**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**Учреждение образования**

**«Гомельский государственный технический университет**

**имени П.О.Сухого»**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

направление специальности 1-40 05 01-12 Информационные системы и технологии (в игровой индустрии)

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовой работе**

по дисциплине «Программирование сетевых приложений»

на тему: «Игровое приложение «Ферма» для двух игроков по сети с использование среды разработки *Unity*, стека протоколов *TCP/IP*»

Исполнитель: студент группы ИТИ-41

Дубовцов И.Д.

Руководитель: заведующий кафедрой

Курочка К.С.

Дата проверки: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата допуска к защите: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата защиты: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подписи членов комиссии

по защите курсовой работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Гомель 2023

СОДЕРЖАНИЕ

[Введение 4](#_Toc152514360)

[1 Обзор и анализ различных программных средств для трехмерного моделирования 5](#_Toc152514361)

[1.1 Виды программного обеспечения для трехмерного моделирования 5](#_Toc152514362)

[1.2 Технический обзор и анализ программы 3ds Max в области трехмерного моделирования и визуализации 6](#_Toc152514363)

[1.3 Технический обзор и анализ программы Blender в области трехмерного моделирования и визуализации 8](#_Toc152514364)

[2 Описание инструментов программы 3ds Max 10](#_Toc152514365)

[2.1 Моделирование объектов 10](#_Toc152514366)

[2.2 Анимация 15](#_Toc152514367)

[2.3 Визуализация 16](#_Toc152514368)

[2.4 Материалы V-ray 19](#_Toc152514369)

[2.5 Наложение звука 21](#_Toc152514370)

[3 Создание помещения с минималистичным интрьером для танцев, персонажи 22](#_Toc152514371)

[3.1 Моделирование помещения 22](#_Toc152514372)

[3.3 Персонажи 26](#_Toc152514373)

[3.5 Создание графики с помощью V-ray 27](#_Toc152514374)

[3.6 Добавление звуков на сцену 29](#_Toc152514375)

[Заключение 31](#_Toc152514376)

[Список использованных источников 32](#_Toc152514377)

# **ВВЕДЕНИЕ**

Популяризация компьютерной игровой индустрии начала свое зарождение с обычной идеи создания игры, в которой игрок перемешает по экрану световые лучи, возникнувшей еще до начала 1970-х годов. В это время взаимодействие человека и вычислительной техники уже стало непреложным фактором.

Видео игры проникли во много сфер современной жизни. На сегодняшний день индустрия компьютерных игр является одной из самых прибыльных и скоростных по развитию отраслей компьютерных технологий и, вместе с этим, глобального сектора развлечений. Рынок компьютерных игр становится все глубже и уже к 2019 году вырос в среднем на 11% и обошел ряд конкурентных рынков, к примеру, кино-, музыкальная индустрия.

В этом проекте будет разработана кооперативная игра жанра Аркада. Главная же особенность такого жанра – упрощенный игровой процесс и графика. Изначально аркадами были названы игры на игровых автоматах. Этим фактом и обусловлена их специфика: автоматы не подходили для сложных сюжетных игр, их ставили в местах большого скопления людей, поэтому они должны были быть привлекательными для широкой целевой аудитории, включая и детей, и взрослых, а соответственно, максимально простыми. Таким образом, разрабатывая игру жанра аркада, можно сказать, что будет разработана казуальная игра.

Казуальная игра содержит простые правила и простое управление. В таких играх нет необходимости обладать особыми навыками и располагать большим количеством времени. Основная аудитория казуальных игр – профессиональные геймеры и фанаты мобильного гейминга. Более 50 процентов пользователей мобильных приложений играют в игры, что уступает только приложениям для социальных сетей с точки зрения затраченного времени [1]. Казуальные игры живут не долго, но хорошо зарабатывают на показе рекламы. Казуальные игроки устанавливают в среднем в десять раз больше приложений в целом и в пять раз больше приложений со внутриигровыми покупками, чем пользователи других жанров [1]. Игра *Fall Guys* является свежим примером казуальной игры. Она стала популярна за пределами мобильной платформы и продалась тиражом в десять миллионов копий, что является отличным результатом.

Для создания игр разработчики, художники и дизайнеры используют самые разные программы. Как и сами технологии, все эти приложения постоянно меняются под влиянием потребностей и амбиций разработчиков. В курсовой работе автор раскроет поставленную тему и разработает сетевое двухмерное игровое приложение. Для разработки игры и написания кода использован язык программирования С#, среда разработки *Visual Studio 2022, Unity*.

**1 ИГРОВЫЕ ПРИЛОЖЕНИЯ И СРЕДСТСТВА ИХ РАЗРАБОТКИ**

**1.1 Многопользовательские аркадные игровые приложения**

Многопользовательские аркадные игры занимают особое место в сфере развлечений, предоставляя игрокам уникальный опыт совместного взаимодействия в виртуальных мирах. Этот подраздел посвящен рассмотрению представителей аркадного жанра, ориентированных на многопользовательский режим. Аркадные игры, как жанр, привлекают внимание своей простотой управления, динамичным геймплеем и часто выразительным дизайном. Таким образом, чтобы понять, что такое аркадные игры, стоит рассмотреть их достойных представителей.

*1.1.1 WormsBattlegrounds* – компьютерная игра серии *Worms* в жанре пошаговой стратегии, разработанная и изданная британской компанией *Team17*.

На рисунке 1.1 представлен кадр из игры *WormsBattlegrounds*.



Рисунок 1.1 – Кадр из игры *WormsBattlegrounds*

Игровой процесс сохраняет схожесть с предшествующими частями серии: управление командой червяков осуществляется игроком, который использует разнообразное оружие, такое как базуки, гранаты, овцы и бананы, с целью уничтожения червяков других команд. При этом каждая команда выполняет выстрелы поочередно, и побеждает та команда, в которой хотя бы один червяк остается живым, в то время как червяки противников погибают.

