# СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc59150661)

[1 СРЕДСТВА РАЗРАБОТКИ ИГРОВЫХ СЕТЕВЫХ ПРИЛОЖЕНИЙ 5](#_Toc59150662)

[1.1 Обзор игрового движка *Unity* 5](#_Toc59150663)

[1.2 Обзор игрового движка *Unreal Engine* 4 7](#_Toc59150664)

[1.3 Сравнение игровых движков *Unity* и *Unreal Engine* 4 11](#_Toc59150665)

[1.4 Протоколы передачи данных транспортного уровня *UDP* и *TCP* 12](#_Toc59150666)

[1.5 Актуальность разработки компьютерных игр в настоящее время 14](#_Toc59150667)

[2 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СЕТЕВОГО ПРИЛОЖЕНИЯ «ШУТЕР ОТ ПЕРВОГО ЛИЦА» 16](#_Toc59150668)

[2.1 Архитектура и синхронизация сетевого взаимодействия 17](#_Toc59150669)

[2.2 Сетевой компонент приложения «Шутер от первого лица» 19](#_Toc59150670)

[2.3 Физика игрового приложения «Шутер от первого лица» 21](#_Toc59150671)

[3 ВЕРИФИКАЦИЯ И АПРОБАЦИЯ РАЗРАБОТАННОГО СЕТЕВОГО ПРИЛОЖЕНИЯ «ШУТЕР ОТ ПЕРВОГО ЛИЦА» 28](#_Toc59150672)

[ЗАКЛЮЧЕНИЕ 35](#_Toc59150673)

[Список используемых источников 36](#_Toc59150674)

ПРИЛОЖЕНИЕ А [Листинг программы «Шутер от первого лица» 37](#_Toc59150676)

ПРИЛОЖЕНИЕ Б [Функциональная схема приложения 55](#_Toc59150678)

ПРИЛОЖЕНИЕ В [Диаграмма классов 56](#_Toc59150680)

ПРИЛОЖЕНИЕ Г [Руководство пользователя 57](#_Toc59150682)

ПРИЛОЖЕНИЕ Д [Руководство программиста 61](#_Toc59150683)

ПРИЛОЖЕНИЕ Е [Руководство системного программиста 62](#_Toc59150684)

# ВВЕДЕНИЕ

Игры наравне с наукой и работой относятся к одним из ключевых видов человеческой деятельности. Люди с незапамятных времён увлекаются играми, чтобы снять напряжение, расслабиться и забыть на время о проблемах окружающего мира. Индустрия компьютерных игр, в наше время, очень сильно развилась, и может легко, быстро и относительно недорого предоставить игру любого жанра и удовлетворить предпочтения каждого игрока, а вычислительная мощность современных цифровых устройств, с помощью фантастической красоты компьютерной графики, может надолго затянуть его в виртуальный мир, наполненный потрясающими пейзажами и завораживающими приключениями.

Актуальность разработки сетевых игр жанра «Шутер от первого лица» объясняется тем, что это один из самых популярных жанров компьютерных экшен-игр. Главной целью игрока, как правило, является остаться в живых и выполнить миссию, которая в процессе требует уничтожить противников из огнестрельного или другого дистанционного оружия. Игры данного жанра позволяют развлечь любого, кому нравится соревноваться и демонстрировать такие личные качества, как ум, смекалка, реакция, точность, стратегическое планирование и внимательность. В настоящее время компьютерные видеоигры помимо развлечения также несут обучающий, расслабляющий и даже лечебный характер. Ведь ни для кого не секрет, что всего лишь пара часов, проведённая за любимой игрой, может поднять настроение и избавить игрока от депрессии. На данный момент, видеоигры являются очень легкодоступным источником удовольствия. В них можно играть как одному, так и в компании друзей. Видеоигры доступны на большинстве современных гаджетов: смартфонах, компьютерах, консолях, телевизорах и даже на некоторых электронных наручных часах. В видеоигры играют люди всех ориентаций и возрастных категорий, поэтому в них легко можно познакомиться и найти компанию по разделяемым интересам.

Разработка видеоигр является очень увлекательным и прибыльным занятием. Чаще всего популярные видеоигры относятся к категории бесплатных онлайн-игр, но за счёт размещения рекламы и продажи костюмов для игровых персонажей разработчики получают хороший доход. Также разработка игр вносит свой вклад в развитие культуры, так как любимые игры вдохновляют людей писать музыку, монтировать видео и рисовать картины. Видеоигры сближают людей и помогают им лучше узнать друг друга и найти общий язык. Из всего вышеперечисленного ясно, что разработка игр весьма актуальна на данный момент и в будущем будет только развиваться.

# СРЕДСТВА РАЗРАБОТКИ ИГРОВЫХ СЕТЕВЫХ ПРИЛОЖЕНИЙ

## 1.1 Обзор игрового движка *Unity*

*Unity*3*D* – кроссплатформенный игровой движок от компании *Unity* *Technologies*. История создания движка довольно интересная и поучительная. Она интересная, потому что двое парней захотели сделать игру, но для этого им не подходили существующие инструменты. И они решили сделать свой движок, а потом уже делать на нем игру. И после того, как они сделали движок, они поняли, что им не так-то и интересно делать игры, а больше нравится заниматься непосредственно движком. Так и началась история одного из самых известных и мощных движков. А поучительная эта история потому, что никогда не знаешь, чем обернется то или иное начинание [1, с. 5].

*Unity* – это бесплатный движок. Единственное ограничение – при запуске игры показывается логотип *Unity*, от которого можно избавиться, только купив расширенную версию. Доступность и простота движка – это то, что привлекло многих к разработке игр на нём.

*Unity* использует компонентно-ориентированный подход. Все в игре – это объект *GameObject*, куда добавлены различные компоненты. Например, если мы делаем платформер, мы добавляем *GameObject*, и к нему добавляем графический компонент для рендеринга игрока и компонент управления, чтобы можно было перемещать игрока по сцене, используя клавиатуру или мышь. Таких различных компонентов можно добавить любое количество к любому *GameObject*. То есть, создание игры в *Unity* – это добавление *GameObject*-ов, и добавление им нужных компонентов.

Но игровой движок не так уж и прост, как кажется на первый взгляд. Чтобы сделать что-то нетривиальное, придется создавать свои компоненты. В терминах *Unity* они называются скриптами. Пишутся они на языке *C*#. Написание своих компонентов – это довольно сложное занятие. Фактически, это обычное программирование. Так что, без знания языка программирования придется не просто.

*Unity* – хороший выбор для создания средних и лёгких по сложности проектов, как для ПК, так и для мобильных устройств. Вдобавок, разработку упрощает огромное сообщество пользователей, которое всегда помогает разобраться с возникшими вопросами, и большое количество готовых ассетов, включая скрипты.

На *Unity* не рекомендуется разрабатывать огромные проекты *AAA*-класса. Сама по себе идея со скриптами хороша, но довольно медлительна. К тому же, язык *C*# – интерпретируемый. Несмотря на все *JIT*-оптимизации, он медленней, чем тот же *C*++. Сотни объектов на сцене со сложными компонентами могут убить производительность.

Впрочем, все зависит от мастерства программиста. Существуют исключения, когда опытные люди создавали на *Unity* большие проекты с хорошей производительностью. Правда, им приходилось много чего переделывать в движке под себя. Так что, перед тем как браться за что-то грандиозное, требуется хорошо разобраться во внутреннем устройстве движка и его деталях. Пользовательский интерфейс игрового движка *Unity* представлен на рисунке 1.1:

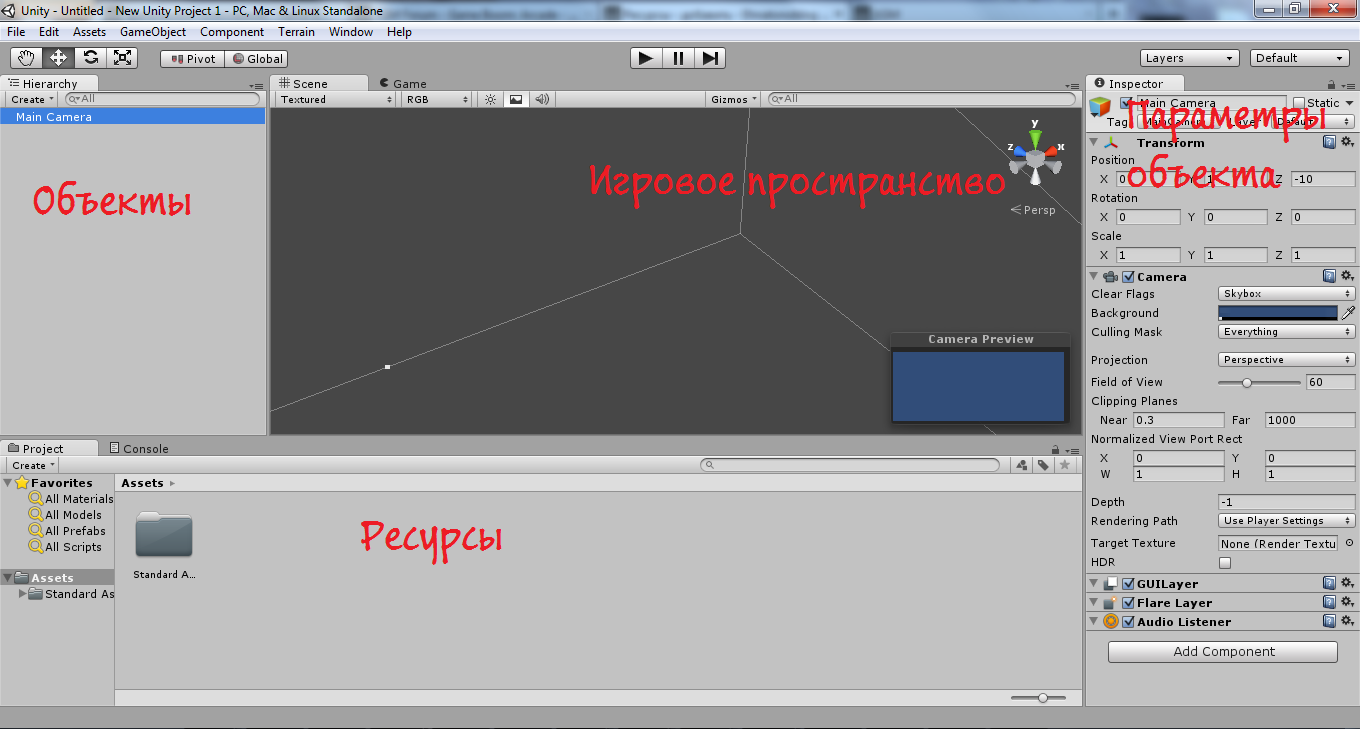


Рисунок 1.1 – Пользовательский интерфейс игрового движка *Unity*

Один из козырей *Unity* – это список поддерживаемых платформ, где может запускаться приложения. *Unity* работает почти везде – на ПК (все операционные системы), на *Android*, на *iOS*, на *SmartTV*, в браузере, на различных экзотических системах – например, *Tizen* *OS*. Написание скриптов для игровых объектов происходит в *MonoDevelop* (идет по умолчанию), либо при использовании стороннего редактора. Многие используют *Visual Studio*. Некоторые настраивают для этих целей *Sublime Text*.

Мощный плюс *Unity* – это ассеты. Все в игре, включая код, картинки, представляется ассетами. Ассеты можно экспортировать, импортировать. Таким образом, сторонние разработчики могут делать целые заготовки для игр. Все, что вам останется – это заменить картинки, подправить скрипты – и можно считать игру уникальной. Нюансом является то, что различные ассеты могут быть несовместимы между собой как в прямом смысле, так и не подходить по стилю. Также есть специальный онлайн-магазин – *Unity Asset Store*. Там продаются готовые ассеты от сторонних разработчиков. Любой желающий может сделать свой ассет, и выложить его в продажу в этом магазине. Некоторые люди сделали на этом целый бизнес, благодаря большому рынку *Unity*-пользователей. Также важный момент, что магазин доступен прямо из редактора *Unity*. То есть, добавление новых ассетов максимлально упрощается. Можно зайти в магазин, кликнуть на нужный ассет, и он сразу скачается и добавится в текущий проект. Быстро и удобно [2, с. 28].

За счёт своей популярности *Unity* имеет огромное комьюнити. Если у пользователя возник какой-то вопрос в процессе разработки, скорей всего, он уже много раз задавался, и столько же раз уже был решен. Информации по игровому движку очень много, в том числе и официальной. Это огромный плюс движка по сравнению с другими. Разработчик не чувствует себя одиноким и при решении проблем ему всегда помогут. Если требуется написать свою игру, но не хочется сильно заморачиваться с программированием – движок *Unity* подходит лучше других. Набросать объектов на сцену, и связать их между собой получится и без знания языка программирования. В случае возникновения проблем большое сообщество пользователей и документация с официального сайта помогут справиться с любыми трудностями.

## Обзор игрового движка *Unreal Engine* 4

Фотореализм – это первое слово, которое приходит на ум, когда думаешь об *Unreal* *Engine* (*UE*4). При использовании технологий студии разработки движка Epic Games получаются настолько реалистичные кат-сцены, что их сложно отличить от реальной съёмки. Благодаря этому движок широко используется и за пределами игр, в таких секторах, как архитектура, дизайн продукта и кинопроизводство. При помощи его возможностей создавались такие картины как «Звёздные войны» и «Мандалорианец».

