

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.....	4
1 Технологии разработки игрового приложения « <i>Ping-Pong</i> ».....	5
1.1 Сравнение графических стеков технологий.....	5
1.2 Сравнение языков программирования.....	11
1.3 Перечень используемых технологий	14
2 Проектирование архитектуры и структуры приложения « <i>Ping-Pong</i> »	15
2.1 Описание игры	15
2.2 Структура игрового приложения	16
2.3 Описание классов игровых объектов.....	17
2.4 Описание классов <i>DirectX</i>	22
2.5 Описание логики взаимодействия классов	23
3 Тестирование и апробация игрового приложения « <i>Ping-Pong</i> »	25
3.1 Функционал классов игровых объектов	25
3.2 Функционал классов <i>DirectX</i>	29
3.3 Функционал служебных классов.....	30
3.4 Тестирование приложения	31
3.5 Апробация приложения.....	32
Заключение	35
Список использованных источников	36
Приложение А	37
Приложение Б.....	73
Приложение В.....	74
Приложение Г	75
Приложение Д.....	76
Приложение Ж.....	77

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, 3D 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 множества 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. Далее 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*.

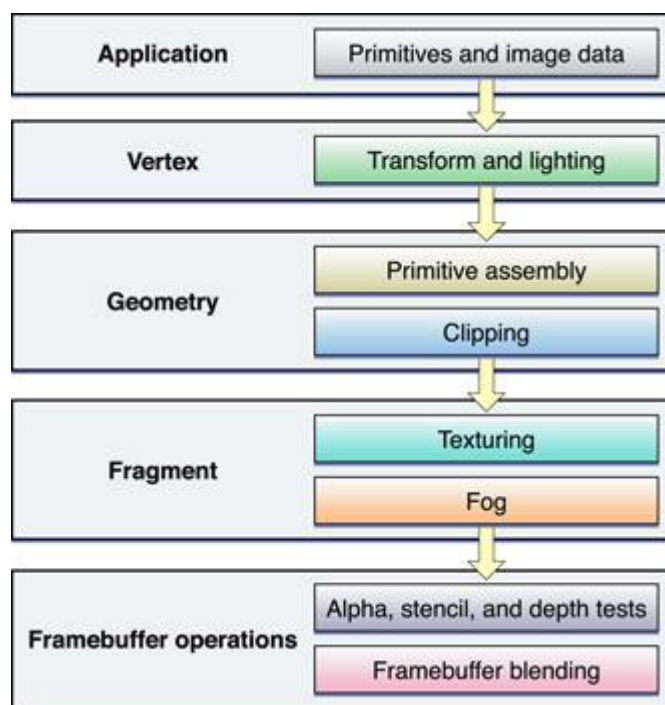


Рисунок 1.1 – Конвейер стандартных функций *OpenGL*

DirectX – это графический стек технологий, разработанный компанией *Microsoft* для операционной системы *Windows*. *DirectX* включает в себя несколько компонентов, таких как *Direct3D* для рендеринга 3D-графики, *Direct2D* для 2D-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* – высокоуровневый 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;

- *Direct3D* – прикладной интерфейс, позволяющий производить работу с трехмерной графикой;

- *DirectDraw* / *Direct2D* – 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*. *Direct3D* позиционировался как замена *OpenGL* в игровой сфере. В середине 1996 года был выпущен *DirectX 2.0*. Добавили интерфейс *Direct3D*, 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 3D-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*.

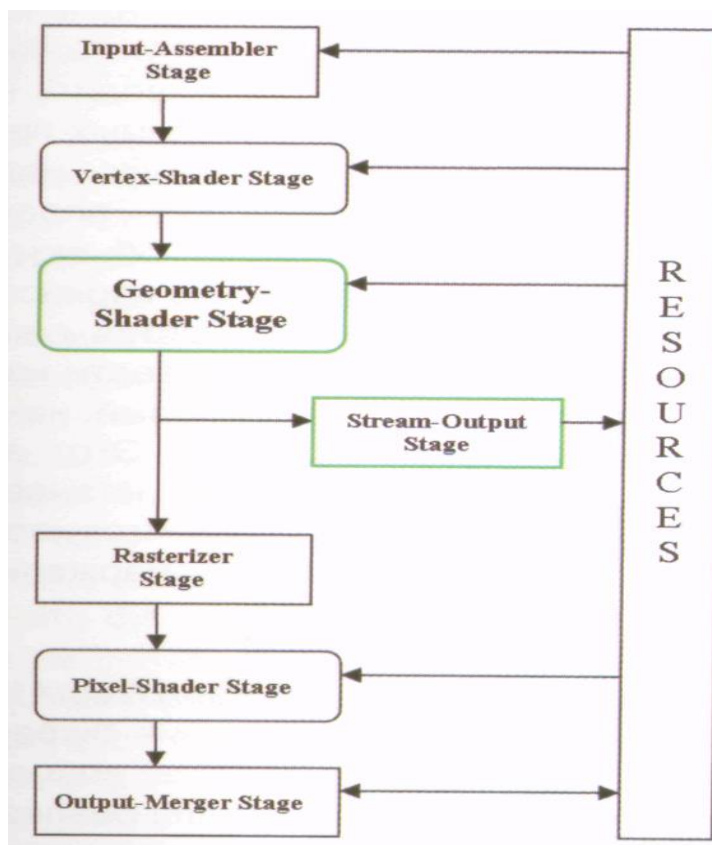


Рисунок 1.2 – Графический конвейер *DirectX 10*

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 3D-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 *2D* объектов и реализовать спрайтовую графику и отобразить *2D* 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#*. Далее 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 предоставляет 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 включает 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.

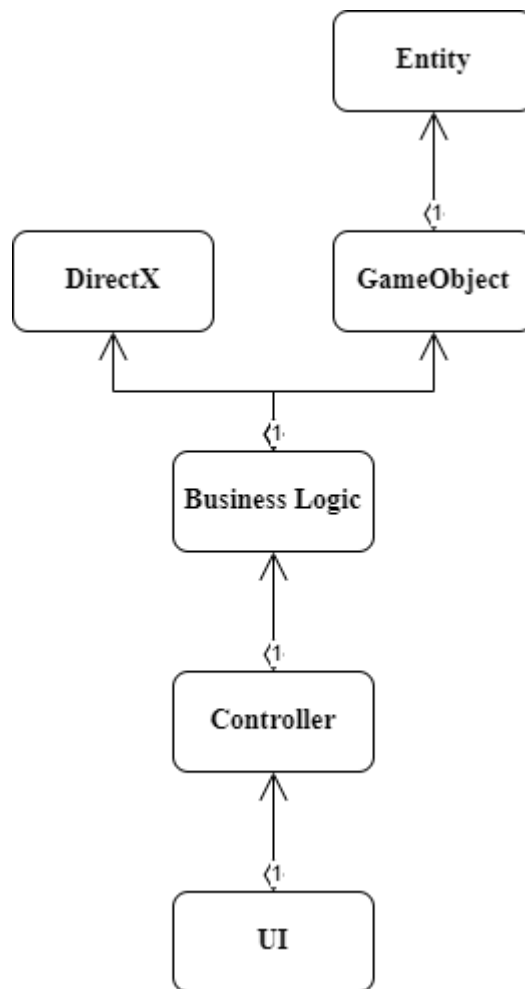


Рисунок 2.1 – Архитектура игрового приложения

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.

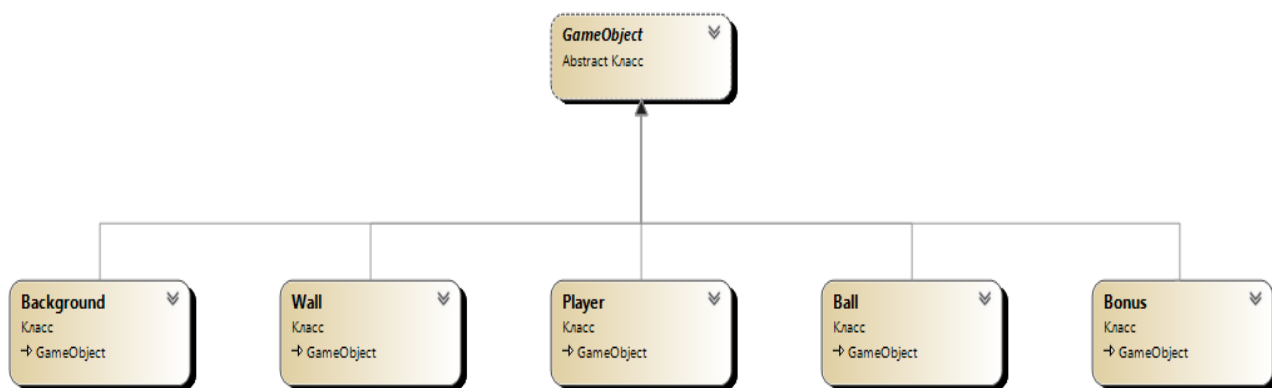


Рисунок 2.2 – Иерархия классов *Entity*

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 выбранного типа бонуса.

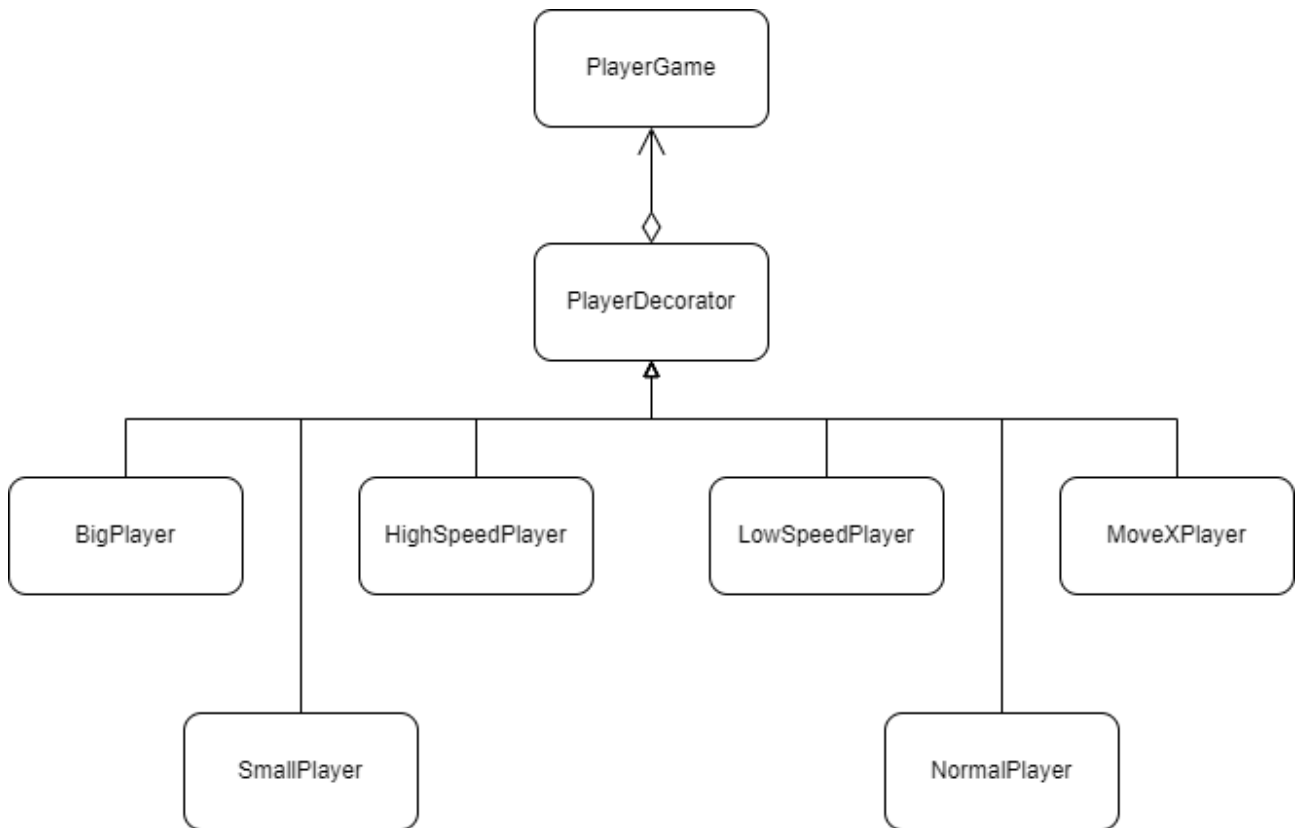


Рисунок 2.3 – Иерархия классов декоратора

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.

