

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: «Игровое приложение «Экшен-платформер» с использованием  
графики *DirectX*»

Исполнитель: студент группы ИТИ-21  
Дегтеров Д.В.

Руководитель: преподаватель-стажер  
Малиновский И.Л.

Дата проверки: \_\_\_\_\_

Дата допуска к защите: \_\_\_\_\_

Дата защиты: \_\_\_\_\_

Оценка работы: \_\_\_\_\_

Подписи членов комиссии  
по защите курсового проекта: \_\_\_\_\_

Гомель 2022



## СОДЕРЖАНИЕ

Введение.....	4
1 Описание используемых технологий.....	5
1.1 Понятие видеоигры.....	5
1.2 Платформа <i>.NET</i> .....	6
1.3 Язык программирования <i>C#</i> .....	7
1.4 Платформа пользовательского интерфейса <i>WFA</i> .....	9
1.5 Графическая библиотека <i>DirectX</i> .....	11
1.6 Отличительные особенности технологии <i>DirectX</i> в сравнении с <i>OpenGL</i> .....	12
1.7 <i>SharpDX</i> как оболочка <i>DirectX</i> .....	14
1.8 Требования к игровому приложению.....	15
2 Программная реализация игрового приложения «Экшен-платформер».....	16
2.1 Паттерны проектирования «Фабричный метод» и «Декоратор» как средства реализации приложения.....	16
2.2 Проектирование приложения методом декомпозиции.....	17
2.3 Структура данных приложения.....	20
3 Верификация и апробация игры «Экшен-платформер».....	24
3.1 Принцип работы приложения.....	24
3.2 Результаты тестирования приложения.....	25
3.3 Результаты верификации приложения.....	27
Заключение.....	30
Список использованных источников.....	31
ПРИЛОЖЕНИЕ А Листинг программы «Экшен-платформер».....	32
ПРИЛОЖЕНИЕ Б Руководство пользователя.....	89
ПРИЛОЖЕНИЕ В Руководство программиста.....	91
ПРИЛОЖЕНИЕ Г Руководство системного программиста.....	92
ПРИЛОЖЕНИЕ Д Функциональная схема приложения.....	93

## ВВЕДЕНИЕ

Игровую индустрию в настоящее время воспринимают в качестве сектора экономики, связанного с разработкой, продвижением и продажей компьютерных игр. На данный момент времени создание видеоигр является одним из наиболее крупных сегментов индустрии развлечений. Масштабы индустрии можно сопоставить с кинематографом, а прибыль за последние несколько лет показывает лучшие показатели.

Актуальность темы данного курсового проекта обусловлена тем, что в наше время большое количество людей, имеющих разнообразные интересы, играют в свободное время в компьютерные игры, причём это не только школьники или прогульщики-студенты. Бизнесмены, политики, инженеры, художники – люди самых разных профессий, встречаются среди игроков. Всех их объединяет общая цель – познавать в виртуальных мирах что-то новое, неисследованное, испытать удачу и получить удовольствие, как от игрового процесса, так и от достигнутых в игре результатов.

В соответствии с поставленной задачей необходимо:

- разработать библиотеку классов, описывающих игровые сущности;
  - разработать классы, которые реализуют графическую составляющую приложения с помощью *API DirectX*;
  - отобразить разработанную игровую сцену в окне *Windows Forms*.
- Приложение должно иметь возможность масштабирования, т.е. можно добавлять новые сущности без изменения уже написанного кода.

# 1 ОПИСАНИЕ ИСПОЛЬЗУЕМЫХ ТЕХНОЛОГИЙ

## 1.1 Понятие видеоигры

Видеоигра – это электронная игра, которая включает взаимодействие с пользовательским интерфейсом или устройством ввода, таким как джойстик, контроллер, клавиатура или устройство обнаружения движения, для создания визуальной обратной связи. Эта обратная связь отображается на устройстве отображения видео, таком как телевизор, монитор, сенсорный экран или гарнитура виртуальной реальности. Видеоигры часто дополняются звуковой обратной связью, воспроизводимой через динамики или наушники, а иногда и другими типами обратной связи, включая тактильные технологии.

Видеоигры, как и большинство других форм медиа, можно разделить на жанры. Однако, в отличие от фильмов или телевидения, в которых используются визуальные или повествовательные элементы, видеоигры обычно делятся на жанры в зависимости от их взаимодействия с игровым процессом, поскольку это основное средство взаимодействия с видеоигрой. Настройка повествования не влияет на игровой процесс; шутер остается шутером, независимо от того, происходит ли он в мире фантазий или в открытом космосе. Исключением является жанр игр ужасов, используемый для игр, основанных на повествовательных элементах фантастики ужасов, сверхъестественного и психологического ужаса.