С 1998-го года *Epic* *Games* радует своих пользователей техническими демонстрациями, показывающими, на что способен *Unreal* *Engine*, от трассировки лучей до технологий реального времени, физики и систем разрушений. Такие возможности делают его предпочтительным движком в разработке *ААА*-игр. И по сегодняшний день *Unreal* *Engine* доказывает свою состоятельность в разработке различных жанров и стилей игр, но его успех основан на проектах с реалистичной графикой. Возможности Unreal Engine представлены на рисунке 1.2:



Рисунок 1.2 – Фотореализм игрового движка *Unreal Engine*

*UE*4 поддерживает широкий спектр платформ: *Windows* *PC*, *PlayStation* 4, *Xbox* *One*, *Mac* *OS* *X*, *iOS*, *Android*, *AR*, *VR*, *Linux*, *SteamOS* и *HTML*5. На *Unreal* *Engine* разрабатывают свои проекты более чем семи с половиной миллионов разработчиков, из-за чего его можно считать популярным. На *Unreal Engine* можно разрабатывать самые разнообразные проекты. Гибкость *UE*4 является одной из основных причин его популярности. Каким бы ни был проект нестандартным, данный движок сможет справиться с этой задачей.

Доступ ко всему исходному коду *C*++ для движка – одна из причин, по которым *UE*4 настолько гибок. Код можно скачать с *GitHub*, любой желающий может настроить его под себя, и это совершенно бесплатно. Для сравнения, Unity предоставляет только профессиональным и корпоративным клиентам доступ к своему исходному коду – и это стоит дорого. Разработка и выпуск игры, сделанной с помощью *Unreal Engine*, полностью бесплатный. Плата в размере 5% выплачивается *Epic Games* только тогда, когда игра приносит доход в размере 3000 долларов в квартал.

*Unreal Engine* упрощает разработку игр для нетехнических пользователей с помощью *Blueprint*. Визуальная скриптовая система *UE*4, *Blueprint*, представленная на рисунке 1.3, является основной причиной того, что движок является отличным инструментом для конструирования игровых прототипов. В *Blueprint* вы перетаскиваете узлы и добавляете связи, чтобы соединить их и добавить логику. Этот узловой интерфейс позволяет дизайнерам, не имеющим технической подготовки, исследовать свои идеи без необходимости знать, как кодировать.

Движок *Unreal Engine* отлично подходит для крупномасштабных игр. Как показывают его достижения в качестве развития *AAA*-игр, движок хорошо масштабируется для больших команд и игр с большим числом игроков. *Unreal Engine* опустила планку до кроссплатформенной разработки. *UE*4 не только может создавать большие игры, но и его функции рендеринга отлично показывают себя на разных платформах. Конвейер рендеринга является современным, поддерживая огромное разнообразие платформ от мобильных до высококлассных компьютеров. Портирование проектов *Unreal Engine* на различные платформы особенно ценно при работе в виртуальной реальности, так как при разработке требуется учитывать комфорт пользователя.

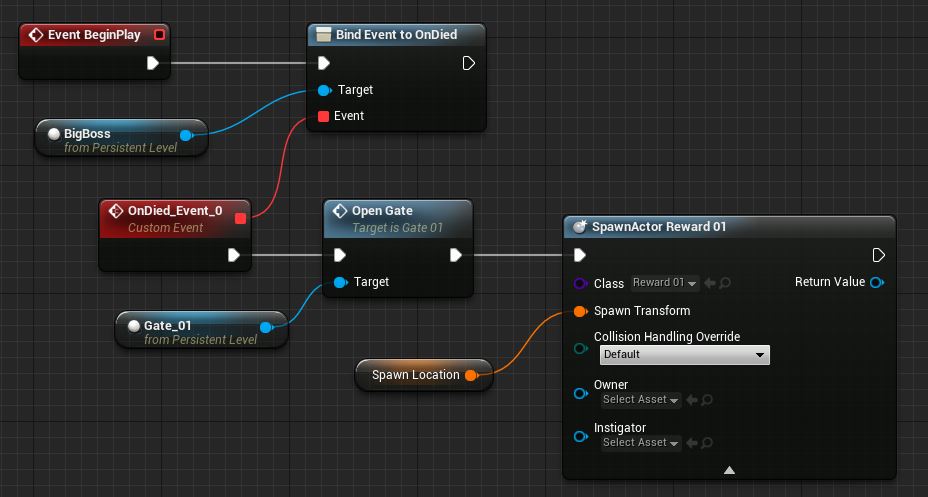


Рисунок 1.3 – Визуальная скриптовая система *Blueprint*

*Unreal Engine* имеет широкий набор инструментов и отличный редактор. Простота использования движка демонстрируется широким спектром функций и инструментов в интуитивно понятном редакторе с дружелюбным интерфейсом, доступном для многих настроек. Широкий спектр инструментов предоставляет решения практически для каждой макро задачи, которую придется решать: макет сцены, включая ландшафт и листву, кинематографию, визуальные сценарии, систему частиц, анимационную сеть, создание материалов, световое отображение, управление движением, уровень детализации, физику, основные модели искусственного интеллекта, инструменты отслеживания производительности и памяти для оптимизации [3, с. 98].

*Unreal Engine* имеет хорошую официальную и общественную поддержку. Будучи широко используемым движком, отдача комьюнити производится на высшем уровне. Стоит отметить, что уровень предоставляемой поддержки может зависеть от того, какой разработчик имеет доступ к движку. Платящий пользователь *UE*4 с пользовательской лицензией будет иметь непосредственное внимание команды разработчиков *Epic Games*, в то время как свободным пользователям, придется больше полагаться на богатство учебников и интернет-ресурсов для поиска ответов. Пользовательский интерфейс *Unreal Engine* представлен на рисунке 1.4:

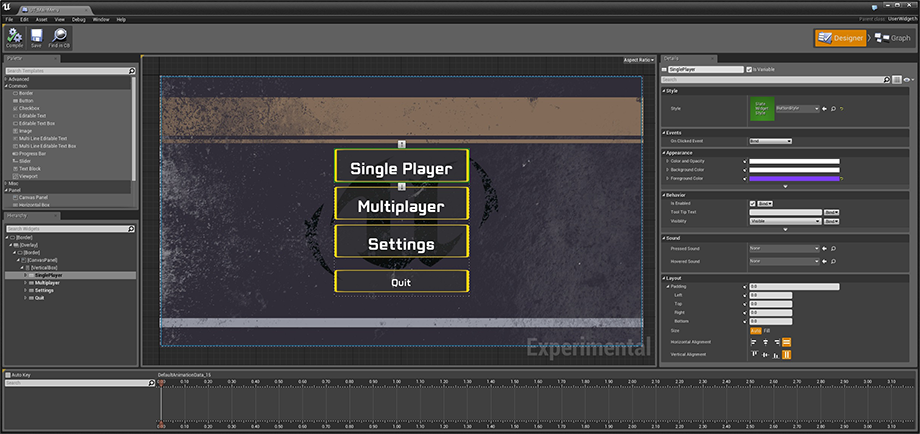


Рисунок 1.4 – Пользовательский интерфейс *Unreal Engine*

Движку *Unreal Engine* сложно бороться с большим количеством данных. В момент проектирования больших игровых уровней способность хорошо их редактировать в редакторе начинает страдать, а во время отладки возникают проблемы с производительностью, которые трудно решить. *UE*4 не предназначен для *RPG* игр, визуализации и управления огромным количеством оружия, брони, расходных материалов, разговоров и так далее. К примеру, игра *Fortnite*, разработанная на этом движке, представляет собой большую игру с открытым миром, но в ней намного меньше данных, чем в большинстве проектов своего жанра. Она не вызывает заметных зависаний, но стоит её ещё немного расширить новым игровым контентом, как проблемы тут же дадут о себе знать. *Unreal Engine* нуждается в точной настройке для удовлетворения нестандартных потребностей разработчика, а чтобы правильно настроить движок, потребуется много времени и нервов. К тому же, это сложно будет сделать без технического образования. В то время как *UE*4 обслуживает менее опытных разработчиков благодаря технологии *Blueprint*, некоторые аспекты движка остаются трудными для понимания пользователям, у которых отсутствует хорошая техническая подготовка. Наличие серьёзного опыта программирования на *С*++ поможет с правильной настройкой окружения, но даже с этим багажом знаний *UE*4 имеет довольно крутую кривую обучения [4, с. 113].

## 1.3 Сравнение игровых движков *Unity* и *Unreal Engine* 4

Для того чтобы подытожить и выяснить какой игровой движок оптимальнее использовать при разработке игрового приложения жанра «сетевой шутер» необходимо разобрать плюсы и минусы каждого по отдельности [5].

К плюсам игрового движка *Unity* можно отнести следующее:

* кроссплатформенный;
* позволяет быстро разрабатывать игры за счёт готовых шаблонов (ассетов);
* содержит встроенный магазин ассетов;
* хорошо подходит для создания небольших проектов;
* имеет низкий порог вхождения;
* реализует конструктивное создание объектов, что значительно упрощает разработку;
* наличие большого комьюнити, видеоуроков и документации;
* имеет дружелюбный интерфейс;
* является условно бесплатным.

Минусы игрового движка *Unity*:

* закрытый исходный код;
* плохо оптимизирован для разработки больших игровых проектов с открытым миром.

К плюсам игрового движка *Unreal Engine* можно отнести:

* возможность создавать фотореалистичные проекты;
* кроссплатформенная разработка;
* открытый исходный код;
* бесплатный;
* наличие технологии конструктивной разработки игр *Blueprint*;
* дружелюбный интерфейс;
* наличие большого комьюнити.

Помимо многочисленных плюсов *Unreal Engine* также имеет ряд серьёзных недостатков:

* проседание в производительности при большой нагрузке данными;
* наличие проблем с отладкой больших игр;
* высокий порог вхождения;
* движок имеет избыточное число инструментов для разработки маленьких проектов;
* сложная индивидуальная настройка редактора и функций движка.

Из перечисленных положительных и отрицательных сторон рассматриваемых игровых движков можно сделать вывод, что для разработки простой некоммерческой игры в рамках курсового проекта оптимальнее будет использовать движок *Unity*, так как он позволяет больше времени уделить проектированию игры и непосредственно её дизайну, нежели настройке движка и редактора под собственные нужды. К тому же, большинство инструментов *Unreal Engine* будут излишними для разработки мелкой сетевой игры.

## 1.4 Протоколы передачи данных транспортного уровня *UDP* и *TCP*

Для реализации сетевого взаимодействия необходимо проанализировать существующие протоколы транспортного уровня для того, чтобы определить с помощью чего пользователи смогут корректно обмениваться пакетами данных и обрабатывать их. Это в свою очередь позволит выбрать наиболее оптимальный протокол, по которому будет функционировать сетевое приложение. Транспортный уровень основан на двух ключевых протоколах – *UDP* (*User Datagram Protocol*) и *TCP* (*Transmission Control Protocol*).

Протокол пользовательских дейтаграмм (*UDP*) – это протокол транспортного уровня, при котором компьютерные приложения могут посылать сообщения называемые декаграммами другим хостам по *IP*-сети без необходимости предварительной отправки сигнального сообщения для приложения дополнительного сетевого маршрута и установки специальных каналов передачи данных, что является длительным и ресурсозатратным процессом. Протокол *UDP* использует простую модель передачи данными, без установления соединения между сокетами, упорядочивания сообщений и гарантии их доставки. Таким образом, *UDP* предоставляет ненадёжный сервис, и дейтаграммы могут прийти не по порядку, дублироваться или вовсе исчезнуть в сети. *UDP* подразумевает, что проверка ошибок и их исправление либо не нужны, либо должны исполняться в приложении. Чувствительные ко времени приложения часто используют *UDP*, так как предпочтительнее сбросить пакеты повторно, чем ждать задержавшиеся пакеты в сети, что может оказаться невозможным в системах реального времени. Протокол *UDP* имеет такие положительные особенности, как:

* отсутствие необходимости в установлении ресурсозатратного соединения между хостами;
* более быстрая передача данных относительно протокола *TCP*.

При необходимости исправления ошибок на сетевом уровне интерфейса приложение может задействовать протокол *TCP*, разработанный для этой цели.

Протокол управления передачей (*TCP*) обеспечивает передачу данных в нужной последовательности и без ошибок. По данному протоколу работают потоковые сокеты. Потоковые сокеты – это сокет с установленным соединением, состоящий из потока байт, который может быть двунаправленным, то есть через эту конечную точку приложение может и передавать, и получать данные. Благодаря протоколу *TCP* потоковый сокет гарантирует исправление ошибок, обрабатывает доставку и сохраняет последовательность данных. На него можно положиться в доставке упорядоченных, сдублированных данных [6].

Потоковый сокет также подходит для передачи больших объемов данных, поскольку накладные расходы, связанные с установлением отдельного соединения для каждого отправляемого сообщения, может оказаться неприемлемым для небольших объемов данных. Для потоковых сокетов путь формируется до начала передачи сообщений. Тем самым гарантируется, что обе участвующие во взаимодействии стороны принимают и отвечают. Если приложение отправляет получателю два сообщения, то гарантируется, что эти сообщения будут получены в той же последовательности [7].

Однако отдельные сообщения могут дробиться на пакеты, и способа определить границы записей не существует. При использовании *TCP* этот протокол берет на себя разбиение передаваемых данных на пакеты соответствующего размера, отправку их в сеть и сборку их на другой стороне. Приложение знает только, что оно отправляет на транспортный уровень определенное число байт и другая сторона получает эти байты. В свою очередь *TCP* эффективно разбивает эти данные на пакеты подходящего размера, получает эти пакеты на другой стороне, выделяет из них данные и объединяет их вместе [8].

Из всего вышеперечисленного следует, что протокол *TCP* обладает такими положительными особенностями, как:

* нумерация пакетов и гарантированная передача информации в неизменном виде, последовательности и без потерь;
* возможность работы в дуплексном режиме, то есть в одном пакете можно передавать данные и подтверждать получение предыдущего пакета;
* установка надёжного соединения;
* полный контроль процесса обмена информацией.

Подытожив можно прийти к выводу, что протокол *TCP* имеет большее количество положительных особенностей и соответственно лучше подходит для обмена информацией за продолжительное время игровой сессии в игре «Шутер от первого лица».