Основным инновационным элементом в игре стала физика объектов и жидкостей. Некоторые объекты на ландшафтах, такие как канистры, гайки и баллоны, подвластны движению при толчке или под воздействием выстрела. Кроме того, выстрел по некоторым из этих объектов влияет на ход битвы; например, взрыв канистры с бензином создает огонь вокруг нее, а взрыв баллона отравляет червяков, находящихся поблизости. Попадание червяка в воду на случайном участке влечет за собой утрату пяти единиц здоровья каждый ход. Также присутствует оружие, позволяющее добавлять на ландшафт больше воды.

Игра включает режимы кампании для одиночного игрока с разнообразными настройками оружия и ландшафтов, а также многопользовательские варианты игры за одним компьютером или через сеть. Кроме того, предоставляется возможность настройки внешнего вида червяков, например, изменение их шляп или голосов. Версии *Deluxe* и *Extreme* для портативного компьютера и *PS Vita* с *PlayStation 3* включают в себя дополнения, представленные новыми ландшафтами, миссиями и одеждой для червяков. Эти дополнения изначально были выпущены для консолей и компьютеров.

*1.1.2 Super Meat Boy* – аркадная компьютерная инди-игра в жанре платформера, разработанная командой *Team Meat*.

В игре акцент делается на высокой точности управления персонажем, где игрок управляет «Мясным пацаном» – квадратным персонажем, лишенным кожи, с задачей спасти «Пластыревую девушку» от «Доктора зародыша».

На рисунке 1.2 представлен кадр из игры *Super Meat Boy*.



Рисунок 1.2 – Кадр из игры *Super Meat Boy*

Геймплей включает в себя преодоление уровней с различными ловушками, такими как циркулярные пилы, иглы и соль, с использованием способности скользить по стенам.

Количество жизней не ограничено, и после смерти персонаж мгновенно возрождается в начале уровня. После успешного прохождения уровня игроку предоставляется возможность просмотра повтора последних 40 попыток прохождения.

Структура игры включает пять основных глав, каждая из которых содержит по 20 уровней. Для открытия босса игроку необходимо успешно завершить не менее 17 уровней в любом порядке. Пройдя пятую главу, открывается шестая глава под названием «*The End*», состоящая из пяти этапов. После ее завершения игрок сталкивается с финальным боссом – «*Dr. Fetus»*. После победы над «*Dr. Fetus»* открывается седьмая глава «*The Cotton Alley*», представляющая собой 20 сложных уровней, где доступен только персонаж «Пластыревая девушка». Общее число уровней в игре достигает приблизительно 350.

Игра также предлагает дополнительные элементы, такие как синие порталы (далее *warp*-зоны) – синие порталы, ведущие игрока в мини-игры с ретро-стилизацией. Каждая глава включает в себя по четыре *warp*-зоны. Присутствует также особая *warp*-зона красного цвета в светлом мире, предоставляющая бонус в виде разблокировки дополнительных персонажей. Игрок может также встретить видоизмененную «пластыревую девушку» на случайном уровне после прохождения основной главы.

В ранних стадиях разработки структура игры включала 15 уровней в каждой главе, разделенных на наборы по 3 этапа, с 16 пластырями на главу и отсутствием темного мира и warp-зон. Постепенно разработчики отошли от системы наборов, сделав все уровни глав изначально доступными.

**1.2 Игровой движок *Unity***

*Unity* – это профессиональный игровой движок, позволяющий создавать видеоигры для различных платформ.

Движок предоставляет множество функциональных возможностей, которые задействуются в различных играх. Реализованная на конкретном движке игра получает все функциональные возможности, к которым добавляются ее собственные игровые ресурсы и код игрового сценария.

Приложение *Unity* предлагает моделирование физических сред, карты нормалей, преграждение окружающего света в экранном пространстве (*Screen* *Space* *Ambient* *Occlusion*, *SSAO*), динамические тени. Подобные наборы функциональных возможностей есть во многих игровых движках, но *Unity* обладает двумя основными преимуществами над другими передовыми инструментами разработки игр. Это крайне производительный визуальный рабочий процесс и сильная межплатформенная поддержка. Визуальный рабочий процесс – достаточно уникальная вещь, выделяющая *Unity* из большинства сред разработки игр.

Альтернативные инструменты разработки зачастую представляют собой набор разрозненных фрагментов, требующих контроля, а в некоторых случаях библиотеки, для работы с которой нужно настраивать собственную интегрированную среду разработки (*Integrated Development Environment, IDE*), цепочку сборки и прочее в этом роде. В *Unity* же рабочий процесс привязан к тщательно продуманному визуальному редактору. Именно в нем будут компоноваться сцены будущей игры, связывая игровые ресурсы и код в интерактивные объекты. Он позволяет быстро и рационально создавать профессиональные игры, обеспечивая невиданную продуктивность разработчиков и предоставляя в их распоряжение исчерпывающий список самых современных технологий в области видеоигр. На рисунке 1.3 представлен интерфейс приложения Unity.