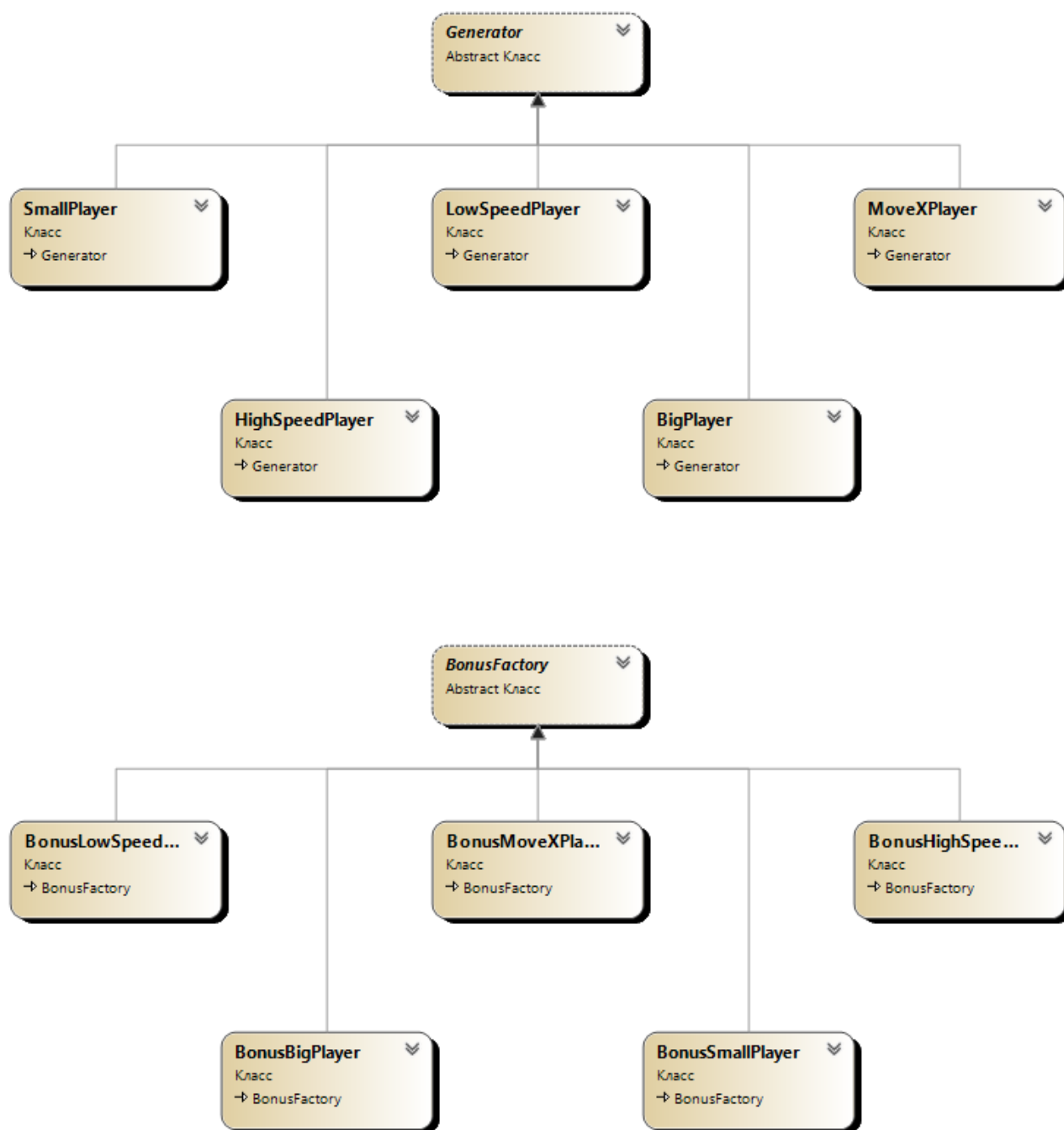


Рисунок 2.4 – Иерархия фабричного метода

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, реализовав 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.

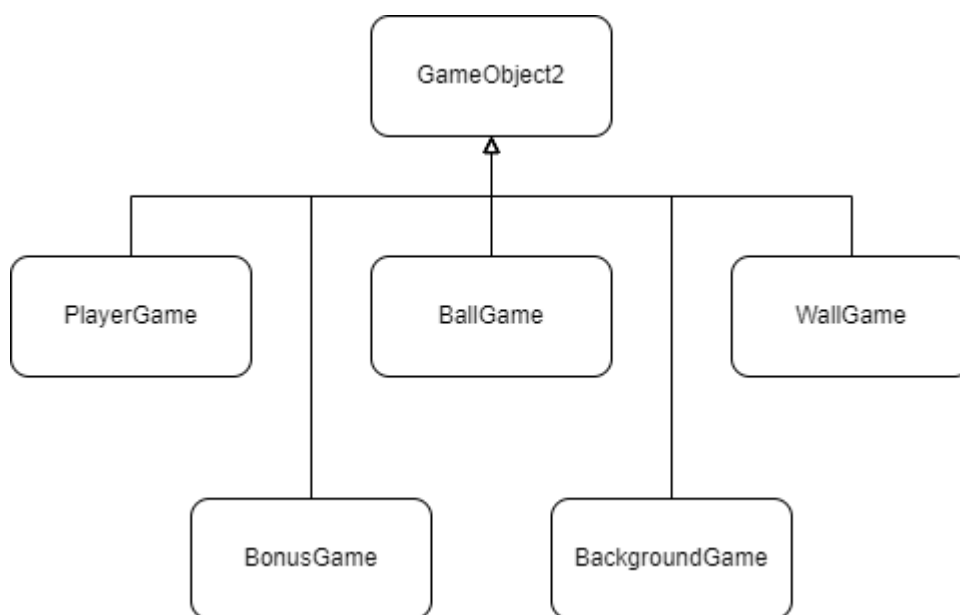


Рисунок 2.5 – Иерархия классов *GameObject*

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

- фабрики для 2D-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 2D-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 подачи. 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 *Vector2* 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 позиционирование 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 *Vector2* и хранит в себе начальное положение игрока на игровом поле. Поле *_giftLimit* хранит в себе логическое значение и определяет может ли игрок получить 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 *Vector2 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 *GameObject2*;

– *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 *GameObject2*;

– *MoveXPlayer* – класс, который наследуется от абстрактного класса декоратора. Он устанавливает значение публичного поля *Gorizont* для игрока *true*, тем самым разрешает 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 *GameObject2*, устанавливает значение публичного поля *Gorizont* 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 *DX2D*. Поля *_height* и *_width* имеют вещественный тип одиночной точности и определяют высоту и ширину спрайта соответственно. Поле *_center* имеет тип *Vector2*, 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 продемонстрировать 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 *GameObject2*. 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 *DX2D*, тем самым определяется отображаемая картинка 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 *GameObject2* 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 2D-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 2D-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* содержит логику 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.

Тестирование	Длительн...	Г
▲ ✓ Tests (7)	141 мс	
▲ ✓ Tests (7)	141 мс	
▲ ✓ BallTests (1)	< 1 мс	
✓ MoveTest	< 1 мс	
▲ ✓ PlayerTests (6)	141 мс	
✓ GiftOffTest	< 1 мс	
✓ GiftOnTest	< 1 мс	
✓ GoalTest	140 мс	
✓ MoveTest	1 мс	
✓ ReturnPositionTest	< 1 мс	
✓ ServeTest	< 1 мс	

Рисунок 3.1 – Модульные тесты

На рисунке 3.2 показана сводка по группе выполненных тестов.

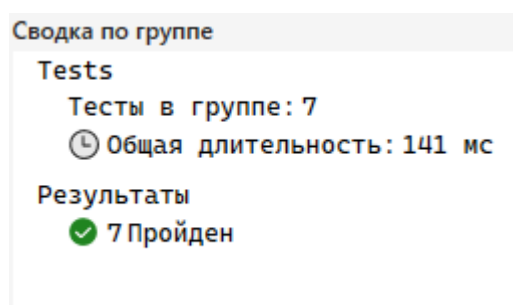


Рисунок 3.2 – Модульные тесты

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.

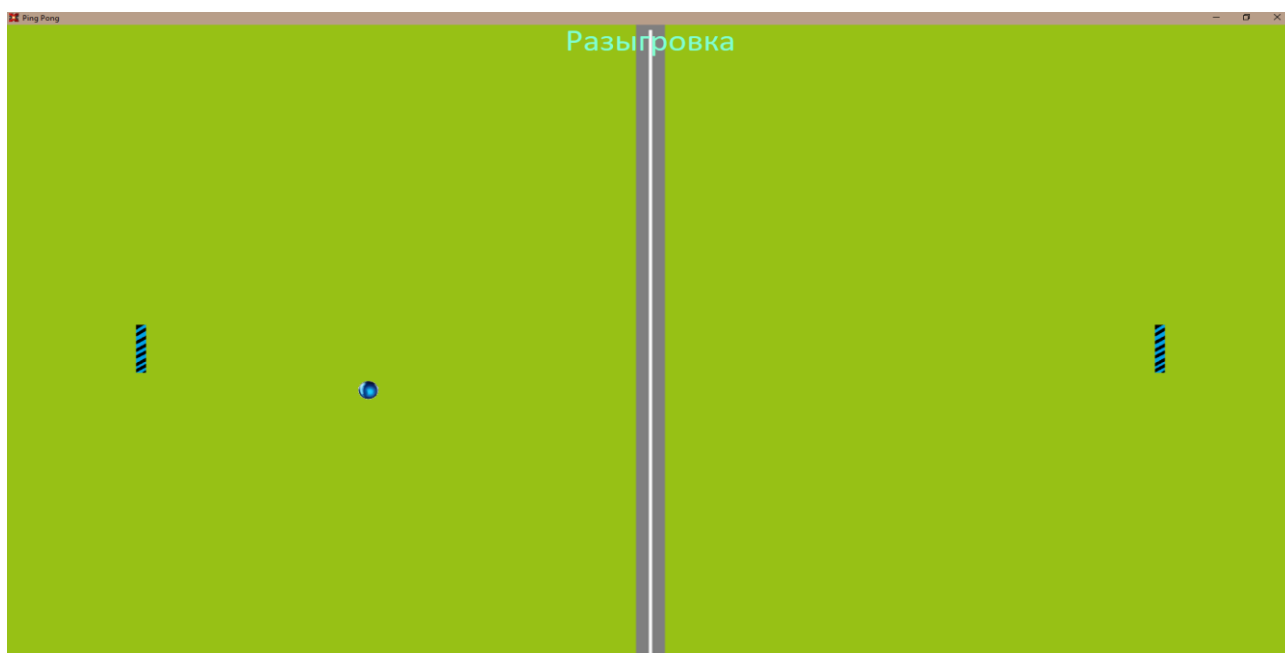


Рисунок 3.3 – Игровое поле с разыгрывкой

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

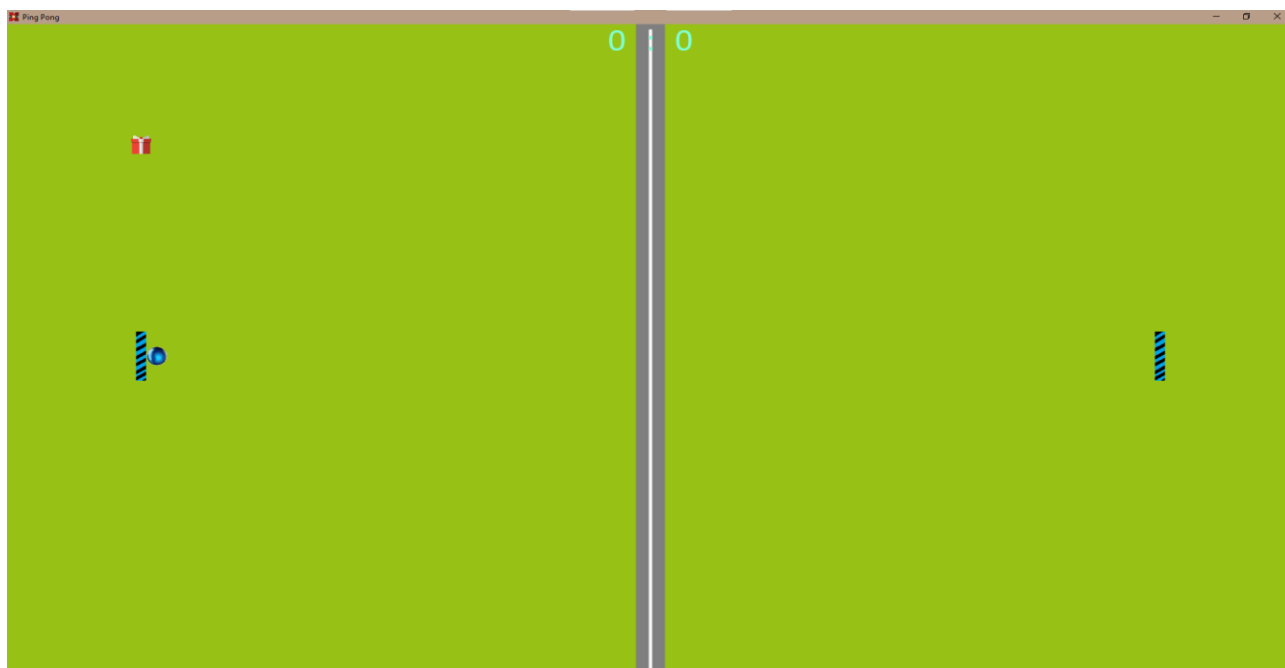


Рисунок 3.4 – Игровое поле со счётом и бонусом

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 разыгрывка вверху экрана. Игроки могут управлять ракеткой по вертикали игрового поля, задействуя клавиши передвижения: *W*, *A*, *S*, *D* – для первого игрока; *Up*, *Left*, *Down*, *Right* – для второго. Клавиша *Space* служит для подачи для обоих игроков. После разыгрывки подач, вместо надписи разыгрывка 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.