Названия жанров обычно говорят сами за себя с точки зрения типа игрового процесса, например, экшн, ролевая игра или стрелялка, хотя некоторые жанры произошли от влиятельных произведений. Названия могут со временем меняться, поскольку игроки, разработчики и СМИ придумывают новые термины; например, шутеры от первого лица изначально назывались «клонами *Doom*» на основе игры 1993 года. Существует иерархия игровых жанров с жанрами высшего уровня, такими как «шутер» и «экшн», которые в целом охватывают основной игровой стиль игры, и несколько поджанров с конкретной реализацией, например, шутер от первого лица и шутер от третьего лица. шутер от человека. Также существуют некоторые кросс-жанровые типы, которые относятся к нескольким жанрам высшего уровня, таким как приключенческая игра.

Цели видеоигры:

- 1) Казуальные игры рассчитаны на массовую аудиторию. Они часто поддерживают возможность включаться и выходить из игры по требованию, например, во время поездок на работу или во время обеденных перерывов;
- 2) Развивающие игры – специальное программное обеспечение, используется дома и в классах, чтобы помочь в обучении пользователю,

которое было разработано для обеспечения интерактивности и развлечений, связанных с элементами игрового дизайна;

3) Художественная игра – предназначена для того, чтобы вызывать у игрока эмоции или сочувствие, бросая вызов социальным нормам и предлагая критику через интерактивность среды видеоигр. Они могут не иметь каких-либо условий победы, и предназначены для того, чтобы игрок мог исследовать игровой мир и сценарии.

## 1.2 Платформа .NET

Платформа .NET была официально представлена в 2002 году и почти моментально стала основой современной разработки программного обеспечения. Платформа позволяет использовать большое количество языков программирования (включая C#, VB.NET и F#) для взаимодействия друг с другом. Код, написанный на C#, может быть использован при написании приложения на другом языке программирования. В 2016 году был официально запущен .NET Core. Как и .NET, .NET Core позволяет языкам взаимодействовать между собой (хотя поддерживается ограниченное количество языков). Что более важно, эта новая технология больше не ограничена работой в операционной системе Windows, но также может работать (и разрабатываться) на MacOS и Linux. Независимость от платформы открыла C# для гораздо большего числа разработчиков. В 2020 году был выпущен .NET 5, «Core» в имени был опущен, с целью показать, что эта версия представляет собой унификацию всего .NET.

Платформа .NET Core – это программная платформа для создания веб-приложений и сервисов на операционных системах Windows, MacOS и Linux, а также приложения Windows Forms и WPF в операционных системах Windows.

Основные функции .NET Core:

- Обратная совместимость с существующим кодом: существующее программное обеспечение, работающее на .NET Framework может работать с новыми версиями .NET Core;

- Поддержка множества языков программирования: приложения на .NET Core могут создаваться с использованием разных языков программирования: C#, F#, VB.NET и других;

- Общий механизм выполнения, используемый всеми языками .NET Core: один из аспектов этого механизма – чётко определенный набор типов, который понимает каждый язык .NET Core;

- Языковая интеграция: .NET Core поддерживает наследование, обработку исключений и отладку кода на разных языках программирования.

Например, вы можете определить базовый класс в *C#* и расширить этот тип в *Visual Basic*;

- Обширная библиотека базовых классов: содержит тысячи предопределённых типов, позволяющие создать свои библиотеки, простые консольные приложения, графические приложения и веб-сайты промышленного уровня;

- Упрощённая модель развёртывания: библиотеки *.NET Core* не регистрируются в системный реестр. Платформа *.NET Core* позволяет использовать несколько версий фреймворка, гармонично сосуществующих на одной машине;

- Расширенная поддержка командной строки: интерфейс командной строки (*CLI*) это кроссплатформенный набор инструментов для разработки и сборки приложений *.NET Core*;

Основные компоненты *.NET*:

- 1) *Common Type System* или *CTS*. Спецификация полностью описывает все возможные типы данных и все программные конструкции, поддерживаемые средой выполнения. Определяет, как эти сущности могут взаимодействовать между собой, и описывает, как они должны быть представлены в формате метаданных;

- 2) *Common Language Runtime* – исполняющая среда для байт-кода *CIL* (*MSIL*), в который компилируются программы, написанные на *.NET*-совместимых языках программирования. Основная задача *CLR* – автоматическое обнаружение, загрузка и управление типами *.NET*. Также среда *CLR* заботится о ряде низкоуровневых деталей – управление памятью, обработка потоков, выполнение разных проверок, связанных с безопасностью;

- 3) *Common Language Specification* является спецификацией, связанной с *CTS*, которая определяет подмножество общих типов и программных конструкций, с которыми могут работать все языки программирования *.NET Core*;

- 4) *Base Class Libraries* представляет набор библиотек базовых классов, доступных для всех пользователей *.NET*. *BCL* не только инкапсулирует различные примитивы, но также обеспечивает поддержку ряда сервисов, необходимых для работы большинству реальных приложений.[1, с. 3-12]

### 1.3 Язык программирования *C#*

*C#*, как объектно-ориентированный язык, имеет схожую с другими языками программирования функциональность. Этот язык поддерживает наследование, полиморфизм, инкапсуляцию, перегрузку операторов, статическую типизацию. Объектно-ориентированный метод позволяет,

разбивая одну большую задачу на множество мелких, создавать гибкие и расширяемые приложения.