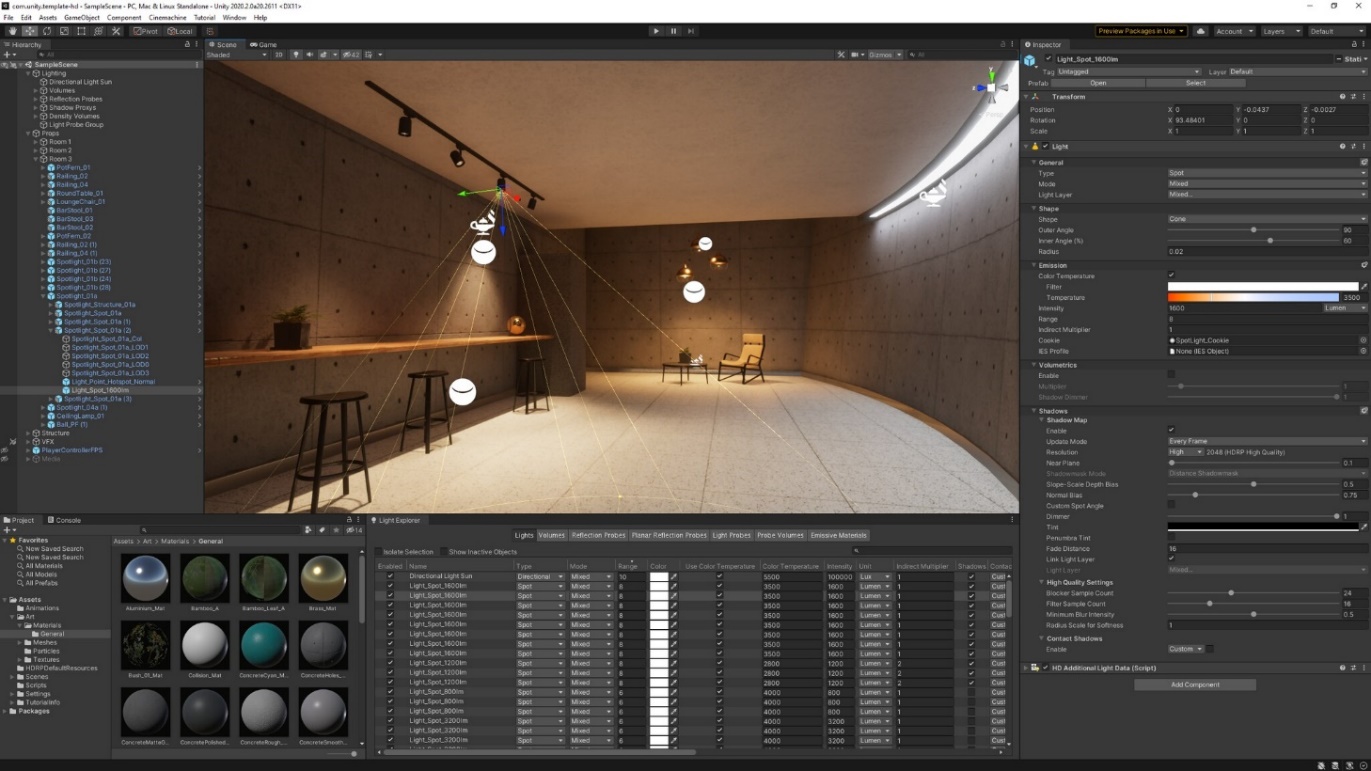


Рисунок 1.3 – Интерфейс *Unity* приложения

Редактор особенно удобен для процессов с последовательным улучшением, например, циклов создания прототипов или тестирования. Даже после запуска игры остается возможность модифицировать в нем объекты и двигать элементы сцены. Настраивать можно и сам редактор. Для этого применяются сценарии, добавляющие к интерфейсу новые функциональные особенности и элементы меню.

Дополнением к производительности, которую обеспечивает редактор, служит сильная межплатформенная поддержка набора инструментов *Unity*. В данном случае это словосочетание подразумевает не только места развертывания (игру можно развернуть на персональном компьютере, в интернете, на мобильном устройстве или на консоли), но и инструменты разработки (игры создаются на машинах, работающих под управлением как *Windows*, так и *Mac OS*). Эта независимость от платформы явилась результатом того, что изначально приложение *Unity* предназначалось исключительно для компьютеров *Mac*, а позднее было перенесено на машины с операционными системами семейства *Windows*. Первая версия появилась в 2005 году, а к настоящему моменту вышли уже пять основных версий (с множеством небольших, но частых обновлений). Изначально разработка и развертка поддерживались только для машин *Mac*, но через несколько месяцев вышло обновление, позволяющее работать и на машинах с *Windows*. В следующих версиях добавлялись все новые платформы развертывания, например межплатформенный веб-плеер в 2006-м, *iPhone* в 2008-м, *Android* в 2010-м и даже такие игровые консоли, как *Xbox* и *PlayStation*. Позднее появилась возможность развертки в *WebGL* – новом фреймворке для трехмерной графики в веб-браузерах. Немногие игровые движки поддерживают такое количество целевых платформ развертывания, и ни в одном из них развертка на разных платформах не осуществляется настолько просто.

Изначально *Unity* предназначался для создания трехмерных игр, но у него есть и другие варианты применения. Начиная с версии 4.3, выпущенной в конце 2013 года, в *Unity* появилась возможность отображения двумерной графики, хотя и раньше с помощью этого инструмента разрабатывали двумерные игры (особенно мобильные, в создании которых помогла кросс-платформенная природа *Unity*). Изначально для эмуляции двумерной графики в трехмерных сценах требовался сторонний фреймворк (например, *2D* *Toolkit* от *Unikron Software*). В конечном счете основной редактор и игровой движок поменяли, встроив в него двумерную графику. Именно с этой функциональностью и знакомит данная глава. Рабочий процесс при создании двумерной и трехмерной графики в *Unity* примерно одинаков и включает в себя импорт графических ресурсов, перетаскивание их в сцену и написание сценариев, которые затем будут присоединены к объектам. Основной вид ресурсов, необходимых для создания двумерной графики, называется «спрайтом».

Дополнением к этим основным достоинствам идет и третье, менее бросающееся в глаза преимущество в виде модульной системы компонентов, которая используется для конструирования игровых объектов. «Компоненты» в такой системе представляют собой комбинируемые пакеты функциональных элементов, поэтому объекты создаются как наборы компонентов, а не как жесткая иерархия классов. В результате получается альтернативный (и обычно более гибкий) подход к объектно-ориентированному программированию, в котором игровые объекты создаются путем объединения, а не наследования.

Оба подхода схематично показаны на рисунке 1.4.