## 1.5 Актуальность разработки компьютерных игр в настоящее время

С каждым годом популярность компьютерных игр растет, все больше и больше людей играют в компьютерные игры. Результатом технологического развития, а также массового распространения персональных компьютеров, планшетов и мобильных устройств, стал все более расширяющийся рынок развлечений, что является веским аргументом в пользу проектов по созданию новых игр.

Актуальность разработки игр становится все более неоспоримой, рынок игр увеличивается в геометрической прогрессии. Самые удачные проекты продаются миллионными тиражами. Зарождение игровой индустрии началось в 1970-х годах одновременно с продажей первых компьютеров, рассчитанных на массового потребителя.

За 40 с небольшим лет индустрия компьютерных игр развилась до таких масштабов, что уже во многом опережает своих ближайших конкурентов: киноиндустрию, музыкальную индустрию, шоу-бизнес. Многие разработчики рассматривают индустрию компьютерных игр лишь как сектор экономики, в котором можно неплохо заработать. Но, на самом деле, создание игр – это ещё и инструмент культуры.

Ретро игры были придуманы много лет назад, но и сейчас они пользуются большой востребованностью. Яркий пример – *Mario*, серия компьютерных аркад, в которые с удовольствием играют пользователи нескольких поколений. Но она бы не была такой популярной, если бы не имела развитие. «*Super Mario Maker*» – игра, в которой игрок имеет возможность не только проходить уровни, но и создавать свои собственные.

На сегодняшний день возрождение старых игр может являться актуальным и выгодным проектом. На рисунке 1.5 представлена текущая динамика доходов игровой индустрии, а также приблизительные планы развития на ближайшую пару лет:



Рисунок 1.5 – Динамика доходов игровой индустрии по категориям в мире

Исходя из данных представленных на диаграмме, можно сделать вывод, что разработка новых игровых приложений будет актуальна и прибыльна ещё долгие годы, а производство новой и более мощной вычислительной техники позволит не только удовлетворить все потребности потребителей, но и удешевить разработку игр, что, в свою очередь, увеличит доход игровых разработчиков.

# ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СЕТЕВОГО ПРИЛОЖЕНИЯ «ШУТЕР ОТ ПЕРВОГО ЛИЦА»

В процессе разработки сетевого приложения «Шутер от первого лица» необходимо будет разработать сеттинг и механику игрового мира. Сетевая часть приложения должна основываться на взаимодействии сокетов протокола *TCP*/*IP*. Игроку потребуется сражаться по сети против людей, которые на протяжении игрового раунда будут противостоять ему и всеми силами препятствовать в выполнении главной цели ­– за короткое время заработать как можно больше очков и занять первое место. Игроку потребуется обладать следующими возможностями:

* перемещаться по игровому полю;
* совершать прыжки;
* взаимодействовать с игровыми объектами;
* носить вооружение;
* стрелять, нанося урон конкурентам, препятствуя их победе;
* защищаться от противников;
* пополнять боезапас;
* восстанавливать здоровье;
* умирать и возрождаться;
* повышать свой рейтинг.

В игре должны присутствовать различного рода бонусы. В качестве них будут выступать патроны, броня и аптечки. После смерти от вражеских персонажей, или самоубийства, игрок должен будет воскрешаться на точке старта с полным здоровьем и только одним видом оружия.

Рендеринг игровых объектов требуется осуществлять средствами игрового движка *Unity* и перспективной проекции.

*HUD* (*Heads*-*Up Display*) должен быть отображён при помощи средств *UnityUI* и оповещать игроков о текущем уровне здоровья, защиты, патронов и набранных очках приоритета. Также в игре должны присутствовать визуальные или звуковые эффекты на такие события, как:

* стрельба;
* подъём бонуса;
* прыжок;
* приземление;
* столкновение;
* победа и поражение.

Необходимо реализовать конец игры по истечению времени игрового раунда и объявить победителя, который успел набрать максимальное количество очков приоритета.

## 2.1 Архитектура и синхронизация сетевого взаимодействия

Разработка многопользовательских игр – это сложный процесс, который требует осторожности и обширного багажа знаний об устройстве сетевого взаимодействия между клиентами и сервером. Если сетевое взаимодействие реализовано неправильно, то игровое приложение может иметь проблемы с производительностью, причины которых достаточно сложно выявить, а игроки, пользуясь данной уязвимостью, смогут использовать запрещённые возможности для того, чтобы превзойти своих конкурентов и нечестно достичь победы.

При разработке многопользовательских игр чаще всего используют архитектуру сети, основанную на технологии передачи данных от пользователя к пользователю (*peer*-*to*-*peer*). Данная технология позволяет избежать создания выделенного сервера и позволить пользователям напрямую соединяться друг с другом. Несмотря на тот факт, что пользователи могут соединяться прямо друг с другом, игра все же продолжается разрабатываться в терминах клиент-серверной логики. Первый пользователь или же «пир», который подсоединяется к игре, становится хостом, то есть клиентом и сервером в одном лице. Все в дальнейшем присоединившиеся к игре пользователи затем соединяются только с хостом.

Все игроки, которые не являются хостами, являются пирами. Хост действует как сервер для всей игры. Ключевое отличие хоста от центрального сервера в том, что хостом может быть любой игрок, в то время как постоянный центральный сервер запускается разработчиком игры. Также хост может быть полноправным участником игры, в то время как сервер обычно является лишь условием связи игроков.

Сетевая архитектура *peer*-*to*-*peer* соединения продемонстрирована на рисунке 2.1:

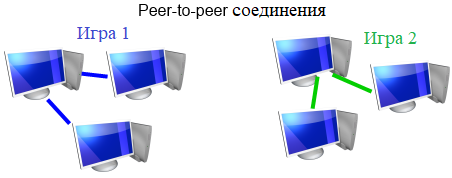


Рисунок 2.1 – Архитектура *peer*-*to*-*peer* соединения

Вдобавок, при предоставлении игрокам возможности, соединяться между собой напрямую, проблема, связанная с хостингом сервера, пропадает и, тем самым, сокращаются лишние денежные траты.

Для того чтобы присоединившийся пир был в состоянии соединиться с хостом, необходимо знать где последний находится и как к нему присоединиться. Это включает три вещи: *IP*-адрес, информация о наличии роутера, который накладывает ограничения на соединение, и информация о том, можно ли обойти эти ограничения. Хостом может быть кто угодно, где угодно в мире, невозможно установить заранее, кто будет соединен с кем. Так что для того, чтобы позволить пользователям найти друг друга, нужен отдельный центральный сервер, который называется сигнальным сервером.

Сигнальный сервер по-настоящему не является хостом для игр. Первый игрок в комнате является хостом и передает данные другим пирам. Сигнальный сервер – это просто центральное место для игроков, куда следует идти, чтобы найти друг друга. Когда пир хочет присоединиться к хосту, то сигнальный сервер транслирует информацию о *IP* адресе и детали связи между двумя. Как только пир успешно соединяется с хостом, сигнальный сервер больше не подключен и игроки обмениваются информацией друг с другом напрямую. Так как разрабатываемое игровое приложение рассчитывается для игры пользователей находящихся в одной локальной сети, то разработка сигнального сервера не является целесообразной.

Иногда при разработке сетевых приложений возникают проблемы связанные с синхронизацией данных. Для того чтобы их избежать существуют действенные методы позволяющие предотвратить некорректный обмен данными между клиентом и сервером. Одним из таких методов является предсказание действий на стороне клиента, который позволяет исключить ошибки, основанные на задержках при передаче данных по сети. Такие ошибки могут проявляться в ситуациях, когда клиент видит мир в настоящем времени, но из-за сетевых разрывов обновления с сервера приходят о состоянии мира в прошлом. Это происходит потому, что сервер не успевает получить актуальную информацию от клиента и формирует ответ на основе старых данных, которыми он владеет. Данная проблема решается путём нумерации сообщений клиента и добавлений в серверный ответ информации о последнем обработанном запросе. Также клиенту необходимо хранить у себя копии всех команд отправляемых на сервер и сверять их нумерацию с серверными ответами, в случае чего корректируя новые запросы.

Данный подход механизма синхронизации предоставляет игроку иллюзию отзывчивости, предсказывая и параллельно обрабатывая дальнейшие внутриигровые события, в то время, когда сервер обновляет полученную от клиента информацию и формирует актуальный ответ. Алгоритм синхронизации клиент-серверного взаимодействия на основе предсказаний дальнейшего поведения игры продемонстрирован на рисунке 2.2:

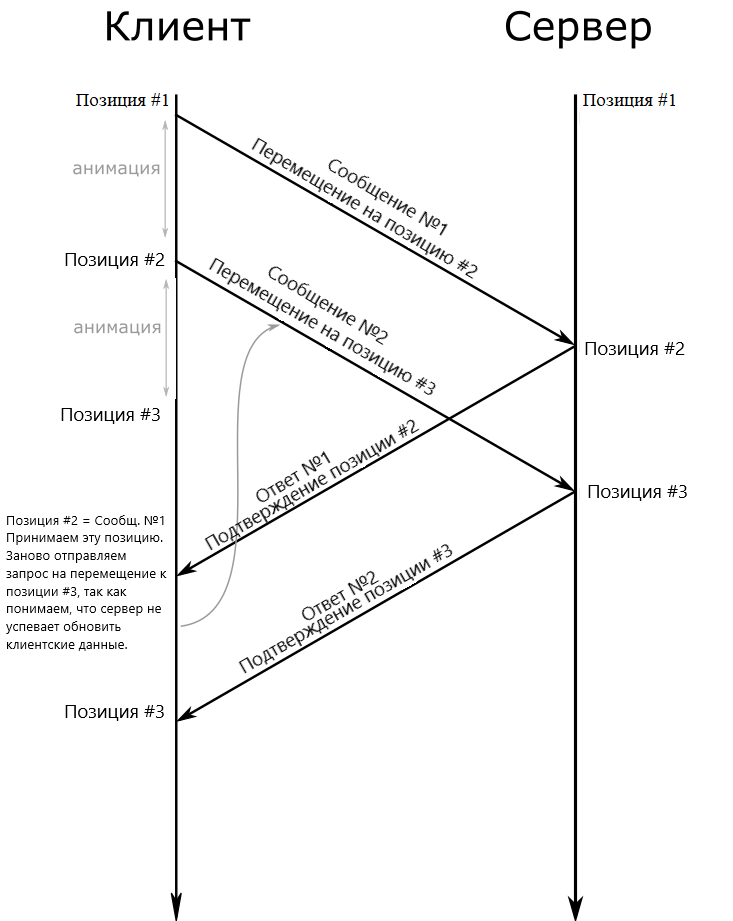


Рисунок 2.2 – Алгоритм синхронизации клиент-серверного взаимодействия

Данный способ позволяет избавиться от большинства проблем связанных с разрывом интернета, исключить жульничество со стороны клиентов, прост для понимания и реализации, отчего довольно часто применяется в большинстве современных сетевых игр.

## 2.2 Сетевой компонент приложения «Шутер от первого лица»

Сетевая часть игрового приложения реализована при помощи встроенного компонента *Unity* –сетевого менеджера(*NetworkManager*). Данный компонент позволяет упростить управление аспектами сети в многопользовательских играх. Интерфейс данного компонента продемонстрирован на рисунке 2.3:

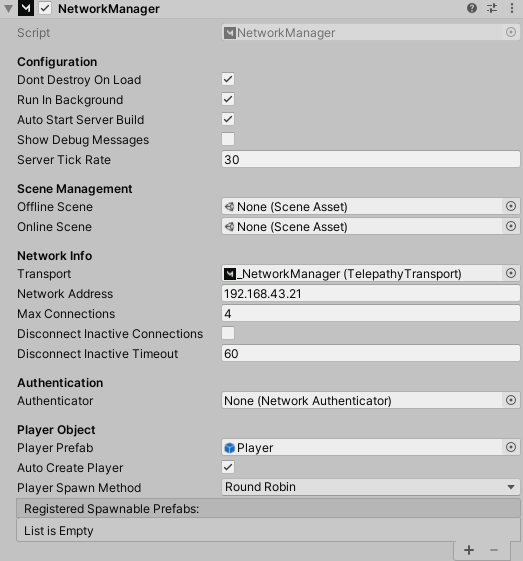


Рисунок 2.3 – Сетевой менеджер в окне инспектора *Unity*

Возможности сетевого менеджера:

* управление состоянием игры;
* управление появлением игроков на сцене;
* управление сценой;
* логирование отладочной информации;
* наследование в собственные сценарии и переопределение существующих методов под требуемые задачи;
* тонкая настройка под большинство необходимых нужд разработчика.

Сетевой менеджер – это основной управляющий компонент многопользовательской игры. Опытные разработчики в любой момент могут расширить его возможности благодаря концепции наследования в собственных *C*#-сценариях и настроить его поведение, переопределив любые, предоставляемые им, перехватчики виртуальных функций. Данный компонент удобно объединяет множество полезных функций в одном месте и максимально упрощает создание, запуск и отладку многопользовательских игр.

Вдобавок к сетевому менеджеру идёт небольшой графический интерфейс *HUDNetworkManager*, который позволяет игроку перед началом игры выбрать необходимый игровой режим. Сетевая многопользовательская игра может работать в трех вариациях – как клиент, как выделенный сервер или как «хост», который одновременно является и клиентом, и сервером. Данный графический интерфейс продемонстрирован на рисунке 2.4:



Рисунок 2.4 – Графический интерфейс компонента *NetworkManager*

В каком бы режиме игра ни запускалась (клиент, сервер или хост), используются свойства сетевого адреса и сетевого порта. В клиентском режиме игра пытается подключиться к указанному адресу и порту. В режиме сервера или хоста игра прослушивает входящие соединения на указанном порту.

