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 курсовому проекту
по дисциплине «Объектно-ориентированное программирование»

на тему: «*WPF*-приложение «*Ping-Pong*» с использованием графики *DirectX*»

Исполнитель: студент группы ИТИ-22
Санин В. С.
Руководитель: преподаватель
Гуменников Е. Д.

Дата проверки: _____
Дата допуска к защите: _____
Дата защиты: _____

Оценка работы: _____

Подписи членов комиссии
по защите курсового проекта: _____

Гомель 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

История игровой индустрии насчитывает несколько десятков лет. За это время игры, выпускаемые на рынок, не только претерпели эволюцию с точки зрения технологий, но также превратились из продуктов, разработанных маленькими командами, в огромную индустрию, производящую крупные игровые проекты с помощью огромных команд. Первые игры были примитивными, но способными привлечь внимание.

Создание видеоигр является одним из крупнейших сегментов развлекательной индустрии. Развитие игровой индустрии сопоставимо с развитием киноиндустрии. С каждым годом растет количество и качество игровых продуктов. Сегодня создание игр включает использование векторной графики, 3*D* объектов, сглаживания и трассировки лучей. Игры разрабатываются как небольшими командами, так и крупными компаниями.

Производители игр выбирают различные решения для создания игр, включая платформу, язык программирования, графические средства и стилистику игры. От выбора зависят масштаб проекта, оптимизация на разных платформах и сложность разработки. Важно выделить основные методы разработки игрового продукта, язык программирования и технологию графики.

На сегодняшний день в мире игровой индустрии выделяются две категории видеоигр: крупные проекты и небольшие инди-игры. Небольшие команды разработчиков выделяются своими идеями и нестандартными решениями в геймдизайне. Технологии, позволяющие создавать игры, включают язык программирования *C# .NET* и технологию *DirectX* для визуализации игрового процесса.

Цель данного проекта – создать игру «*Ping-Pong*» для соревновательной игры двух игроков. Необходимо спроектировать архитектуру и иерархию классов для легкой масштабируемости и модификации игрового приложения. Также требуется проявить геймдизайнерские навыки для определения стилистики и графического отображения игры с использованием графического *API*. Важно также протестировать разработанное приложение и доказать, что оно соответствует всем требованиям.

1 ТЕХНОЛОГИИ РАЗРАБОТКИ ИГРОВОГО ПРИЛОЖЕНИЯ «*PING-PONG*»

1.1 Сравнение графических стеков технологий

В наше время есть много способов для реализации и построения сложных проектов и разных задач. Задачи могут быть решены с использованием одного из множество различных языков программирования, например: *C#, Java, C++* и другие. Также кроме языков программирования есть набор *API* для создания игрового приложения, например *DirectX* и *OpenGL*, каждая из *API* предоставляет доступ к использованию графических элементов в разрабатываемом приложении. Разработчик должен знать, как ему решить заданную задачу исходя из условий и требований, какие методы *API* выбрать для своего разрабатываемого приложения и какой язык лучше всего использовать для его написания.

OpenGL и *DirectX* – два из наиболее распространенных графических стеков технологий, используемых для разработки графических приложений и игр. Эти технологии имеют свои особенности и различия, которые могут быть важными для разработчиков при выборе стека технологий для своих проектов. Дальше будут рассмотрены графические *API OpenGL* и *DirectX* с последующим их сравнением.

OpenGL (*Open Graphics Library*) – это кроссплатформенный графический стек технологий, который был создан в 1992 году компанией *Silicon Graphics* и находится под управлением *Khronos Group* [1, с. 191]. Он используется для создания высокопроизводительных приложений и игр на различных операционных системах, включая *Windows, Mac OS* и *Linux*. *OpenGL* предоставляет разработчикам возможность работать с *3D*-графикой.

Реализация функционала *OpenGL* базируется на библиотеках. Сам по себе *OpenGL* содержит только набор геометрических примитивов в виде точек, линий и многоугольников. Иногда этого функционала может не хватить для реализации нужной задачи, тогда были созданы высокоуровневые библиотеки, которые дают более широкий аспект геометрических фигур и трёхмерных объектов, это делает разработку легче, а следовательно процесс разработки будет ускорен. В состав функциональных возможностей интерфейса входит:

– удаление невидимых линий и поверхностей. *Z*-буферизация;
– двойная буферизация;
– сглаживание;
– работа с цветом;
– использование списков изображений;
– работа с освещением;
– работа с текстурами.

Представленные функции реализованы по модели клиент-сервер: приложение отправляет команды *OpenGL*, который выступает в качестве сервера и выполняет поступающие команды. Такая модель выполнения позволяет создавать приложения, использующие *OpenGL* и работающие на разных компьютерах.

Конвейер стандартных функций *OpenGL* представляет собой последовательность этапов обработки графической информации, которая происходит при рендеринге трехмерных сцен. Он состоит из нескольких этапов, каждый из которых выполняет определенные операции. Ниже приведены основные этапы конвейера *OpenGL*:

1) Этап вершинного шейдера (*Vertex Shader*): в этом этапе каждая вершина трехмерной модели проходит через вершинный шейдер, который выполняет программные вычисления над каждой вершиной. Здесь можно изменять положение вершин, применять матрицы трансформации, вычислять освещение и так далее;

2) Этап сборки геометрии (*Geometry Assembly*): на этом этапе геометрия формируется из обработанных вершин, и выполняются операции, такие как преобразование вершин, генерация новых геометрических примитивов и отсечение невидимых граней;

3) Этап геометрического шейдера (*Geometry Shader*): опциональный этап, в котором можно выполнять дополнительные вычисления над геометрией. Например, можно генерировать новые вершины, создавать тесселяцию, выполнять объемное освещение и другие операции;

4) Этап растеризации (*Rasterization*): на этом этапе геометрия преобразуется в пиксели на экране. Происходит определение, какие пиксели должны быть заполнены, а какие игнорироваться. Это включает определение границ треугольников, вычисление интерполированных значений для каждого пикселя и выполнение отсечения по границам экрана;

5) Этап фрагментного шейдера (*Fragment Shader*): каждый пиксель, прошедший через растеризацию, обрабатывается фрагментным шейдером. Здесь выполняются вычисления, связанные с освещением, текстурированием, прозрачностью и другими эффектами. Результатом является окончательный цвет самого пикселя;

6) Этап вывода на экран (*Framebuffer*): на этом последнем этапе окончательные пиксели отображаются на экране. Они могут быть выведены на экран или использованы для дальнейшей обработки или прохода по конвейеру для рендеринга других объектов.

На рисунке 1.1 показан конвейер стандартных функций *OpenGL*.

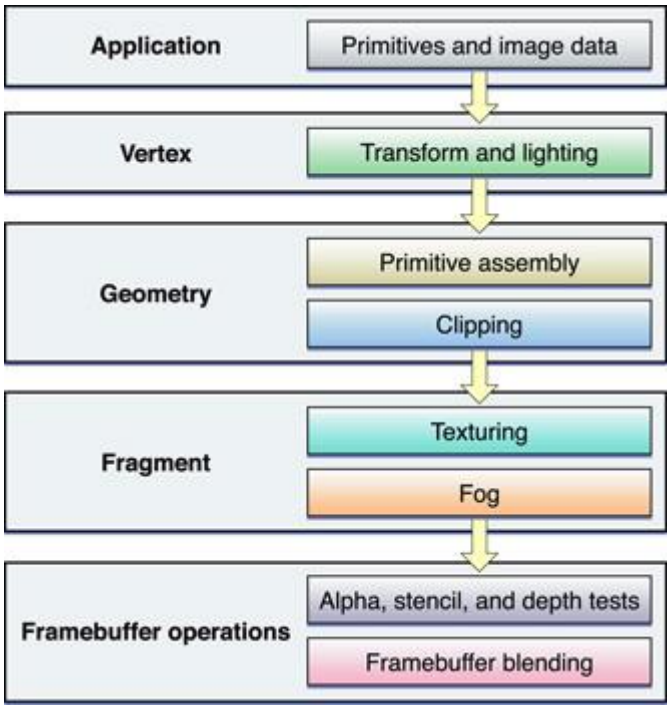

Рисунок 1.1 – Конвейер стандартных функций *OpenGL*

DirectX – это графический стек технологий, разработанный компанией *Microsoft* для операционной системы *Windows*. *DirectX* включает в себя несколько компонентов, таких как *Direct*3*D* для рендеринга 3*D*-графики, *Direct*2*D* для 2*D*-графики и *DirectCompute* для общих вычислений на графических процессорах (*GPU*) [2, с. 40]. *DirectX* был создан с целью обеспечения высокой производительности игр на операционной системе *Windows*.

Для подробного описания функции, которые может выполнять *DirectX* можно выделить:

– *DirectPlay* – высокоуровневые программный интерфейс, позволяющий осуществлять коммуникацию программы и удаленных сервисов. Данный интерфейс позволяет осуществить реализацию межсетевого взаимодействия прикладных программ;

– *DirectInput* – это интерфейс, позволяющий получать доступ к различным источникам ввода-вывода, таким как клавиатура, мышь и игровые геймпады;

– *Direct*3*D* – прикладной интерфейс, позволяющий производить работу с трехмерной графикой;

– *DirectDraw* / *Direct*2*D* – менеджер для управления памятью графических поверхностей, которые работают на платформе *Windows*. Используется для работы с двумерной графикой;

– *DirectSound* – интерфейс работы со звуком. Позволяет производить операции с микшированием и позиционированием источников звукового сигнала.

Первый релиз *DirectX* был выпущен в сентябре 1995 года под названием «*Windows Game SDK*». Ещё до появления *DirectX Microsoft* включила *OpenGL* в ОС *Windows NT*. *Direct*3*D* позиционировался как замена *OpenGL* в игровой сфере. В середине 1996 года был выпущен *DirectX* 2.0. Добавили интерфейс *Direct*3*D*, поддерживающий вывод уже трехмерной графики. Изменения в *DirectX* 8.0. были куда существеннее чем в предыдущих версиях, например появились пиксельные и вершинные шейдеры, которые вычисляют параметры графических объектов, также внедрили технологию мультитекстурирования. На одном полигоне стало возможно использовать две и более текстуры одновременно, что значительно увеличило производительность 3*D*-графики.