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг игрового приложения «*Ping-Pong*»

Листинг программного класса *Drawer.cs*:

```
using _PingPongLibrary._GameObject;
using SharpDX;
using SharpDX.Direct2D1;
using System;

namespace PingPongLibrary.DirectX
{
    /// <summary>
    /// Класс, реализующий отрисовку игровых объектов
    /// </summary>
    public class Drawer
    {
        private DX2D _dx2d;
        private Vector2 _translation;
        private Matrix3x2 _matrix;
        Matrix3x2 last;

        public Drawer(DX2D dx2d)
        {
            _dx2d = dx2d;
        }
        /// <summary>
        /// Метод, реализующий отображение статических игровых объектов
        /// </summary>
        /// <param name="gameObject">Объект класса GameObject2</param>
        /// <exception cref="Exception">Обработка исключения нулевого индекса</exception>
        public void DrawObject(GameObject2 gameObject)
        {
            if (_dx2d.Bitmaps[gameObject.ActiveSprite.BitmapIndex] == null) throw new Exception("bitmap was null");
            Bitmap bmp = _dx2d.Bitmaps[gameObject.ActiveSprite.BitmapIndex];
            RectangleF rect = gameObject.Rect;
            _dx2d.RenderTarget.DrawBitmap(_dx2d.Bitmaps[gameObject.ActiveSprite.BitmapIndex],
                rect,
                1,
                SharpDX.Direct2D1.BitmapInterpolationMode.Linear);
        }
        /// <summary>
        /// Метод, реализующий отображение динамических игровых объектов
        /// </summary>
        /// <param name="gameObject">Объект класса GameObject2</param>
        public void Draw(GameObject2 gameObject)
        {
            float scale = gameObject.Scale;
            _translation.X = (gameObject.PositionOfCenter.X - gameObject.ActiveSprite.Center.X);
            _translation.Y = (gameObject.PositionOfCenter.Y - gameObject.ActiveSprite.Center.Y);
            // Порядок перемножения матриц в "прямом иксе 2 измерения" перевернут с ног на голову:
            // слева - вращение и масштаб (в произвольном порядке между собой), справа - перенос
            _matrix = Matrix3x2.Rotation(gameObject.ActiveSprite.Angle, gameObject.PositionOfRotation) *
                Matrix3x2.Scaling(scale, scale, Vector2.Zero) *
                Matrix3x2.Translation(_translation);

            // Получаем из инфраструктурного объекта "цель" отрисовки
            WindowRenderTarget r = _dx2d.RenderTarget;
        }
    }
}
```

```

last = r.Transform;
// Устанавливаем матрицу координатных преобразований
r.Transform = _matrix;

// Ресурс
int opacity = 1;
Bitmap bitmap = _dx2d.Bitmaps[gameObject.ActiveSprite.BitmapIndex];
r.DrawBitmap(bitmap, opacity, SharpDX.Direct2D1.BitmapInterpolationMode.Linear);
r.Transform = last;
}
/// <summary>
/// Метод, который отображает текущий счёт игроков
/// </summary>
/// <param name="score1">Счёт первого игрока</param>
/// <param name="score2">Счёт второго игрока</param>
public void ScoreDraw(int score1, int score2)
{
    /*_dx2d.RenderTarget.DrawRectangle(new RectangleF((_dx2d.RenderTarget.Size.Width / 2) - 90, 0, 80, 50),
_dx2d.QuanBrush);
_dx2d.RenderTarget.DrawRectangle(new RectangleF((_dx2d.RenderTarget.Size.Width / 2) - 10, 0, 20, 50),
_dx2d.QuanBrush);
_dx2d.RenderTarget.DrawRectangle(new RectangleF((_dx2d.RenderTarget.Size.Width / 2) + 10, 0, 80, 50),
_dx2d.QuanBrush);*/

    _dx2d.RenderTarget.DrawText($"{score1}", _dx2d.TextFormatStats, new RectangleF((_dx2d.RenderTar-
get.Size.Width / 2) - 90, 0, 80, 50), _dx2d.QuanBrush);
    _dx2d.RenderTarget.DrawText(":", _dx2d.TextFormatStats, new RectangleF((_dx2d.RenderTarget.Size.Width /
2) - 10, 0, 20, 50), _dx2d.QuanBrush);
    _dx2d.RenderTarget.DrawText($"{score2}", _dx2d.TextFormatStats, new RectangleF((_dx2d.RenderTar-
get.Size.Width / 2) + 10, 0, 80, 50), _dx2d.QuanBrush);
}
/// <summary>
/// Метод, который отображает результат игры
/// </summary>
/// <param name="flag">Победа первого игрока</param>
public void ResultDraw(bool flag)
{
    /*_dx2d.RenderTarget.DrawRectangle(new RectangleF((_dx2d.RenderTarget.Size.Width / 2) - 90, 0, 80, 50),
_dx2d.QuanBrush);
_dx2d.RenderTarget.DrawRectangle(new RectangleF((_dx2d.RenderTarget.Size.Width / 2) - 10, 0, 20, 50),
_dx2d.QuanBrush);
_dx2d.RenderTarget.DrawRectangle(new RectangleF((_dx2d.RenderTarget.Size.Width / 2) + 10, 0, 80, 50),
_dx2d.QuanBrush);*/
    if (flag)
    {
        _dx2d.RenderTarget.DrawText($"Player 1 is won", _dx2d.TextFormatEndGame, new Rectan-
gleF((_dx2d.RenderTarget.Size.Width / 2) - 750, (_dx2d.RenderTarget.Size.Height / 2) - 150, 1500, 150), _dx2d.Quan-
Brush);
    }
    else
    {
        _dx2d.RenderTarget.DrawText($"Player 2 is won", _dx2d.TextFormatEndGame, new Rectan-
gleF((_dx2d.RenderTarget.Size.Width / 2) - 750, (_dx2d.RenderTarget.Size.Height / 2) - 150, 1500, 150), _dx2d.Quan-
Brush);
    }
    _dx2d.RenderTarget.DrawText($"~Press ESC to start new game~", _dx2d.TextFormatStartNewGame, new
RectangleF((_dx2d.RenderTarget.Size.Width / 2) - 750, (_dx2d.RenderTarget.Size.Height / 2) + 35, 1500, 75),
_dx2d.RedBrush);
}
/// <summary>
/// Метод, который отображает разыгрывку перед началом игры
/// </summary>

```

```

        public void ServeDraw()
        {
            //_dx2d.RenderTarget.DrawRectangle(new RectangleF((_dx2d.RenderTarget.Size.Width / 2) - 150, 0, 300, 50),
            _dx2d.QuanBrush);

            _dx2d.RenderTarget.DrawText($"Разыгрровка", _dx2d.TextFormatStats, new RectangleF((_dx2d.RenderTar-
            get.Size.Width / 2) - 150, 0, 300, 50), _dx2d.QuanBrush);
        }
    }
}

```

Листинг программного класса *DX2D.cs*:

```

using SharpDX;
using SharpDX.Direct2D1;
using SharpDX.DirectWrite;
using SharpDX.WIC;
using SharpDX.Windows;
using System.Collections.Generic;

namespace PingPongLibrary.DirectX
{
    /// <summary>
    /// Класс, реализующий основную логику DirectX
    /// </summary>
    public class DX2D
    {
        private SharpDX.Direct2D1.Factory _factory;
        /// <summary>
        /// Фабрика для создания 2D объектов
        /// </summary>
        public SharpDX.Direct2D1.Factory Factory { get => _factory; }

        private SharpDX.DirectWrite.Factory _writeFactory;
        /// <summary>
        /// Фабрика для работы с текстом
        /// </summary>
        public SharpDX.DirectWrite.Factory WriteFactory { get => _writeFactory; }

        private WindowRenderTarget _renderTarget;
        /// <summary>
        /// "Цель" отрисовки
        /// </summary>
        public WindowRenderTarget RenderTarget { get => _renderTarget; }

        private ImagingFactory _imagingFactory;
        /// <summary>
        /// Фабрика для работы с изображениями (WIC = Windows Imaging Component)
        /// </summary>
        public ImagingFactory ImagingFactory { get => _imagingFactory; }

        private TextFormat _textFormatStats;
        /// <summary>
        /// Формат текста для вывода счёта и разыгрровки
        /// </summary>
        public TextFormat TextFormatStats { get => _textFormatStats; }

        private TextFormat _textFormatEndGame;
        /// <summary>
        /// Формат текста для вывода победителя в конце игры
        /// </summary>
    }
}

```

```

public TextFormat TextFormatEndGame { get => _textFormatEndGame; }

private TextFormat _textFormatStartNewGame;
/// <summary>
/// Формат текста для вывода сообщения: какую кнопку нужно нажать, чтобы начать новую игру
/// </summary>
public TextFormat TextFormatStartNewGame { get => _textFormatStartNewGame; }
// Четыре кисти для текста
private Brush _redBrush;
/// <summary>
/// Кисть для сообщения: какую кнопку нужно нажать, чтобы начать новую игру
/// </summary>
public Brush RedBrush { get => _redBrush; }
private Brush _quanBrush;
/// <summary>
/// Кисть для отрисовки счёта, розыгрыша и победителя в конце игры
/// </summary>
public Brush QuanBrush { get => _quanBrush; }

// Коллекция спрайтов
// Есть конечно атлас спрайтов SpriteBatch
// https://docs.microsoft.com/en-us/windows/win32/api/d2d1\_3/nn-d2d1\_3-id2d1spritebatch
// но при его использовании не поддерживается антиалиасинг, можно использовать только
AntialiasMode.Aliased
// Источник: https://issue.life/questions/55093299
// Таким образом он отлично подойдет, если спрайты вращать не надо
private List<SharpDX.Direct2D1.Bitmap> _bitmaps;
public List<SharpDX.Direct2D1.Bitmap> Bitmaps { get => _bitmaps; }

// В конструкторе создаем все объекты
public DX2D(RenderForm form)
{
    // Создание фабрик для 2D объектов и текста
    _factory = new SharpDX.Direct2D1.Factory();
    _writeFactory = new SharpDX.DirectWrite.Factory();

    // Инициализация "прямой рисовалки":
    // Свойства отрисовщика
    RenderTargetProperties renderProp = new RenderTargetProperties()
    {
        DpiX = 0,
        DpiY = 0,
        MinLevel = FeatureLevel.Level_10,
        PixelFormat = new SharpDX.Direct2D1.PixelFormat(SharpDX.DXGI.Format.B8G8R8A8_UNorm, AlphaMode.Premultiplied),
        Type = RenderTargetType.Hardware,
        Usage = RenderTargetUsage.None
    };
    // Свойства отрисовщика, связанные с окном (дискриптор окна, размер в пикселях и способ представле-
    ния результирующего изображения)
    HwndRenderTargetProperties winProp = new HwndRenderTargetProperties()
    {
        Hwnd = form.Handle,
        PixelSize = new Size2(form.ClientSize.Width, form.ClientSize.Height),
        PresentOptions = PresentOptions.None // Immediately // None - vSync
    };
    // Создание "цели" и задание свойств сглаживания
    _renderTarget = new WindowRenderTarget(_factory, renderProp, winProp);
    _renderTarget.AntialiasMode = AntialiasMode.PerPrimitive;
    _renderTarget.TextAntialiasMode = SharpDX.Direct2D1.TextAntialiasMode.Cleartype;

    // Создание "картиночной" фабрики