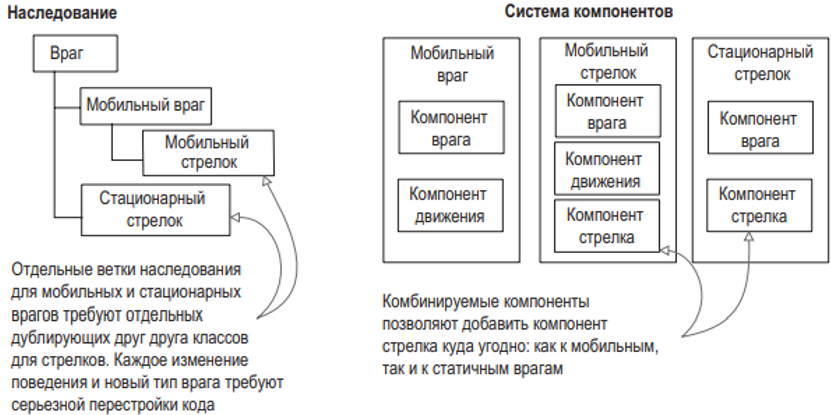


Рисунок 1.4 – Сравнение наследования с компонентной системой

Каждое изменение поведения и новый тип врага требуют серьезной перестройки кода. Комбинируемые компоненты позволяют добавить компонент стрелка куда угодно: как к мобильным, так и к статичным врагам. В компонентной системе объект существует в горизонтальной иерархии, поэтому объекты состоят из наборов компонентов, а не из иерархической структуры с наследованием, в которой разные объекты оказываются на разных ветках дерева. Такая компоновка облегчает создание прототипов, потому что взять нужный набор компонентов куда быстрее и проще, чем перестраивать цепочку наследования при изменении каждого объекта. Разумеется, ничто не мешает написать код, реализующий вашу собственную компонентную систему, но в *Unity* уже существует вполне надежный вариант такой системы, органично встроенный в визуальный редактор. Эта система дает возможность не только управлять компонентами программным образом, но и соединять и разрывать связи между ними в редакторе. Разумеется, возможности не ограничиваются составлением объектов из готовых деталей; в своем коде вы можете воспользоваться наследованием и всеми наработанными на его базе шаблонами проектирования [2, c.17].

**1.3 Сетевое взаимодействие в играх**

Сетевое взаимодействие представляет собой установление связи между двумя или более компьютерами. Основная концепция заключается в установлении взаимоотношений между клиентом (компьютером, осуществляющим запрос информации) и сервером (компьютером, ответственным за обработку запросов информации). Сервер может представлять собой выделенный хост-компьютер, принимающий запросы от всех участников, а также компьютер одного из игроков, на котором запущена игра и который действует в роли сервера для остальных участников. С момента запуска сервера и подключения клиента эти два компьютера приобретают возможность взаимообмена информацией, соответствующей игровому процессу.

Существуют две основные модели взаимодействия в сфере сетевых игр.

Первая модель, известная как *Peer-to-peer*, представляет собой архитектуру, основанную на равноправном взаимодействии программ. В этой модели каждая запущенная копия игры обладает равными правами и обязана отслеживать состояние других копий, осуществляя передачу сообщений между ними. Эта модель эффективна для игр с небольшим количеством участников, однако при увеличении числа игроков наблюдается значительный поток сообщений, необходимых для поддержания игрового процесса. Схематическое представление модели *peer-to-peer* представлено на рисунке 1.5.

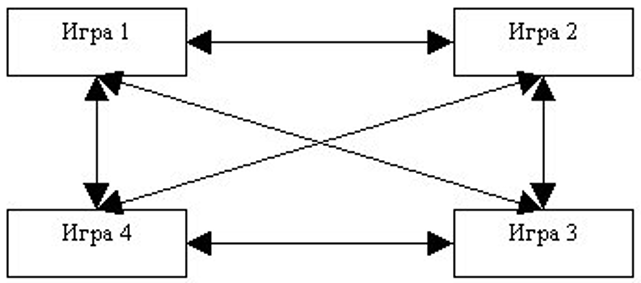


Рисунок 1.5 – *Peer-to-peer*

Часто в этой модели одна из игр назначается хостом, к которому обращаются новые игроки для участия. Однако хост не управляет обменом сообщениями между играми.

Вторая модель взаимодействия в сетевых играх - модель *Client/Server* (Клиент/Сервер). Здесь выделяются два типа программ: программа-сервер, ответственная за организацию и взаимодействие в игре, и программы-клиенты, обменивающиеся сообщениями только с сервером, не взаимодействуя между собой напрямую. Такой подход позволяет эффективно использовать пропускную способность каналов связи и поддерживать участие множества игроков в игре одновременно. Модель взаимодействия *Client/Server* схематически изображена на рисунке 1.6.

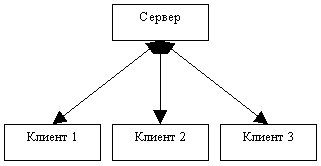


Рисунок 1.6 – *Client/Server*

Каждая игра в модели одноранговой архитектуры является самостоятельной единицей, обеспечивающей полноценную организацию игрового процесса. Каждая копия игры функционирует независимо, передавая информацию о своем состоянии другим копиям и получая данные о состоянии других игр. В клиент-серверной схеме взаимодействия существует несколько подходов, включая разделение вычислений между сервером и клиентами.

Существуют и другие модели организации сетевого взаимодействия игр, такие как кольцевая система обмена сообщениями или использование нескольких игровых серверов.

Относительно видов сетевых игр, их можно разделить на два основных типа. Первый - пошаговые игры (*Turn Based Games*), к примеру, шахматы или компьютерные версии настольных игр. Второй тип - игры реального времени (*Real Time Games*), представленные, например, игрой «*Dota 2»*. При этом пошаговые игры лучше переносят задержки в сетевой связи, тогда как игры реального времени могут столкнуться с серьезными проблемами при даже небольших сбоях в связи.

Многие современные игры, не предназначенные для многопользовательского режима, поддерживают хранение результатов игр на специальных интернет-сервисах производителей, что позволяет игрокам соревноваться «заочно».

**1.4 Язык программирования *C#***

*С#* – это язык с *C*-подобным синтаксисом. Здесь он близок в этом отношении к *C++* и *Java* [3, c.21].

Будучи объектно-ориентированным языком, он много перенял у *Java* и *С++*. Как и *Java*, *C#* изначально предназначался для веб-разработки, и примерно 75% его синтаксических возможностей такие же, как у *Java*. *C#* также называют «очищенной версией *Java*». Ещё 10% *C#* позаимствовал из *C++* и 5% – из *Visual Basic*. Оставшиеся 10% *C#* – это реализация собственных идей разработчиков. Объектно-ориентированный подход позволяет строить с помощью *C#* крупные, но в то же время гибкие, масштабируемые и расширяемые приложения.

