

1. Реализована автоматическая система начисления очков в глобальную таблицу лидеров после каждого завершенного матча.
2. Реализована глобальная таблица лидеров на уровне приложения с разделением на игровые лиги.
3. Реализовано подключение к лобби перед началом матча в трех разных стилях.

Во время выполнения практических задач были улучшены необходимые в разработке навыки чтения и понимания официальной документации, улучшены навыки программирования на ЯП C++, закреплены базовые знания работы с игровыми сессиями и знания сетевого программирования в UE 4, также были получены практические навыки работы с программным интерфейсом управления таблицами лидеров, программным интерфейсом создания лобби. Также были закреплены навыки по работе и настройке приложения на Developer Portal. Немало важны полученные знания о сетевом программировании в Unreal Engine, которые в дальнейшем будут использованы для разработки подобных графических приложений с сетевой частью.

В результате выполнения проекта был разработан шаблон многопользовательской игры с возможностью играть с другими игроками по сети Интернет, который поддерживает возможности современных многопользовательских игр. Важным преимуществом проекта является его полная разработка на языке программирования C++, что позволит в дальнейшем не терять производительность и скорость приложения, а также редактировать исходный код под любые нужды. В настоящее время проект уже может быть размещен на площадках цифровой дистрибуции и быть использован в качестве начального шаблона в другом проекте.

Дальнейшая работа над проектом предполагает проведение комплекса мер по улучшению визуальной составляющей и других аспектов таких, как реализация правдоподобной анимации стрельбы, анимации ликвидации игрока и других анимаций, оптимизация проекта с точки зрения рендеринга, а также решение проблем с возможными межсетевыми задержками.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Services Documentation. Unreal Engine Documentation [Электронный ресурс] // Режим доступа: <https://dev.epicgames.com/docs> (дата обращения 25.11.2022).
2. Auth Interface. Unreal Engine Documentation [Электронный ресурс] // Режим доступа: <https://dev.epicgames.com/docs/epic-account-services/auth-interface> (дата обращения 26.11.2022).
3. Platform Interface. Unreal Engine Documentation [Электронный ресурс] // Режим доступа: <https://dev.epicgames.com/docs/game-services/eos-platform-interface> (дата обращения 26.11.2022).
4. EOS Game Services. Unreal Engine Documentation [Электронный ресурс] // Режим доступа: <https://dev.epicgames.com/docs/game-services> (дата обращения 26.11.2022).
5. EOS Account Services. Unreal Engine Documentation [Электронный ресурс] // Режим доступа: <https://dev.epicgames.com/docs/epic-account-services>. (дата обращения 27.11.2022).
6. EOS SDK Errors Code. Unreal Engine Documentation [Электронный ресурс] // Режим доступа: <https://dev.epicgames.com/docs/epic-online-services/sdk-error-codes>. (дата обращения 01.12.2022).
7. Traveling in Multiplayer. Unreal Engine Documentation [Электронный ресурс] // Режим доступа: <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Networking/Travelling/> (дата обращения 20.04.23).
8. Расширяем возможности UObject в Unreal Engine 4. Хабр [Электронный ресурс] // Режим доступа: <https://habr.com/ru/companies/pixonix/articles/475622/> (дата обращения 24.04.23).
9. RPCs. Unreal Engine Documentation [Электронный ресурс] // Режим доступа: <https://docs.unrealengine.com/4.27/enUS/InteractiveExperiences/Networking/Actors/RPCs/> (дата обращения: 25.04.23).

10. Using Epic Online Services as a Voice Chat provider. Unreal Engine Documentation [Электронный ресурс] // Режим доступа: <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Online/VoiceInterface/EOSVoiceChatPlugin/> (дата обращения: 20.05.23).
11. Voice chat on listen servers. EOS Online Subsystem [Электронный ресурс] // Режим доступа: [https://docs.repdoint.games/eos-online-subsystem/docs/voice\\_chat\\_listen\\_servers](https://docs.repdoint.games/eos-online-subsystem/docs/voice_chat_listen_servers) (дата обращения 22.05.23).
12. Добавление голосового чата в игру Unreal Engine 4 Habr [Электронный ресурс] // Режим доступа: <https://habr.com/ru/articles/717474/> (дата обращения 30.05.23)
13. Уильям Шериф. Unreal Engine 4.x Scripting with C++ Cookbook / Уильям Шериф, Стивен Уиттл, Джон Доран. — Packt Publishing, 2019 г. — 708 с.
14. Арам Куксон. Unreal Engine 4 Game Development in 24 Hours, Sams Teach Yourself / Арам Куксон, Райан Даулингсока, Клинтон Крамплер. — Москва: Бомбора, 2019 г. — 528 с.
15. Маркус Ромеру. Blueprints Visual Scripting for Unreal Engine 2nd Edition / Маркус Ромеру, Брендэн Сьюэлл. — Packt Publishing, 2019 г. — 380 с.

## ПРИЛОЖЕНИЕ А

Список графических материалов:

1. Виджеты при запуске приложения
2. Виджеты при создании сессии и подключении к сессии
3. Виджеты с результатами завершившегося матча
4. Внутриигровое меню
5. Начало игрового процесса
6. Работа приложения на игровом уровне
7. Таблица лидеров на уровне приложения
8. Управление приложением на уровне лобби