```

```

_imagingFactory = new ImagingFactory();

// Задание форматов текста
_textFormatStats = new TextFormat(_writeFactory, "Calibri", 52);
_textFormatStats.ParagraphAlignment = ParagraphAlignment.Center;
_textFormatStats.TextAlignment = TextAlignment.Center;
_textFormatEndGame = new TextFormat(_writeFactory, "Ink Free", FontWeight.Bold, FontStyle.Italic, 170);
_textFormatEndGame.ParagraphAlignment = ParagraphAlignment.Center;
_textFormatEndGame.TextAlignment = TextAlignment.Center;
_textFormatStartNewGame = new TextFormat(_writeFactory, "Ink Free", FontWeight.Normal, FontStyle.Italic,
50);
_textFormatStartNewGame.ParagraphAlignment = ParagraphAlignment.Center;
_textFormatStartNewGame.TextAlignment = TextAlignment.Center;
// Создание кистей для текста
_redBrush = new SolidColorBrush(_renderTarget, Color.Red);
_quanBrush = new SolidColorBrush(_renderTarget, Color.Aquamarine);
}

/// <summary>
/// Метод для загрузки изображения в коллекцию
/// </summary>
/// <param name="imageFileName">Путь к изображению</param>
/// <returns>Индекс изображения в коллекции</returns>
public int LoadBitmap(string imageFileName)
{
    // Чтение изображения из bmp
    // System.Drawing.Bitmap bitmap = new System.Drawing.Bitmap("image.bmp"); - для слабаков

    // Декодер формата
    BitmapDecoder decoder = new BitmapDecoder(_imagingFactory, imageFileName, DecodeOp-
tions.CacheOnDemand);
    // Берем первый фрейм
    BitmapFrameDecode frame = decoder.GetFrame(0);
    // Также нужен конвертер формата
    FormatConverter converter = new FormatConverter(_imagingFactory);
    converter.Initialize(frame, SharpDX.WIC.PixelFormat.Format32bppPRGBA, BitmapDitherType.Ordered4x4,
null, 0.0, BitmapPaletteType.Custom);
    // Вот теперь можно и bitmap
    SharpDX.Direct2D1.Bitmap bitmap = SharpDX.Direct2D1.Bitmap.FromWicBitmap(_renderTarget, converter);

    // Освобождаем неуправляемые ресурсы
    Utilities.Dispose(ref converter);
    Utilities.Dispose(ref frame);
    Utilities.Dispose(ref decoder);

    // Добавляем изображение в коллекцию
    if (_bitmaps == null) _bitmaps = new List<SharpDX.Direct2D1.Bitmap>(4);
    _bitmaps.Add(bitmap);
    return _bitmaps.Count - 1;
}

/// <summary>
/// Освобождает неуправляемые ресурсы
/// </summary>
public void Dispose()
{
    for (int i = _bitmaps.Count - 1; i >= 0; i--) // foreach здесь не пойдет, поскольку итератор нельзя передавать
как ref
    {
        SharpDX.Direct2D1.Bitmap bitmap = _bitmaps[i];
        _bitmaps.RemoveAt(i);
        Utilities.Dispose(ref bitmap);
    }
}

```



```

    }
    Utilities.Dispose(ref _quanBrush);
    Utilities.Dispose(ref _redBrush);
    Utilities.Dispose(ref _textFormatEndGame);
    Utilities.Dispose(ref _textFormatStartNewGame);
    Utilities.Dispose(ref _textFormatStats);
    Utilities.Dispose(ref _imagingFactory);
    Utilities.Dispose(ref _renderTarget);
    Utilities.Dispose(ref _factory);
}
}
}

```

Листинг программного класса *InputManager.cs*:

```

using SharpDX;
using SharpDX.DirectInput;
using SharpDX.Windows;

namespace PingPongLibrary.DirectX
{
    /// <summary>
    /// Реализует подключение клавиатуры и обновление её состояния
    /// </summary>
    public class InputManager
    {
        // Экземпляр объекта "прямого ввода"
        private DirectInput _directInput;

        // Поля и свойства, связанные с клавиатурой
        private Keyboard _keyboard;
        private KeyboardState _keyboardState;
        /// <summary>
        /// Определяет состояние операции с клавиатурой
        /// </summary>
        public KeyboardState KeyboardState { get => _keyboardState; }
        private bool _keyboardUpdated;
        /// <summary>
        /// Определяет обновление состояния клавиатуры
        /// </summary>
        public bool KeyboardUpdated { get => _keyboardUpdated; }
        private bool _keyboardAcquired;

        // В конструкторе создаем все объекты и пробуем получить доступ к клавиатуре
        public InputManager(RenderForm renderForm)
        {
            _directInput = new DirectInput();

            _keyboard = new Keyboard(_directInput);
            _keyboard.Properties.BufferSize = 16;
            _keyboard.SetCooperativeLevel(renderForm.Handle, CooperativeLevel.Foreground | CooperativeLevel.NonEx-
clusive);
            AcquireKeyboard();
            _keyboardState = new KeyboardState();
        }

        /// <summary>
        /// Получение доступа к клавиатуре
        /// </summary>
        private void AcquireKeyboard()
        {

```

```

        try
        {
            _keyboard.Acquire();
            _keyboardAcquired = true;
        }
        catch (SharpDXException e)
        {
            if (e.ResultCode.Failure)
                _keyboardAcquired = false;
        }
    }

    /// <summary>
    /// Обновление состояния клавиатуры
    /// </summary>
    public void UpdateKeyboardState()
    {
        // Если доступ не был получен, пробуем здесь
        if (!_keyboardAcquired) AcquireKeyboard();

        // Пробуем обновить состояние
        ResultDescriptor resultCode = ResultCode.Ok;
        try
        {
            _keyboard.GetCurrentState(ref _keyboardState);
            // Успех
            _keyboardUpdated = true;
        }
        catch (SharpDXException e)
        {
            resultCode = e.Descriptor;
            // Отказ
            _keyboardUpdated = false;
        }

        // В большинстве случаев отказ из-за потери фокуса ввода
        // Устанавливаем соответствующий флаг, чтобы в следующем кадре попытаться получить доступ
        if (resultCode == ResultCode.InputLost || resultCode == ResultCode.NotAcquired)
            _keyboardAcquired = false;
    }

    /// <summary>
    /// Освобождение неуправляемых ресурсов
    /// </summary>
    public void Dispose()
    {
        Utilities.Dispose(ref _keyboard);
        Utilities.Dispose(ref _directInput);
    }
}

```

Листинг программного класса *TimeHelper.cs*:

```

using System;
using System.Diagnostics;

namespace PingPongLibrary.DirectX
{
    /// <summary>
    /// Предназначен для работы со временем

```

```

/// </summary>
public class TimeHelper
{
    // Таймер
    private Stopwatch _watch;

    // Текущее время в секундах
    private float _time;
    /// <summary>
    /// Служит для подсчета времени
    /// </summary>
    public float Time { get => _time; }

    // В конструкторе создаем экземпляр таймера и вызываем метод Reset
    public TimeHelper()
    {
        _watch = new Stopwatch();
        Reset();
    }

    /// <summary>
    /// Рассчитывает значение поля _time
    /// </summary>
    public void Update()
    {
        // Обновление подсчитываемых значений; Должен вызываться в начале каждого кадра
        // Текущее значение счетчика тиков
        long ticks = _watch.Elapsed.Ticks;
        // Вычисляем текущее время и интервал между текущим и прошлым кадрами
        _time = (float)ticks / TimeSpan.TicksPerSecond;
    }

    /// <summary>
    /// Обнуляет значение поля _watch и запускает счетчик заново
    /// </summary>
    public void Reset()
    {
        _watch.Reset();
        _watch.Start();
    }
}

```

Листинг программного класса *BigPlayer.cs*:

```

using _PingPongLibrary._GameObject;

namespace PingPongLibrary.Entity.PlayerDecorator
{
    /// <summary>
    /// Класс, реализующий увеличение ракетки
    /// </summary>
    public class BigPlayer : PlayerDecorator
    {
        public BigPlayer(PlayerGame player) : base(player)
        {
            player.Resize(new SharpDX.Size2((int)player.ActiveSprite.Width, (int)player.ActiveSprite.Heigth + 20));
        }
    }
}

```

Листинг программного класса *HighSpeedPlayer.cs*:

```
using _PingPongLibrary._GameObject;

namespace PingPongLibrary.Entity.PlayerDecorator
{
    /// <summary>
    /// Класс, реализующий увеличение скорости ракетки
    /// </summary>
    public class HighSpeedPlayer : PlayerDecorator
    {
        public HighSpeedPlayer(PlayerGame player) : base(player)
        {
            player.Player.Speed = player.Player.Speed * 1.2f;
        }
    }
}
```

Листинг программного класса *LowSpeedPlayer.cs*:

```
using _PingPongLibrary._GameObject;

namespace PingPongLibrary.Entity.PlayerDecorator
{
    /// <summary>
    /// Класс, реализующий уменьшение скорости ракетки
    /// </summary>
    public class LowSpeedPlayer : PlayerDecorator
    {
        public LowSpeedPlayer(PlayerGame player) : base(player)
        {
            player.Player.Speed = player.Player.Speed * 0.8f;
        }
    }
}
```

Листинг программного класса *MoveXPlayer.cs*:

```
using _PingPongLibrary._GameObject;

namespace PingPongLibrary.Entity.PlayerDecorator
{
    /// <summary>
    /// Класс, реализующий передвижение ракетки по оси абсцисс
    /// </summary>
    public class MoveXPlayer : PlayerDecorator
    {
        public MoveXPlayer(PlayerGame player) : base(player)
        {
            player.Player.Gorisont = true;
        }
    }
}
```

Листинг программного класса *NormalPlayer.cs*:

```

using _PingPongLibrary._GameObject;

namespace PingPongLibrary.Entity.PlayerDecorator
{
    /// <summary>
    /// Класс, реализующий возврат игрока в первоначальное состояние
    /// </summary>
    public class NormalPlayer : PlayerDecorator
    {
        public NormalPlayer(PlayerGame player) : base(player)
        {
            player.NormalSize();
            player.Player.Speed = 1;
            player.Player.Gorisont = false;
            player.Player.GiftOff();
            player.Time = 0;
        }
    }
}

```

Листинг программного класса *PlayerDecorator.cs*:

```

using _PingPongLibrary._GameObject;
using System.Collections.Generic;

namespace PingPongLibrary.Entity.PlayerDecorator
{
    /// <summary>
    /// Абстрактный класс, реализующий изменение ракетки
    /// </summary>
    public abstract class PlayerDecorator : PlayerGame
    {
        protected PlayerDecorator(PlayerGame player) : base(player.Player, new List<Sprite>() { player.ActiveSprite })
        {
        }
    }
}

```

Листинг программного класса *SmallPlayer.cs*:

```

using _PingPongLibrary._GameObject;

namespace PingPongLibrary.Entity.PlayerDecorator
{
    /// <summary>
    /// Класс, реализующий уменьшение ракетки
    /// </summary>
    public class SmallPlayer : PlayerDecorator
    {
        public SmallPlayer(PlayerGame player) : base(player)
        {
            player.Resize(new SharpDX.Size2((int)player.ActiveSprite.Width, (int)player.ActiveSprite.Heigth - 20));
        }
    }
}

```

Листинг программного класса *Background.cs*:

```

using SharpDX;

namespace PingPongLibrary.Entity
{
    /// <summary>
    /// Класс, реализующий графическое представление заднего фона
    /// </summary>
    public class Background : GameObject
    {
        public Background(Vector2 position) : base(position)
        {

        }
    }
}

```

Листинг программного класса *Ball.cs*:

```

using SharpDX;
using System;

namespace PingPongLibrary.Entity
{
    /// <summary>
    /// Класс, реализующий графическое представление шарика
    /// </summary>
    public class Ball : GameObject
    {
        protected float width { get; set; }
        protected float height { get; set; }
        /// <summary>
        /// Направление движения мячика
        /// </summary>
        public Vector2 Direction;
        private Random _rnd;

        public Ball(Vector2 position) : base(position)
        {
            width = 5;
            height = 5;
            speed = 1;
            _rnd = new Random();
            switch (_rnd.Next(2))
            {
                case 0: Direction = new Vector2(-1, 0); break;
                case 1: Direction = new Vector2(1, 0); break;
            }
        }

        /// <summary>
        /// Метод, реализующий передвижение шарика по игровому полю
        /// </summary>
        public void Move()
        {
            PositionOfCenter += Direction * speed;
        }
    }
}

```

Листинг программного класса *Bonus.cs*:

```
using SharpDX;

namespace PingPongLibrary.Entity
{
    /// <summary>
    /// Перечисление типов бонусов
    /// </summary>
    public enum TypeBonus
    {
        Big,
        HighSpeed,
        LowSpeed,
        Small,
        MoveX
    }
    /// <summary>
    /// Класс, реализующий графическое представление бонуса
    /// </summary>
    public class Bonus : GameObject
    {
        private TypeBonus _type;
        /// <summary>
        /// Тип бонуса
        /// </summary>
        public TypeBonus Type => _type;
        public Bonus(Vector2 position, TypeBonus type) : base(position)
        {
            _type = type;
        }
    }
}
```