Передача данных в сетевом менеджере осуществляется благодаря транспорту *Telepathy*, который работает на основе сокетов *TCP* и *C*#-классов *TcpListener* и *TcpClient* из пространства имён *System*.*Net*.*Sockets*. Транспорт *Telepathy* реализован по принципу *KISS*, который гласит, что большинство систем работают лучше всего, если они остаются простыми, а не усложненными; поэтому простота должна быть ключевой целью в проектировании, и следует избегать ненужной сложности. *Telepathy* – это быстрый и надёжный транспорт, который разработан для крупномасштабных *MMO*-сетей. Данный транспорт использует кадрирование, благодаря чему все отправленные данные будут гарантировано получены в той же последовательности, что и при отправлении. Благодаря этой технологии можно параллельно обрабатывать тысячи соединений и их коммуникацию. *Telepathy* был разработан для того, чтобы упростить создание сетевых игр жанра *MMORPG*.

## 2.3 Физика игрового приложения «Шутер от первого лица»

Объекты любого игрового мира, как и в жизни, должны иметь ряд основных физических свойств. К ним можно отнести:

* направление движения. Так как игровой мир представляет собой трёхмерное пространство, то оно будет вычисляться с помощью углов поворота вокруг осей *x*, *y*, *z* и относительно стартового положения объекта. Это позволит сущностям сцены смещаться не только вдоль конкретной оси, но и по направлению взгляда;
* скорость перемещения. Она позволит по ситуации ускорить или же замедлить объекты в конкретный момент времени. Её потребуется дополнительно корректировать с помощью произведения на время меж кадрового интервала, чтобы скорость перемещения объектов не зависела от *FPS* (*Frames Per Second*);
* силу гравитации. Она позволит притягивать объекты к земле, если те будут парить в воздухе.

Разрабатываемое графическое приложение требуется отображать при помощи перспективной проекции. Перспективная проекция, которая представлена на рисунке 2.5 – это изображение трёхмерных фигур на так называемой картинной плоскости способом, представляющим собой геометрическую идеализацию оптических механизмов зрения, фотографии:

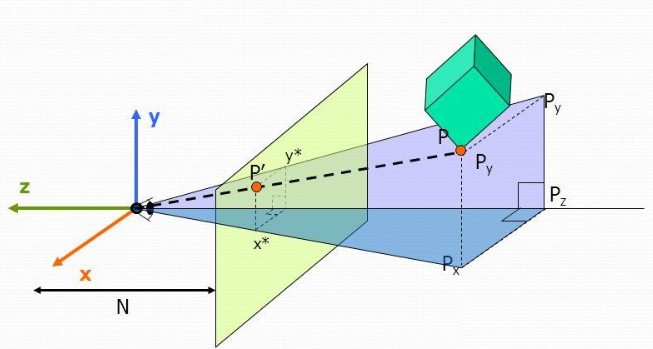
**

Рисунок 2.5 – Перспективная проекция

Перспективная проекция позволит максимально реалистично отображать расположение трёхмерных объектов в виртуальном пространстве.

Все объекты игрового мира состоят из вершин. Под вершинами подразумеваются точки в трёхмерном пространстве, которые содержат три значения – смещения по каждой оси относительно центра координат. Они позволяют отслеживать и корректировать положение объекта в игровом мире.

Основными функциями, для изменения положения объектов на сцене, являются перемещение и вращение. Для простого перемещения объекта достаточно сместить каждую его вершину на определённое значение вдоль конкретной оси. Пример перемещения трёхмерного объекта представлен на   
рисунке 2.6:

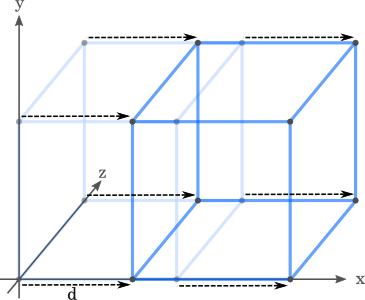


Рисунок 2.6 – Перемещение объектов в трёхмерном пространстве

Перемещение объекта не вдоль конкретной оси, а перпендикулярно его фронтальной стороне является более сложной операцией, так как для её реализации потребуется дополнительно вычислить направление, в которое смотрит объект. Это можно сделать благодаря углам, на которые он повёрнут. К ним относятся:

* *pitch* – угол поворота относительно оси *x*,в радианах;
* *yaw* – угол поворота относительно оси *y*,в радианах;
* *roll* – угол поворота относительно оси *z*,в радианах.

Координаты направления, вдоль которого смотрит объект, вычисляются по формулам 2.1 – 2.3:

(2.1)

(2.2)

(2.3)

Благодаря возможности отслеживать направление объекта можно перемещать вдоль него персонажа и посылать в эту сторону пули в момент   
стрельбы.

Поворот трёхмерного объекта относительно своего центра осуществляется благодаря четырём операциям. Изначально объект перемещается в начало координат, затем производится произведение каждой его вершины на матрицу поворота для вычисления новой позиции, после чего объект возвращается на своё предыдущее место расположения, и к определённому углу, в зависимости от того, вокруг какой оси вращался объект, прибавляется угол , на который повернули объект, в радианах. Наглядное вращение трёхмерного объекта и матрицы поворота, относительно каждой оси, представлены на   
рисунке 2.7:

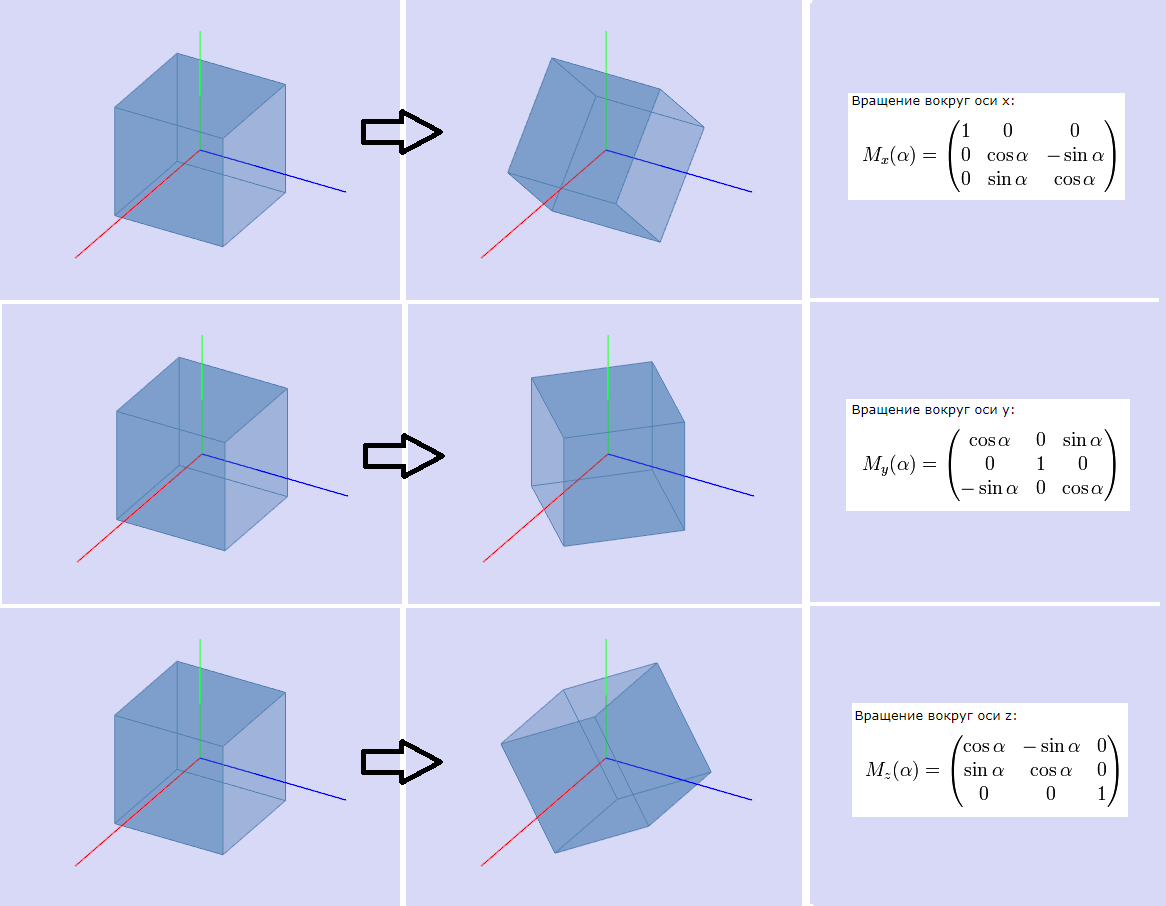


Рисунок 2.7 – Вращение трёхмерного объекта с помощью матриц поворота

Ежесекундные небольшие изменения положения объектов на небольшое расстояние при помощи предыдущих преобразований позволяют создать анимацию, благодаря которой игровые объекты могут казаться живыми и привлекать к себе внимание игрока, что в свою очередь влечёт ещё более глубоким погружением его в виртуальный мир. Благодаря такому эффекту будут плавно передвигаться игровые персонажи, и вращаться бонусы. Пример на рисунке 2.8:



Рисунок 2.8 – Анимирование игровых объектов

В процессе перемещения объектов на игровой сцене потребуется реализовать функцию разрешения столкновений. Она будет препятствовать объектам проваливаться друг через друга. Для того чтобы осуществить такую задачу понадобится создать вокруг каждого объекта ограничивающую маску по его крайним точкам. Пример ограничивающей маски представлен на рисунке 2.9:

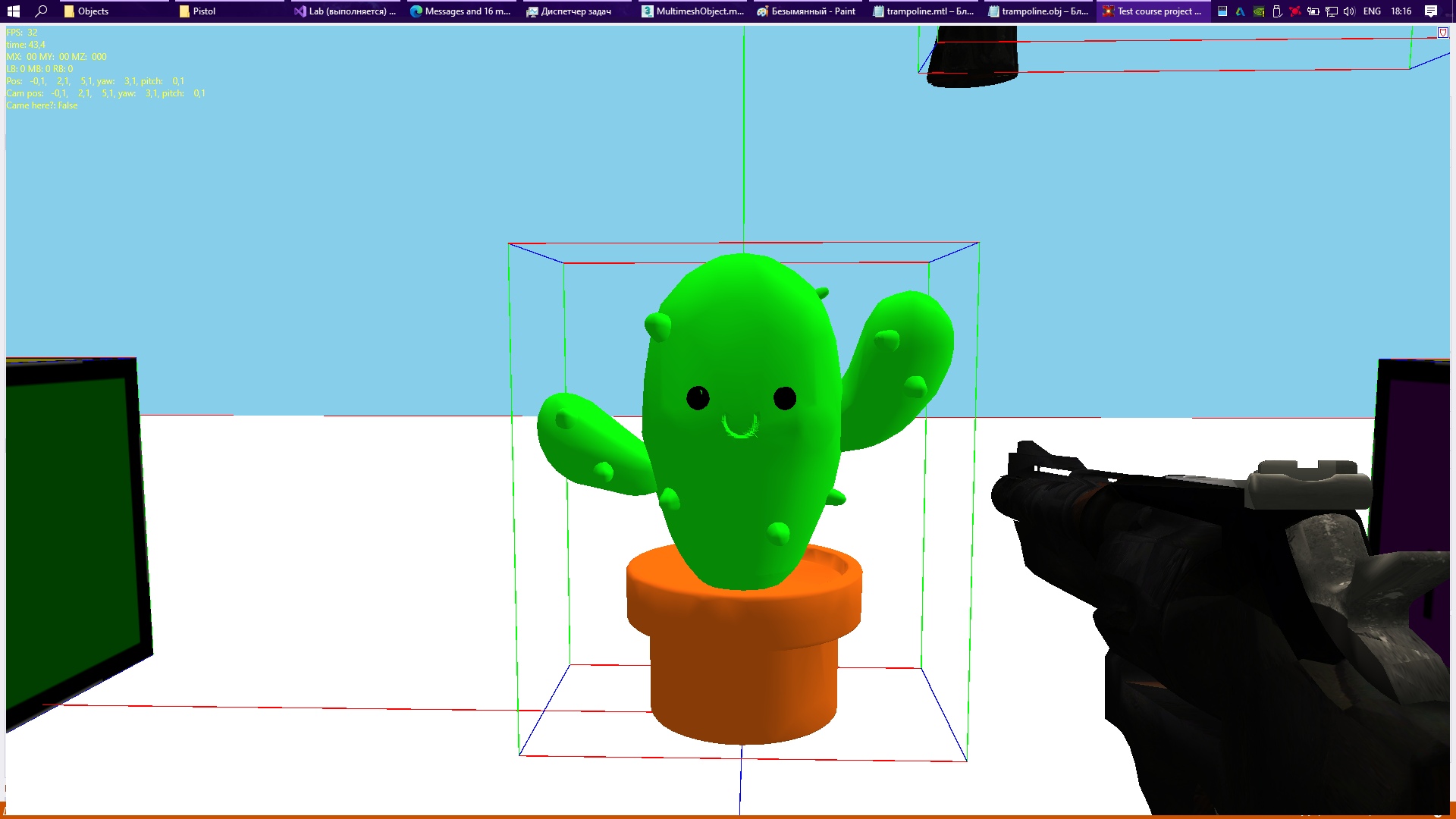


Рисунок 2.9 – Ограничивающая маска

Сам процесс разрешения коллизий довольно прост, при каждом перемещении объекта требуется проверять, не пересеклась ли его ограничивающая маска с масками ближайших объектов сцены и, если маски пересекаются, перемещать объект в обратном направлении на то же расстояние. Благодаря этому можно будет избежать неприятных провалов сквозь пол и стены. Но стоит также позаботиться о размере самой ограничивающей маски, так как, если она будет слишком большой, то в противном случае персонажам будет казаться, что они врезаются в пустоту.

Также, если в момент возникновения столкновения определять тип препятствующего перемещению объекта, то можно дополнительно наложить эффект на перемещаемую сущность, если препятствием окажется бонус или пуля.

Большую часть необходимого функционала для реализации физических и геометрических преобразований над игровыми объектами содержат в себе два ключевых компонента игрового движка *Unity* – это *Rigidbody*, который характеризует твёрдое тело игрового объекта и *Collider*, который представляет собой ограничивающую маску крайних вершин меша для обнаружения столкновений.