Конвейер *DirectX* 10, также известный как графический конвейер *DirectX* 10, является частью *DirectX API*, который предоставляет возможности для программирования графики и взаимодействия с графическими устройствами. Конвейер *DirectX* 10 представляет собой последовательность этапов обработки графических данных.

Конвейер *DirectX* 10 состоит из следующих этапов:

1) Входная сборка (*Input Assembly*): в этом этапе графические данные, такие как вершины, индексы и примитивы, подготавливаются для обработки в следующих этапах;

2) Вершинный шейдер (*Vertex Shader*): вершинный шейдер выполняет преобразование и обработку каждой вершины входных данных. Он может изменять положение, цвет, текстурные координаты и другие атрибуты вершин;

3) Тесселяция (*Tessellation*): этот этап вводится в *DirectX* 10 и позволяет более детальное разбиение примитивов (таких как треугольники) на более мелкие фрагменты для повышения детализации геометрии;

4) Геометрический шейдер (*Geometry Shader*): геометрический шейдер принимает примитивы, созданные вершинным шейдером или тесселяцией, и может выполнять дополнительные операции, такие как генерация новых примитивов или преобразование геометрии;

5) Растеризация (*Rasterization*): в этом этапе примитивы преобразуются в фрагменты, которые будут отображаться на экране. Растеризация также определяет, какие фрагменты находятся внутри границ примитивов и могут быть закрашены;

6) Пиксельный (фрагментный) шейдер (*Pixel Shader*): пиксельный шейдер применяется к каждому фрагменту, определяет его цвет и другие атрибуты на основе входных данных. Он может выполнять текстурирование, освещение, эффекты и другие операции для создания финального изображения;

7) Выходная сборка (*Output Merger*): в этом последнем этапе фрагменты объединяются, выполняются операции смешивания цветов и глубины, а затем выводятся на экран.

На рисунке 1.2 показан графический конвейер *DirectX* 10.

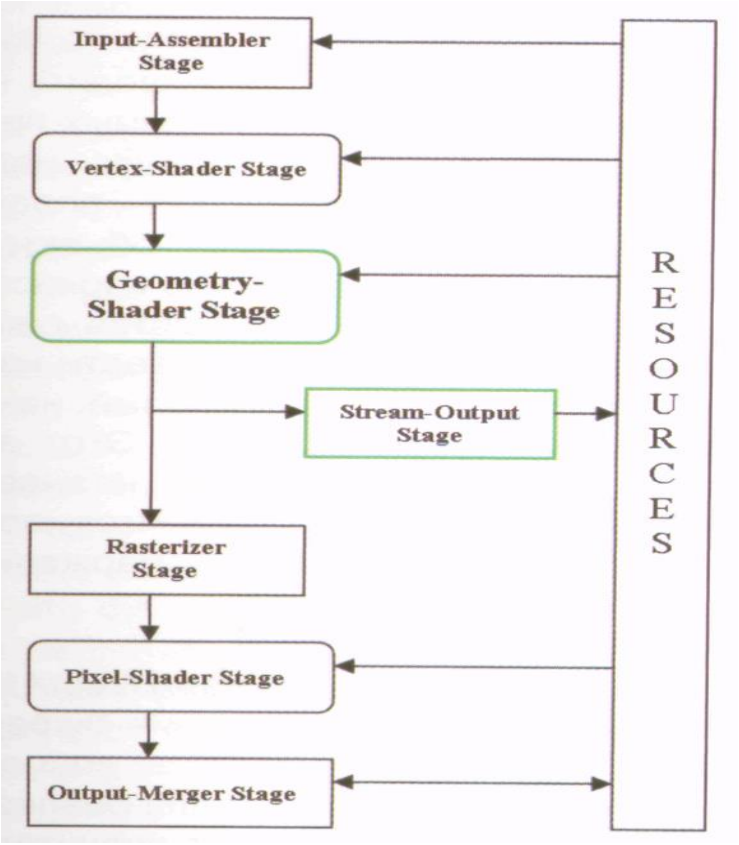

Рисунок 1.2 – Графический конвейер *DirectX* 10

Сравним *API* и драйвера *OpenGL* и *DirectX*. *OpenGL* и *DirectX* предоставляют различные *API* для программирования графических приложений. *OpenGL* использует *API* на основе языка Си, в то время как *DirectX* использует *API* на основе объектно-ориентированного языка *C++*. *API OpenGL* более прост и прямолинеен, что облегчает его использование, особенно для начинающих разработчиков. С другой стороны, *DirectX API* более сложен и требует более высокой квалификации разработчиков.

DirectX также включает в себя набор драйверов для взаимодействия с графическими устройствами, которые могут быть использованы для различных операций, таких как отображение графики на экране или ввода пользовательских действий. Эти драйверы обеспечивают высокую производительность и поддерживают широкий спектр графических устройств.

Дальше речь пойдёт о поддержке платформ, данными стеками технологий. *OpenGL* является кроссплатформенным *API* и может быть использован на различных операционных системах, включая *Windows*, *Mac OS* и *Linux* [3, с. 34]. Это

делает *OpenGL* более подходящим для разработки кроссплатформенных приложений, тогда как *DirectX* наиболее подходит для разработки игр и приложений, ориентированных на платформу *Windows*.

Одним из преимуществ *DirectX* является его интеграция с операционной системой *Windows*. *DirectX* может использоваться в связке с другими технологиями *Microsoft*, такими как *.NET* и *Windows Forms*, что обеспечивает удобный и совместимый способ разработки приложений для *Windows*.

С другой стороны, поддержка *OpenGL* на различных платформах делает его более гибким и масштабируемым, что позволяет разработчикам создавать приложения для различных операционных систем, не прибегая к переписыванию кода для каждой платформы.

Теперь же сравним их производительность. Одним из ключевых факторов при выборе графического стека технологий является производительность. *OpenGL* и *DirectX* предоставляют схожие возможности для рендеринга 3*D*-графики, но производительность может зависеть от конкретных условий использования.

DirectX имеет репутацию более высокой производительности, особенно в играх на *Windows*, где он может использовать полную мощность графической карты. С другой стороны, *OpenGL* может предоставлять более широкий спектр возможностей для оптимизации производительности приложений, благодаря его кроссплатформенной природе и открытой архитектуре.

Сравним сообщество и поддержку. Как и любая технология, *OpenGL* и *DirectX* имеют свои сообщества разработчиков и пользователей, которые могут предоставлять поддержку, решать проблемы и обмениваться знаниями и опытом. Оба стека технологий имеют активные сообщества, которые постоянно совершенствуют и улучшают их возможности.

Однако *DirectX* может иметь преимущество в доступности обучения и поддержке, благодаря своей популярности среди разработчиков игр на *Windows*. С другой стороны, поддержка *OpenGL* может быть более ограниченной, особенно для менее популярных платформ.

И в заключение подведём итоги. *OpenGL* и *DirectX* являются двумя из наиболее распространенных графических стеков технологий, используемых для разработки графических приложений и игр. Оба стека технологий имеют свои преимущества и недостатки, которые могут быть важными для разработчиков при выборе стека технологий для своих проектов.

DirectX является наиболее подходящим для разработки игр и приложений, ориентированных на платформу *Windows*, благодаря своей интеграции с операционной системой и возможностям для оптимизации производительности. Однако, *OpenGL* является кроссплатформенным *API* и может быть использован на

различных операционных системах, что делает его более гибким и масштабируемым.

В зависимости от конкретных потребностей проекта, разработчики могут выбирать между *OpenGL* и *DirectX*, основываясь на факторах, таких как производительность, кроссплатформенность, уровень поддержки и сообщества разработчиков.

Одной из проблем *OpenGL* является проблема производительности и качества, под разные платформы могут быть подключены различные функции и параметры, что при переносе может повлечь потерю качества изображения или отключения библиотеки. Для того, чтобы избежать данных проблем при работе с проектом на основе библиотеки *OpenGL* для *ОС Windows* лучше будет использовать *API DirectX*. Да и к тому же нам не нужна кроссплатформенность.

На языке *C#* имеется библиотека *SharpDX*, которая отображает функционал *DirectX* и идеально подходит для реализации поставленной задачи, а именно позволяет задействовать рендеринг 2*D* объектов и реализовать спрайтовую графику и отобразить 2*D* карту.

1.2 Сравнение языков программирования

Язык программирования – важная часть разработки программного обеспечения. Разные языки разработаны для выполнения различных инженерных задач, будь то разработка программного обеспечения под микроконтроллеры или же разработка больших корпоративных приложений с использованием веб технологий и средств работы с различными хранилищами. Следовательно, от выбранного языка зависят многие аспекты разработки, такие как общий объем написанного кода, а также подходы к проектированию и организации разработки.

На данный момент существует три наиболее распространенных подхода в программировании: процедурный, объектно-ориентированный и функциональный подходы. Объектно-ориентированный подход является наиболее подходящим методом решения поставленной задачи, так как он является наиболее распространенным и удобным для разработки игровых приложений.

Для использования возможностей объектно-ориентированного подхода существует большое количество языков программирования. Наиболее популярными из которых являются языки *Java SE* и *C# .NET*.

Java и *C#* – два из наиболее популярных языков программирования, используемых для создания различных типов приложений. Оба языка имеют свои преимущества и недостатки, которые могут быть важными для разработчиков при выборе языка для своих проектов. В данной статье мы рассмотрим сравнение *Java* и *C#* на 1000 слов.

Java – это объектно-ориентированный язык программирования, который был создан компанией *Sun Microsystems* в 1995 году. Он широко используется для создания приложений, работающих на различных платформах, таких как *Windows*, *Mac OS* и *Linux*. *Java* использует виртуальную машину *Java* (*JVM*) для запуска приложений, что позволяет их использовать на любой платформе, на которой установлена *JVM*.

C# – это язык программирования, разработанный компанией *Microsoft* в 1999 году [4, с. 24]. Он является объектно-ориентированным языком программирования, который используется для создания приложений, работающих на операционной системе *Windows*. *C#* использует *.NET Framework* для запуска приложений, которые могут быть написаны на других языках *.NET*, таких как *Visual Basic* и *F#*. Дальше будут рассмотрены синтаксисы и структуры таких языков программирования как *C#* и *Java* с последующим их сравнением.

Java и *C#* имеют схожий синтаксис и структуру, благодаря тому что они оба являются объектно-ориентированными языками программирования. Они используют классы, интерфейсы, наследование и полиморфизм для описания структуры приложения и его функциональности.