*C#* уже давно поддерживает много полезных функций:

– инкапсуляция;

– наследование;

– полиморфизм;

– перегрузка операторов;

– статическая типизация [4, c.43].

При этом он всё ещё активно развивается, и с каждой новой версией появляется всё больше интересного – например лямбды, динамическое связывание, асинхронные методы.

По сравнению с другими языками *C#* довольно молод, но в то же время он уже прошёл большой путь. Первая версия языка вышла вместе с релизом *Microsoft Visual Studio .NET* в феврале 2002 года. Текущей версией языка является версия *C# 8.0*, которая вышла в сентябре 2019 года вместе с релизом *.NET Core 3* [6].

У «шарпа» выделяют много преимуществ:

– поддержка подавляющего большинства продуктов *Microsoft*;

– бесплатность ряда инструментов для небольших компаний и некоторых индивидуальных разработчиков (*Visual Studio*, облако *Azure*, *Windows Server*, *Parallels Desktop* для *Mac Pro*);

– типы данных имеют фиксированный размер (32-битный *int* и 64-битный *long*), что повышает «мобильность» языка и упрощает программирование, так как вы всегда знаете точно, с чем вы имеете дело;

– автоматическая «сборка мусора», это значит, что нам в большинстве случаев не придётся заботиться об освобождении памяти. Вышеупомянутая общеязыковая среда *CLR* сама вызовет сборщик мусора и очистит память;

– большое количество «синтаксического «сахара» (специальных конструкций, разработанных для понимания и написания кода);

– низкий порог вхождения;

– с помощью *Xamarin* на *C#* можно писать программы и приложения для таких операционных систем, как *iOS*, *Android*, *MacOS* и *Linux* [5, c.38].

Но есть у *C#* и некоторые недостатки:

– приоритетная ориентированность на платформу *Windows*;

– язык бесплатен только для небольших фирм, индивидуальных программистов, стартапов и учащихся;

– инструментарий *C#* позволяет решать широкий круг задач, язык действительно очень мощный и универсальный.

На нём часто разрабатывают:

– веб-приложения;

– игры;

– мобильные приложения для *Android* или *iOS*;

– программы под *Windows*.

Перечень возможностей разработки практически не имеет ограничений благодаря широчайшему набору инструментов и средств. Конечно, всё это можно реализовать при помощи других языков. Но некоторые из них узкоспециализированные, а в некоторых придётся использовать дополнительные инструменты сторонних разработчиков. В *C#* решить широкий круг задач возможно быстрее, проще и с меньшими затратами времени и ресурсов.

2 АРХИТЕКТУРА И СТРУКТУРА ПРИЛОЖЕНИЯ

2.1 Архитектура приложения

Представление надежной и эффективной архитектуры сетевого приложения является критическим компонентом в разработке современных многопользовательских игр и приложений. Процесс пришествия к пониманию важности архитектуры и структуры приложений был пошаговым, основанным на накопленном опыте, столкновениях с проблемами и поиске эффективных решений для их решения.

Её главная цель – определить структуру, компоненты и взаимодействия между ними в приложении. Она позволяет увидеть «общую картинку» приложения и оценить его целостность, а также обеспечить гибкость и расширяемость приложения в будущем.

Некоторые из основных причин, по которым нужно разрабатывать архитектуру приложения:

– четкое определение структуры приложения и его компонентов;

– оценка сложности приложения и определение возможных рисков и проблем;

– обеспечение гибкости и расширяемости приложения в будущем;

– упрощение процесса разработки за счет более четкого понимания того, что нужно разрабатывать и как компоненты приложения взаимодействуют друг с другом;

– улучшение качества кода и снижение затрат на его сопровождение.

Разработка архитектуры может помочь определить, какие компоненты нужны для реализации игры, как они будут взаимодействовать друг с другом, какие алгоритмы нужны для обработки данных и как эти компоненты будут связаны с игровым движком. Это поможет упростить процесс разработки и обеспечить более гибкую и расширяемую архитектуру для будущих доработок и улучшений игры.

Чтобы правильно построить архитектуру приложения стоит использовать метод декомпозиции для правильной оценки задачи и последующем её разбиении на отдельные компоненты с целью оптимизации разработки приложения и дальнейшей поддержки.

Декомпозиция – это разделение большого и сложного на небольшие простые части. При постановке задач декомпозировать – значит разбить абстрактную большую задачу на маленькие задачи, которые можно легко оценить.

Таким образом, правильное разбиение задачи на отдельные компоненты приведет к оптимизированию разработки архитектуры и структуры приложения.

Результат разбиения задачи на отдельные компоненты показан на рисунке 2.1.

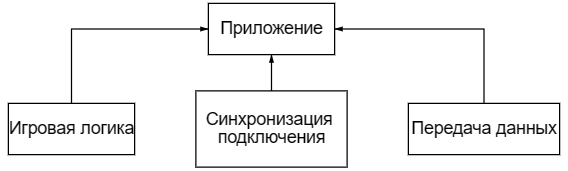


Рисунок 2.1 – Результат разбиения задачи на отдельные компоненты

Игровая логика будет разрабатываться на игровом движке *Unity*. Он будет отвечать за отображение графических элементов, также на нем будут разработаны механики и сценарии игры.

Сетевое взаимодействие между игроками будет обусловливаться их синхронизацией подключения и передачей данных между ними. Синхронизация подключения и передача данных будут разработаны средствами платформы .Net и протокола *TCP*.

После этого стоит выбрать шаблон сетевого взаимодействия. Здесь нужно учитывать, что в игре будет небольшое кол-во игроков. В таком случае подойдет шаблон «*P2P*», который относится к категории симметричных клиент-серверных шаблонов.

Симметричность в данном контексте означает отсутствие в сети подразделения на клиентов и серверы. В этом шаблоне одна система выступает и как клиент, и как сервер. Каждая система, также называемая пиром, отправляет запросы другим пирам сети и в то же время получает и обслуживает запросы от других пиров этой сети. Такая схема сильно отличается от традиционной клиент-серверной сети, в которой клиент должен только отправлять запрос и ожидать его обработки сервером.