*Rigidbody* позволяет игровым объектам взаимодействовать с окружающей средой посредством применения к ним физических сил. Для реалистичного перемещения твёрдых тел применяются силы вращения, тяжести, трения и другие. Основные свойства данного компонента:

* *Mass –* масса объекта (по умолчанию в килограммах);
* *Drag* – воздушное сопротивление оказываемое на объект пока он перемещается под воздействием внешних сил;
* *Use Gravity –* сила гравитации;
* *Is Kinematic –* свойство, при включении которого объект не будет управляться физическим движком, и сможет управляться только при помощи своей трансформации;
* *Interpolate –* свойство позволяет включитьсглаживание трансформации основанное на трансформации из предыдущего кадра;
* *Collision Detection* – свойство позволяет предотвратить проникновение быстро перемещающихся объектов сквозь другие объекты без определения столкновений;
* *Constraints –* ограничения движения твёрдого тела по необходимой координате.

Компонент *Rigidbody* способен быстро смоделировать реалистичные движения физических тел, затратив минимум усилий и вычислительных средств.

Коллайдеры это ещё один тип компонентов, которые должны быть добавлены к физическим объектам наряду с компонентом твёрдого тела, чтобы задействовать, отлавливать и корректно обрабатывать столкновения. Если два твёрдых тела врезаются друг в друга, физический движок не будет просчитывать столкновение, пока к обоим объектам не будет назначен компонент коллайдера. Твёрдые тела, не имеющие коллайдеров, будут просто проходить сквозь друг друга при просчёте столкновений. В *Unity* существует целое подмножество компонентов наследуемых от класса *Collider*:

* *Box* *Collider* – примитивная форма куба;
* *Sphere* *Collider* – примитивная форма сферы;
* *Capsule* *Collider* – примитивная форма капсулы;
* *Mesh* *Collider* – коллайдер из меша объекта, который не может сталкиваться с другим меш-коллайдером;
* *Wheel* *Collider* – форма колеса с дополнительным функционалом для разработки автомобилей;
* *Terrain* *Collider* – обработчик столкновений с ландшафтной системой *Unity*.

Также существуют составные коллайдеры. Составные коллайдеры – это комбинации примитивных коллайдеров вместе, действующих как единое твердое тело. При тщательном позиционировании и выборе размеров составные коллайдеры часто могут довольно хорошо аппроксимировать форму объекта, сохраняя при этом низкие затраты на процессор. Дополнительная гибкость может быть достигнута за счет наличия дополнительных коллайдеров на дочерних элементах физических объектов (например, блоки могут вращаться относительно локальных осей родительского объекта). Их применение очень полезно, когда существует модель, которая была бы слишком сложной или дорогостоящей с точки зрения производительности для точного моделирования ограничивающей маски, и требуется имитировать столкновение формы оптимальным образом с использованием простых приближений. Визуализация составного коллайдера продемонстрирована на рисунке 2.10:

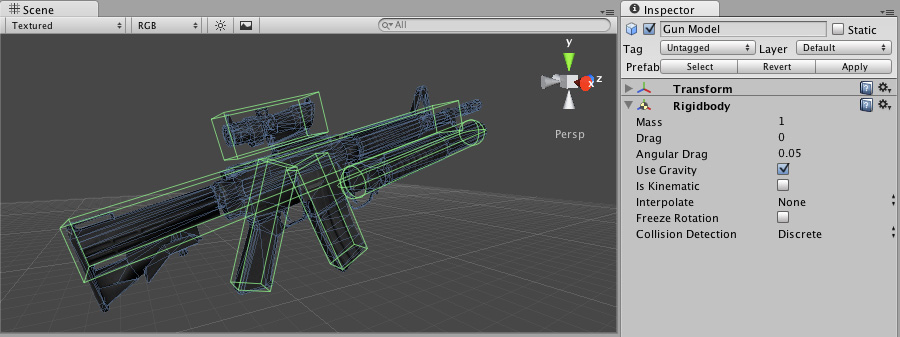


Рисунок 2.10 – Составной коллайдер

Когда происходят столкновения, физический движок вызывает функции с определенными именами в любых скриптах, прикрепленных к задействованным объектам. Разработчик может поместить любой код в этих функциях, чтобы реагировать на событие столкновения. Например, можно воспроизвести звуковой или визуальный эффект столкновения, когда пуля врезается в препятствие

*OnCollisionEnter* функция вызывается при первом обновлении физики, при котором обнаруживается столкновение. Во время обновлений, когда контакт поддерживается, вызывается *OnCollisionStay* и, наконец, *OnCollisionExit* указывает, что контакт был прерван. Данная событийная система столкновений очень удобна и позволяет разработчику больше сконцентрироваться на механике разрабатываемой игры, вместо продумывания архитектурных решений.

# ВЕРИФИКАЦИЯ И АПРОБАЦИЯ РАЗРАБОТАННОГО СЕТЕВОГО ПРИЛОЖЕНИЯ «ШУТЕР ОТ ПЕРВОГО ЛИЦА»

После запуска разработанного игрового приложения появляется меню с подключением, где требуется выбрать одно из следующих возможностей:

* стать сервером и создателем игровой комнаты, куда бы в будущем могли подключаться другие игроки;
* стать клиентом и подключиться к уже существующему серверу по сетевому адресу и порту;
* объединить в себе две перечисленные роли и стать хостом;

Сетевое меню продемонстрировано на рисунке 3.1:

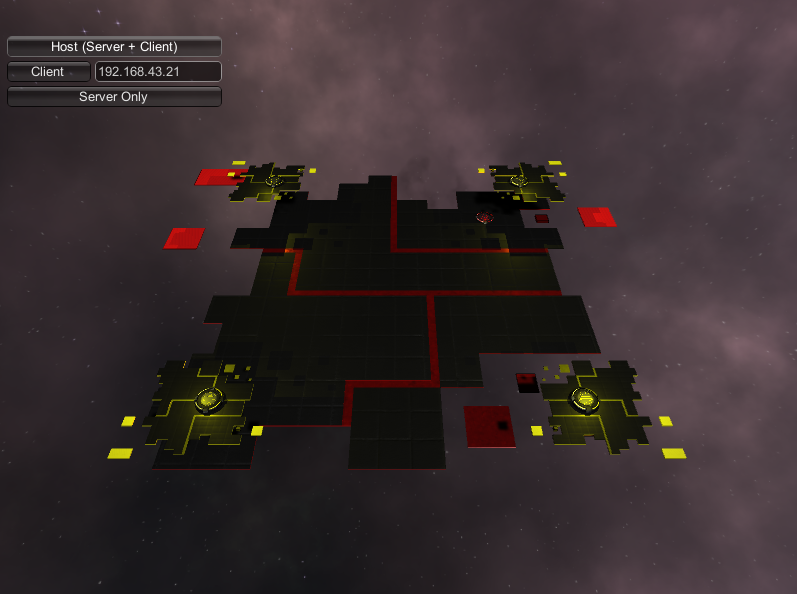


Рисунок 3.1 – Сетевое меню

Сетевое меню отображается на фоне игровой карты, которая представляет собой парящие космические платформы на фоне звёздного неба и отчётливо делится на две разные части, как по размеру, так и по цветовой палитре. Первая часть – это красная зона. Красная зона является самой опасной, так как на ней неожиданно возрождаются ранее убитые, или только что вошедшие в игру, игроки с полным запасом здоровья и вооружения, поэтому в этом месте велика вероятность встретиться лицом к лицу с сильным врагом.

При старте игры игроки имеют при себе:

– полный запас здоровья;

– заряженную штурмовую винтовку;

– запас пуль из тридцати единиц;

– нулевой запас прочности защитного поля (брони);

– обнулённый рейтинг в таблице рекордсменов.

Игроки умирают, когда их запас здоровья опускается ниже нуля, и воскрешаются через пять секунд на определённых красных генераторах, которые продемонстрированы на рисунке 3.2:

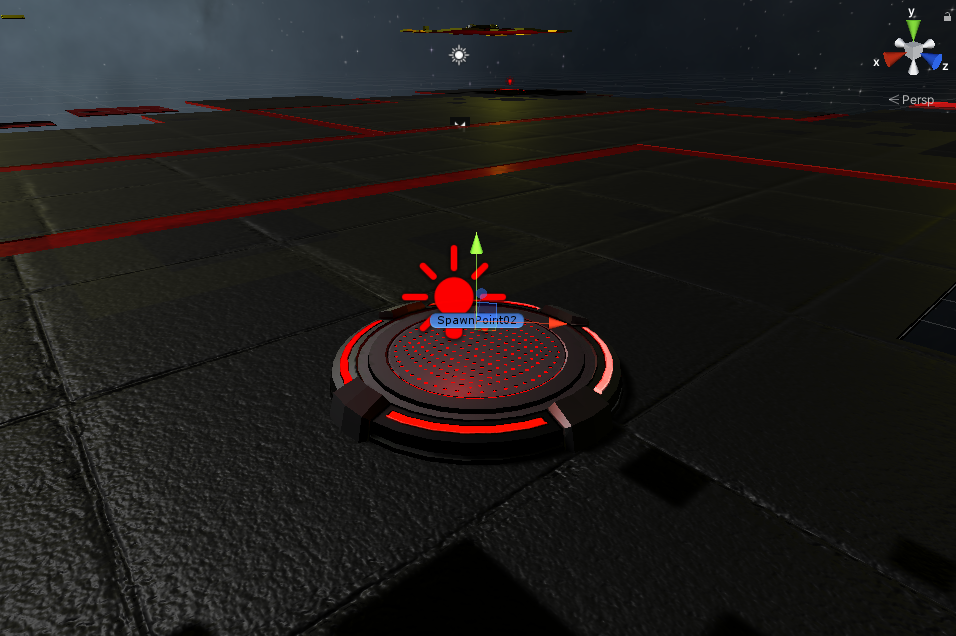


Рисунок 3.2 – Красный генератор для воскрешения игроков

Данные объекты расположены на противоположных краях карты для того, чтобы при старте игры у игроков было достаточно времени, чтобы укрыться от вражеского огня, отыскать полезные бонусы и заработать больше очков рейтинга. Красные генераторы нельзя уничтожить или заблокировать таким образом, чтобы на них не смог появиться вражеский игрок. При обнаружении красных генераторов требуется занять более выгодное место за укрытиями, которыми являются парящие в небе платформы или ящики, продемонстрированные на рисунке 3.3:



Рисунок 3.3 – Ящик, представляющий собой укрытие

Данное препятствие является статичным объектом и его никак нельзя уничтожить, передвинуть, прострелить или пройти насквозь. В подобных объектах не содержится ничего ценного, и они требуются лишь для того, чтобы игроки могли укрыться за ними от своих врагов в трудный момент времени.

Вторая часть игровой карты представляет собой жёлтую зону. Жёлтая зона является самой безопасной и выгодной позицией с точки зрения игрока, так как она хорошо укреплена препятствиями от вражеского огня, находится на возвышенности, что позволяет контролировать к ней все подходы, имеет отличный обзор и жёлтый генератор. На жёлтом генераторе время от времени создаются полезные призы. Призы делятся на три категории – это аптечки, броня и патроны. Они появляются на жёлтых генераторах случайным образом и пропадают через десять секунд, если ни один игрок не успевает вовремя их подобрать. Бонусы нельзя уничтожить выстрелом из оружия или заблокировав жёлтый генератор ближайшим твёрдым объектом. Игроки могут улучшать своё состояние и повышать личный рейтинг при помощи:

– выстрела во вражеского игрока. Это прибавляет стрелку пять единиц рейтинга к счёту;

– использования бонуса «аптечка». Это восстанавливает игроку двадцать единиц здоровья и прибавляет ему десять единиц рейтинга;

– использования бонуса «броня». Это восстанавливает игроку десять единиц защитного поля и прибавляет ему двадцать единиц рейтинга;

– использования бонуса «амуниция». Это добавляет игроку тридцать пуль на штурмовую винтовку и прибавляет ему тридцать единиц рейтинга;

– убийства врага. Это прибавляет ему сто единиц рейтинга.

Если учесть, что штурмовая винтовка имеет урон в пять единиц здоровья, то полное уничтожение врага с полными жизнями принесёт игроку прибавку к рейтингу в размере двух сот единиц. Генератор призов на безопасной и выгодной жёлтой зоне продемонстрирован на рисунке 3.4:

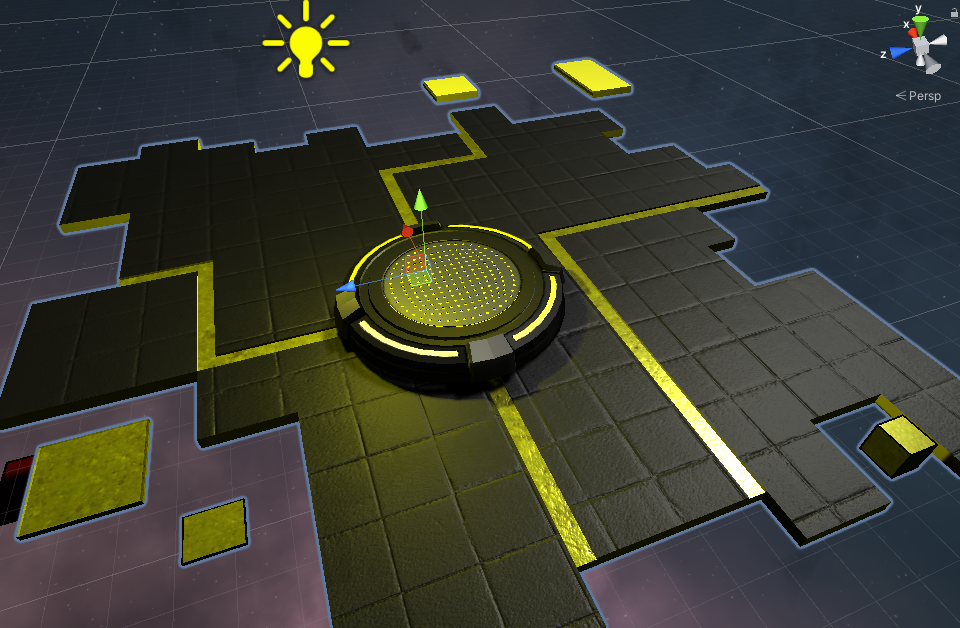


Рисунок 3.4 – Генератор призов

Жёлтых зон на карте больше чем красных, их количество составляет четыре единицы, но их совокупная площадь не превышает одной красной зоны.