Листинг программного класса *BonusFactory.cs*:

```
using SharpDX;

namespace PingPongLibrary.Entity
{
    /// <summary>
    /// Абстрактный класс BonusFactory
    /// </summary>
    public abstract class BonusFactory
    {
        protected Vector2 vec;
        /// <summary>
        /// Абстрактный метод
        /// </summary>
        /// <returns>Возвращает реализацию абстрактного класса Generator</returns>
        public abstract Generator GetBonus();
    }
    /// <summary>
    /// Класс, описывающий увеличенного игрока
    /// </summary>
    public class BonusBigPlayer : BonusFactory
    {
        public BonusBigPlayer(Vector2 vec) : base()
        {
            this.vec = vec;
        }
    }
}
```

```

    }
    /// <summary>
    /// Переопределённый метод
    /// </summary>
    /// <returns>Возвращает реализацию абстрактного класса Generator</returns>
    public override Generator GetBonus()
    {
        return new BigPlayer(vec);
    }
}
/// <summary>
/// Класс, описывающий уменьшенного игрока
/// </summary>
public class BonusSmallPlayer : BonusFactory
{
    public BonusSmallPlayer(Vector2 vec) : base()
    {
        this.vec = vec;
    }
    /// <summary>
    /// Переопределённый метод
    /// </summary>
    /// <returns>Возвращает реализацию абстрактного класса Generator</returns>
    public override Generator GetBonus()
    {
        return new SmallPlayer(vec);
    }
}
/// <summary>
/// Класс, описывающий игрока с увеличенной скоростью
/// </summary>
public class BonusHighSpeedPlayer : BonusFactory
{
    public BonusHighSpeedPlayer(Vector2 vec) : base()
    {
        this.vec = vec;
    }
    /// <summary>
    /// Переопределённый метод
    /// </summary>
    /// <returns>Возвращает реализацию абстрактного класса Generator</returns>
    public override Generator GetBonus()
    {
        return new HighSpeedPlayer(vec);
    }
}
/// <summary>
/// Класс, описывающий игрока с уменьшенной скоростью
/// </summary>
public class BonusLowSpeedPlayer : BonusFactory
{
    public BonusLowSpeedPlayer(Vector2 vec) : base()
    {
        this.vec = vec;
    }
    /// <summary>
    /// Переопределённый метод
    /// </summary>
    /// <returns>Возвращает реализацию абстрактного класса Generator</returns>
    public override Generator GetBonus()
    {
        return new LowSpeedPlayer(vec);
    }
}

```



```

    }
}
/// <summary>
/// Класс, описывающий движение игрока по горизонтали
/// </summary>
public class BonusMoveXPlayer : BonusFactory
{
    public BonusMoveXPlayer(Vector2 vec) : base()
    {
        this.vec = vec;
    }
    /// <summary>
    /// Переопределённый метод
    /// </summary>
    /// <returns>Возвращает реализацию абстрактного класса Generator</returns>
    public override Generator GetBonus()
    {
        return new MoveXPlayer(vec);
    }
}

/// <summary>
/// Абстрактный класс Generator
/// </summary>
public abstract class Generator
{
    protected Bonus bonus;
    /// <summary>
    /// Абстрактный метод
    /// </summary>
    /// <returns>Возвращает объект класса Bonus</returns>
    public abstract Bonus GetGenerator();
}
/// <summary>
/// Класс, описывающий увеличенного игрока
/// </summary>
public class BigPlayer : Generator
{
    public BigPlayer(Vector2 vec) : base()
    {
        bonus = new Bonus(vec, TypeBonus.Big);
    }
    /// <summary>
    /// Переопределённый метод
    /// </summary>
    /// <returns>Возвращает объект класса Bonus</returns>
    public override Bonus GetGenerator()
    {
        return bonus;
    }
}
/// <summary>
/// Класс, описывающий уменьшенного игрока
/// </summary>
public class SmallPlayer : Generator
{
    public SmallPlayer(Vector2 vec) : base()
    {
        bonus = new Bonus(vec, TypeBonus.Small);
    }
    /// <summary>
    /// Переопределённый метод

```

```

    /// </summary>
    /// <returns>Возвращает объект класса Bonus</returns>
    public override Bonus GetGenerator()
    {
        return bonus;
    }
}
/// <summary>
/// Класс, описывающий игрока с увеличенной скоростью
/// </summary>
public class HighSpeedPlayer : Generator
{
    public HighSpeedPlayer(Vector2 vec) : base()
    {
        bonus = new Bonus(vec, TypeBonus.HighSpeed);
    }
    /// <summary>
    /// Переопределённый метод
    /// </summary>
    /// <returns>Возвращает объект класса Bonus</returns>
    public override Bonus GetGenerator()
    {
        return bonus;
    }
}
/// <summary>
/// Класс, описывающий игрока с уменьшенной скоростью
/// </summary>
public class LowSpeedPlayer : Generator
{
    public LowSpeedPlayer(Vector2 vec) : base()
    {
        bonus = new Bonus(vec, TypeBonus.LowSpeed);
    }
    /// <summary>
    /// Переопределённый метод
    /// </summary>
    /// <returns>Возвращает объект класса Bonus</returns>
    public override Bonus GetGenerator()
    {
        return bonus;
    }
}
/// <summary>
/// Класс, описывающий движение игрока по горизонтали
/// </summary>
public class MoveXPlayer : Generator
{
    public MoveXPlayer(Vector2 vec) : base()
    {
        bonus = new Bonus(vec, TypeBonus.MoveX);
    }
    /// <summary>
    /// Переопределённый метод
    /// </summary>
    /// <returns>Возвращает объект класса Bonus</returns>
    public override Bonus GetGenerator()
    {
        return bonus;
    }
}
}

```

Листинг программного класса *GameObject.cs*:

```
using SharpDX;

namespace PingPongLibrary.Entity
{
    /// <summary>
    /// Абстрактный класс, для обобщённой реализации игровых объектов
    /// </summary>
    public abstract class GameObject
    {
        protected Vector2 _positionOfCenter;
        /// <summary>
        /// Поле описывает позиционирование по осям X и Y
        /// </summary>
        public Vector2 PositionOfCenter { get => _positionOfCenter; set => _positionOfCenter = value; }
        protected float speed;
        /// <summary>
        /// Поле для реализации скорости игрока и мяча
        /// </summary>
        public float Speed { get => speed; set => speed = value; }
        public GameObject(Vector2 position, int speed = 0)
        {
            _positionOfCenter = position;
            this.speed = speed;
        }
    }
}
```

Листинг программного класса *Player.cs*:

```
using SharpDX;

namespace PingPongLibrary.Entity
{
    /// <summary>
    /// Перечисление направлений движения игрока
    /// </summary>
    public enum Direction
    {
        Up,
        Down,
        Left,
        Right
    }
    /// <summary>
    /// Класс, реализующий графическое представление игрока
    /// </summary>
    public class Player : GameObject
    {
        protected float width { get; set; } // ширина ракетки игрока
        protected float height { get; set; } // высота ракетки игрока
        private int _score;
        /// <summary>
        /// Счёт игрока
        /// </summary>
        public int Score { get => _score; }
        private bool _gorisont;
        /// <summary>
        /// Определяет возможность игрока перемещаться по горизонтали, если true то может
    }
}
```

```

/// </summary>
public bool Gorisont { get => _gorisont; set => _gorisont = value; }
private Vector2 _position;
/// <summary>
/// Первоначальная позиция игрока
/// </summary>
public Vector2 Position { get => _position; }
private bool _giftLimit;
/// <summary>
/// Определяет, сможет ли игрок получить бонус, если true, то нет
/// </summary>
public bool GiftLimit { get => _giftLimit; }
private bool _turnserve;
/// <summary>
/// Определяет подачу игроков, если true то игрок подаёт
/// </summary>
public bool TurnServe { get => _turnserve; }

public Player(Vector2 position) : base(position)
{
    _position = position;
    _gorisont = false;
    _giftLimit = false;
    width = 5;
    height = 20;
    speed = 1;
    _score = 0;
    _turnserve = false;
}
/// <summary>
/// Метод, реализующий очередь подач
/// </summary>
public void Serve()
{
    if (_turnserve)
    {
        _turnserve = false;
    }
    else
    {
        _turnserve = true;
    }
}
/// <summary>
/// Метод, реализующий движение игрока
/// </summary>
/// <param name="key">Направление движения игрока</param>
public void Move(Direction key)
{
    switch (key)
    {
        case Direction.Up:
            _positionOfCenter.Y -= speed;
            break;
        case Direction.Down:
            _positionOfCenter.Y += speed;
            break;
        case Direction.Right:
            if (_gorisont && PositionOfCenter.X < _position.X + 200)
                _positionOfCenter.X += speed;
            break;
        case Direction.Left:

```

```

        if (_gorisont && PositionOfCenter.X > _position.X - 200)
            _positionOfCenter.X -= speed;
        break;
    }
}
/// <summary>
/// Метод, реализующий подсчёт очков
/// </summary>
public void Goal()
{
    _score++;
}
/// <summary>
/// Метод, реализующий возврат игрока на начальную позицию по оси абсцисс
/// </summary>
/// <returns>Непринадлежность игрока своей линии по завершению действия бонуса</returns>
public bool ReturnPosition()
{
    if (PositionOfCenter.X < _position.X)
    {
        _positionOfCenter.X += 1;
        return true;
    }

    if (PositionOfCenter.X > _position.X)
    {
        _positionOfCenter.X -= 1;
        return true;
    }

    return false;
}
/// <summary>
/// Метод, реализующий запрет спавна бонусов
/// </summary>
public void GiftOn()
{
    _giftLimit = true;
}
/// <summary>
/// Метод, реализующий разрешение спавна бонусов
/// </summary>
public void GiftOff()
{
    _giftLimit = false;
}
}
}

```

Листинг программного класса *Sprite.cs*:

```

using SharpDX;
using System;

namespace PingPongLibrary.Entity
{
    /// <summary>
    /// Класс, представляющий сам спрайт игрового объекта
    /// </summary>
    public class Sprite
    {

```

```

private static readonly float _Pi = (float)Math.PI;
private static readonly float _2Pi = 2.0f * (float)Math.PI;

private float _angle;
/// <summary>
/// Описывает угол поворота загружаемого спрайта
/// </summary>
public float Angle
{
    get => _angle;
    set
    {
        _angle = value;
        if (_angle > _Pi) _angle -= _2Pi;
        else if (_angle < -_Pi) _angle += _2Pi;
    }
}

// Индекс картинки спрайта в коллекции
private int _bitmapIndex;
/// <summary>
/// Указывает на номер спрайта, загруженного в метод LoadBitmap класса DX2D
/// </summary>
public int BitmapIndex { get => _bitmapIndex; set => _bitmapIndex = value; }

private float _width;
/// <summary>
/// Ширина спрайта в пикселях
/// </summary>
public float Width { get => _width; }
private float _height;
/// <summary>
/// Высота спрайта в пикселях
/// </summary>
public float Height { get => _height; }
/// <summary>
/// Для увеличения или уменьшения загруженного спрайта
/// </summary>
public Size2 Size { get => new Size2((int)_width, (int)_height); }
// Положение центра спрайта в пикселях относительно верхнего левого края картинки
private Vector2 _center;
/// <summary>
/// Описывает центр спрайта, как точку центра высоты и ширины
/// </summary>
public Vector2 Center { get => _center; }
public Sprite(int bitmapIndex, int width, int height, float angle)
{
    _bitmapIndex = bitmapIndex;
    _angle = angle;
    _width = width;
    _height = height;

    _center.X = _width / 2.0f;
    _center.Y = _height / 2.0f;
}
/// <summary>
/// Предназначен для смены размеров спрайта у игрового объекта
/// </summary>
/// <param name="size">Новый размер спрайта</param>
public void Resize(Size2 size)
{
    _width = size.Width;

```

```

        _height = size.Height;
    }
}

```

Листинг программного класса *Wall.cs*:

```

using SharpDX;

namespace PingPongLibrary.Entity
{
    /// <summary>
    /// Класс, реализующий графическое представление стены
    /// </summary>
    public class Wall : GameObject
    {
        public Wall(Vector2 position) : base(position)
        {
        }
    }
}

```

Листинг программного класса *BackgroundGame.cs*:

```

using _PingPongLibrary._GameObject;
using PingPongLibrary.Entity;
using System.Collections.Generic;

namespace PingPongLibrary.GameObject
{
    /// <summary>
    /// Класс, реализующий объект заднего фона
    /// </summary>
    public class BackgroundGame : GameObject2
    {
        public BackgroundGame(Background gameObject, Sprite sprite) : base(gameObject, new List<Sprite>() { sprite })
        {
            _scale = 1;
        }
    }
}

```

Листинг программного класса *BallGame.cs*:

```

using PingPongLibrary.Entity;
using SharpDX;
using System.Collections.Generic;

namespace _PingPongLibrary._GameObject
{
    /// <summary>
    /// Класс, реализующий объект мячика
    /// </summary>
    public class BallGame : GameObject2
    {
        private Ball _ball;
        /// <summary>

```

```

    /// Позволяет взаимодействовать с логикой данного игрового объекта
    /// </summary>
    public Ball Ball { get { return _ball; } }
    public BallGame(Ball gameObject, List<Sprite> sprites) : base(gameObject, sprites) { _ball = gameObject; _scale
= 1; }

    /// <summary>
    /// Метод, реализующий привязку мяча к игроку в момент подачи
    /// </summary>
    /// <param name="rect">Центр игрока для дальнейшей привязки к нему мяча</param>
    public void SetPosition(Vector2 rect)
    {
        _ball.PositionOfCenter = rect;
    }
}