Одним из ключевых различий между *Java* и *C#* является то, что *Java* поддерживает множественное наследование интерфейсов, тогда как *C#* поддерживает только одно наследование. Это означает, что в *Java* класс может реализовывать несколько интерфейсов, тогда как в *C#* класс может наследовать только один другой класс.

Еще одним различием является то, что *C#* поддерживает свойства (*properties*), которые позволяют устанавливать и получать значения переменных через методы, в то время как *Java* использует методы *get* и *set* для этой цели.

Java и *C#* имеют свои инструменты разработки, которые могут быть использованы для создания приложений на этих языках. *Eclipse* и *NetBeans* являются популярными инструментами разработки для *Java*, тогда как *Microsoft Visual Studio* является наиболее популярным инструментом разработки для *C#*. Оба инструмента разработки предоставляют широкий набор функций, таких как отладка, автодополнение кода, интеграция с системами управления версиями и тестирование.

Visual Studio предоставляет больше инструментов для создания пользовательских интерфейсов и быстрого создания приложений, в то время как инструменты разработки для *Java* имеют большую гибкость и могут быть легко настроены для различных типов приложений.

Производительность является ключевым фактором при выборе языка программирования для проекта. Оба языка имеют схожие возможности для оптимизации производительности, такие как использование многопоточности и сборки мусора.

C# имеет преимущество в производительности благодаря использованию *Just-In-Time* (*JIT*) компиляции, которая компилирует код в машинный код во время выполнения программы. Это позволяет достичь более высокой производительности приложений, в том числе при работе с большими объемами данных.

Java, с другой стороны, использует *Ahead-Of-Time* (*AOT*) компиляцию, которая компилирует код в машинный код до запуска приложения. Это может быть недостатком при работе с большими объемами данных, но может быть преимуществом при работе с приложениями, требующими быстрой загрузки и запуска.

Java и *C#* имеют широкое сообщество разработчиков и пользователей, которые могут предоставлять поддержку и обмениваться знаниями и опытом. Оба языка имеют активные сообщества, которые постоянно совершенствуют и улучшают их возможности.

Однако *C#* может иметь преимущество в доступности обучения и поддержке, благодаря своей популярности среди разработчиков на платформе *Windows*. *Java*, с другой стороны, имеет большую распространенность и поддержку на различных платформах, что делает его более гибким и масштабируемым.

Java и *C#* – это два из наиболее популярных языков программирования, которые используются для создания различных типов приложений. Оба языка имеют свои преимущества и недостатки, которые могут быть важными для разработчиков при выборе языка для своих проектов.

Java является кроссплатформенным языком программирования и может быть использован на различных операционных системах, благодаря виртуальной машине *Java* [5, с. 74]. *Java* также поддерживает множественное наследование интерфейсов и имеет большую гибкость в создании приложений различных типов. Однако, *Java* может иметь недостаток в производительности при работе с большими объемами данных.

C# является языком программирования, ориентированным на платформу *Windows*, и может использоваться для создания приложений, работающих на этой платформе [6, с. 55]. *C#* имеет интеграцию с *.NET Framework* и *Microsoft Visual Studio*, что обеспечивает удобный и совместимый способ разработки приложений для *Windows*. *C#* также имеет преимущество в производительности благодаря использованию *JIT* компиляции.

При выборе между *Java* и *C#* для своего проекта разработчикам следует учитывать несколько факторов, таких как тип приложения, платформа, производительность, доступность обучения и сообщества разработчиков. Оба языка являются хорошими выборами для создания различных типов

приложений, и правильный выбор будет зависеть от конкретных потребностей проекта.

На языке *C#* имеются множество библиотек для работы с *DirectX* в качестве прикладного интерфейса для реализации графики в создаваемом приложении. Для использования *DirectX* можно использовать библиотеку *SharpDX*, которая поставляет компоненты интерфейса для работы с двухмерными и трёхмерными объектами.

1.3 Перечень используемых технологий

При рассмотрении предлагаемых программных решений применимо задаче, выделяется ряд инструментов, наиболее подходящих к использованию в игровом приложении.

Список предлагаемых инструментов включал в себя:

– графические библиотеки: *DirectX* и *OpenGL*;

– языки программирования: *C# .NET* и *Java*.

Список выбранных инструментов включает в себя:

– *DirectX API* (*SharpDX*) как главный инструмент для работы с графикой в приложении;

– *C# .NET* как язык программирования, на котором будет разработана основная часть игрового приложения.

2 ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ И СТРУКТУРЫ ПРИЛОЖЕНИЯ «*PING-PONG*»

2.1 Описание игры

Чтобы правильно составить архитектуру игрового приложения, необходимо представлять сам игровой процесс, для этого нужно описать его.

Игра представляет собой противостояние двух игроков друг против друга, значит в игре будет содержаться два игрока.

Ping-Pong с бонусами – это игровое приложение, основанное на классической игре «пинг-понг» или «теннис». Игрок управляет ракеткой, отбивающей мячик, его целью является отбить мячик от своей ракетки так, чтобы противник не успел отбить, тем самым противник пропустит мяч, а игрок заработает очко в свою пользу.

Однако в игре также присутствуют бонусы, которые добавляют дополнительные интерес и разнообразие в игровой процесс. Бонусы могут появляться случайным образом и предлагать различные преимущества одному из игроков. Бонусы в этой игре *Ping-Pong* могут включать:

– увеличение размера ракетки: бонус, который временно увеличивает размер ракетки игрока, облегчая отбивание мяча;

– ускорение ракетки: бонус, который временно увеличивает скорость ракетки игрока, позволяя более быстро реагировать на мяч;

– уменьшение размера ракетки противника: бонус, который временно уменьшает размер ракетки противника, усложняя отбивание мяча;

– замедление ракетки: бонус, который временно уменьшает скорость ракетки противника, усложняя отбивание мяча;

– передвижение ракетки по горизонтали: бонус, который временно позволяет ракетке игрока перемещаться по горизонтали, тем самым добавляя в игру динамичности.

Игра *Ping-Pong* с бонусами предлагает игрокам дополнительные возможности и стратегические решения, а также увлекательный игровой процесс, которым можно наслаждаться с друзьями.

Так как игра содержит двух игроков, то игроки, после появления на игровом поле будут передвигаться по игровому миру, используя устройство ввода в виде клавиатуры, где каждому игроку будет предоставляться своя область управления на клавиатуре. Игроки не могут подбирать более одного бонуса за определённый промежуток времени, необходимо ждать пока действие предыдущего бонуса закончится. Игра завершается, когда у одного из игроков количество очков будет 11, и разрыв очков будет больше двух, или при счёте 7:0, при этом выводится соответствующее сообщение, чтобы игроки понимали кто победил и как начать новую игру.

2.2 Структура игрового приложения

Необходимо определить и разработать структуру игрового приложения, где будет описана связь между различными её компонентами. В связи с поставленной задачей потребуются компоненты *DirectX* и *Object*. На рисунке 2.1 представлена архитектура разрабатываемого игрового приложения со всеми необходимыми компонентами и связью между ними.

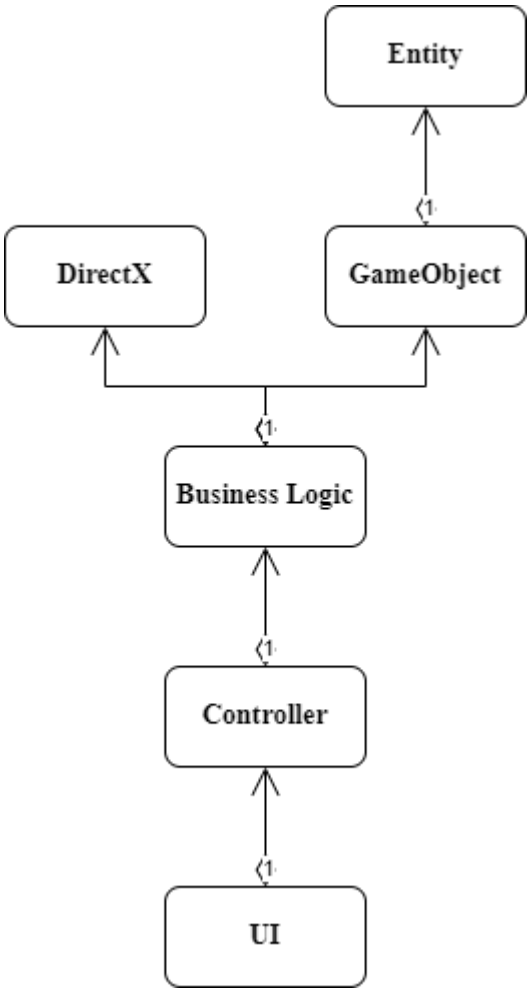

Рисунок 2.1 – Архитектура игрового приложения

В архитектуру игрового приложения входит:

– *UI* – компонент, необходимый для графического интерфейса пользователя, через него будет реализовано главное меню приложения;

– *Controller* – компонент, который требуется для связи *UI* и самого игрового процесса;

– *Business Logic* – компонент, который отвечает за логику взаимодействия игровых объектов, графического отображения и музыкального сопровождения;

– *DirectX* – компонент, отвечающий за реализацию *DirectX* и отображение объектов на игровом поле;

– *GameObject* – компонент, который отвечает за графическую информацию об используемых объектах;

– *Entity* – компонент, который описывает логику игровых объектов.

Графический интерфейс пользователя представлен в виде *WPF* приложения. Контроллер представлен методом передачи информации об игроках и запуске игрового приложения. Логика приложения является самым объемным компонентом и отвечает за запуск игры, вызов методов отображения игровых объектов, методов передвижения игроков и окончание игры. *DirectX* отвечает за фабрики, класс с методами для отображения игровых объектов и класс для связи с клавиатурой, чтобы считывать нажатие и определять действие игрока. *GameObject* представлен в виде классов, каждый из которых определяет размер объекта на игровом поле и используемый спрайт. *Entity* содержит классы, которые содержат позицию объекта относительно игрового поля и их основную логику.

2.3 Описание классов игровых объектов

После того как была описана структура игрового приложения можно рассматривать основные компоненты по отдельности, а именно их классы. Классов компонента *Entity* будет больше всего, т.к. в игре будут присутствовать сущность игрока, стены, мяча, бонуса и заднего фона. К каждому классу компонента *Entity* необходимо сопоставить класс компонента *GameObject*, если данный класс должен отображаться на игровом поле. Остальные компоненты будут иметь один или несколько файлов.