Понятие строго типизированного языка программирования – это неформальный термин, говорящий о сравнительной способности системы типов языка. Строгого общепринятого определения для данного термина не существует. Для C# система типов достаточно богата, чтобы выражать связи между данными, но, когда речь заходит об отношениях между типами, то тут уже возникают вопросы. Тем не менее, язык C# по большей мере является типобезопасным: тип выражения, определённый на этапе компиляции, гарантирован статическим анализом. Исключениями являются явные операции преобразования типа и специальный тип *dynamic*, а также небезопасный код, который может привести к компрометации типобезопасности.

Особенности C#:

- 1) Автоматическая сборка мусора;
- 2) Значимые типы, допускающие значение *null*;
- 3) Обнаружение и обработка ошибок;
- 4) Синтаксис языка запросов к источнику данных (*LINQ*);
- 5) Все типы наследуются от базового типа *object*.

Программы на C# выполняются в среде .NET, виртуальной системе выполнения, называемой CLR, и наборе библиотек классов.

Исходный код, написанный на C#, компилируется в промежуточный язык (IL), соответствующий спецификации CLI. Код и ресурсы IL, такие как растровые изображения и строки, хранятся в сборке, обычно с расширением *.dll*. Сборка содержит манифест, предоставляющий информацию о типах, версии и культуре сборки.

Когда программа C# выполняется, сборка загружается в среду CLR. CLR выполняет компиляцию *Just-In-Time (JIT)* для преобразования кода IL в собственные машинные инструкции. CLR предоставляет другие службы, связанные с автоматической сборкой мусора, обработкой исключений и управлением ресурсами. Код, выполняемый средой CLR, иногда называют «управляемым кодом». «Неуправляемый код» компилируется в собственный машинный язык, предназначенный для конкретной платформы.

Определяет структуру и поведение любых данных в C#. Объявление типа может включать его члены, базовый тип, интерфейсы, которые он реализует, и операции, разрешенные для этого типа. *Переменная* – это метка, которая ссылается на экземпляр определенного типа.

В C# существуют две разновидности типов: *ссылочные типы* и *типы значений*. Переменные типа значений содержат непосредственно данные, а в переменных ссылочных типов хранятся ссылки на нужные данные, которые именуются объектами. Две переменные ссылочного типа могут ссылаться на



один и тот же объект, поэтому может случиться так, что операции над одной переменной затронут объект, на который ссылается другая переменная. Каждая переменная типа значения имеет собственную копию данных, и операции над одной переменной не могут затрагивать другую (за исключением переменных параметров *ref* и *out*).

Идентификатор – это имя переменной. Идентификатор – это последовательность символов Юникода без пробелов. Идентификатор может быть зарезервированным словом C#, если он имеет префикс @. При взаимодействии с другими языками в качестве идентификатора может быть полезно использовать зарезервированное слово.

Типы значений в C# делятся на простые типы, типы перечислений, типы структур, типы, допускающие значение *NULL*, и типы значений кортежей. Ссылочные типы в C# подразделяются на типы классов, типы интерфейсов, типы массивов и типы делегатов.

Основными понятиями организации в C# являются программы, пространства имен, типы, члены и сборки. В программе объявляются типы, которые содержат члены. Эти типы можно организовать в пространства имен. Примерами типов являются классы, структуры и интерфейсы. К членам относятся поля, методы, свойства и события. При компиляции программы на C# упаковываются в сборки. Сборки обычно имеют расширение *.exe* файла или *.dll*, в зависимости от того, реализуют ли они *.exe* или библиотеки соответственно [5].

## 1.4 Платформа пользовательского интерфейса *WFA*

*Windows Forms* – это современная система пользовательского интерфейса для *Windows*. Она обеспечивает один из самых эффективных способов создания классических приложений с помощью визуального конструктора в *Visual Studio*. Такие функции, как размещение визуальных элементов управления путем перетаскивания, упрощают создание классических приложений.

В *Windows Forms* можно разрабатывать графически сложные приложения, которые просто развертывать, обновлять, и с которыми удобно работать как в автономном режиме, так и в сети. Приложения *Windows Forms* могут получать доступ к локальному оборудованию и файловой системе компьютера, на котором работает приложение.

*Windows Forms* – это технология пользовательского интерфейса для *.NET*, представляющая собой набор управляемых библиотек, которые упрощают выполнение стандартных задач, таких как чтение из файловой системы и запись в нее. С помощью среды разработки, такой как *Visual Studio*, можно создавать интеллектуальные клиентские приложения *Windows Forms*,

которые отображают информацию, запрашивают ввод пользователя и взаимодействуют с удаленными компьютерами по сети.

В *Windows Forms* форма — это визуальная поверхность, на которой выводится информация для пользователя. Обычно приложение *Windows Forms* строится путем добавления элементов управления в формы и создания кода для реагирования на действия пользователя, такие как щелчки мыши или нажатия клавиш. Элемент управления — это отдельный элемент пользовательского интерфейса, предназначенный для отображения или ввода данных.