```

Листинг программного класса *BonusGame.cs*:

```

using _PingPongLibrary._GameObject;
using PingPongLibrary.Entity;
using System.Collections.Generic;

namespace PingPongLibrary.GameObject
{
    /// <summary>
    /// Класс, реализующий объект бонуса
    /// </summary>
    public class BonusGame : GameObject2
    {
        private Bonus _bonus;
        /// <summary>
        /// Позволяет взаимодействовать с логикой данного игрового объекта
        /// </summary>
        public Bonus Bonus => _bonus;
        public BonusGame(Bonus gameObject, Sprite sprites) : base(gameObject, new List<Sprite>() { sprites })
        {
            _bonus = gameObject;
            _scale = 1;
        }
    }
}

```

Листинг программного класса *GameObject2.cs*:

```

using PingPongLibrary.Entity;
using SharpDX;
using System.Collections.Generic;

namespace _PingPongLibrary._GameObject
{
    /// <summary>
    /// Абстрактный класс, для обобщённого отображения игровых объектов
    /// </summary>
    public abstract class GameObject2
    {
        protected GameObject _gameObject;
        /// <summary>

```



```

/// Описывает класс игрового объекта
/// </summary>
public GameObject gameObject { get => _gameObject; }
protected List<Sprite> _sprites;
protected Sprite _activeSprite;
protected Vector2 _positionOfRotation;
/// <summary>
/// Точка, относительно которой спрайт будет вращаться
/// </summary>
public Vector2 PositionOfRotation { get { return ActiveSprite.Center; } set => _positionOfRotation = value; }
/// <summary>
/// Логика спрайта, его ширина, высота, поворот и номер необходи-мого спрайта из коллекции спрайтов,
что хранится в классе DX2D
/// </summary>
public Sprite ActiveSprite { get => _activeSprite; set => _activeSprite = value; }
protected RectangleF _rect;
/// <summary>
/// Определяет прямоугольник объекта
/// </summary>
public RectangleF Rect { get => GetRect(); }

protected float _scale;
/// <summary>
/// Представляет собой масштабирование объекта при отрисовки его спрайта на экране
/// </summary>
public float Scale { get => _scale; }

private Size2 _size;
public GameObject2(GameObject gameObject, List<Sprite> sprites)
{
    _gameObject = gameObject;
    _sprites = new List<Sprite>();
    foreach (var sprite in sprites)
    {
        _sprites.Add(sprite);
    }
    _activeSprite = _sprites[0];
    _size = new Size2(15, 80);
}
/// <summary>
/// Рассчитывает точку верхнего левого угла прямоугольника в виде разницы между позиционированием
объекта на игровом поле и центром отображен-ного спрайта
/// </summary>
/// <returns>Возвращает прямоугольник объекта</returns>
RectangleF GetRect()
{
    _rect = new RectangleF(_gameObject.PositionOfCenter.X - ActiveSprite.Center.X, _gameObject.Position-
OfCenter.Y - ActiveSprite.Center.Y, _activeSprite.Width, _activeSprite.Heigth);
    return _rect;
}
/// <summary>
/// Предназначен для смены размеров спрайта у игрового объекта
/// </summary>
/// <param name="size">Новый размер спрайта</param>
public void Resize(Size2 size)
{
    foreach (var sprite in _sprites)
    {
        sprite.Resize(size);
    }
}
/// <summary>

```

```

    /// Предназначен для смены размеров спрайта у игрового объекта на определённый размер
    /// </summary>
    public void NormalSize()
    {
        foreach (var sprite in _sprites)
        {
            sprite.Resize(_size);
        }
    }
}

```

Листинг программного класса *PlayerGame.cs*:

```

using PingPongLibrary.Entity;
using System.Collections.Generic;

namespace _PingPongLibrary._GameObject
{
    /// <summary>
    /// Класс, реализующий объект игрока
    /// </summary>
    public class PlayerGame : GameObject2
    {
        private Player _player;
        /// <summary>
        /// Позволяет взаимодействовать с логикой данного игрового объекта
        /// </summary>
        public Player Player { get => _player; }
        private float _time;
        /// <summary>
        /// Служит для определения времени действия бонуса на игрока
        /// </summary>
        public float Time { get => _time; set => _time = value; }
        public PlayerGame(Player gameObject, List<Sprite> sprites) : base(gameObject, sprites) { _player = gameObject; }
    }
}

```

Листинг программного класса *WallGame.cs*:

```

using _PingPongLibrary._GameObject;
using PingPongLibrary.Entity;
using System.Collections.Generic;

namespace PingPongLibrary.GameObject
{
    /// <summary>
    /// Класс, реализующий объект стены
    /// </summary>
    public class WallGame : GameObject2
    {
        private Wall _wall;
        /// <summary>
        /// Позволяет взаимодействовать с логикой данного игрового объекта
        /// </summary>
        public Wall Wall { get { return _wall; } }
        public WallGame(Wall gameObject, List<Sprite> sprites) : base(gameObject, sprites) { _wall = gameObject;
        _scale = 1; }
    }
}

```

```
}  
}
```

Листинг программного класса *Game.cs*:

```
using _PingPongLibrary._GameObject;  
using PingPongLibrary.DirectX;  
using PingPongLibrary.Entity;  
using PingPongLibrary.Entity.PlayerDecorator;  
using PingPongLibrary.GameObject;  
using SharpDX;  
using SharpDX.Direct2D1;  
using SharpDX.DirectInput;  
using SharpDX.Windows;  
using System;  
using System.Collections.Generic;  
using System.Windows.Forms;  
  
namespace PingPongLibrary  
{  
    /// <summary>  
    /// Класс, содержащий основную логику игры, связывает логические компоненты объектов и компоненты от-  
    рисовки  
    /// </summary>  
    public class Game  
    {  
        private static float _unitsPerHeight = 20.0f;  
  
        private RenderForm _renderForm;  
        /// <summary>  
        /// Окно программы  
        /// </summary>  
        public RenderForm RenderForm { get => _renderForm; }  
  
        // Инфраструктурные объекты  
        private DX2D _dx2d;  
        /// <summary>  
        /// Экземпляр класса DX2D  
        /// </summary>  
        public DX2D DX2D { get => _dx2d; }  
        private InputManager _dInput;  
  
        WindowRenderTarget _target;  
        Size2F _targetSize;  
  
        // Клиентская область порта отрисовки в устройство-независимых пикселях  
        private RectangleF _clientRect;  
  
        private float _scale;  
        /// <summary>  
        /// Коэффициент масштабирования  
        /// </summary>  
        public float Scale { get => _scale; }  
  
        // Помощник для работы со временем  
        private TimeHelper _timeHelper;  
  
        // Остальные приватные поля, используемые для разных целей  
        private BackgroundGame _background;  
        private PlayerGame _firstPlayer;  
        private PlayerGame _secondPlayer;
```

```

private WallGame _topWall;
private WallGame _botWall;
private BallGame _ball;
private int _ballIndex;
private int _giftIndex;
private int _playerIndex;
private bool _serveraffle;
private bool _serve;
private Random _rnd;
private BonusGame _bonus;
private BonusFactory _bonusFactory;
private Generator _generator;
private bool _giftdraw;
private PhysicsEngine _physicsEngine;
private bool _gameEnd;
private bool _gameReturn;
/// <summary>
/// Значение перезапуска приложения
/// </summary>
public bool GameReturn { get => _gameReturn; }

Drawer _drawer;

// В конструкторе создаем форму, инфраструктурные объекты, подгружаем спрайты, создаем помощник
// для работы со временем
// В конце вызываем ресайзинг формы для вычисления масштаба и установки пределов по горизонтали и
// вертикали
public Game()
{
    _renderForm = new RenderForm("Ping Pong");
    _renderForm.WindowState = FormWindowState.Maximized;
    _renderForm.IsFullscreen = true;

    _dx2d = new DX2D(_renderForm);
    _dInput = new InputManager(_renderForm);

    _target = _dx2d.RenderTarget;
    _targetSize = _target.Size;
    // Индексы наших картинок
    int backgroundIndex = _dx2d.LoadBitmap("..\\..\\..\\table.png");
    _playerIndex = _dx2d.LoadBitmap("..\\..\\..\\paddle.png");
    _ballIndex = _dx2d.LoadBitmap("..\\..\\..\\ball.png");
    _giftIndex = _dx2d.LoadBitmap("..\\..\\..\\gift.png");
    // Инициализация наших отображаемых объектов
    _background = new BackgroundGame(new Entity.Background(new Vector2(_targetSize.Width / 2, _targetSize.Height / 2)), new Sprite(backgroundIndex, (int)_targetSize.Width, (int)_targetSize.Height, 0));

    _firstPlayer = new PlayerGame(new Player(new Vector2(200, 540)), new List<Sprite> { new Sprite(_playerIndex, 15, 80, 0) });
    _secondPlayer = new PlayerGame(new Player(new Vector2(1720, 540)), new List<Sprite> { new Sprite(_playerIndex, 15, 80, 0) });

    _topWall = new WallGame(new Wall(new Vector2(_targetSize.Width / 2, 0)), new List<Sprite> { new Sprite(0, (int)_targetSize.Width, 10, 0) });
    _botWall = new WallGame(new Wall(new Vector2(_targetSize.Width / 2, _targetSize.Height)), new List<Sprite> { new Sprite(0, (int)_targetSize.Width, 10, 0) });

    _ball = new BallGame(new Ball(new Vector2(_targetSize.Width / 2, _targetSize.Height / 2)), new List<Sprite> { new Sprite(_ballIndex, 32, 32, 0) });

```

```

    _gameReturn = false;
    _gameEnd = false;
    _physicsEngine = new PhysicsEngine();
    _rnd = new Random();
    _giftDraw = false;
    _serve = false;
    _serverAffle = true;
    _drawer = new Drawer(_dx2d);
    _timeHelper = new TimeHelper();
    RenderForm_Resize(this, null);
    float scale = _dx2d.RenderTarget.Size.Width / _dx2d.RenderTarget.PixelSize.Width;
}
/// <summary>
/// Вызываются методы отображения объектов на игровом поле, методы колли-зии, методы проверки окон-
чания игры, а также в данном методе заключена логика передвижения игроков
/// </summary>
private void RenderCallback()
{
    if (!_gameEnd)
    {
        // Вызываем обновление состояния "временного" помощника и объектов ввода
        _timeHelper.Update();
        _dInput.UpdateKeyboardState();

        // Для расчетов масштаба берем не клиентскую область формы, которая в честных пикселях, а
        RenderTarget-a
        WindowRenderTarget target = _dx2d.RenderTarget;
        Size2F targetSize = target.Size;
        _clientRect.Width = targetSize.Width;
        _clientRect.Height = targetSize.Height;

        // Начинаем вывод графики
        _target.BeginDraw();
        // Перво-наперво - очистить область отображения
        _target.Clear(Color.Black);

        // Прорисовка наших объектов
        _drawer.Draw(_background);

        _drawer.DrawObject(_firstPlayer);
        _drawer.DrawObject(_secondPlayer);
        _drawer.Draw(_ball);

        EndGame();

        // Движение игроков (ракеток) с условием на коллизию
        if (_dInput.KeyboardUpdated)
        {
            if (_dInput.KeyboardState.IsPressed(Key.W) && !PlayerColWalls(_firstPlayer, Direction.Up))
            {
                for (int i = 0; i < 10 && !PlayerColWalls(_firstPlayer, Direction.Up); i++)
                {
                    _firstPlayer.Player.Move(Direction.Up);
                }
            }
            if (_dInput.KeyboardState.IsPressed(Key.S) && !PlayerColWalls(_firstPlayer, Direction.Down))
            {
                for (int i = 0; i < 10 && !PlayerColWalls(_firstPlayer, Direction.Down); i++)
                {
                    _firstPlayer.Player.Move(Direction.Down);
                }
            }
        }
    }
}

```

```

        if (_dInput.KeyboardState.IsPressed(Key.D) && !PlayerColWalls(_firstPlayer, Direction.Right) && _first-
Player.Player.Gorizont)
        {
            for (int i = 0; i < 10 && !PlayerColWalls(_firstPlayer, Direction.Right); i++)
            {
                _firstPlayer.Player.Move(Direction.Right);
            }
        }
        if (_dInput.KeyboardState.IsPressed(Key.A) && !PlayerColWalls(_firstPlayer, Direction.Left) && _first-
Player.Player.Gorizont)
        {
            for (int i = 0; i < 10 && !PlayerColWalls(_firstPlayer, Direction.Left); i++)
            {
                _firstPlayer.Player.Move(Direction.Left);
            }
        }

        if (_dInput.KeyboardState.IsPressed(Key.Up) && !PlayerColWalls(_secondPlayer, Direction.Up))
        {
            for (int i = 0; i < 10 && !PlayerColWalls(_secondPlayer, Direction.Up); i++)
            {
                _secondPlayer.Player.Move(Direction.Up);
            }
        }
        if (_dInput.KeyboardState.IsPressed(Key.Down) && !PlayerColWalls(_secondPlayer, Direction.Down))
        {
            for (int i = 0; i < 10 && !PlayerColWalls(_secondPlayer, Direction.Down); i++)
            {
                _secondPlayer.Player.Move(Direction.Down);
            }
        }
        if (_dInput.KeyboardState.IsPressed(Key.Left) && !PlayerColWalls(_secondPlayer, Direction.Left) &&
_secondPlayer.Player.Gorizont)
        {
            for (int i = 0; i < 10 && !PlayerColWalls(_secondPlayer, Direction.Left); i++)
            {
                _secondPlayer.Player.Move(Direction.Left);
            }
        }
        if (_dInput.KeyboardState.IsPressed(Key.Right) && !PlayerColWalls(_secondPlayer, Direction.Right) &&
_secondPlayer.Player.Gorizont)
        {
            for (int i = 0; i < 10 && !PlayerColWalls(_secondPlayer, Direction.Right); i++)
            {
                _secondPlayer.Player.Move(Direction.Right);
            }
        }
        if (_dInput.KeyboardState.IsPressed(Key.Space) && !_serveraffle)
        {
            _serve = true;
        }
    }

    if (!_serveraffle)
    {
        _drawer.ScoreDraw(_firstPlayer.Player.Score, _secondPlayer.Player.Score);
        if (_giftdraw)
        {
            _drawer.Draw(_bonus);
            GiftCol();
        }
    }
    else