Таким образом, это будет работать следующим так: пользователь будет выбирать роль сервера или клиента. Если была выбрана роль сервера, то пользователю нужно будет ожидать подключения клиента. Если была выбрана роль клиента, то пользователю нужно будет подключаться к серверу. В результате чего у каждого игрока будет запущена игровая логика и в это время будет обрабатываться весь пользовательский ввод локально, а результаты ввода будут отправлены другому пользователю.

На рисунке 2.2 представлена архитектура «*P2P*» шаблона.

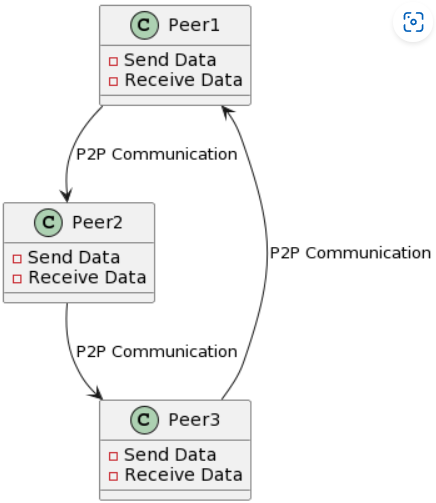


Рисунок 2.2 – «*P2P*» шаблон

Помимо того, что этот шаблон хорошо подходит к сетевым приложениям, в которых количество клиентов мало, он также ещё прост в реализации.

**2.2 Структура приложения**

После определения архитектуры приложения можно перейти к разработке структуры приложения.

Игрокам нужно будет подключаться и в ходе геймплея обмениваться данными. Таким образом, рационально было бы отделить сетевое взаимодействие от игровой логики. В таком случае, можно осуществлять подключение игроков на одной сцене, а на другой реализовать игровую логику.

После подключения игроков данные о подключении будут использоваться в сцене с игровой логикой для передачи данных по сети. В этом случае нужно как-то сохранять данные о подключении. Здесь можно столкнуться с проблемой того, что при переключении сцен в Unity ресурсы сцены будут очищены. В таком случае есть два варианта решения проблемы: сохранять данные в статические переменные или запускать фоновую вспомогательную сцену. Сохранение данных в статических переменных является плохой практикой, так как, в случае, когда полей данных будет много очищать их вручную будет неудобно. В случае с фоновой сценой также будут применяться статические переменные, но, так как данные будут находится в фоновой сцене, *Unity* сам позаботиться об их очистке.

В конченом варианте, сохранение данных в отдельной фоновой сцене выглядит неплохо.

В Unity последовательностью загрузки сцен управляет сам разработчик. Первым делом будет загружаться сцена для синхронизации подключения игроков. Чтобы сохранить данные о подключении пользователей нужно гарантировать, что фоновая сцена, с помощью которой данные о подключении будут переноситься и использоваться в другой сцене с игровой логикой, будет запущена первой. Это позволит проверить среду запуска приложения на пригодность прежде, чем запускать остальные сцены.

*2.2.1* Bootstrapp сцена – это место, где определяется простая сцена, которая содержит только системные объекты игры. Цель состоит в том, чтобы инициализировать основные системы игры, проверить готовность среды и систем к загрузке игры, а затем загрузить основную игру. Такая сцена загружается всего один раз в самом старте и больше никогда.

Схема сцен приложения представлена на рисунке 2.3.

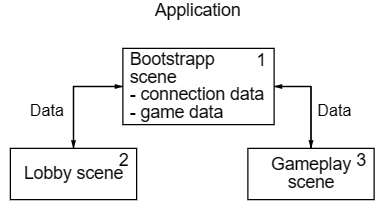


Рисунок 2.3 – Схема сцен приложения

*2.2.2* *Lobby* сцена – это сцена, которая будет отвечать за синхронизацию подключения игроков и загрузку *Gameplay* сцены. Игроку будет представлена возможность стать «хостом» или клиентом. При ожидании подключения важно сохранять отзывчивость приложения. Ведь это основа того, что оставляет пользователя в приложении во время каких-либо загрузок. Поэтому, во время подключения игрокам будет предоставлена возможность прервать операцию подключения и вернуться в начало *Lobby* сцены.

*Lobby* сцена будет отвечать за 2 этапа: установление соединения и загрузка уровня. Во время установления соединения со стороны клиента нужно будет ввести строку подключения. В случае с сервером, нужно будет просто ожидать подключения. Во время загрузки уровня будет отображаться условный процесс загрузки, при этом загрузку уровня прервать будет нельзя.

На рисунке 2.4 показана структура разработанных классов для синхронизации подключения игроков.

*2.2.3* *Gameplay* сцена – на этой сцене будет рассчитываться вся игровая логика и общение с клиентами. Данные для общения с клиентами будут браться из *Bootstrapp* сцены.

После разделения задачи на отдельные компоненты можно начать разработку необходимого программного кода, который будет реализовывать все программные компоненты.

2.3 Описание разработанных классов

В этом разделе будут описаны классы и скрипты, реализующие разработанные компоненты задачи. Под скриптами будут подразумеваться классы, которые наследуются от *MonoBehaviour*.

Стоит начать с классов для сетевого взаимодействия.

TCPBase – абстрактный класс, предоставляет основные методы для обмена данными между клиентом и сервером по сети. Содержит методы:

– InitializeTCPSocket(): cоздает и возвращает новый экземпляр сокета TCP;

– SendAsync<T>(): асинхронная отправка данных;

– RecvAsync<T>(): асинхронный прием данных;

– SendFixAsync<T>(): отправка данных фиксированного размера;

– RecvFixAsync<T>(): прием фиксированных данных;

– Stop(): завершает соединение с сервером;

– CheckConnetcion(): проверяет, подключен ли сокет;

– StopSocket(): останавливает сокет и освобождает ресурсы;

– GetLastError(): возвращает сообщение об ошибке;

– GetLocalPoint() и GetRemotePoint(): возвращают строковое представление локальной и удаленной конечных точек.