При выполнении пользователем какого-либо действия с формой или одним из ее элементов управления создается событие. Приложение реагирует на эти события, как задано в коде, и обрабатывает события при их возникновении.

В *Windows Forms* предусмотрено множество элементов управления, которые можно добавлять в формы. Например, элементы управления могут отображать текстовые поля, кнопки, раскрывающиеся списки, переключатели и даже веб-страницы. Если предусмотренные элементы управления не подходят для ваших целей, в *Windows Forms* можно создавать собственные пользовательские элементы управления с помощью класса *UserControl*.

В *Windows Forms* имеются многофункциональные элементы управления пользовательского интерфейса, позволяющие эмулировать функции таких сложных приложений, как *Microsoft Office*. С помощью элементов управления *ToolStrip* и *MenuStrip* вы можете создавать панели инструментов и меню, которые содержат текст и изображения, отображают подменю и размещают другие элементы управления, такие как текстовые поля и поля со списками.

Используя функцию перетаскивания конструктора *Windows Forms* в *Visual Studio*, можно легко создавать приложения *Windows Forms*. Просто выделите элемент управления с помощью курсора и поместите его на нужное место в форме. Для преодоления трудностей, связанных с выравниванием элементов управления, конструктор предоставляет такие средства, как линии сетки и линии привязки. С помощью элементов управления *FlowLayoutPanel*, *TableLayoutPanel* и *SplitContainer* можно гораздо быстрее создавать сложные макеты форм.

Наконец, если нужно создать свои собственные элементы пользовательского интерфейса, пространство имен *System.Drawing* содержит широкий набор классов, необходимых для отрисовки линий, кругов и других фигур непосредственно на форме.

Архитектура *WPF* изображена на рисунке 1.1.

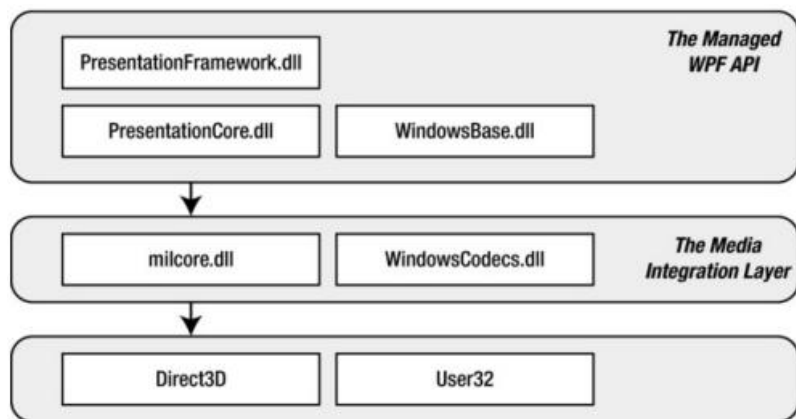


Рисунок 1.1 – Архитектура *WPF*

На рисунке 1.1 присутствуют следующие компоненты:

- 1) *PresentationFramework.dll*: Содержит типы верхнего уровня, которые представляют собой окна, панели, кнопки и другие элементы управления;
- 2) *PresentationCore.dll*: содержит базовые типы, такие как *UIElement* и *Visual*, от которых наследуются все формы и элементы управления;
- 3) *WindowsBase.dll*: содержит ещё больше базовых типов, которые можно использовать и вне *WPF*, например: *DispatcherObject* и *DependencyObject*, которые систему для зависимых свойств;
- 4) *Milcore.dll*: ядро системы отображения *WPF* и основа уровня интеграции мультимедиа(*MIL*);
- 5) *WindowsCodecs.dll*: низкоуровневая библиотека, обеспечивающая поддержку различных форматов медиафайлов;
- 6) *Direct3D*: низкоуровневая библиотека, через которую обрабатывается вся графика *WPF*;
- 7) *User32*: используется для определения того, какая программа получает ресурсы. Не играет роли в отрисовке элементов управления.

Самый важный фактор заключается в том, что вся графика вне зависимости от того какая на машине установлена видеокарта, отображается через *Direct3D* [2, с. 11].

## 1.5 Графическая библиотека *DirectX*

*DirectX* – это набор библиотек, которые обеспечивают низкоуровневый доступ к аппаратным компонентам, таким как видеокарты, звуковая карта и память. Если говорить простым языком, то *DirectX* позволяет играм общаться с видеокартой. Во времена *DOS* игры имели прямой доступ к видеокартам и материнской плате. Но в *Windows95* доступ к низкоуровневому оборудованию был ограничен в качестве меры безопасности. Это означало,

что игры больше не могли взаимодействовать с низкоуровневыми аппаратными функциями и это было проблемой. Поэтому, чтобы облегчить получение доступа к ресурсам видеокарты был представлен *DirectX* – посредник между игрой и видеокартой [3, с. 39-42].