```

```

        {
            GiftSpawn();
        }
    }
    else
    {
        _drawer.ServeDraw();
    }

    if (!_firstPlayer.Player.Gorizont)
    {
        for (int i = 0; i < 10 && _firstPlayer.Player.ReturnPosition(); i++)
        {
        }
    }
    if (!_secondPlayer.Player.Gorizont)
    {
        for (int i = 0; i < 10 && _secondPlayer.Player.ReturnPosition(); i++)
        {
        }
    }

    if (_firstPlayer.Time <= _timeHelper.Time && _firstPlayer.Player.GiftLimit)
    {
        _firstPlayer = new NormalPlayer(_firstPlayer);
    }
    if (_secondPlayer.Time <= _timeHelper.Time && _secondPlayer.Player.GiftLimit)
    {
        _secondPlayer = new NormalPlayer(_secondPlayer);
    }

    BallCol();
    BallColWalls();
    BallIsOut();
    // Передвижение шарика
    if (!_serveraffle && !_serve)
    {
        PingPongServe();
    }
    else
    {
        for (int i = 0; i < 7; i++)
        {
            _ball.Ball.Move();
        }
    }

    _target.EndDraw();
}
if (_dInput.KeyboardUpdated)
{
    _dInput.UpdateKeyboardState();
    if (_dInput.KeyboardState.IsPressed(Key.Escape) && _gameEnd == true)
    {
        this._renderForm.Close();

        _gameReturn = true;
    }
}
}
}
/// <summary>
/// Реализована логика привязки шарика к игроку, который будет подавать

```

```

/// </summary>
private void PingPongServe()
{
    if (_firstPlayer.Player.TurnServe)
    {
        _ball.SetPosition(new Vector2(_firstPlayer.Rect.Right + _ball.Rect.Width / 2, _firstPlayer.Rect.Center.Y));
    }
    else if (_secondPlayer.Player.TurnServe)
    {
        _ball.SetPosition(new Vector2(_secondPlayer.Rect.Left - _ball.Rect.Width / 2, _secondPlayer.Rect.Center.Y));
    }
    _ball.Ball.Speed = 1;
}
/// <summary>
/// Реализуется логика коллизии игрока со стенами
/// </summary>
/// <param name="player">Объект класса PlayerGame</param>
/// <param name="key">Направление движения игрока</param>
/// <returns>Возвращает столкновение игрока со стеной</returns>
private bool PlayerColWalls(PlayerGame player, Direction key)
{
    RectangleF pl = new RectangleF();
    switch (key)
    {
        case Direction.Up:
            pl = new RectangleF(player.Rect.Left, player.Rect.Top - 1, player.Rect.Width, player.Rect.Height);
            break;
        case Direction.Down:
            pl = new RectangleF(player.Rect.Left, player.Rect.Top + 1, player.Rect.Width, player.Rect.Height);
            break;
    }
    /*_dx2d.RenderTarget.DrawRectangle(pl, DX2D.RedBrush);*/
    if (pl.Intersects(_topWall.Rect)) { player.Player.Move(Direction.Down); return true; }
    if (pl.Intersects(_botWall.Rect)) { player.Player.Move(Direction.Up); return true; }
    return false;
}
/// <summary>
/// Реализует коллизию шарика со стенами
/// </summary>
private void BallColWalls()
{
    if (_ball.Rect.Intersects(_topWall.Rect)) { _physicsEngine.UpdateBall(_ball); }
    if (_ball.Rect.Intersects(_botWall.Rect)) { _physicsEngine.UpdateBall(_ball); }
}
/// <summary>
/// Определяет коллизию шарика с игроком
/// </summary>
private void BallCol()
{
    /*_dx2d.RenderTarget.DrawRectangle(pl, DX2D.RedBrush);*/
    if (_ball.Rect.Intersects(_firstPlayer.Rect)) { _ball.Ball.Direction = new Vector2(1, 0); _physicsEngine.UpdatePlayer(_ball, _firstPlayer); }
    if (_ball.Rect.Intersects(_secondPlayer.Rect)) { _ball.Ball.Direction = new Vector2(-1, 0); _physicsEngine.UpdatePlayer(_ball, _secondPlayer); }
}
/// <summary>
/// Реализует логику коллизии игрока и бонуса
/// </summary>
/// <returns>Возвращает столкновение игрока с бонусом</returns>
private bool GiftCol()
{

```



```

        if (_bonus.Rect.Intersects(_firstPlayer.Rect)) { _giftdraw = false; GiftUsage((_bonus.Bonus.Type == TypeBo-
nus.Small || _bonus.Bonus.Type == TypeBonus.LowSpeed) ? _secondPlayer : _firstPlayer); return true; }
        if (_bonus.Rect.Intersects(_secondPlayer.Rect)) { _giftdraw = false; GiftUsage((_bonus.Bonus.Type ==
TypeBonus.Small || _bonus.Bonus.Type == TypeBonus.LowSpeed) ? _firstPlayer : _secondPlayer); return true; }
        return false;
    }
    /// <summary>
    /// Запускает таймер действия бонуса, а также определяет его тип
    /// </summary>
    /// <param name="player">Объект класса PlayerGame</param>
    private void GiftUsage(PlayerGame player)
    {
        player.Time = _timeHelper.Time + 5f; player.Player.GiftOn();
        switch (_bonus.Bonus.Type)
        {
            case TypeBonus.Big:
                player = new Entity.PlayerDecorator.BigPlayer(player);
                break;
            case TypeBonus.HighSpeed:
                player = new Entity.PlayerDecorator.HighSpeedPlayer(player);
                break;
            case TypeBonus.LowSpeed:
                player = new Entity.PlayerDecorator.LowSpeedPlayer(player);
                break;
            case TypeBonus.Small:
                player = new Entity.PlayerDecorator.SmallPlayer(player);
                break;
            case TypeBonus.MoveX:
                player = new Entity.PlayerDecorator.MoveXPlayer(player);
                break;
        }
    }
    /// <summary>
    /// Определяет вышел ли мяч за пределы поля или нет
    /// </summary>
    private void BallIsOut()
    {
        if (_ball.Rect.Center.X < 0)
        {
            if (!_serveraffle)
            {
                _secondPlayer.Player.Goal();
                _serve = false;
                ChoiceServe();
            }
            else
            {
                _serveraffle = false;
                _secondPlayer.Player.Serve();
            }
            PingPongServe();
        }
        if (_ball.Rect.Center.X > _targetSize.Width)
        {
            if (!_serveraffle)
            {
                _firstPlayer.Player.Goal();
                _serve = false;
                ChoiceServe();
            }
            else
            {

```

```

        _serveraffle = false;
        _firstPlayer.Player.Serve();
    }
    PingPongServe();
}
}
/// <summary>
/// Реализует логику определения очереди подачи
/// </summary>
private void ChoiceServe()
{
    if (_firstPlayer.Player.Score >= 10 && _secondPlayer.Player.Score >= 10)
    {
        if (_secondPlayer.Player.TurnServe)
        {
            _ball.Ball.Direction = new Vector2(1, 0);
            _secondPlayer.Player.Serve();
            _firstPlayer.Player.Serve();
        }
        else
        {
            _ball.Ball.Direction = new Vector2(-1, 0);
            _secondPlayer.Player.Serve();
            _firstPlayer.Player.Serve();
        }
    }
    else if ((_firstPlayer.Player.Score + _secondPlayer.Player.Score) % 2 == 0 && (_firstPlayer.Player.Score +
_secondPlayer.Player.Score) > 0)
    {
        if (_secondPlayer.Player.TurnServe)
        {
            _ball.Ball.Direction = new Vector2(1, 0);
            _secondPlayer.Player.Serve();
            _firstPlayer.Player.Serve();
        }
        else
        {
            _ball.Ball.Direction = new Vector2(-1, 0);
            _secondPlayer.Player.Serve();
            _firstPlayer.Player.Serve();
        }
    }
}
}
/// <summary>
/// Содержится логика по случайной генерации бонуса
/// </summary>
private void GiftSpawn()
{
    if (_rnd.Next(1000) == 100)
    {
        switch (_rnd.Next(2))
        {
            case 0: if (!_firstPlayer.Player.GiftLimit) ChoiceGift(_firstPlayer); break;
            case 1: if (!_secondPlayer.Player.GiftLimit) ChoiceGift(_secondPlayer); break;
        }
    }
}
}
/// <summary>
/// Метод, реализующий случайный выбор генерируемого бонуса
/// </summary>
/// <param name="player">Объект класса PlayerGame</param>
private void ChoiceGift(PlayerGame player)

```

```

{
    Vector2 position = new Vector2(player.Player.Position.X, _rnd.Next((int)_targetSize.Height - 200) + 100);
    switch (_rnd.Next(5))
    {
        case 0:
            _bonusFactory = new BonusBigPlayer(position);
            _generator = _bonusFactory.GetBonus();
            _bonus = new BonusGame(_generator.GetGenerator(), new Sprite(_giftIndex, 32, 32, 0));
            break;
        case 1:
            _bonusFactory = new BonusSmallPlayer(position);
            _generator = _bonusFactory.GetBonus();
            _bonus = new BonusGame(_generator.GetGenerator(), new Sprite(_giftIndex, 32, 32, 0));
            break;
        case 2:
            _bonusFactory = new BonusHighSpeedPlayer(position);
            _generator = _bonusFactory.GetBonus();
            _bonus = new BonusGame(_generator.GetGenerator(), new Sprite(_giftIndex, 32, 32, 0));
            break;
        case 3:
            _bonusFactory = new BonusLowSpeedPlayer(position);
            _generator = _bonusFactory.GetBonus();
            _bonus = new BonusGame(_generator.GetGenerator(), new Sprite(_giftIndex, 32, 32, 0));
            break;
        case 4:
            _bonusFactory = new BonusMoveXPlayer(position);
            _generator = _bonusFactory.GetBonus();
            _bonus = new BonusGame(_generator.GetGenerator(), new Sprite(_giftIndex, 32, 32, 0));
            break;
    }
    _giftdraw = true;
}
/// <summary>
/// Реализует логику окончания игры
/// </summary>
private void EndGame()
{
    bool flag = true;
    if (_firstPlayer.Player.Score >= 11 && (_firstPlayer.Player.Score - _secondPlayer.Player.Score) >= 2 || _first-
Player.Player.Score == 7 && _secondPlayer.Player.Score == 0)
    {
        _drawer.ResultDraw(flag);
        _gameEnd = true;
    }
    if (_secondPlayer.Player.Score >= 11 && (_secondPlayer.Player.Score - _firstPlayer.Player.Score) >= 2 ||
_secondPlayer.Player.Score == 7 && _firstPlayer.Player.Score == 0)
    {
        flag = false;
        _drawer.ResultDraw(flag);
        _gameEnd = true;
    }
}
/// <summary>
/// Определяет масштабируемость игрового поля относительно изменения раз-мера игрового окна
/// </summary>
/// <param name="sender">Объект</param>
/// <param name="e">Событие</param>
private void RenderForm_Resize(object sender, EventArgs e)
{
    // При ресайзинге обновляем размер области отображения и масштаб
    int width = _renderForm.ClientSize.Width;
    int height = _renderForm.ClientSize.Height;

```

```

        _dx2d.RenderTarget.Resize(new Size2(width, height));
        _clientRect.Width = _dx2d.RenderTarget.Size.Width;
        _clientRect.Height = _dx2d.RenderTarget.Size.Height;
        _scale = _clientRect.Height / _unitsPerHeight;
    }
    /// <summary>
    /// Отображает игровое поле и вызывает метод RenderCallback
    /// </summary>
    public void Run()
    {
        _renderForm.Resize += RenderForm_Resize;
        RenderLoop.Run(_renderForm, RenderCallback);
    }
    /// <summary>
    /// Освобождает неуправляемые ресурсы игрового приложения
    /// </summary>
    public void Dispose()
    {
        _dInput.Dispose();
        _dx2d.Dispose();
        _renderForm.Dispose();
    }
}
}