Cleint – класс представляет клиентскую сторону приложения и предоставляет методы для подключения и отключения от сервера с использованием базовой функциональности, предоставляемой классом **TCPBase.** Конструктор инициализирует объект клиента, устанавливает конечную точку и создает сокет.

Содержит методы:

– TryConnectAsync(): реализует асинхронную попытку подключения к серверу, в случае успеха устанавливает локальную и удаленную конечные точки;

– TryDisconnect(): предоставляет асинхронную возможность отключения от сервера с использованием события DisconnectReuseSocket, что позволяет повторно использовать сокет после отключения, обрабатывает возможные ошибки при отключении;

– Client (конструктор): инициализирует объект клиента, устанавливает конечную точку и создает сокет.

– *GetLastError* (унаследован от *TCPBase*): возвращает строку с описанием последней ошибки;

– *GetLocalPoint* (унаследован от *TCPBase*): возвращает строковое представление локальной конечной точки;

– *GetRemotePoint* (унаследован от *TCPBase*): возвращает строковое представление удаленной конечной точки.

*Server* – класс, который предоставляет функциональность для управления серверным сокетом, привязки к определенной конечной точке, прослушивания входящих подключений, асинхронного принятия подключений, а также проверки и остановки соединений. Конструкторинициализирует объект сервера, устанавливает локальный IP-адрес и создает серверный сокет. Содержит методы:

– SetEndPoint(): устанавливает конечную точку сервера;

– TryBindPoint(): попытка привязать серверный сокет к установленной конечной – точке с обработкой возможных ошибок;

– Listen(): запускает прослушивание входящих подключений;

– TryAcceptAsync(): асинхронно принимает входящее подключение и устанавливает соответствующие конечные точки;

– GetLocalIpAddress(): возвращает локальный IP-адрес сервера;

– Stop (переопределенный от TCPBase): останавливает сокет сервера и, если есть подключение, также останавливает клиентский сокет;

– CheckConnection (переопределенный от TCPBase): проверяет соединение серверного сокета, а также, если есть подключение, проверяет соединения и клиентского сокета.

*JSONStringSpliter* – класс предоставляет функциональность для разделения строки, содержащей несколько JSON-строк, на отдельные JSON-строки. Конструктор JSONStringSpliter инициализирует регулярные выражения для разделения JSON-строк. Метод SplitJSONStrings разделяет строку JSON на список отдельных JSON-строк. Если присутствует разделитель, используется регулярное выражение для разделения, иначе возвращается исходная строка как единственный элемент списка.

PrefixWriterReader – класс предоставляет функциональность для создания и анализа префикса, который используется для указания размера передаваемых данных. Содержит методы:

*– InitializePrefix*(): инициализирует префикс, указывая размер передаваемых данных.

– *WriteInfoPrefix*(): добавляет информацию к префиксу и возвращает результат как строку.

– *ReadInfoPrefix*(): считывает информацию из префикса и возвращает размер передаваемых данных. Если формат префикса не соответствует ожидаемому, генерирует исключение.

Теперь стоит описать классы, которые синхронизируют подключение пользователей.

Класс LobbyConnection отвечает за работу с подключением в лобби игры. Реализует методы для создания серверного и клиентского соединений, обработку событий и вызов соответствующих обработчиков при изменении статуса подключения. Содержит события:

– OnCancelServerConnectionEvent,OnCancelClientConnectionEvent: события отмены соединения на сервере и клиенте соответственно;

– OnStartCreateConnectionEvent: событие начала процесса создания соединения;

– OnCreateConnectionSeccessEvent: событие успешного создания соединения;

– OnCreateConnectionFailedEvent: событие неудачного создания соединения;

– OnCreateServerEndPointEvent: событие создания конечной точки сервера;

– OnCreateConnectionStringFailedEvent: событие неудачного создания строки подключения;

– onServerConnectionCreatedEvent: событие успешного создания серверного соединения;

– onClientConnectionCreatedEvent: событие успешного создания клиентского соединения.

Содержит методы:

**–***Awake*(): инициализирует экземпляры классов для создания соединений при запуске сцены.

– OnCreate(): метод для создания серверного соединения. Вызывает соответствующие события в зависимости от результата операции.

– OnConnect(): метод для создания клиентского соединения. Вызывает соответствующие события в зависимости от результата операции.

– OnServerBack(), OnClientBack(): методы для завершения соединения на сервере и клиенте соответственно.

– CancelConnection(Action cancelConnectionEvent)**:** метод для отмены текущего соединения.

Класс ClientConnectionCreator отвечает за создание и настройку клиентского соединения. Реализует методы для инициализации конечной точки, создания клиентского соединения, и возвращает результат асинхронной операции. Содержит методы:

– ClientConnectionCreator(): конструктор класса, инициализирует экземпляр регулярного выражения для разбора строки конечной точки;

– InitializeEndPoint(string endPoint): метод для инициализации конечной точки на основе строки endPoint. Проверяет корректность формата строки, парсит IP-адрес и порт, и создает объект IPEndPoint;

– CreateClientConnection(CancellationToken cancellationToken): метод для создания клиентского соединения асинхронно. Принимает токен отмены и возвращает результат операции. Создает экземпляр Client, пытается установить соединение с сервером асинхронно с периодической проверкой. При отмене задачи завершает соединение;

– GetClient(): метод возвращает созданный клиентский объект Client.

Класс ServerConnectionCreator ответственен за создание и настройку серверного соединения. Реализует методы для создания сервера, привязки к определенной конечной точке, прослушивания входящих подключений, а также возвращает результат асинхронной операции. Содержит метод создания серверного соединения асинхронно. Принимает токен отмены и делегат для обработки события создания конечной точки сервера. Создает новый экземпляр Server, привязывает его к конечной точке, начинает прослушивание входящих подключений и вызывает соответствующие события. Регистрирует действие по отмене операции. Возвращает результат асинхронной операции – успешное либо неудачное создание соединения.