Каждый класс компонента *Entity* будет содержать позицию относительно игрового поля, а значит есть необходимость в создании абстрактного родительского класса, в котором будет реализовываться позиционирование объектов, а каждый класс будет наследоваться от абстрактного класса объекта, тем самым каждый игровой объект объекта будет реализовывать позиционирование относительно игрового поля. Так же каждый класс должен содержать свой собственный спрайт, для этого необходимо создать класс, в котором будет находиться информация о спрайте:

– угол поворота;

– индекс спрайта;

– ширина;

– высота;

– центр.

Таким образом у каждого объекта компонента *Entity* можно реализовать позиционирование на экране и указать спрайт для него, его размеры и угол поворота. С помощью редактирования этих параметров у объектов открывается

возможность осуществления их передвижения, вращения на игровом поле и смены отображаемого спрайта.

Компонент *Entity* будут содержать такие классы, как:

– *Player* – объект, который реализует логику для игрока. В логике данного класса будут определяться: метод движения игрока в различные стороны, для реализации передвижения будет использоваться список *Direction*, в котором будут перечислены стороны, в которые игрок может двигаться, метод, который будет контролировать очередь подачи, метод, который будет изменять счёт, методы, которые будут контролировать появление бонусов, чтобы эффекты от бонусов не накладывались друг на друга, а также метод, который будет возвращать игрока обратно на позицию после окончания действия определённого бонуса;

– *Background* – объект заднего фона, не содержит логику, а только позиционирование в виде центра экрана;

– *Ball* – объект, в котором реализуется логика мяча. В логике данного класса будет определяться метод передвижения мяча;

– *Wall* – объект стены, не содержит логику, а только позиционирование в виде центра экрана;

– *Bonus* – объект, в котором реализуется логика бонуса. В нем будет содержаться тип бонуса. Для определения типа бонуса будет введено перечисление, в котором описаны названия типов бонусов;

– *Sprite* – объект, в котором будут находиться основные параметры для изображения;

– *PlayerDecorator* – объект, который переопределяет реализацию игрока, модифицирует его;

– *BonusFactory* – объект, который реализует выборку бонуса определённого типа.

На рисунке 2.2 представлена иерархия классов компонента *Entity*, на котором отображена взаимосвязь классов между собой.

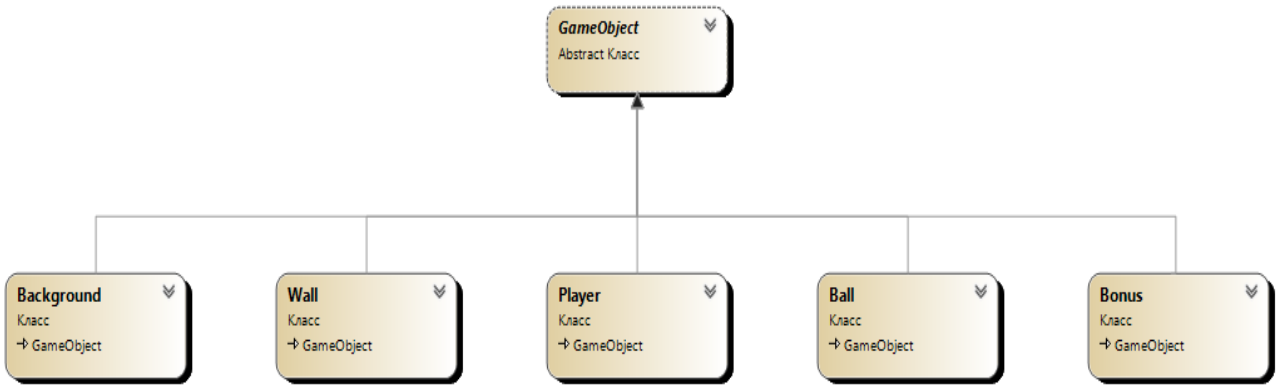

Рисунок 2.2 – Иерархия классов *Entity*

Во время игры, когда игроки будут подбирать бонусы, характеристики одного из них будут изменяться, каждый бонус будет отличаться своим действием: будет увеличивать или уменьшать, ускорять или замедлять ракетку, давать возможность перемещаться по горизонтали. После того как время действия у бонуса истекает, игрока нужно вернуть в прежнее, нормальное состояние.

На рисунке 2.3 изображена структура иерархии декорирования игрока, для возможности изменения характеристик объекта игрока в зависимости от подобранного типа бонуса.

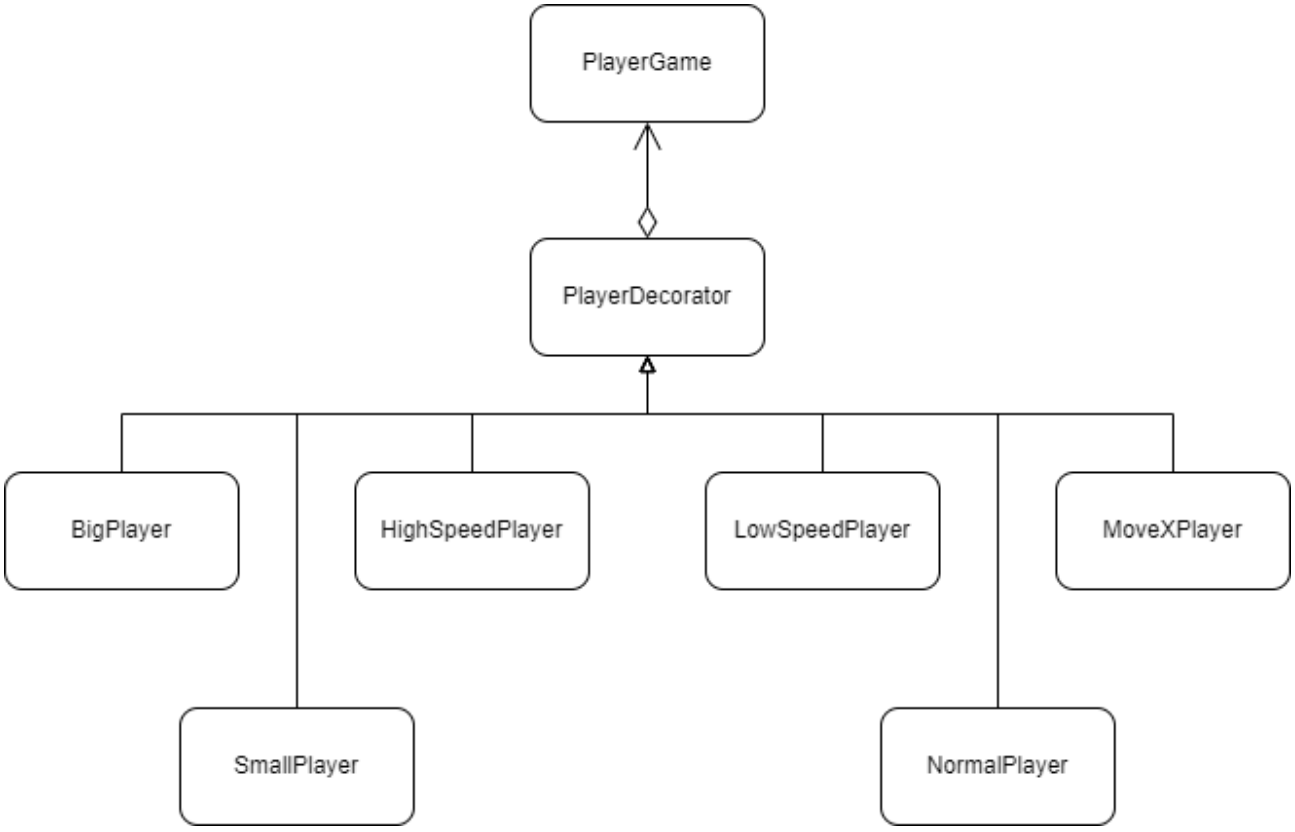

Рисунок 2.3 – Иерархия классов декоратора

Фабричный метод – это паттерн проектирования, который предоставляет интерфейс для создания объектов, но позволяет подклассам решать, какие классы конкретных объектов создавать. Он инкапсулирует создание объектов и делегирует это действие подклассам, чтобы они могли изменять тип создаваемого объекта. Это помогает сделать код более гибким и расширяемым, так как он основывается на абстракции и полиморфизме.

На рисунке 2.4 изображена структура иерархии фабричного метода, для возможности определения вида случайно-генерируемого бонуса на игровом поле.