Таким образом, игроки имеют возможность:

– перемещаться по игровому полю;

– совершать прыжки;

– взаимодействовать с игровыми объектами;

– носить вооружение;

– стрелять, нанося урон конкурентам, препятствуя их победе;

– защищаться от противников;

– пополнять боезапас;

– восстанавливать здоровье;

– умирать и возрождаться;

– повышать свой рейтинг.

Все бонусы, которые случайным образом появляются на жёлтых генераторах, продемонстрированы на рисунках 3.5 – 3.7:

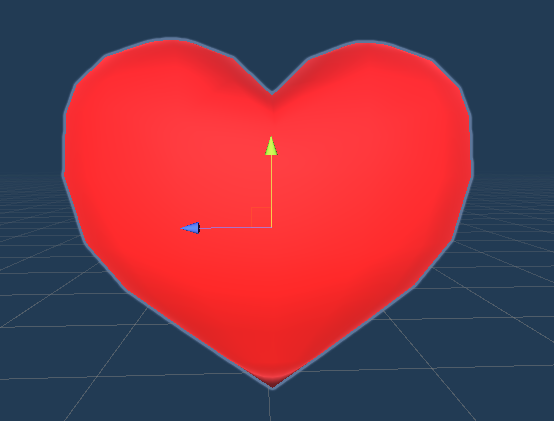


Рисунок 3.5 – Бонус «Здоровье»

Бонус «аптечка» генерируется с 20% вероятностью, для того, чтобы игрокам не приходилось постоянно стоять на жёлтых генераторах и часто лечиться, приобретая большое преимущество в сражениях с другими игроками.

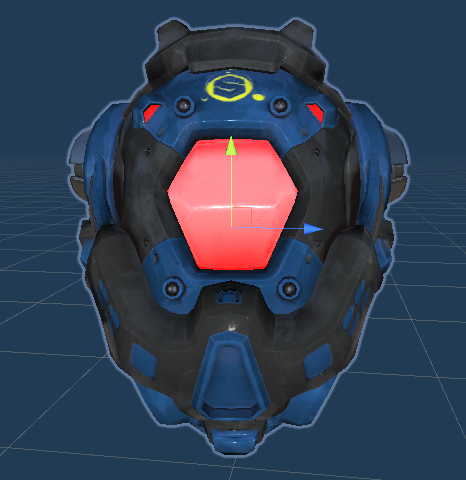


Рисунок 3.6 – Бонус «Броня»

Бонус «броня» генерируется с 30% вероятностью, и каждый игрок может максимально накапливать лишь 50 её единиц. Каждая единица брони уменьшает входящий урон на один процент. Это сделано для того, чтобы игрок не получал неуязвимость при накоплении ста единиц данного бонуса. При смерти вся накопленная в бою броня на игроке обнуляется.

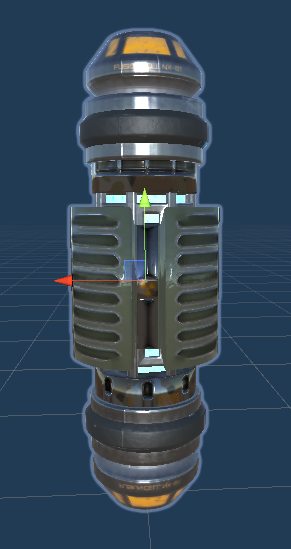


Рисунок 3.7 – Бонус «Амуниция»

Данный бонус генерируется чаще всего – с 50% вероятностью. Это сделано для того, чтобы игроки чаще сражались и не боялись остаться без пуль посреди раунда. У игроков нет ограничения при накоплении патронов, поэтому они смело могут их тратить и стрелять во всё что движется.

При разработке игры был также реализован дружелюбный графический пользовательский интерфейс. Выбор пал на минималистичный стиль для того, чтобы игрок максимально сосредоточился на сражении и его не отвлекали громоздкие значки и частые мерцания индикаторов на экране. В углах экрана расположены полупрозрачные шкалы разных цветов:

– красная шкала в нижнем левом углу экрана – показатель здоровья;

– оранжевая шкала в нижнем левом углу экрана – показатель брони;

– бирюзовая шкала в нижней части экрана – показатель энергии для совершения прыжка;

– зелёная шкала, расположенная в правой верхней части экрана – показатель длительности раунда. Игровой раунд длится пять минут, после чего объявляется победитель, набравший больше всех очков рейтинга. Очки рейтинга расположены прямо под шкалой длительности раунда.

Также в правом нижнем углу экрана расположен счётчик пуль для штурмовой винтовки. Графический пользовательский интерфейс продемонстрирован на рисунке 3.8:

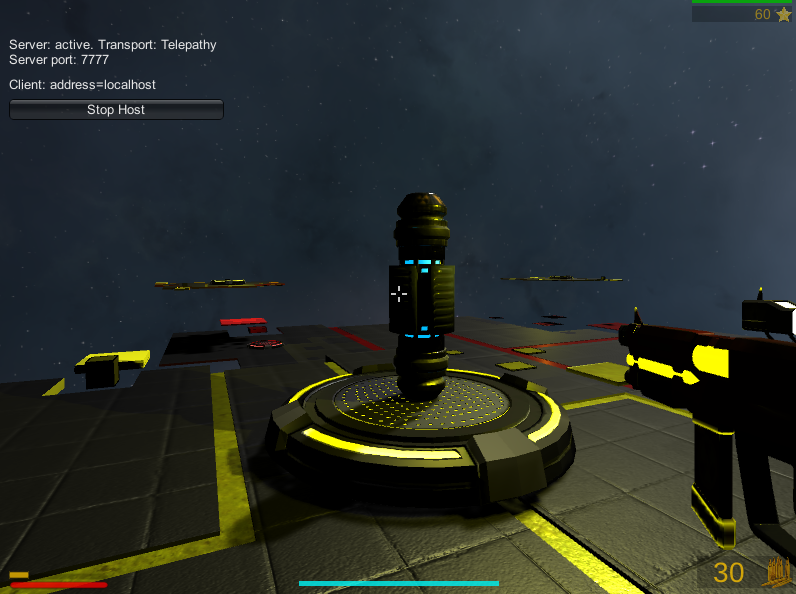


Рисунок 3.8 – Графический пользовательский интерфейс

По истечению времени таймера игрового раунда наступает конец игры и объявляется победитель. Это продемонстрировано на рисунке 3.9:



Рисунок 3.9 – Объявление победителя

Его имя высвечивается на месте счётчика очков рейтинга и, если победителем является локальный игрок, его имя подсвечивается зелёным цветом, иначе – красным.

# ЗАКЛЮЧЕНИЕ

Разработана сетевая игра жанра «Шутер от первого лица», как минимум для двух игроков. В ходе разработки графического игрового приложения решены следующие задачи:

– проведён анализ средств разработки графических приложений;

* разработана сцена, объекты, которые на ней позиционируются, а также игровая логика, сеттинг и механика;
* разработана система генерации игроков, призов и изменения характеристик игровых объектов;
* разработан программный код игры.

При реализации игрового проекта были использованы средства языка программирования *C*#. Графический интерфейс был разработан при помощи общедоступного игрового движка *Unity*. С помощью средств данного игрового движка была реализована сетевая часть на основе стека протоколов *TCP*/*IP*, спроектирована и отображена игровая сцена с, размещёнными на ней, текстурированными игровыми объектами, с которыми следует взаимодействовать пользователю в процессе игры, а также *HUD* с подсказками для игрока. Рендеринг игровых объектов на сцене осуществлялся при помощи перспективной проекции. Игра была наполнена основными визуальными и звуковыми эффектами стрельбы, подъёма бонусов, прыжков, приземления, перемещения. Было разработано и настроено интуитивно понятное управление игровым персонажем. Также, для разнообразия было добавлено разновидные бонусы и декоративные препятствия. Для преодоления заграждений и перемещений на возвышенности была реализована регенерирующая система прыжков. Была спланирована цель игры, которая заключается в том, чтобы за определённое время уничтожить как можно больше противников и набрать максимальное количество очков рейтинга.

Приложение подходит для всех возрастных категорий, так как относится к популярному игровому жанру, принцип которого понятен даже ребёнку, а взаимодействие с окружающей средой происходит интуитивно и беспрепятственно.

# Список используемых источников

1. Хокинг Д. Unity в действии. Мультиплатформенная разработка на C# / Хокинг, Д. – М.: СПб: Питер, 2019. – 215 c.
2. Гибсон Б. Д. Unity и C#. Геймдев от идеи до реализации / Гибсон, Б. Д. – М.: СПб: Питер, 2019. – 228 c.
3. Куксон А. Разработка игр на Unreal Engine 4 за 24 часа / Куксон, А. – М.: Бомбора, 2019. – 298 c.
4. Костер Р. Разработка игр и теория развлечений / Костер, Р. – М.: ДМК, 2018. – 313 c.
5. DirectX и С#. Искусство программирования / БХВ-Петербург, 2018. – 1207 с.
6. Компьютерные сети. Принципы, технологии, протоколы, 5-е издание / В. Олифер, Н. Олифер, 2017. – 880 с.
7. Методы проектирования / М: Мир, 2016. – 326 с.
8. Паттерны проектирования на платформе .NET. – СПб: Питер, 2015. – 320 с.

# ПРИЛОЖЕНИЕ А

(обязательное)

## Листинг программы «Шутер от первого лица»

**Листинг класса *Player:***

using Assets.Scripts.Bonuses;

using Mirror;

using System;

using System.Collections;

using UnityEngine;

public class Player : NetworkBehaviour

{

[SyncVar]

private bool \_isDead = false;

public bool isDead { get => \_isDead; protected set => \_isDead = value; }

[SerializeField]

private int maxArmor = 50;

[SerializeField]

private int maxHealth = 100;

[SerializeField]

private int startBullets = 30;

[SyncVar]

private int currentArmor;

[SyncVar]

private int currentHealth;

[SyncVar]

private int currentBullets;

[SyncVar]

private int currentScore;

[SerializeField]

private Behaviour[] disableOnDeath;

private bool[] wasEnabled;

[Header("Sounds:")]

[SerializeField]

private AudioSource ammoSound;

[SerializeField]

private AudioSource armorSound;

[SerializeField]

private AudioSource healthSound;

public void Setup()

{

wasEnabled = new bool[disableOnDeath.Length];

for (int i = 0; i < wasEnabled.Length; i++)

{

wasEnabled[i] = disableOnDeath[i].enabled;

}

SetDefaults();

}

public int GetScore()

{

return currentScore;

}

public void AddScore(int \_incrementScore)

{

currentScore += \_incrementScore;

}

/\*

private void Update()

{

if (!isLocalPlayer)

{

return;

}

if (Input.GetKeyDown(KeyCode.K))

{

RpcTakeDamage(99999);

}

}

\*/

[ClientRpc]

public void RpcTakeDamage(int \_amount)

{

if (isDead)

{

return;

}

currentHealth -= \_amount;

Debug.Log(transform.name + " now has " + currentHealth + " health.");

if (currentHealth <= 0)

{

Die();

}

}

public int GetBullets()

{

return currentBullets;

}

public void AddBullets(int \_count)

{

currentBullets += \_count;

}

public void TakeAwayBullets(int \_count)

{

currentBullets -= \_count;

if (currentBullets < 0)

{

currentBullets = 0;

}

}

public void Heal(int \_points)

{

currentHealth += \_points;

if (currentHealth > maxHealth)

{

currentHealth = maxHealth;

}

}

public void AddArmor(int \_points)

{

currentArmor += \_points;

if (currentArmor > maxArmor)

{

currentArmor = maxArmor;

}

}

public float GetHealth()

{

return currentHealth;

}

public float GetMaxHealth()

{

return maxHealth;

}

public float GetArmor()

{

return currentArmor;

}

public float GetMaxArmor()

{

return maxArmor;

}

private void Die()

{

isDead = true;

for (int i = 0; i < disableOnDeath.Length; i++)

{

disableOnDeath[i].enabled = false;

}

Collider \_col = GetComponent<Collider>();

if (\_col != null)

{

\_col.enabled = false;

}

Debug.Log(transform.name + " is DEAD!");

StartCoroutine(Respawn());

}

private IEnumerator Respawn()

{

yield return new WaitForSeconds(GameManager.instance.matchSettings.respawnTime);

SetDefaults();

Transform \_spawnPoint = NetworkManager.singleton.GetStartPosition();

transform.position = \_spawnPoint.position;

transform.rotation = \_spawnPoint.rotation;

Debug.Log(transform.name + " respawned.");

}

private void SetDefaults()

{

isDead = false;

currentHealth = maxHealth;

currentArmor = 0;

currentBullets = startBullets;

for (int i = 0; i < disableOnDeath.Length; i++)

{

disableOnDeath[i].enabled = wasEnabled[i];

}

Collider \_col = GetComponent<Collider>();

if (\_col != null)

{

\_col.enabled = true;

}

}

private void OnCollisionEnter(Collision \_collision)

{

AbstractBonus \_bonus = \_collision.gameObject.GetComponent<AbstractBonus>();

if (\_bonus != null)

{

\_bonus.Buff(this);

PlayAudioBonusEffect(\_bonus);

Destroy(\_collision.gameObject);