Класс LevelLoader отвечает за загрузку уровня в лобби, контролируя процесс подключения и сигналов загрузки от сервера или клиента. Реализует методы для обработки создания серверного и клиентского соединений, а также проверку сигналов загрузки. Содержит методы:

**–**Start(): метод вызывается при старте объекта, инициализирует поля, подписывается на события подключения и отключения сервера или клиента.

– OnDisable(): метод вызывается при выключении объекта, Отписывается от событий подключения и отключения сервера или клиента.

– OnServerConnectionCreated(Server server): метод, обрабатывающий событие создания серверного соединения, инициализирует сервер и запускает проверку сигналов загрузки асинхронно, в случае успешной проверки сигнала и отсутствия отмены загружает уровень.

– OnClientConnecitonCreated(Client client): метод, обрабатывающий событие создания клиентского соединения, инициализирует клиента и запускает проверку сигналов загрузки асинхронно, в случае успешной проверки сигнала и отсутствия отмены загружает уровень.

– CheckSignal(Task checkSignalLoadingTask): метод для проверки сигнала загрузки асинхронно, вызывает события начала и конца проверки сигнала. Ожидает завершения задачи проверки сигнала и вызывает событие завершения проверки.

– LoadLevel(SceneName sceneName): метод для асинхронной загрузки уровня по его имени, использует асинхронную операцию SceneManager.LoadSceneAsync и ожидает завершения загрузки.

– OnServerCanceled(): метод, вызываемый при отмене подключения к серверу. Вызывает общий метод отмены операции.

– OnClientCanceled(): метод, вызываемый при отмене подключения к клиенту. Вызывает общий метод отмены операции.

– Cancel(): метод для отмены операции, пытается отменить токен отмены и, в случае возникновения исключения, останавливает сервер и клиента.

Класс LevelSignalChecker отвечает за проверку сигналов загрузки уровня между сервером и клиентом. Реализует методы для проверки сигнала загрузки от сервера и клиента, а также внутренние методы для инициализации и ожидания сигналов. Содержит методы:

**–**CheckServerLevelSignal(): метод для проверки сигнала загрузки уровня от сервера, инициализирует задачи отправки и приема сигнала, ожидает выполнения одной из них и возвращает результат проверки.

– CheckClientLevelSignal(): метод для проверки сигнала загрузки уровня от клиента, инициализирует задачи отправки и приема сигнала, ожидает выполнения одной из них и возвращает результат проверки.

– InitializeLevelLoadCompliteTask(): метод для инициализации задач отправки и приема сигнала загрузки уровня, создает задачу отправки, задачу приема и задачу, завершающуюся при получении сигнала или по истечении времени ожидания.

– WaitForSignalOrReciveSignal(): метод для ожидания сигнала загрузки уровня или приема сигнала от определенного типа соединения, возвращает результат проверки сигнала загрузки уровня.

Класс *User* представляет собой основной компонент для управления пользователем в контексте соединения. Он отвечает за инициализацию и управление соединением, обработку событий загрузки уровня, а также проверку состояния соединения во время выполнения. Содержит методы:

– Start(): метод, вызываемый при старте компонента, регистрирует обработчики событий загрузки уровня и завершения игры;

– OnDisable(): метод, вызываемый при выключении компонента, вызывает завершение игры и останавливает коммуникатор;

– Update(): метод, вызываемый на каждом кадре. проверяет состояние соединения и вызывает завершение игры в случае разрыва соединения;

– InitializeUserBase(): метод для инициализации базы пользователя и установки типа соединения и типа игрока;

–  OnLevelLoaded(): метод, вызываемый при загрузке уровня. Инициализирует коммуникатор, отправляет и принимает сигналы начала коммуникации, и, при успешной проверке, запускает коммуникатор и устанавливает флаг создания соединения;

– OnApplicationQuit(): метод, вызываемый при завершении приложения, останавливает сервер или клиент в зависимости от типа соединения;

– OnGameOver(): метод, вызываемый при завершении игры, останавливает коммуникатор и сбрасывает флаг создания соединения.

Класс PlayerData представляет данные игрока, которые могут сохраняться и использоваться в различных контекстах игры. Он содержит информацию о командах, выполненных командах, несвободных территориях, положении, направлении, состоянии кнопки и деньгах игрока.

Класс Communicator предназначен для обмена данными между клиентом и сервером с определенной частотой, основываясь на предоставленном тике (интервале времени между обменами). Содержит методы:

– Start(): запускает задачу для выполнения коммуникации с определенной частотой;

– CommunicateTask(): задача для выполнения коммуникации, осуществляет отправку данных и получение ответных данных в цикле с учетом указанного тика;

– CommunicateFix(): метод для обмена данными с использованием фиксированного формата, отправляет данные, получает ответные данные и возвращает полученные данные;

– Stop(): останавливает выполнение задачи и очищает данные отправки и получения.

BootSingleton – класс представляет собой обобщенную реализацию паттерна Singleton для Unity-компонентов. Позволяет создать единственный экземпляр объекта для класса-наследника.

3  СОЗДАНИЕ ПОМЕЩЕНИЯ С МИНИМАЛИСТИЧНЫМ ИНТРЬЕРОМ ДЛЯ ТАНЦЕВ, ПЕРСОНАЖИ

# ЗАКЛЮЧЕНИЕ

Список использованных источников

1. «3ds Max. Основы. Как и с чего начать?» [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/326532/ – Дата доступа: 22.10.2023.
2. Горелик, А. И., «Самоучитель 3ds Max 2018» / А. И. Горелик. – СПб.: «Питер», 2018. – 487 с.
3. «Простой Blender. Часть 1» [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/272519/ – Дата доступа: 10.10.2023.
4. Иванов, И. В., «Творчество с Blender» / И. Е. Иванов – СПб.: «Питер», 2017. – 510 с.
5. «Blender Basics 4-rd edition» / Джеймс Кронистер – СПб.: «Питер», 2019. – 570 с.
6. «Самоучитель по Wings 3D» / Ричард Джонс – Издательство: СПб.: «Питер», 2020. – 344 с.
7. Швембергер С., Щербаков И., Горончаровский В. «3ds Max: художественное моделирование и специальные эффекты»: «БХВ-Петербург», 2006. – 320 с.