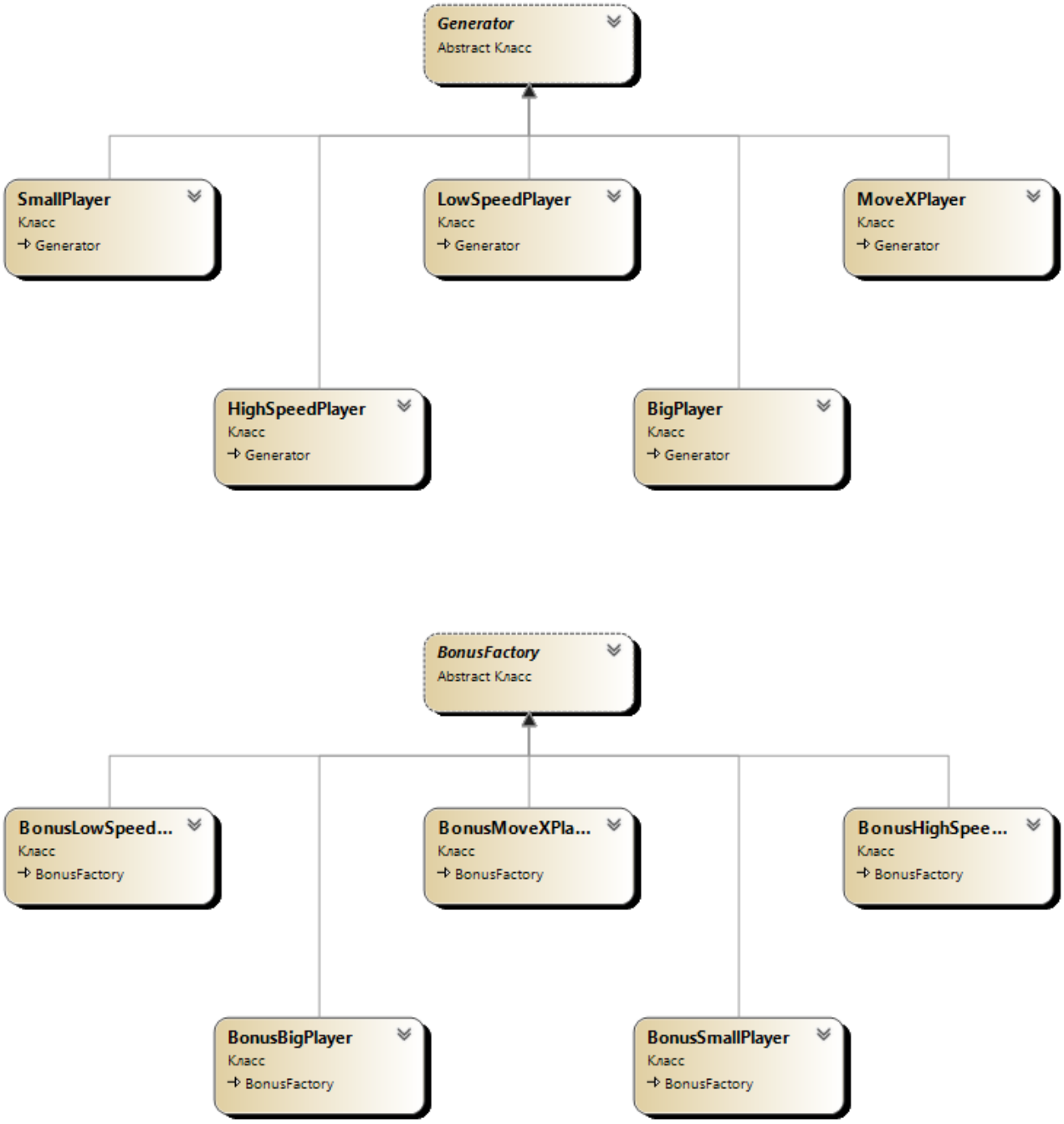

Рисунок 2.4 – Иерархия фабричного метода

Так как для каждого объекта необходима реализация отображения на экране, появляется необходимость в классах компонента *GameObject*, в котором будут реализовываться основные методы, требуемые для отображения. Отображаться будут спрайты на соответствующих объектах игры, чтобы объединить класс спрайта, где находятся основные параметры для изображения и класс

объекта, где прописана логика этого объекта. Требуется наличие абстрактного класса, который будет реализовывать некую логику контейнера, в нем будет находиться и класс объекта, и его спрайт.

Для описания игровой логики взаимодействия с объектами требуется описать границы объекта. Так как поступающие спрайты представляют собой прямоугольную картинку, то лучший выбор для описания границ объектов является использование прямоугольника, с помощью которого будет осуществляться расчет коллизии между разными объектами: шариком и игроком, шариком и стенами, игроком и стенами и игроком и бонусом. Определение прямоугольника будет находиться в абстрактном классе контейнера, чтобы каждый игровой объект, реализуя данный абстрактный класс, мог иметь у себя данный параметр. В абстрактном классе необходим метод, который будет описывать прямоугольник относительно передвижения игрока, чтобы границы игрока передвигались вместе с его спрайтом.

Так как нельзя заранее рассчитать ширину и высоту спрайта, который будет поступать на обработку, хранение и отображение, то появляется необходимость в параметре, который будет регулировать размер поступающего спрайта. Параметр будет определяться в диапазоне от нуля до единицы или больше, если требуется увеличить спрайт больше его стандартных размеров. Данный параметр так же будет определяться в абстрактном классе контейнера и принимать значения в классах компонента *GameObject*. Структура иерархии классов компонента *GameObject* представлена на рисунке 2.5, где описана взаимосвязь между классами.

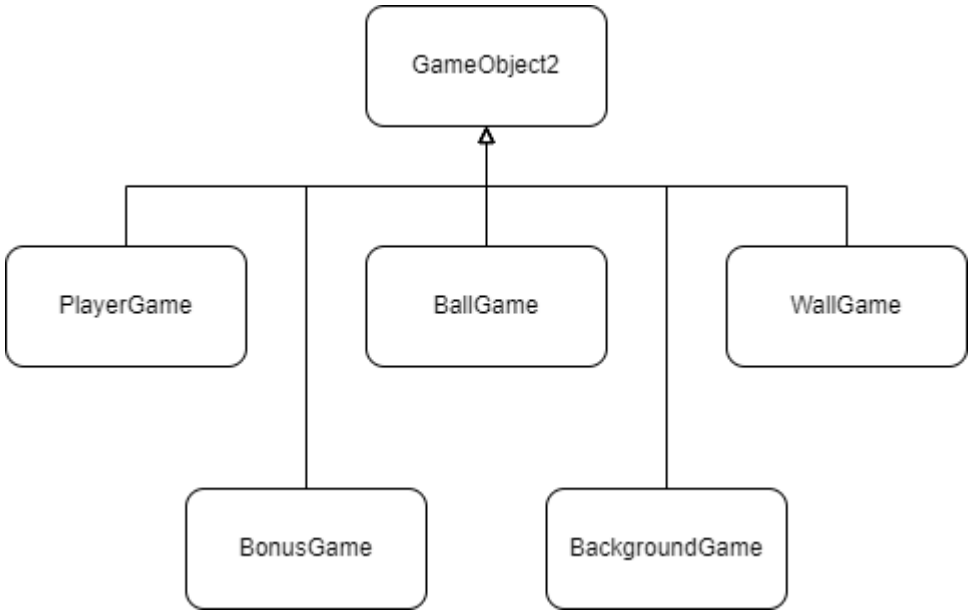

Рисунок 2.5 – Иерархия классов *GameObject*

Класс игрока компонента *GameObject* будет содержать время, чтобы рассчитывать начало действия бонуса и его окончание.

2.4 Описание классов *DirectX*

Классы, реализующие библиотеку *SharpDX*, содержат функциональность *DirectX*, необходимую для отображения игровых объектов и текста на экране.

Для реализации функциональности *DirectX* и отображения 2*D* спрайтовой графики необходим основной класс *DirectX*. Этот класс описывает и хранит следующие фабрики для обработки данных:

– фабрики для 2*D*-графики;

– фабрики для изображений;

– фабрики для работы с текстом.

Каждая фабрика обрабатывает определенный формат данных для получения результатов и вывода их на игровое поле: результаты фабрики 2*D*-графики передаются в игровое окно, и необходимые объекты отображаются на игровом поле. Фабрика изображений используется для декодирования и обработки входного изображения. Фабрика текста реализует отображение текста на экране в различных цветах и шрифтах.

Для того чтобы иметь возможность менять спрайты объекта без повторной загрузки спрайтов, необходимо хранилище для спрайтов. Это хранилище реализовано в виде коллекции объектов изображений, передаваемых в обработанном виде с фабрики изображений. Для пополнения коллекции предусмотрен соответствующий метод.

Двухмерная графика и объекты на игровом поле отображаются классами, реализующими рендеринг игровых объектов; классам, реализующим *DirectX*-фабрики, передается доступ к фабрике, экрану и цели рендеринга.

Классы, отображающие графику и объекты, имеют методы для отображения статических объектов на игровом поле, текста и динамических объектов, которые могут изменять размер, положение и угол поворота относительно центра. Методы для отображения динамических объектов имеют свойства для определения текущей позиции объекта.

Доступ к клавиатуре для управления ракеткой в игре обеспечивается классом, реализующим логику подключения клавиатуры. Этот класс содержит методы для подключения клавиатуры и отслеживания нажатий клавиш, а также методы для обновления состояния клавиатуры.

2.5 Описание логики взаимодействия классов

В логику взаимодействия разных уровней структуры заложена основа работоспособности приложения. Логика отвечает за связь разных классов и их функционирования вместе.

Класс, описывающий логику игры, станет основным классом приложения. Класс будет иметь метод, запускающий игру, и вызывается он из графического интерфейса пользователя сразу после запуска приложения. После запуска игры будет вызываться метод, который описывает рендер каждого кадра на игровом поле, что позволяет отображать объекты и придает плавности движениям игроков и шарика и смену спрайтов объектам. В методе рендера игрового поля будут вызываться методы, которые, используя класс для отрисовки, будут вызывать его методы для отображения объектов на экране. Класс будет иметь метод завершения игры, в нем проверяются условия для окончания игры, и если количество одного из игроков достигает 11 и разница очков больше двух, или счёт 7:0, то метод будет выводить на экран соответствующий текст, описывающий завершение игры и какую кнопку нужно нажать для старта новой игры.

Кроме методов отрисовки игровых объектов в классе будут присутствовать методы для определения логики подач. В одном из этих методов будет определяться кто подаёт, в зависимости от некоторых условий, таких как разница в счёте, например если счёт будет обычный 3:1, то игрок будет подавать дважды, однако если счёт будет больше меньше, то есть 10:10 и дальше с разницей в один мяч, то подавать игроки будут по одной подаче по очереди. В другом же методе в зависимости от того, какое условие соблюдается, то есть как первый метод определил чей мяч, мяч будет закрепляться за одним из игроков, для дальнейшей подачи.

В классе будет реализован метод, содержащий логику коллизии объекта игрока со стенами через пересечение границ прямоугольников, если прямоугольники пересеклись, то игрок не может идти дальше в этом направлении. Однако если же каким-либо образом, например из-за высокой скорости ракетки или из-за какого-то бага, игрок окажется в стене, то есть его прямоугольник пересечётся с прямоугольником стены, то игрока будет выталкивать обратно на игровую площадку.

Кроме метода коллизии объекта игрока со стенами, будет реализован метод коллизии объекта мячика со стенами. Если мяч прилетит в стену, то есть границы его прямоугольника пересекутся с границами прямоугольника стены, то мяч отлетит от неё и продолжит движение в определённом направлении, в зависимости от физики игры.

Также будет метод, реализующий логику коллизии объекта мяча с игроком. При столкновении мяча и игрока, при пересечении границ их

прямоугольников, мячик будет отскакивать от ракетки в определённом направлении, в зависимости от физики игры.

Также будет реализован метод, определяющий логику коллизии объекта мяча с бонусами, если границы прямоугольников пересекутся, то будет вызываться метод, который реализует применение бонусов.