```

Листинг программного класса *PhysicsEngine.cs*:

```

using _PingPongLibrary._GameObject;
using SharpDX;
using System;

namespace PingPongLibrary
{
    /// <summary>
    /// Класс, реализующий логику физики игры, то есть логика отскоков шарика от стен и игроков
    /// </summary>
    public class PhysicsEngine
    {
        /// <summary>
        /// Рассчитывает и обновляет угол отскока шарика от игрока
        /// </summary>
        /// <param name="ball">Объект класса BallGame</param>
        /// <param name="player">Объект класса PlayerGame</param>
        public void UpdatePlayer(BallGame ball, PlayerGame player)
        {
            // определяем на какую часть ракетки попал мяч
            float paddleHeight = player.Rect.Height;
            float paddleTop = player.Player.PositionOfCenter.Y - paddleHeight / 2f;
            float ballHeight = ball.Rect.Height;
            float ballTop = ball.Ball.PositionOfCenter.Y - ballHeight / 2f;
            float distanceFromTop = ballTop - paddleTop;
            float relativeIntersectY = distanceFromTop / paddleHeight * 2f;
            float angle = MathUtil.DegreesToRadians(45f) * relativeIntersectY;
            // вводим ограничения на угол движения мяча
            if (angle >= 1.3f) { angle = 1.3f; }
            if (angle <= -1.3f) { angle = -1.3f; }
            // изменяем направление движения мяча
            if (ball.Ball.Direction == new Vector2(-1, 0))
            {
                ball.Ball.Direction = new Vector2(-(float)Math.Cos(angle), (float)Math.Sin(angle));
            }
        }
    }
}

```

```

else
{
    ball.Ball.Direction = new Vector2((float)Math.Cos(angle), (float)Math.Sin(angle));
}

// увеличиваем скорость мяча
ball.Ball.Speed += 0.10f;
//MessageBox.Show(angle.ToString());
}
/// <summary>
/// Обновляет угол отскока шарика от стен
/// </summary>
/// <param name="ball">Объект класса BallGame</param>
public void UpdateBall(BallGame ball)
{
    ball.Ball.Direction = new Vector2(ball.Ball.Direction.X, -ball.Ball.Direction.Y);
}
}
}

```

Листинг программного класса *MainWindow.xaml.cs*:

```

using System.Windows;

namespace Course_Project
{
    /// <summary>
    /// Логика взаимодействия для MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            PingPongLibrary.Game game = new PingPongLibrary.Game();
            game.Run();
            while (game.GameReturn) { game = new PingPongLibrary.Game(); game.Run(); }
            game.Dispose();
            InitializeComponent();
        }
    }
}

```

Листинг программного класса *BallTests.cs*:

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using PingPongLibrary.Entity;
using SharpDX;

namespace Tests
{
    /// <summary>
    /// Класс тестирования для класса Ball
    /// </summary>
    [TestClass]
    public class BallTests
    {
        /// <summary>
        /// Тестирование метода Move
        /// </summary>

```

```

[TestMethod]
public void MoveTest()
{
    Ball ball = new Ball(new SharpDX.Vector2(200, 200));
    Vector2 vector2 = ball.PositionOfCenter;
    ball.Direction = new Vector2(1, 0);
    ball.Move();
    Assert.AreEqual(vector2.X + 1, ball.PositionOfCenter.X);
    vector2 = ball.PositionOfCenter;
    ball.Direction = new Vector2(-1, 0);
    ball.Move();
    Assert.AreEqual(vector2.X - 1, ball.PositionOfCenter.X);
}
}
}

```

Листинг программного класса *PlayerTests.cs*:

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using PingPongLibrary.Entity;
using SharpDX;

namespace Tests
{
    /// <summary>
    /// Класс тестирования класса Player
    /// </summary>
    [TestClass]
    public class PlayerTests
    {
        /// <summary>
        /// Тест метода Serve
        /// </summary>
        [TestMethod]
        public void ServeTest()
        {
            Player player = new Player(new SharpDX.Vector2(200, 200));
            Assert.IsFalse(player.TurnServe);

            player.Serve();
            Assert.IsTrue(player.TurnServe);
        }
        /// <summary>
        /// Тест метода Move
        /// </summary>
        [TestMethod]
        public void MoveTest()
        {
            Player player = new Player(new SharpDX.Vector2(200, 200));
            Vector2 vector2 = player.PositionOfCenter;
            player.Move(Direction.Up);
            Assert.AreEqual(vector2.Y - 1, player.PositionOfCenter.Y);

            vector2 = player.PositionOfCenter;
            player.Move(Direction.Down);
            Assert.AreEqual(vector2.Y + 1, player.PositionOfCenter.Y);

            player.Gorisont = true;
            vector2 = player.PositionOfCenter;
            player.Move(Direction.Right);
            Assert.AreEqual(vector2.X + 1, player.PositionOfCenter.X);
        }
    }
}

```

```

        vector2 = player.PositionOfCenter;
        player.Move(Direction.Left);
        Assert.AreEqual(vector2.X - 1, player.PositionOfCenter.X);
    }
    /// <summary>
    /// Тест метода Goal
    /// </summary>
    [TestMethod]
    public void GoalTest()
    {
        Player player = new Player(new SharpDX.Vector2(200, 200));
        int score = 1;
        player.Goal();
        Assert.AreEqual(score, player.Score);
    }
    /// <summary>
    /// Тест метода ReturnPosition
    /// </summary>
    [TestMethod]
    public void ReturnPositionTest()
    {
        Player player = new Player(new SharpDX.Vector2(200, 200));
        Assert.IsFalse(player.ReturnPosition());

        Vector2 vector2 = player.PositionOfCenter;
        player.Gorisont = true;
        player.Move(Direction.Left);
        Assert.IsTrue(player.ReturnPosition());
        Assert.AreEqual(vector2.X, player.PositionOfCenter.X);

        vector2 = player.PositionOfCenter;
        player.Move(Direction.Right);
        Assert.IsTrue(player.ReturnPosition());
        Assert.AreEqual(vector2.X, player.PositionOfCenter.X);
    }
    /// <summary>
    /// Тест метода GiftOn
    /// </summary>
    [TestMethod]
    public void GiftOnTest()
    {
        Player player = new Player(new SharpDX.Vector2(200, 200));
        bool flag = true;
        player.GiftOn();
        Assert.AreEqual(flag, player.GiftLimit);
    }
    /// <summary>
    /// Тест метода GiftOff
    /// </summary>
    [TestMethod]
    public void GiftOffTest()
    {
        Player player = new Player(new SharpDX.Vector2(200, 200));
        bool flag = false;
        player.GiftOff();
        Assert.AreEqual(flag, player.GiftLimit);
    }
}
}

```

ПРИЛОЖЕНИЕ Б (обязательное) Графический интерфейс

Окно игры представлено на рисунке Б.1.

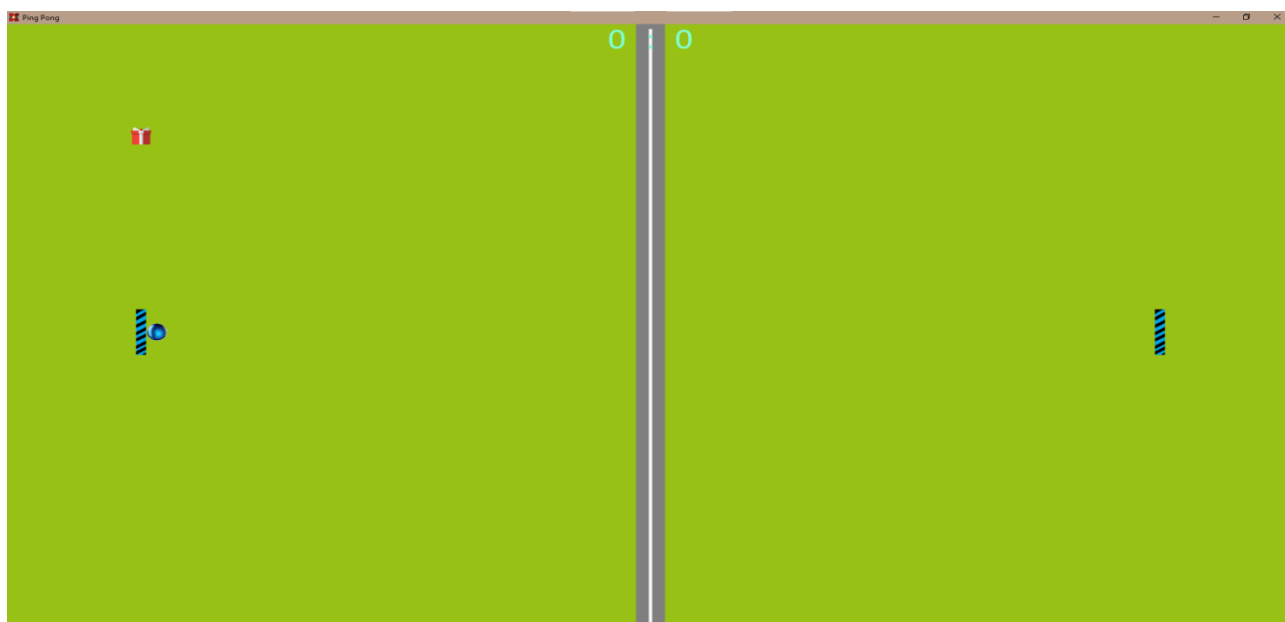


Рисунок Б.1 – Окно игры

Окно конца игры представлено на рисунке Б.2.



Рисунок Б.2 – Конец игры

ПРИЛОЖЕНИЕ В

(обязательное)

Руководство пользователя

1. Введение.

Разработанное приложение предназначено для игры двух игроков на одном устройстве. Приложение обладает следующим функционалом:

- бонусами;
- игровым полем;
- передвижными ракетками, которыми управляют игроки;
- шариком.

2. Назначения и условия применения.

Игровое приложение предназначено для игры двух игроков на одном устройстве. Служит в качестве развлечения и не может применяться в работе. Доступен режим игры только на двух игроков.

Для корректной работы программы и возможности взаимодействия с ней необходимо:

- операционная система *Windows 10*;
- наличие следующего устройства ввода: клавиатура;
- наличие устройства вывода: монитор.

3. Подготовка к работе.

Для начала игры необходимо запустить файл *MainWindow.exe*.

4. Описание операций.

Пользователь может выполнять следующие операции:

- подбор бонуса;
- перемещение на игровом поле;
- совершение подачи.

5. Аварийные ситуации.

Возникновение аварийных ситуаций в приложении практически невозможно. Однако в случае возникновения ошибки, рекомендуется перезапустить приложение.

6. Рекомендации по освоению.

Рекомендации по освоению отсутствуют, поскольку любой пользователь, имеющий опыт работы с компьютером, сразу же сможет освоить данное приложение.

ПРИЛОЖЕНИЕ Г
(обязательное)
Руководство программиста

1. Назначения и условия применения программы.

Игровое приложение предназначено для игры двух игроков на одном устройстве. Служит в качестве развлечения и не может применяться в работе. Доступен режим игры только на двух игроков.

Для корректной работы программы и возможности взаимодействия с ней необходимо:

- операционная система *Windows 10*;
- наличие следующего устройства ввода: клавиатура;
- наличие устройства вывода: монитор.

2. Характеристики программы.

При запуске игрового приложения открывается окно самого приложения с уже начатой игрой, а именно разыгровкой подачи. На появившемся игровом поле находятся две ракетки, которыми игроки управляют с помощью клавиатуры и шар.

3. Обращение к программе.

Приложение запускается путём открытия файла *MainWindow.exe*.

4. Входные и выходные данные.

В приложении в качестве входных данных используется язык программирования *C#*, графика *DirectX*, *WPF* и шаблон проектирования.

5. Сообщения.

При возникновении какой-либо ошибки необходимо перезапустить игровое приложение.

ПРИЛОЖЕНИЕ Д

(обязательное)

Руководство системного программиста

1. Общие сведения о программе.

Игровое приложение предназначено для игры двух игроков на одном устройстве. Служит в качестве развлечения и не может применяться в работе. Доступен режим игры только на двух игроков.

Для корректной работы программы и возможности взаимодействия с ней необходимо:

- операционная система *Windows 10*;
- наличие следующего устройства ввода: клавиатура;
- наличие устройства вывода: монитор.

2. Структура программы.

Игровое приложение разделено на две основные составляющие: графическую и логическую части.

Графическая часть включает в себя пользовательский интерфейс и основную графику приложения. Именно в этой части программы происходит визуализация логической части.

В логической части описываются основные объекты приложения и их взаимодействие между собой.

3. Настройка программы.

Игровое приложение не требует настройки. Для начала игры достаточно запустить игровое приложение.

4. Проверка программы.

Отсутствие окон ошибок и возникновений исключений говорит о том, что программа работает корректно.

5. Дополнительные возможности.

Дополнительные возможности отсутствуют.

6. Сообщения системному программисту.

При возникновении какой-либо ошибки рекомендуется перезапустить игровое приложение.

ПРИЛОЖЕНИЕ Ж
(обязательное)
Функциональная схема приложения