В состав *DirectX* входят:

- 1) *DirectDraw* – для быстрого доступа к видеопамяти;
- 2) *DirectSound* – для вывода аудиоинформации на звуковую карту;
- 3) *DirectPlay* – для организации многопользовательской работы через модем, локальную сеть или *Internet*;
- 4) *DirectInput* – для обработки ввода информации с клавиатуры, мыши или джойстика;
- 5) *Direct3D* – ядро поддержки трёхмерной графики, используемое совместно с *DirectDraw*.

Архитектура *DirectX* представлена на рисунке 1.2.

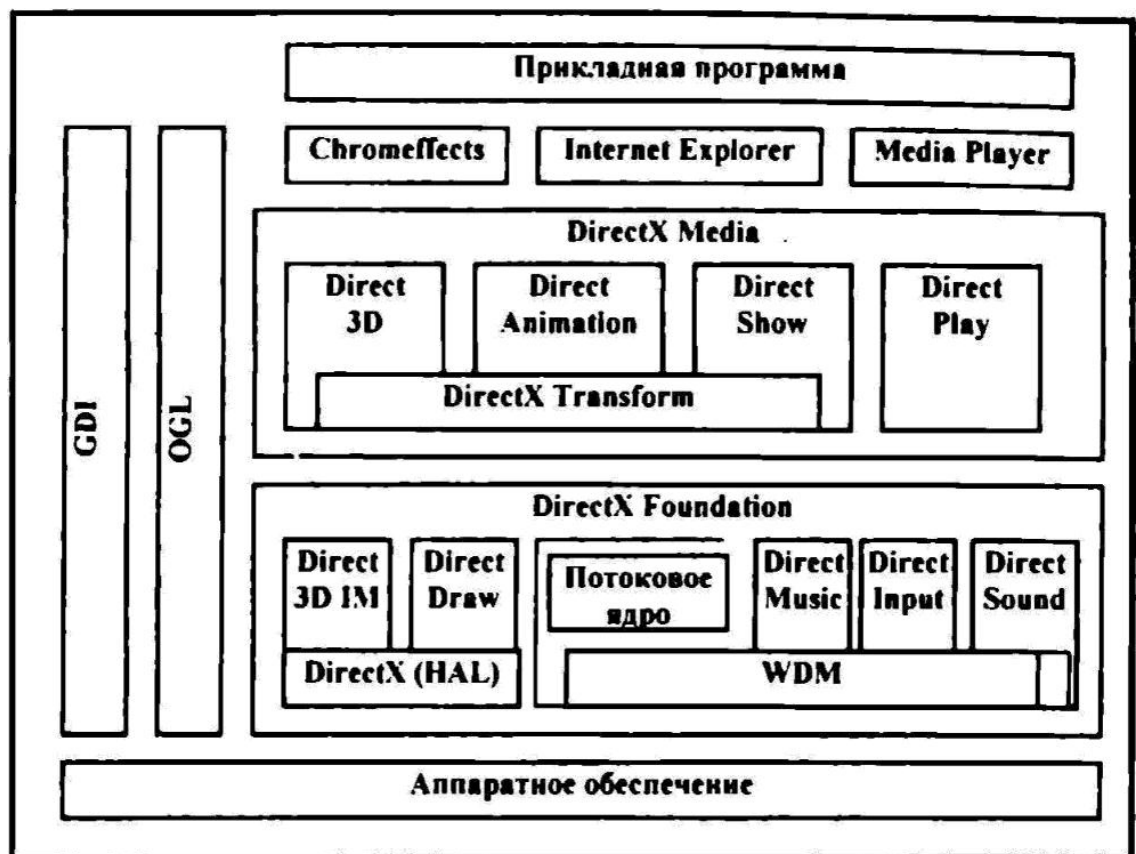


Рисунок 1.2 – Архитектура *DirectX*

## 1.6 Отличительные особенности технологии *DirectX* в сравнении с *OpenGL*

Технологии *DirectX* и *OpenGL* являются конкурирующими интерфейсами. Они оба могут использоваться для визуализации 3D и 2D компьютерной графики. Отсюда можно выделить одно из основных преимуществ *DirectX*: наличие множества различных компонентов, которые позволяют пользователю работать не только с графикой, но и со звуком, текстом, вычислениями с использованием графических процессоров и памятью компьютера.

Представленные интерфейсы прикладного программирования имеют множество различий. Одним из них можно считать поддержку расширений. Так как *DirectX* является проприетарным продуктом, то и изменения в него вносятся исключительно самой компанией разработчиком – *Microsoft*. *OpenGL*, в свою очередь, является свободным API, распространяющийся под лицензией. *OpenGL* представляет собой открытый стандарт, однако некоторые его функции запатентованы. Правки в интерфейс и расширение *OpenGL* осуществляются всем сообществом, в которое входят и ключевые производители видеокарт: *AMD* и *NVIDIA*. В конечном итоге наиболее важные расширения становятся частью основной спецификации. Каждая новая версия *OpenGL* представляет из себя старую версию с добавлением некоторых новых расширений. В то же время новые функции по-прежнему доступны в виде расширений.