Метод, который будет реализовывать применение бонусов, будет иметь входной параметр в виде игрока, чтобы запускать для него таймер, с помощью которого будет определяться время действия бонуса на определённого игрока, а по истечению данного таймера будет реализовываться сбрасывание полученных эффектов от бонусов. Также будет вызываться метод, который будет выполнять избавление от возможности наложения эффектов бонусов друг на друга, за исключением совместного применения положительного и отрицательного бонусов.

Ещё будет определён метод, который будет отвечать за появление бонусов на игровом поле, при чём он должен будет содержать случайную генерацию бонусов, чтобы они не появлялись слишком часто. А также случайное определение игрока, на линии движения которого будет появляться этот бонус.

Логика со случайным определением разновидности бонуса, которая будет содержаться в методе.

Кроме того, будет реализован метод, который будет отвечать за выход мяча за поле, в котором будет просчитываться выход мяча и если такое будет происходить, то будет вызываться другой метод, который будет засчитывать гол.

Чтобы у игроков не возникал такой вопрос как, а какой же счёт в игре, необходимо выводить соответствующую информацию, которая будет отображаться с использованием фабрик *DirectX* и методов класса для отрисовки игровых объектов.

Связь между методами передвижения игрока и классом, который является контроллером клавиатуры, описывается в методе рендера, где определяется статус нажатия необходимой клавиши, и если такая клавиша была активирована, то вызывается метод, в который передается направление движения игрока.

3 ТЕСТИРОВАНИЕ И АПРОБАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ «*PING-PONG*»

3.1 Функционал классов игровых объектов

Каждый игровой класс описывает логику объекта, который будет использоваться в игре, также он имеет свой функциональный класс для отображения на экране.

В пространстве имен *Entity* будут располагаться основные классы, описывающие различные игровые объекты, а также реализовывать логику абстрактного класса *GameObject*. Абстрактный класс позволяет реализовать позиционирование объекта на игровом поле используя поле *_positionOfCenter* типа *Vector*2 из библиотеки *SharpDX*. Поле описывает позиционирование по осям *X* и *Y*. Имея данное поле каждый класс, реализующий игровой объект может быть помещен в определенную точку экрана.

Класс *Player* является самым большим по содержанию классом среди остальных классов данного пространства имен. Данный класс наследуется от абстрактного класса *GameObject*, соответственно при создании объекта игрока будет реализовано его позиционирования на экране. Класс *Player* имеет поле *width*, данное поле имеет вещественный тип с плавающей точкой одиночной точности. Оно будет устанавливать размеры игрока, а именно его ширину. Поле *height* имеет вещественный тип с плавающей точкой одиночной точности. Данное поле также будет устанавливать размеры игрока, но теперь уже его высоту. Поле *_score* имеет целочисленный тип и содержит значение счёта. Поле *_gorisont* хранит в себе логическое значение и определяет сможет ли игрок передвигаться по горизонтали. Поле *_position* имеет тип *Vector*2 и хранит в себе начальное положение игрока на игровом поле. Поле *_giftLimit* хранит в себе логическое значение и определяет может ли игрок получиться бонус или нет. Поле *_turnserve* хранит в себе логическое значение и определяет чья очередь сейчас подавать.

Так же класс *Player* имеет множество методов, которые регулируют некоторую логику игрока. Метод *Serve* представляет логику по определению очереди подачи, в нем проверяется логическое значение поля *_turnserve*, и если оно хранит в себе *true*, то оно меняется на *false*. Иначе же, если оно хранит *false*, то оно меняется на *true*. Метод *Move* имеет входной параметр типа *Direction*, который представляет собой перечисление с направлениями движения: *Up*, *Down*, *Left*, *Right*. Данный метод представляет логику определения необходимого направления движения игрока, если игрок нажимает кнопку движения вверх, то скорость игрока по оси ординат отнимается, а если вниз – прибавляется. Если же игрок нажимает кнопку движения влево, то сначала проверяются такие условия, как логическое значение поля *_gorisont*, оно должно быть *true*, и позиция игрока на

игровом поле, чтобы он не передвигался слишком далеко, например за невидимую часть экрана, а потом уже скорость игрока по оси абсцисс отнимается. Если же игрок нажимает кнопку движения вправо, то скорость игрока по оси абсцисс прибавляется. Расчёты передвижения игрока ведутся таким образом, потому что отсчёт значений *x* и *y* начинается с верхнего левого угла экрана. Метод *Goal* представляет логику счёта, если он вызывается, то значение поля *_score* увеличивается на один. Методы *GiftOn* и *GiftOff*, определяют логику появления бонуса, если вызывается *GiftOn*, то бонусы больше не смогут появляться, до того момента, пока не вызовется метод *GiftOff*. Метод *ReturnPosition* представляет логику возвращения игрока обратно на его линию по оси абсцисс. Данный метод сравнивает позицию игрока по *x* в момент его вызова с начальной позицией этого игрока. Если игрок находится правее, чем его первоначальная абсцисса, то его перемещает влево со скоростью один пиксель в кадр, а так как кадров в секунде явно больше, чем один, то игрока достаточно быстро возвращает на его первоначальную позицию по оси абсцисс. Если же игрок находится левее, то его перемещает вправо, до того момента, пока он не вернётся на свою первоначальную позицию по *x*.

Вторым по важности объектов стал класс *Ball*, который описывает логику мяча. В классе мяча есть поле *width*, данное поле имеет вещественный тип с плавающей точкой одиночной точности. Оно будет устанавливать размеры игрока, а именно его ширину. Поле *height* имеет вещественный тип с плавающей точкой одиночной точности. Данное поле также будет устанавливать размеры игрока, но теперь уже его высоту. Поле типа *Random _rnd*, в котором хранится случайное целочисленное значение, которое определяет в какую сторону полетит мяч при запуске игры, на разыгровке подачи. Публичное поле типа *Vector*2 *Direction*, оно определяет направления полёта мяча на игровом поле. Метод *Move* служит для передвижения мяча по игровому полю, при его вызове мячу придаётся скорость в определённом направлении, которое указывает поле *Direction*.

Класс *Bonus* отвечает за позиционирование бонуса на игровом поле, а также определяет его тип. В классе бонуса есть поле *_type* типа *TypeBonus*, оно определяет какого типа будет бонус. *TypeBonus* представляет собой перечисление разновидностей бонусов: *Big*, *HighSpeed*, *LowSpeed*, *Small*, *MoveX*. От типа бонуса зависит эффект, который будет даваться игроку.

В связи с изменением характеристик игрока было создано пространство имен *PlayerDecorator*, которое находится в пространстве имен *Entity*, куда помещены классы, реализующие декорирование игрока:

– *PlayerDecorator* – абстрактный класс, который представляет собой декоратор для класса игрока и наследуется от класса *PlayerGame*;

– *BigPlayer* – класс, который наследуется от абстрактного класса декоратора. Он изменяет игрока, увеличивая его, а делает он это используя метод *Resize* из класса *GameObject*2;

– *HighSpeedPlayer* – класс, который наследуется от абстрактного класса декоратора. Он изменяет скорость игрока, то есть увеличивает её, умножив на определённый коэффициент;

– *LowSpeedPlayer* – класс, который наследуется от абстрактного класса декоратора. Он изменяет скорость игрока, то есть уменьшает её, умножив на определённый коэффициент;

– *SmallPlayer* – класс, который наследуется от абстрактного класса декоратора. Он изменяет игрока, уменьшая его, а делает он это используя метод *Resize* из класса *GameObject*2;

– *MoveXPlayer* – класс, который наследуется от абстрактного класса декоратора. Он устанавливает значение публичного поля *Gorisont* для игрока *true*, тем самым резрашает игроку перемещаться по оси абсцисс;

– *NormalPlayer* – класс, который наследуется от абстрактного класса декоратора. Он служит для того, чтобы вернуть игрока в нормальное состояние. Класс *NormalPlayer* изменяет игрока, возвращая ему первоначальную скорость, его первоначальные размеры, вызвав метод *NormalSize* из класса *GameObject*2, устанавливает значение публичного поля *Gorisont* для игрока *false*, обнуляет таймер для игрока, а также вызывает метод *GiftOff* из класса *Player*, позволяя тем самым появляться бонусам на поле игрока.

Класс *Background* представляет собой класс, который описывает позиционирование заднего фона игры, поэтому он реализует абстрактный класс *GameObject* для определения позиционирования спрайта относительно центра экрана для отрисовки, и не содержит в себе никакой логики.

Класс *Wall* отвечает за позиционирование стен на игровом поле и не содержит никакой логики.

Класс *Sprite* не является описанием игрового объекта, а представляет сам спрайт. В классе присутствует поле *_angle*, которое имеет вещественный тип с плавающей точкой одиночной точности и описывает угол поворота загружаемого спрайта. Поле *_bitmapIndex* имеет целочисленный тип и указывает на номер спрайта, загруженного в метод *LoadBitmap* класса *DX*2*D*. Поля *_height* и *_width* имеют вещественный тип одиночной точности и определяют высоту и ширину спрайта соответственно. Поле *_center* имеет тип *Vector*2, который предоставляет библиотека *SharpDX*. Данное поле описывает центр спрайта, как точку центра высоты и ширины. Метод *Resize* предназначен для смены размеров спрайта у игрового объекта, чтобы демонстрировать анимацию отображения объекта или изменять его внешний вид.

Класс *BonusFactory* представляет собой совокупность из 12 классов, из которых классы *BonusFactory* и *Generator* являются абстрактными. Класс *BonusFactory* наследуют *BonusBigPlayer*, *BonusSmallPlayer*, *BonusHighSpeedPlayer*, *BonusLowSpeedPlayer*, *BonusMoveXPlayer*. Класс *Generator* наследуют *BigPlayer*, *SmallPlayer*, *HighSpeedPlayer*, *LowSpeedPlayer*, *MoveXPlayer*. Каждый из данных классов реализует определённый объект бонуса.