}

}

[Client]

private void PlayAudioBonusEffect(AbstractBonus \_bonus)

{

if (!isLocalPlayer)

{

return;

}

if (\_bonus is Ammo)

{

ammoSound.Play();

}

else if (\_bonus is Armor)

{

armorSound.Play();

}

else if (\_bonus is Health)

{

healthSound.Play();

}

}

}

**Листинг класса *PlayerController*:**

using Assets.Scripts.Bonuses;

using UnityEngine;

[RequireComponent(typeof(Animator))]

[RequireComponent(typeof(ConfigurableJoint))]

[RequireComponent(typeof(PlayerMotor))]

public class PlayerController : MonoBehaviour

{

[SerializeField]

private float speed = 5f;

[SerializeField]

private float lookSensitivity = 3f;

[SerializeField]

private float thrusterForce = 1000f;

[SerializeField]

private float thrusterFuelBurnSpeed = 1f;

[SerializeField]

private float thresterFuelRegenSpeed = 0.3f;

private float thrusterFuelAmount = 1f;

public float GetThrusterFuelAmount()

{

return thrusterFuelAmount;

}

[Header("Spring settings:")]

[SerializeField]

private float jointSpring = 20f;

[SerializeField]

private float jointMaxForce = 40f;

[Header("Sounds:")]

[SerializeField]

private AudioSource flySound;

// Component caching

private PlayerMotor motor;

private ConfigurableJoint joint;

private Animator animator;

private void Start()

{

motor = GetComponent<PlayerMotor>();

joint = GetComponent<ConfigurableJoint>();

animator = GetComponent<Animator>();

SetJointSettings(jointSpring);

}

private void Update()

{

// Setting target position for spring

// This make the physics acr right when it comes to applying gravity when flying over objects

RaycastHit \_hit;

float \_range = 100f;

if (Physics.Raycast(transform.position, Vector3.down, out \_hit, \_range))

{

joint.targetPosition = new Vector3(0f, -\_hit.point.y, 0f);

}

else

{

joint.targetPosition = new Vector3(0f, 0f, 0f);

}

// Calculate movement velocity as a 3D vector

float \_xMov = Input.GetAxis("Horizontal");

float \_zMov = Input.GetAxis("Vertical");

Vector3 \_movHorizontal = transform.right \* \_xMov;

Vector3 \_movVertical = transform.forward \* \_zMov;

// Final Movement vector

Vector3 \_velocity = (\_movHorizontal + \_movVertical) \* speed;

// Animate Movement

animator.SetFloat("ForwardVelocity", \_zMov);

// Vocalize Movement

PlayMoveEffect(\_velocity);

// Apply movement

motor.Move(\_velocity);

// Calculate rotation as a 3D vector (turning around)

float \_yRot = Input.GetAxisRaw("Mouse X");

Vector3 \_rotation = new Vector3(0f, \_yRot, 0f) \* lookSensitivity;

// Apply rotation

motor.Rotate(\_rotation);

// Calculate camera rotation as a 3D vector (turning around)

float \_xRot = Input.GetAxisRaw("Mouse Y");

float \_cameraRotationX = \_xRot \* lookSensitivity;

// Apply camera rotation

motor.RotateCamera(\_cameraRotationX);

// Calculate the thrusterforce based on player input

Vector3 \_thrusterForce = Vector3.zero;

if (Input.GetButton("Jump") &&

thrusterFuelAmount > 0f)

{

thrusterFuelAmount -= thrusterFuelBurnSpeed \* Time.deltaTime;

if (thrusterFuelAmount >= 0.01f)

{

\_thrusterForce = Vector3.up \* thrusterForce;

SetJointSettings(0f);

}

}

else

{

thrusterFuelAmount += thresterFuelRegenSpeed \* Time.deltaTime;

SetJointSettings(jointSpring);

}

thrusterFuelAmount = Mathf.Clamp(thrusterFuelAmount, 0f, 1f);

// Apply the thruster force

motor.ApplyThruster(\_thrusterForce);

// Show/Hide cursor

if (Input.GetKeyDown(KeyCode.F1))

{

Cursor.lockState = CursorLockMode.Locked;

}

else if (Input.GetKeyDown(KeyCode.F2))

{

Cursor.lockState = CursorLockMode.None;

}

}

/// <summary>

/// Set the force of gravity

/// </summary>

private void SetJointSettings(float \_jointSpring)

{

joint.yDrive = new JointDrive

{

positionSpring = \_jointSpring,

maximumForce = jointMaxForce

};

}

private void PlayMoveEffect(Vector3 \_velocity)

{

if (!flySound.isPlaying)

{

if (\_velocity != Vector3.zero)

{

flySound.Play();

}

}

}

}

**Листинг класса *PlayerMotor*:**

using UnityEngine;

[RequireComponent(typeof(Rigidbody))]

public class PlayerMotor : MonoBehaviour

{

[SerializeField]

private Camera cam;

private Vector3 velocity = Vector3.zero;

private Vector3 rotation = Vector3.zero;

private float cameraRotationX = 0f;

private float currentCameraRotationX = 0f;

private Vector3 thrusterForce = Vector3.zero;

[SerializeField]

private float cameraRotationLimit = 85f;

private Rigidbody rb;

private void Start()

{

rb = GetComponent<Rigidbody>();

}

/// <summary>

/// Gets a movement vector

/// </summary>

public void Move(Vector3 \_velocity)

{

velocity = \_velocity;

}

/// <summary>

/// Gets a rotational vector

/// </summary>

public void Rotate(Vector3 \_rotation)

{

rotation = \_rotation;

}

/// <summary>

/// Gets a rotational vector for the camera

/// </summary>

public void RotateCamera(float \_cameraRotationX)

{

cameraRotationX = \_cameraRotationX;

}

/// <summary>

/// Gets a force vector for our thrusters

/// </summary>

public void ApplyThruster(Vector3 \_thrusterForce)

{

thrusterForce = \_thrusterForce;

}

/// <summary>

/// Run every physics iteration

/// </summary>

private void FixedUpdate()

{

PerformMovement();

PerformRotation();

}

/// <summary>

/// Perform movement based on velocity variable

/// </summary>

private void PerformMovement()

{

if (velocity != Vector3.zero)

{

rb.MovePosition(rb.position + velocity \* Time.fixedDeltaTime);

}

if (thrusterForce != null)

{

rb.AddForce(thrusterForce \* Time.fixedDeltaTime, ForceMode.Acceleration);

}

}

/// <summary>

/// Perform rotation

/// </summary>

private void PerformRotation()

{

rb.MoveRotation(rb.rotation \* Quaternion.Euler(rotation));

if (cam != null)

{

// Set our rotation and clamp it

currentCameraRotationX -= cameraRotationX;

currentCameraRotationX = Mathf.Clamp(currentCameraRotationX, -cameraRotationLimit, cameraRotationLimit);

// Apply our rotation to the transform of our camera

cam.transform.localEulerAngles = new Vector3(currentCameraRotationX, 0f, 0f);

}

}

}

**Листинг класса *PlayerSetup*:**

using Mirror;

using System;

using UnityEngine;

[RequireComponent(typeof(Player))]

[RequireComponent(typeof(PlayerController))]

public class PlayerSetup : NetworkBehaviour

{

[SerializeField]

private Behaviour[] componentsToDisable;

[SerializeField]

private string remoteLayerName = "RemotePlayer";

[SerializeField]

private string dontDrawLayerName = "DontDraw";

[SerializeField]

private GameObject playerGraphics;

[SerializeField]

private GameObject playerUIPrefab;

private GameObject playerUIInstance;

private Camera sceneCamera;

/// <summary>

/// Activate when spawn

/// </summary>

private void Start()

{

Player \_player = GetComponent<Player>();

// Disable components of uncontrollable players

if (!isLocalPlayer)

{

DisableComponents();

AssignRemoteLayer();

}

else

{

sceneCamera = Camera.main;

if (sceneCamera != null)

{

sceneCamera.gameObject.SetActive(false);

}

// Disable player graphics for local player

Util.SetLayerRecursively(playerGraphics, LayerMask.NameToLayer(dontDrawLayerName));

// Create PlayerUI

playerUIInstance = Instantiate(playerUIPrefab);

playerUIInstance.name = playerUIPrefab.name;

// Configure PlayerUI

PlayerUI ui = playerUIInstance.GetComponent<PlayerUI>();

if (ui == null)

{

Debug.LogError("No PlayerUI component on PlayerUI prefab.");

}

ui.SetController(GetComponent<PlayerController>());

ui.SetPlayer(\_player);

}

\_player.Setup();

}

public override void OnStartClient()

{

base.OnStartClient();

string \_netID = GetComponent<NetworkIdentity>().netId.ToString();

Player \_player = GetComponent<Player>();

GameManager.RegisterPlayer(\_netID, \_player);

}

private void AssignRemoteLayer()

{

gameObject.layer = LayerMask.NameToLayer(remoteLayerName);

}

private void DisableComponents()

{

for (int i = 0; i < componentsToDisable.Length; i++)

{

componentsToDisable[i].enabled = false;

}

}

/// <summary>

/// Activate when leave the game

/// </summary>

private void OnDisable()

{

Destroy(playerUIInstance);

if (sceneCamera != null)

{

sceneCamera.gameObject.SetActive(true);

}

GameManager.UnRegisterPlayer(transform.name);

}

}

**Листинг класса *PlayerShoot*:**

using Mirror;

using System;

using UnityEngine;

[RequireComponent(typeof(WeaponManager))]

public class PlayerShoot : NetworkBehaviour

{

private const string PLAYER\_TAG = "Player";

[SerializeField]

private int scoreForHit = 5;

[SerializeField]

private int scoreForKill = 100;

[SerializeField]

private Camera cam;

[SerializeField]

private LayerMask mask;

[Header("Sounds:")]

[SerializeField]

private AudioSource shotSound;

private PlayerWeapon currentWeapon;

private WeaponManager weaponManager;

private void Start()

{

if (cam == null)

{

Debug.LogError("PlayerShoot: No camera referenced!");

enabled = false;

}

weaponManager = GetComponent<WeaponManager>();

}

private void Update()

{

currentWeapon = weaponManager.GetCurrentWeapon();

if (currentWeapon.fireRate <= 0)

{

if (Input.GetButtonDown("Fire1"))

{

Shoot();

}

}

else

{

if (Input.GetButtonDown("Fire1"))

{

InvokeRepeating("Shoot", 0f, 1f / currentWeapon.fireRate);

}

else if (Input.GetButtonUp("Fire1"))

{

CancelInvoke("Shoot");

}

}

}

/// <summary>

/// Is called on the server when a player shoots

/// </summary>

[Command]

private void CmdOnShoot()

{

RpcDoShootEffect();

}

/// <summary>

/// Is called on all clients when we need to do a shoot effect

/// </summary>

[ClientRpc]

void RpcDoShootEffect()

{

weaponManager.GetCurrentGraphics().muzzleFlash.Play();

}

/// <summary>

/// Is called on the server when we hit comething

/// Takes in the hit point and normal of the surface

/// </summary>

[Command]

private void CmdOnHit(Vector3 \_pos, Vector3 \_noraml)

{

RpcDoHitEffect(\_pos, \_noraml);

}

/// <summary>

/// Is called on all clients

/// </summary>

[ClientRpc]

void RpcDoHitEffect(Vector3 \_pos, Vector3 \_normal)

{

GameObject \_hitEffect = Instantiate(weaponManager.GetCurrentGraphics().hitEffectPrefab, \_pos, Quaternion.LookRotation(\_normal));

Destroy(\_hitEffect, 2f);

shotSound.Play();

}

[Client]

private void Shoot()

{

if (!isLocalPlayer)

{

return;

}

// Take away bullets

Player \_player = GetComponent<Player>();

if (\_player.GetBullets() <= 0)

{

return;

}

\_player.TakeAwayBullets(1);

// We are shooting, call the OnShoot method on the server

CmdOnShoot();

RaycastHit \_hit;

if (Physics.Raycast(cam.transform.position, cam.transform.forward, out \_hit, currentWeapon.range, mask))

{

if (\_hit.collider.tag == PLAYER\_TAG)

{

CmdPlayerShot(\_hit.collider.name, \_player.name, currentWeapon.damage);

}

// We hit something, call the OnHit method on the server

CmdOnHit(\_hit.point, \_hit.normal);

}

}

[Command]

private void CmdPlayerShot(string \_targetPlayerID, string \_haterPlayerID, int \_damage)

{

Debug.Log($"{\_targetPlayerID} take damage from {\_haterPlayerID}.");

Player \_targetPlayer = GameManager.GetPlayer(\_targetPlayerID);

\_targetPlayer.RpcTakeDamage(\_damage);

// Reward for a hit/kill

Player \_haterPlayer = GameManager.GetPlayer(\_haterPlayerID);

\_haterPlayer.AddScore(scoreForHit);

if (\_targetPlayer.GetHealth() - \_damage <= 0)

{

\_haterPlayer.AddScore(scoreForKill);

}

}

}

# ПРИЛОЖЕНИЕ Б

(обязательное)

## Функциональная схема приложения

# ПРИЛОЖЕНИЕ В

(обязательное)

## Диаграмма классов

## ПРИЛОЖЕНИЕ Г

(обязательное)

## Руководство пользователя

1. Введение.

Руководство пользователя информирует пользователя базовыми знаниями по эксплуатации игрового приложения.

Разработанное программное приложение предназначено для организации игрового процесса между множеством игроков по сети. Оно предполагает наличие нескольких игроков, которые соревнуются между собой в достижении высшего рейтинга, уничтожая друг друга и собирая бонусы. Управление игроком осуществляется с помощью клавиатуры и мыши.

Программное средство обладает следующим функционалом:

* наличие меню для игровых настроек;
* наличие меню для реализации игры по локальной сети;
* игровая сцена;
* управления игровым персонажем в трёх измерениях;
* наличие преград и бонусов.