Еще одно существенное различие двух технологий – кроссплатформенность. Проприетарный *DirectX* реализован официально лишь в семействе операционных систем *Windows*, включая те версии, которые используются в семействе игровых консолей *Xbox*.

*OpenGL* имеет реализации, доступные на многих платформах, включая *Microsoft Windows*, *Unix*-системы, такие как *MacOS*, *Linux*. *Nintendo* и *Sony* разработали свои собственные библиотеки, которые похожи, но не идентичны *OpenGL*. *OpenGL* было выбрано в качестве основной графической библиотеки для *Android*, *BlackBerry*, *iOS* и *Symbian* в форме *OpenGL ES*.

*DirectX* основан на технологии *COM*, одним из преимуществ которой является то, что API можно использовать на любом *COM*-совместимом языке, в частности: *Object Pascal (Delphi)*, *C++*, *C#* и др.

*OpenGL* – спецификация, реализованная на языке программирования *C*, которая может быть использована и на других языках. Как API, *OpenGL* не зависит ни от одной функции языка программирования и может быть вызван практически из любого языка. В *OpenGL* используется так называемая машина состояний (конечный автомат). Результат вызовов функций *OpenGL* зависит от внутреннего состояния, и может изменять его. В *OpenGL*, чтобы получить доступ к конкретному объекту (например, текстуре), нужно сначала выбрать его в качестве текущего функцией *glBindTexture*, а затем уже можно влиять на объект, например, задание содержимого текстуры осуществляется вызовом *glTexImage2D*.

На рынке профессиональной графики более востребованным считался *OpenGL*, нежели *DirectX*, в то время как набор интерфейсов прикладного программирования от *Microsoft* считался наиболее пригодным в основном для компьютерных игр. Однако, на данный момент как *OpenGL*, так и *DirectX* имеют достаточно большое перекрытие в функциональности, которое может быть использовано для большинства общих целей, причем операционная система часто является главным критерием, диктующим, что используется: *DirectX* является общим выбором для *Windows*, а *OpenGL* – почти для всего остального.

При рассмотрении игровой индустрии можно с уверенностью убедиться, что большинство производителей игровых приложений отдают свое предпочтение технологии *DirectX* благодаря гибкости и расширенному набору библиотек. Игр, где присутствует полноценная поддержка *OpenGL* не так уж и много. Производители игровых приставок оснащают свои продукты собственными *API* для максимизации производительности, что делает сравнение *OpenGL* и *DirectX* актуальным лишь для платформы ПК.

Таким образом, менее продвинутый и реже обновляемый *DirectX* является наиболее предпочтительным для тех, кто использует операционную систему *Windows* и стремится получить максимальную производительность во время игрового процесса.

### 1.7 *SharpDX* как оболочка *DirectX*

*SharpDX* является низкоуровневой оболочкой *DirectX*. Большая часть функциональности *DirectX* предоставляется через *COM*-интерфейсы. *IUnknown* является базовым интерфейсом для *COM*-объектов и предоставляет следующие методы:

- *Int AddReference()*;
- *Int ReleaseReference()*;
- *HRESULT QueryInterface(ref Guid interfaceGuid, out IntPtr pComObj)*.

Каждый *COM*-объект содержит внутренний счетчик ссылок и уничтожается только тогда, когда вызывается *ReleaseReference()* и счетчик становится равным 0. Все *COM*-объекты *SharpDX* наследуются от *ComObject*, который реализует интерфейс *IDisposable*. Метод *Dispose* на самом деле является сокращением метода *COM ReleaseReference()*. *SharpDX ComObject* предоставляет базовый *COM*-указатель через свойство *NativePointer*. В большинстве ситуаций этот указатель не нужно использовать напрямую. Также нужно обратить внимание на то, что многие свойства и методы *DirectX* возвращают *COM*-объекты. В *SharpDX*, в большинстве случаев, если эти методы будут вызваны, то они вернут другой экземпляр объекта *.NET*,

даже если исходный *COM*-указатель тот же, но вызов *COM*-метода увеличивает ссылку на возвращаемый *COM*-объект и должен быть утилизирован/освобождён.

Почти все *API*-интерфейсы *Windows* используют целое число в качестве возвращаемого значения для всех методов и функций. Обычно его называют *HRESULT*.

В зависимости от значения *HRESULT*:

- $< 0$  означает, что произошла ошибка;
- $= 0$  означает, что метод выполнен успешно.
- $> 0$  в редких случаях используется для возврата статуса отсутствия ошибки в зависимости от метода.

Несмотря на то, что некоторые общие значения *HRESULT* заранее определены, большинство *API DirectX* имеют собственный набор кодов ошибок.

В *SharpDX* большинство методов автоматически создают *SharpDXException*, если *HRESULT*  $< 0$ .