В пространстве имен *GameObject* содержатся классы, которые реализуют функционал отображения на игровом поле. Все классы данного пространства реализуют абстрактный класс *GameObject*2. Абстрактный класс описывает основную логику для связки логических и графических частей игровых объектов, тем самым позволяет определять поля действия объекта, описывая его границы. В абстрактном классе находится поле *_gameObject*, которое имеет тип абстрактного класса *GameObject* и описывает класс игрового объекта. В поле типа игрового объекта содержится вся необходимая для его логика. Поле *_sprites* имеет тип класса *Sprite*, в котором находится логика спрайта, его ширина, высота, поворот и номер необходимого спрайта из коллекции спрайтов, что хранится в классе *DX*2*D*, тем самым определяется отображаемая картинки на экране. Поле *_rect* имеет тип *RectangleF*, который предоставляется библиотекой *SharpDX*, и определяет прямоугольник объекта. Прямоугольник характеризует границу объекта, и именно с помощью его реализована логика коллизии объектов и регистрация уничтожения объектов. Поле *_scale* имеет тип вещественного числа с плавающей точкой одиночной точности и представляет собой масштабирование объекта при отрисовки его спрайта на экране, тем самым можно регулировать размер каждого объекта отдельно при выводе его спрайта на экран, не регулирования размер самой загружаемой картинки в коллекцию. Абстрактный класс *GameObject*2 имеет метод *GetRect*, который рассчитывает точку верхнего левого угла прямоугольника в виде разницы между позиционированием объекта на игровом поле и центром отображенного спрайта. Таким образом метод возвращает прямоугольник, описывающий границы объекта, который передвигается по игровому полю вместе с объектом.

Класс *BackgroundGame* не содержит логики, т.к. весь смысл данного класса в связывании позиционирования заднего фона относительно центра экрана и соответствующим отображаемым спрайтом.

Класс *BallGame* имеет только поле *_ball*, которое имеет тип класса *Ball* и позволяет взаимодействовать с логикой данного игрового объекта.

Класс *BonusGame* имеет только поле *_bonus*, которое имеет тип класса *Bonus* и позволяет взаимодействовать с логикой данного игрового объекта.

Класс *PlayerGame* имеет поле *_time* вещественного типа с плавающей точкой одиночной точности для определения времени действия бонуса. Так же класс

имеет поле *_player*, который имеет тип класса *Player* и позволяет обращаться к логике класса игрока сразу из класса *PlayerGame*.

Класса *WallGame* имеет только поле *_wall*, которое имеет тип класса *Wall* и позволяет взаимодействовать с логикой данного игрового объекта.

3.2 Функционал классов *DirectX*

Основные классы, реализующие функциональность *DirectX*, содержатся в пространстве имен *DirectX*. Классы реализуют отображение графики.

Класс *DX2D* описывает базовую функциональность для реализации графики. Он содержит поля, описывающие фабрики: поле *_factory* представляет фабрику 2*D*-графики, *_imgFactory* – фабрику изображений, а *TextFactory* – фабрику текста. Поле *_renderTarger*, поле, описывающее объект отображения; поле *_textFormat* описывает формат текста при отображении на экране; поля *_quanBrush* и *_redBrush* описывают установленные кисти, синюю и красную соответственно. Синим цветом обозначается счет, который отображается вместо счета в начале игры, а также записывается результат игры, который выиграл этот цвет, после окончания игры. Также, после окончания игры, красный цвет указывает, какую кнопку нужно нажать, чтобы начать новую игру. Поле *_bitmaps* представляет коллекцию, содержащую спрайты, загруженные и обработанные с помощью фабрики изображений в методе *GetImageLoad*. Метод *Dispose* позволяет освободить неуправляемые ресурсы библиотеки *SharpDX*.

Класс *Drawer* представляет логику отображения игровых объектов на экране с помощью *RenderTagret*. Этот класс принимает класс *DX2D*, который является ядром для реализации функций *DirectX* для реализации 2*D*-графики. Этот класс имеет поле *_translation* типа *Vector2* из библиотеки *SharpDX*, которое описывает положение объекта на игровом поле. Это поле используется для определения поля *_matrix* типа *Matrix3x2* из библиотеки *SharpDX*. Поле *_matrix* представляет собой перемножение матриц вращения, масштабирования и позиционирования объекта на игровом поле. Поле *last* также имеет тип *Matrix3x2*. Когда объект отрисовывается, поле *_matrix* вычисляется и применяется к объекту отображения, после чего объект можно перемещать по игровому полю без изменения каких-либо параметров. После отрисовки игрового объекта значение цели отрисовки равно значению поля *last* и возвращаются исходные параметры цели отрисовки. Метод *Draw* позволяет использовать матрицы для отображения динамических объектов, свойства которых могут изменяться в процессе игры. Метод *DrawObject* реализует отображение статических объектов без матриц вращения, масштабирования или позиционирования; метод *ScoreDraw* реализует отображение очков на игровом поле. Поскольку текст является статическим объектом, использование матрицы можно игнорировать; метод *ResultDraw* отображает

результаты игры на игровом поле; метод *ServeDraw* отображает другой текст вместо счета в начале игры.

Класс *InputManager* реализует подключение клавиатуры и может быть использован для отслеживания нажатия определенных клавиш. Этот класс имеет поле *_directInput* типа *DirectInput*, определяющее устройство ввода библиотеки *SharpDX*; поле *_keyboard* типа *Keyboard* представляет класс работы с клавиатурой. Поле *_keyboardState* определяет состояние операции с клавиатурой. Это поле используется для контроля нажатия определенной клавиши на клавиатуре. Поле *_keyboardAcquired* логического типа определяет состояние подключения клавиатуры. Метод *AcquireKeyboard* разрешает подключение клавиатуры. Метод *UpdateKeyboardState*, если подключение клавиатуры отключено, обновляет подключение клавиатуры, а также обновляет состояние нажатия клавиш. Метод *Dispose* реализует интерфейс *IDisposable* и освобождает неуправляемые ресурсы библиотеки *SharpDX*.

3.3 Функционал служебных классов

К служебным классам относятся классы *TimeHelper*, *Game* и *PhysicsEngine*, которые являются вспомогательными.

Класс *Game* является самым большим классом, который связывает компоненты графики и логики объектов. В нем существуют поля, которые хранят объекты всех классов. В данном классе реализуется метод *Run*, вызывающий со страницы *UI* и отображает окно с игровым полем, а также метод *RenderCallback*, который рендерит каждый кадр. В методе *RenderCallback* вызываются методы отображения объектов на игровом поле, методы коллизии, методы проверки окончания игры, а также в данном методе заключена логика передвижения игроков. В методе *PingPongServe* реализована логика привязки шарика к игроку, который будет подавать. В методе *PlayerColWalls* реализуется логика коллизии игрока со стенами. Метод *BallColWalls* реализует коллизию шарика со стенами. Метод *BallCol* определяет коллизию шарика с игроком. Метод *GiftCol* реализует логику коллизии игрока и бонуса. Метод *GiftUsage* запускает таймер действия бонуса, а также определяет его тип. Метод *BallIsOut* определяет вышел ли мяч за пределы поля или нет. Метод *ChoiceServe* реализует логику определения очереди подачи. Метод *GiftSpawn* содержится логика по случайной генерации бонуса, шанс появления бонуса у одного из игроков одна тысячная. Если срабатывает шанс появления бонуса, то случайным образом выбирается один из двух игроков, на линии которого появится соответствующий бонус. Метод *EndGame* реализует логику окончания игры, если выполняются необходимые условия. Метод *RenderForm_Resize* определяет масштабируемость игрового поля относительно изменения размера игрового окна. При запуске приложения вызывается метод

Run, который отображает игровое поле и вызывает метод *RenderCallback*, обрабатывающий каждый кадр и отображающий игровые объекты. Метод *Dispose* освобождает неуправляемые ресурсы игрового приложения.

В классе *PhysicsEngine* реализуется логика физики игры, то есть логика отскоков шарика от стен и игроков. Метод *UpdateBall* обновляет угол отскока шарика от стен. Метод *UpdatePlayer* рассчитывает и обновляет угол отскока шарика от игрока. Также этот метод ограничивает угол отскока шарика, чтобы он долго не бился об стены и игроки не ждали. Ещё метод *UpdatePlayer* увеличивает скорость шарика при каждом столкновении с игроком, прибавляя к скорости мяча определённый коэффициент.

Класс *TimeHelper* предназначен для работы со временем. Поле *_watch* имеет тип *Stopwatch*, который реализует подсчет времени. Метод *Reset* обнуляет значение поля *_watch* и запускает счетчик заново. Метод *Update* рассчитывает значение поля *_time*, представляющее подсчет времени. Данный класс необходим, чтобы рассчитать время для длительности бонусов, плавного перемещения игроков, плавного перемещения шарика.

3.4 Тестирование приложения

Важной частью всего приложения является его тестирование. Оно необходимо для проверки правильной работоспособности созданного приложения.

С помощью модульных тестов был протестирован основной функционал игровых классов через проверку каждого его метода, чтобы проверить правильную работоспособность созданного игрового приложения. На рисунке 3.1 отображены успешно пройденные тесты.

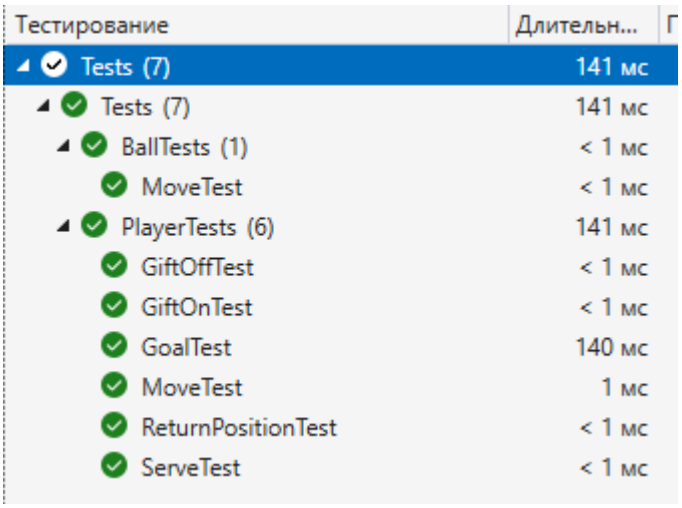

Рисунок 3.1 – Модульные тесты

На рисунке 3.2 показана сводка по группе выполненных тестов.

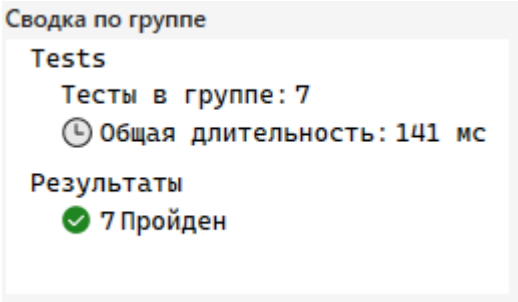

Рисунок 3.2 – Модульные тесты

Системное тестирование игровых приложений идеально подходит для тестирования данного приложения, поскольку оно похоже на эксплуатацию готового продукта и заменяет создание множества двойных тестов. Системное тестирование позволяет отслеживать и исправлять ошибки в процессе использования приложения.