Ход использования приложения:

Для использования программного средство пользователь должен быть ознакомлен с:

– настоящим руководством пользователя;

– правилами использования ПК.

1. Назначение и условия применения.

Разработанное программное приложение предназначено для мультиплеерной симуляции военно-футуристической войны. Данное приложение несёт развлекательный характер и предназначено для азартных игроков, которым интересны разнотипные соревнования.

Для корректной работы программного средства необходимо соблюдение следующих требований:

– поддерживаемые операционные системы *Windows XP* и выше;

– наличие стандартной клавиатуры и мыши;

– наличие следующих устройств вывода: экран, подключаемый по *HDMI*.

1. Подготовка к работе.

Для запуска приложения необходимо запустить скомпилированный исполняемый файл *3D\_TEST\_SHOOTER\_3.exe* из каталога с игрой.

Если все инструкции соблюдены и приложение не выдает никаких сообщений об ошибках, значит, программа работает исправно.

1. Описание операций

В ходе разработки игрового приложения было реализовано меню создания и подключения к игре по локальной сети, меню настройки игры и игровая сцена, на которой происходят основные действия игры. В окне для подключения пользователь может стать хостом – кнопка *Host* (*Server + Client*), подключится к уже созданной игре – кнопка *Client*, и стать сервером – кнопка *Server Only.* Для подключения к существующей игре необходимо в поле ввода ввести *IP*-адрес сервера. Это изображает рисунок Г.1:

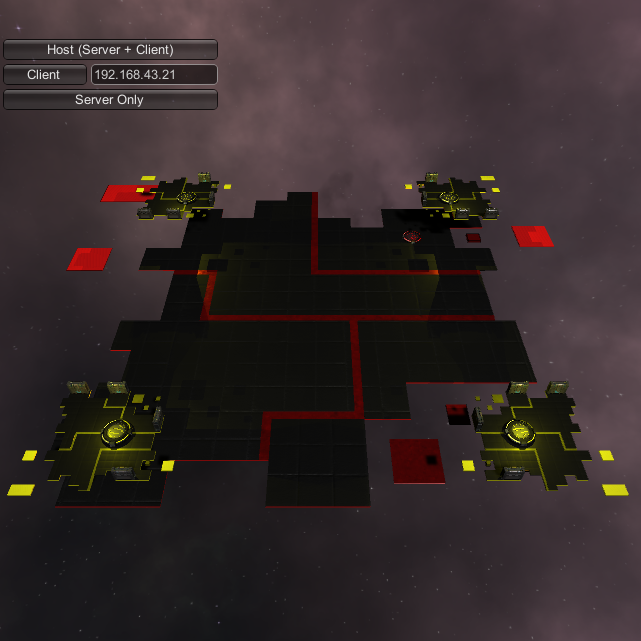


Рисунок Г.1 – Меню для подключения по локальной сети

Игрок всегда может выйти из игры в сетевое меню при нажатии на кнопку *Stop connect*.Окно с игровой сценой при старте как хост или клиент продемонстрировано на рисунке Г.2:

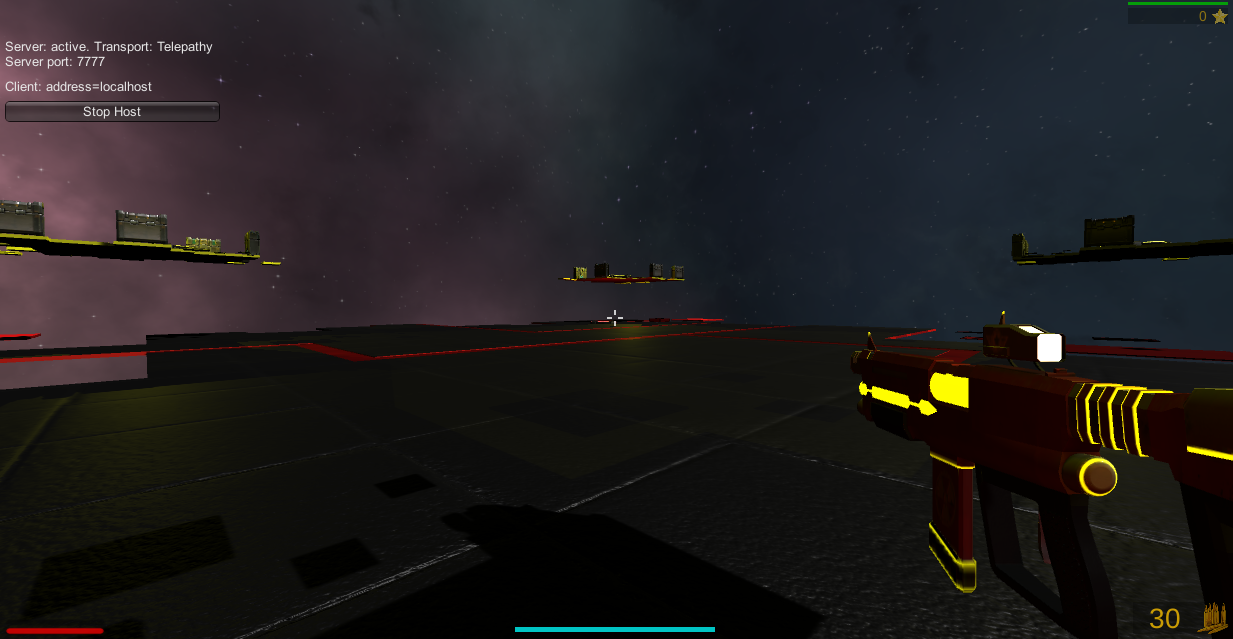


Рисунок Г.2 – Игровая сцена приложения для хоста и клиента

При старте как сервер с высоты птичьего полёта будет демонстрироваться игровая карта, на которой будут появляться подключённые клиенты. Серверная сцена продемонстрирована на рисунке Г.3:

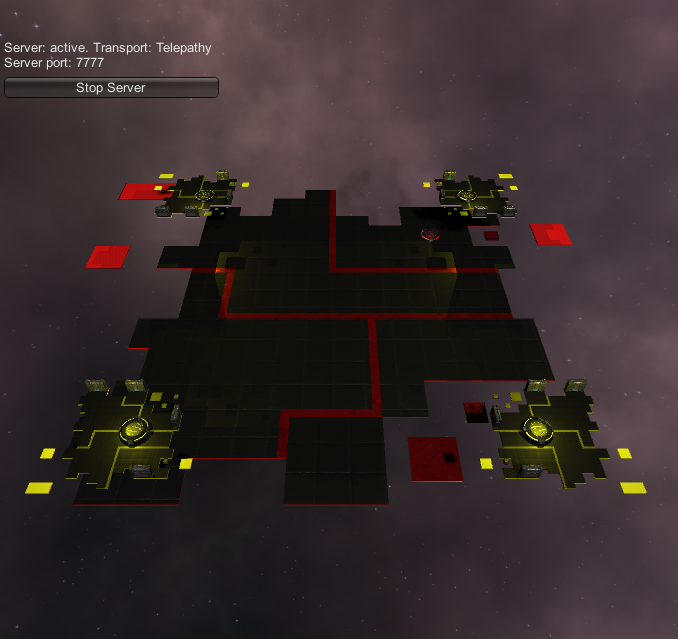


Рисунок Г.3 – Вид сцены от лица создателя сервера

Сервер всегда можно выключить при нажатии на кнопку *Stop Server*, после чего соединение со всеми подключёнными клиентами будет разорвано.

На игровой сцене случайным образом появляются бонусы в виде космических объектов, которые улучшают игровых персонажей, которые первыми успеют их подобрать. Противники окрашены в красный цвет и стараются противостоять игроку, пытаясь убить его игрового персонажа. Игроку необходимо следить за индикатором здоровья, который изображён в левом нижнем углу экрана, чтобы не умереть и дать врагам получить дополнительные очки рейтина за свою смерть. Призы продемонстрированы на рисунке Г.4:

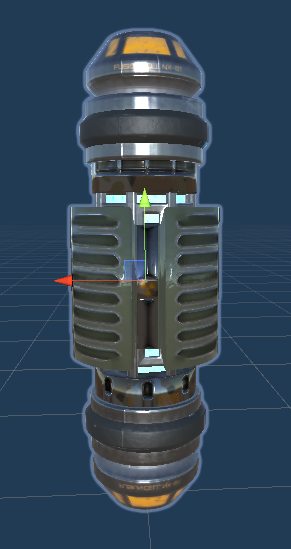
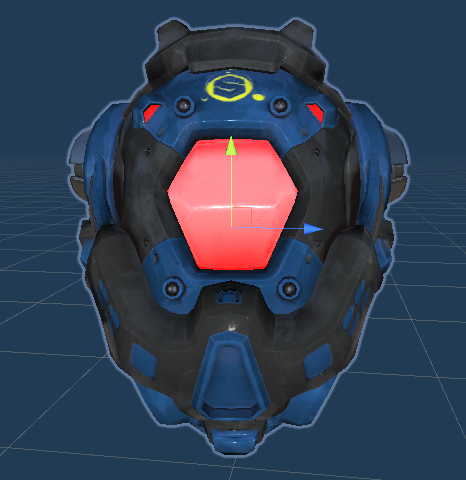
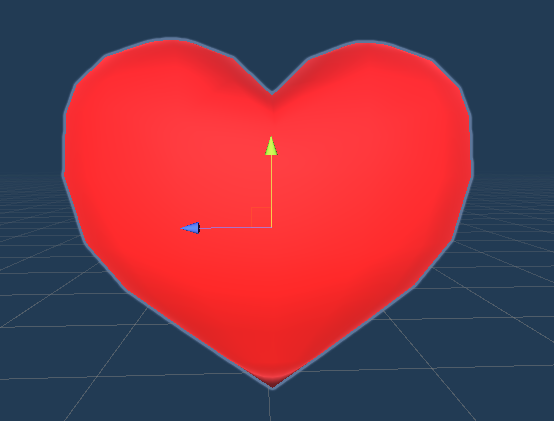


Рисунок Г.4 – Бонусы повышающие характеристики игрока

Управление игроком осуществляется при помощи клавиш *A* – для перемещения игрока влево, *D* – для перемещения игрока вправо, *W* – для перемещения игрока вперёд, *S* – для перемещения игрока назад соответственно. Для совершения прыжка используется клавиша *Space* Выстрел из оружия производится при помощи левой кнопки мыши.

Чтобы победить игроку необходимо набрать больше всех очков за время длительности игрового раунда. Результат игрового процесса представлен на рисунке 3.8.

1. Аварийные ситуации.

Чтобы избежать ошибок при использовании данного игрового приложения, необходимо соблюдать порядок действий и условия использования, описанные в главе 3 данного руководства пользователя.

В случае непредвиденного «зависания» программы рекомендуется завершить процесс в диспетчере задач и запустить снова. Или можно подождать некоторое время, так как существует возможность зависания программы за счет наличия библиотек, используемых в игровом приложении.

1. Рекомендации по освоению.

Игровое приложение рекомендуется запускать при наличии двух игроков для правильного функционирования.

**ПРИЛОЖЕНИЕ Д**

(обязательное)

## Руководство программиста

1. Назначение и условия применения.

Разработанное программное приложение предназначено для организации игрового процесса между несколькими игроками по локальной сети. Жанр игры – «Шутер от первого лица». Он предполагает наличие двух и более игроков, которые соревнуются в уничтожении врагов и сборе бонусов. Управление игроком осуществляется с помощью клавиатуры и мыши.

Для корректной работы программного средства необходимо соблюдение следующих требований:

* поддерживаемые операционные системы *Windows XP* и выше;
* наличие стандартной клавиатуры и мыши;
* наличие следующих устройств вывода: экран, подключаемый по *HDMI*.

1. Характеристики программы.

При запуске открывается меню сетевых настроек для игрового приложения, в котором выбирается тип запуска игры: хост; клиент; сервер, затем начинается игровая сессия и у игроков появляется возможность управлять своими персонажами.

1. Обращение к программе

Приложение запускается путем открытия исполняющегося файла *3D\_TEST\_SHOOTER\_3.exe*, который находится в каталоге с игрой.

1. Входные и выходные данные.

Входной информацией являются данные, которые предоставляют игроки путём нажатия на клавиши клавиатуры и кнопок мыши. Выходной информацией являются кадры на экране с результатом игрового процесса.

1. Сообщения.

В ходе выполнения приложения никаких сообщений не предусмотрено.

**Приложение Е**

(обязательное)

## Руководство системного программиста

1. Назначение и условия применения.

Разработанное программное приложение предназначено для организации игрового процесса между несколькими игроками по локальной сети. Жанр игры – «Шутер от первого лица». Он предполагает наличие двух и более игроков, которые соревнуются в уничтожении врагов и сборе бонусов. Управление игроком осуществляется с помощью клавиатуры и мыши.

Для корректной работы программного средства необходимо соблюдение следующих требований:

– поддерживаемые операционные системы *Windows XP* и выше;

– наличие стандартной клавиатуры и мыши;

– наличие следующих устройств вывода: экран, подключаемый по *HDMI*.

2. Структура программы.

Структурно приложение разделено на две рабочие области – область настроек для сетевой игры и область игрового поля.

Окно с настройками сетевой игры состоит из:

* кнопки создания сервера и подключения к нему как хост;
* кнопки подключению к серверу как клиент и поля ввода сетевого адреса;
* кнопки создания сервера.

Окно с игровой сценой состоит из двух разделенных областей для каждого пользователя. Первая область отвечает за *GUI* с индикаторами характеристик игрока, а вторая за обзор самой игровой сцены.

1. Настройка программы

Настройка программы осуществляется путем запуска исполняемого файла.

1. Проверка программы.

Отсутствие сообщения об ошибках свидетельствует, о корректной работе приложения.

1. Дополнительные возможности

Дополнительных возможностей программа не имеет

1. Сообщения системному программисту.

В ходе выполнения программы необходимо выбрать тип запуска игры (как хост, клиент или сервер). Других действий программа не требует.