Часто *HRESULT* недостаточно для отладки вызова метода. Но, к счастью, большинство *API DirectX* предоставляют уровень отладки, который можно активировать через определённый метод. [4]

## 1.8 Требования к игровому приложению

В результате анализа вида компьютерных игр, информационных технологий и программных средств, позволяющих разработать игровое приложение, были выдвинуты следующие требования для разработки:

- создать игровое приложение «Экшен-платформер» для двух игроков;
- использование платформы пользовательского интерфейса *Windows Forms*;
- использование шаблона проектирования «фабричный метод» для реализации бонусов;
- использование шаблона проектирования «декоратор» для временного изменения улучшения характеристик игроков;
- использование спрайтовой графики *DirectX API* для отображения персонажей и объектов на игровом поле;

## 2. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ «Экшен-платформер»

### 2.1 Паттерны проектирования «Фабричный метод» и «Декоратор» как средства реализации приложения

Шаблон проектирования «фабричный метод» является порождающим паттерном (он определяет интерфейс для создания объекта), но оставляет подклассам задачу о создании конкретного класса. Данный паттерн использует механизм полиморфизма, при котором классы всех конечных типов наследуются от одного базового класса.

Преимущества и недостатки «фабричного метода»:

- классы фабрик избавляют программиста от необходимости встраивать в код зависящие от приложения классы. Взаимодействие происходит только с интерфейсом класса, поэтому он может работать с любыми реализациями конкретных классов;
- создание объектов с помощью паттерна внутри класса является довольно гибким решением. Класс фабрики создает в подклассах операции-зацепки для предоставления расширенной версии объектов;
- клиентом для создания лишь одного нового объекта, наследуемого от интерфейса, необходимо создать и наследника класса фабрики. В данном случае порождение наследников оправдано, т.к. клиенту необходимо создавать наследников фабрики [6]. Если такой необходимости нет, тогда не следует вынуждать клиента работать с дополнительным уровнем наследников.

В игре «Экшен-платформер» паттерн применим к классам, которые динамически добавляются и удаляются со сцены, что требует постоянного создания новых объектов. Такими классами в игре являются эффекты, накладываемые на персонажей в результате игры. Поэтому существует необходимость в создании абстрактных классов фабрик и их наследников с реализацией каждого объекта.

Такая структура позволит вносить изменения в проект без возникновения проблем. На рисунке 2.1 представлена схема реализации паттерна «фабричный метод» в нашем игровом приложении.

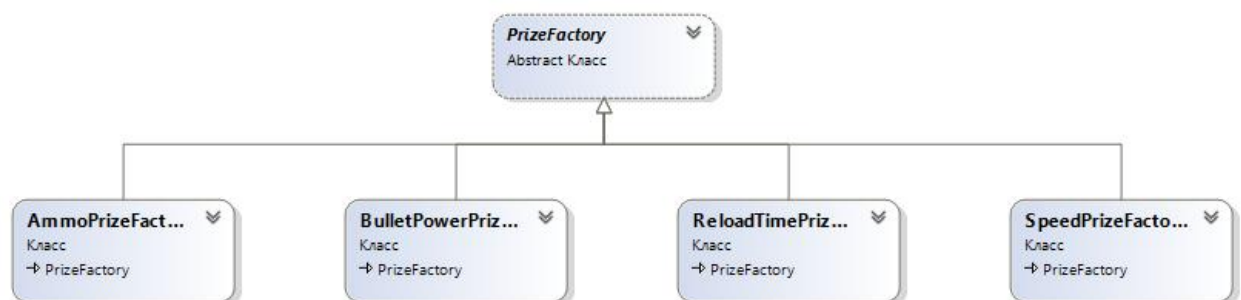


Рисунок 2.1 – Схема реализации паттерна «фабричный метод»



В игре «Экшен-платформер» часто необходимо расширять функциональность эффектов. Для каждого объекта потребуется создать класс базового декоратора. Он предоставит механизм подключения компонента и обеспечит переадресацию всех методов и свойств [7]. Затем с помощью наследования создать классы конкретных декораторов, реализовав в них только методы и свойства с изменённой функциональностью. Такая структура позволит получать различные объекты с отличающимися параметрами путем декорирования. В нашем случае декоратор применяется для изменения характеристик игрока.

На рисунке 2.2 представлена схема реализации паттерна «декоратор» для изменения характеристик игрока.