Листинг игрового приложения представлен в приложении А.

3.5 Апробация приложения

На рисунке 3.3 показано игровое поле, только что запущенного приложения.

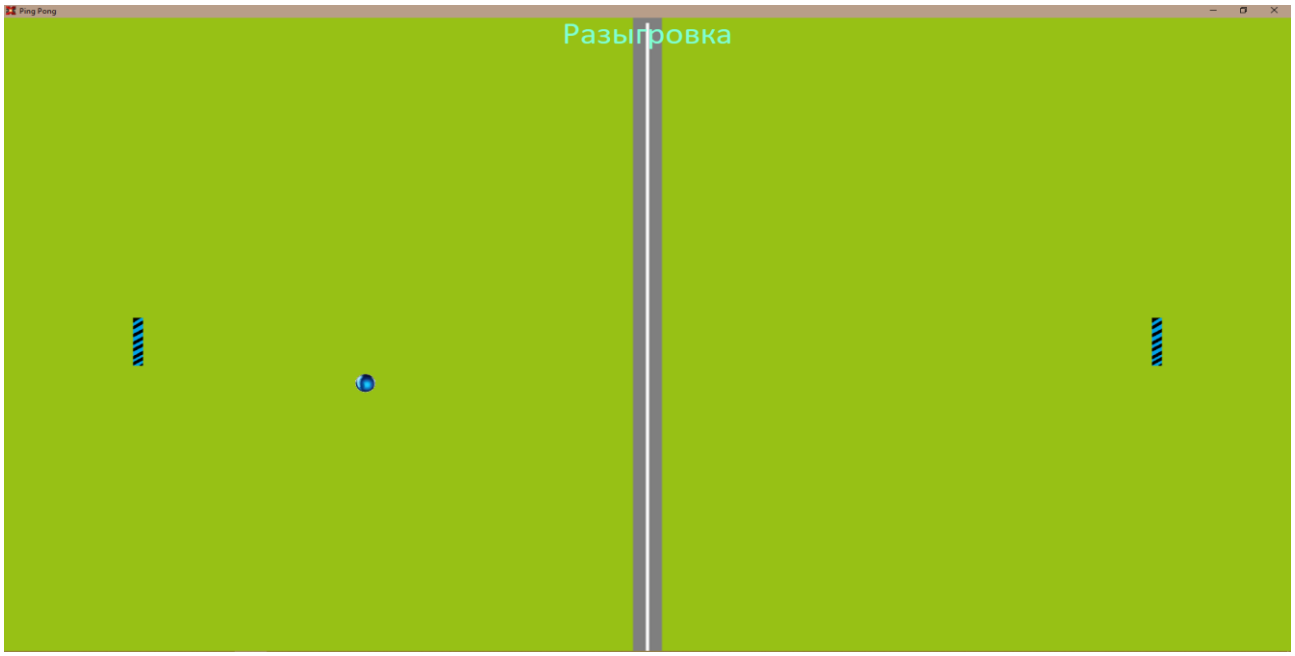

Рисунок 3.3 – Игровое поле с разыгровкой

На рисунке 3.4 показано игровое поле приложения с бонусом, которое уже прошло стадию разыгровки подач.

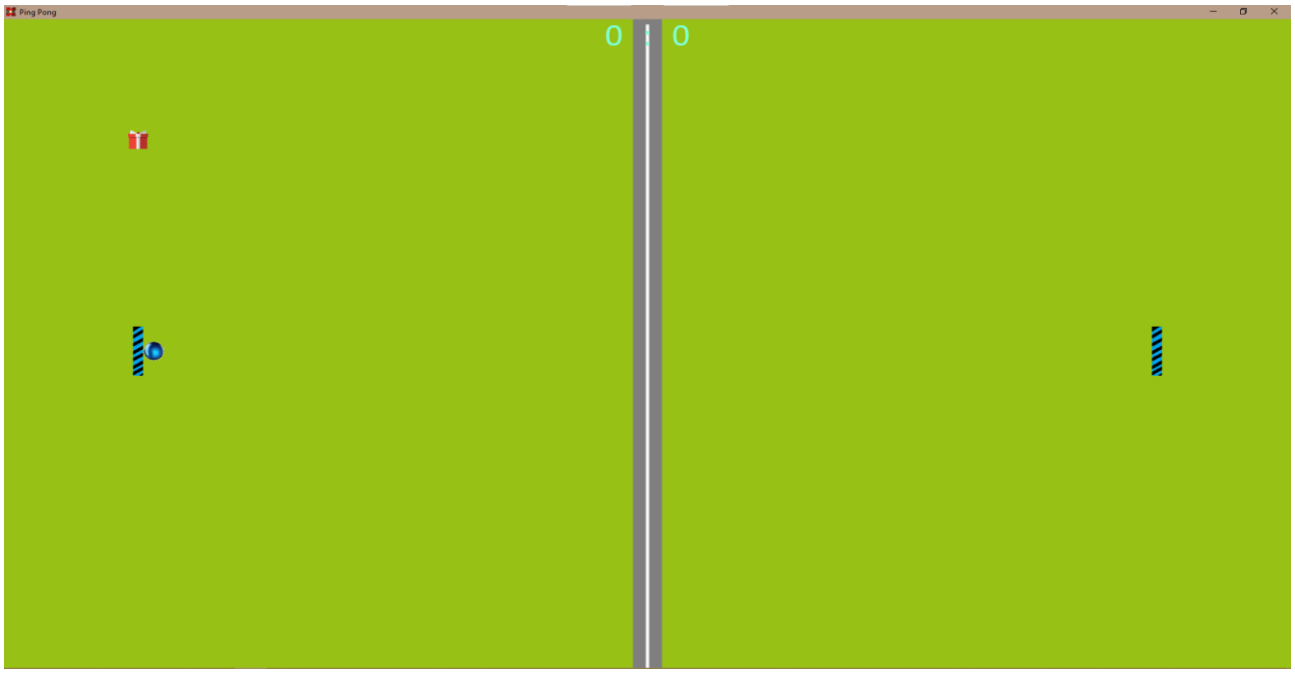

Рисунок 3.4 – Игровое поле со счётом и бонусом

На рисунке 3.5 изображено окончание игры, с выводом победителя и кнопки для начала новой игры.

Рисунок 3.5 – Окончание игры

На игровом поле отображаются два игрока и мяч, а также разыгровка вверху экрана. Игроки могут управлять ракеткой по вертикали игрового поля, задействую клавиши передвижения: *W, A, S, D* – для перового игрока; *Up, Left, Down, Right* – для второго. Клавиша *Space* служит для подачи для обоих игроков. После разыгровки подач, вместо надписи разыгровка в верхней части экрана появится счёт 0:0 и начинают появляться бонусы. При поднятии бонусов игроками, меняются их характеристики, следовательно их спрайт может растягиваться и сжиматься. При счёте 7:0 или, когда у одного из игроков наберётся 11 очков и разница будет больше двух, игра заканчивается. В конце игры на экран выводится надпись с оповещением победителя и кнопкой, при нажатии на которую, перезагрузится окно игрового приложения, и игра начнётся по новой.

Графический интерфейс и все ресурсы игры приведены в приложении Б.

Инструкции по эксплуатации приведены в приложении С.

ЗАКЛЮЧЕНИЕ

Основные стеки графических технологий и языки программирования, наиболее подходящие для решения поставленной задачи, были выбраны и описаны на ранней стадии разработки игрового приложения. Информация, предоставленная для сравнения, была отобрана на основе отобранных литературных источников. При описании графических стеков и языков программирования были рассмотрены преимущества и недостатки, благодаря чему была сделана оценка использования графических *API* и языков программирования для разработки игровых приложений и сделаны выводы.

После того, как сравнение показало выбор графического стека и языка программирования для разработки игрового приложения, была описана общая архитектура приложения, а также разработана иерархическая структура классов различных архитектурных компонентов, чтобы показать взаимосвязь различных классов.

С разработанной архитектурой приложения и иерархией классов было написано игровое приложение, состоящее из разработанной структуры и классов, выполняющих поставленную задачу. Для проверки правильности работы приложения и выполнения всех функциональных частей были проведены соответствующие тесты, которые подтвердили правильность работы приложения.

Данное игровое приложение является уникальным в своем жанре. На рынке игровых приложений редко можно встретить игры, подобные этой, особенно те, которые похожи по механике и общей идее игры. Это приложение может стать отличным способом сбежать от однотипных сложных проектов с различными тяжелыми механиками. Разнообразие карт не только поддерживает интерес к игре, но и дает игрокам доступ к целостной комбинации приобретаемых шахт, позволяя сделать собственный выбор и разнообразить игровой процесс. Реализация на основе спрайтов визуально потрясающе передает атмосферу игры, а добавление фоновой музыки и звуковых эффектов по мере действий игрока и реакции окружения усиливает погружение в захватывающий игровой процесс.

Курсовая работа выполнена самостоятельно, проверена в системе «Антиплагиат». Процент оригинальности составляет 98,84%. Цитирования обозначены ссылками на публикации, указанные в «Списке использованных источников».

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Муххамад, М. М. *OpenGL Development Cookbook* / М. М. Муххамад. – Великобритания, 2013. – 326 с
2. Шерод, А. *Beginning DirectX* 11 *Game Programming* / А. Шерод, В. Джонс. – *Detroit*: *Cengage Learning*, 2011. – 385 с
3. Боресков, А. В. Программирование компьютерной графики. Современный *OpenGL* / А. В. Боресков. – Москва: ДМК Пресс, 2019. – 372 с
4. Рихтер, Дж. *CLR via C#*. Программирование на платформе *Microsoft .NET Framework* 4.5 на языке *C#*. 4-е изд. — СПб.: Питер, 2013. — 896 с.
5. Эккель, Б. Философия *Java*. 4-е издание / Б. Эккель. – СПб.: Питер, 2006. – 638 с
6. Шилдт, Г.С. *C#*: Учебный курс / Г.С. Шилдт. – СПб.: Питер, 2002. – 512с.