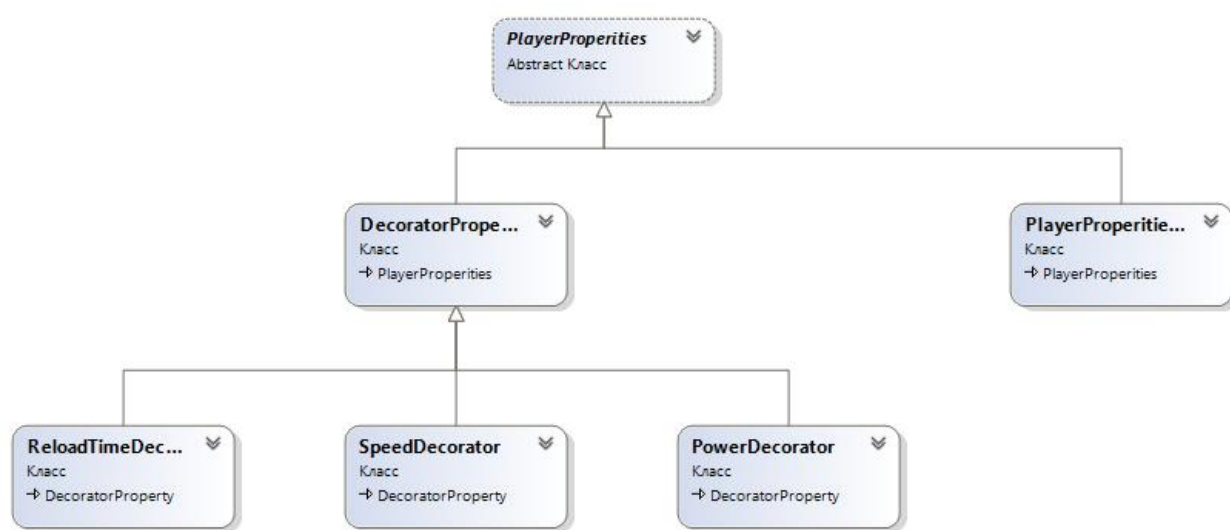


Рисунок 2.2 – Схема реализации паттерна «декоратор»

## 2.2 Проектирование приложения методом декомпозиции

Для реализации игровой логики требуется создать игровой движок – проект, разделяющий центральные компоненты игры (например, окно отображения и его настройки, ввод с клавиатуры, игровые объекты, имеющие определённый набор компонентов для отображения и корректного поведения на сцене) от её логики. Проект движка определяется библиотекой классов и не имеет графических компонентов для визуальной разработки, он содержит технологии перемещения по вертикали и горизонтали, отображение графики и анимацию, распознавание столкновений твердых тел и «триггеров» [8].

Игровой движок предполагает наличие определенного алгоритма работы. В нашем случае алгоритм состоит из подготовки ресурсов для корректной работы приложения, циклического рендеринга (воспроизведения, отрисовки) объектов на протяжении всей игры и высвобождения ресурсов

перед закрытием. На рисунке 2.3 представлен алгоритм работы игрового движка.



Рисунок 2.3 – Алгоритм работы игрового движка

Для проектирования архитектуры игрового движка используется процесс декомпозиции. Декомпозиция – это универсальный научный метод решения сложных задач, основанный на использовании структуры задачи и её разбиении на более простые подзадачи [9]. Необходимо спроектировать и разработать подзадачи, а после объединить их в общее решение. Каждый элемент декомпозиции является уникальным, что позволяет сформировать и отличить составляющие одного объекта от другого.

Процесс декомпозиции позволил разработать обобщённый алгоритм работы проекта игровой логики. Ключевым элементом в алгоритме является сцена, которая создает, отображает, динамически добавляет и удаляет элементы в игре. Также сцена отслеживает конец игры. В алгоритме каждый блок действий представляет собой этап разработки, который разбивается на подзадачи. На рисунке 2.4 представлен алгоритм работы проекта игровой логики.



Рисунок 2.4 – Алгоритм работы проекта игровой логики

Каждый этап разработки позволяет получить информацию о необходимых ресурсах и действиях и, при необходимости, сведения о реализации данных действий. После сбора информации можно чётко обозначить цель и последовательность небольших задач для её достижения. Также, оценивая каждый этап разработки, легче сформировать структуру приложения.

Главным преимуществом использования декомпозиции является решение задачи с минимальными затратами и максимальной гибкостью. Каждый этап можно протестировать и выявить ошибки, а после заняться рефакторингом кода. Это позволяет избежать возникновения ошибок при объединении этапов.

Средство разработки *SharpDX* позволяет создавать собственное окно отображения. Однако для вывода статистики потребуется интегрировать его в *Windows Forms* и создать элементы для отображения данных из проекта игровой логики [10]. Элементы выводят количество набранных очков, количество пунктов здоровья каждого из игроков, выбранное оружие, количество патронов в выбранном оружии, эффект, действующий на персонажа и победителя игры. Для оповещения об изменении состояния

игровых объектов необходимо использовать делегаты. Также, на *Windows Forms* придётся возложить создание экземпляров игрового движка и игровой логики и высвобождение ресурсов этих элементов при необходимости.

## 2.3 Структура данных приложения

В данной подглаве будут подробно рассмотрены ключевые проекты игрового приложения, его классы, а также то, что они в себя включают.

Проект *EngineLibrary*, являющийся в нашем случае игровым движком, включает в себя следующие классы:

- *RenderingApplication* – основной компонент проекта *EngineLibrary*. В конструкторе данного класса создается экземпляр класса *RenderForm*, представляющий собой окно рендеринга, и инициализация свойств: название окна, размер окна, возможность изменять размер окна, режим отображения окна верхнего уровня;

- *RenderingSystem* – отвечает за инициализацию необходимых настроек и создание целевого окна рендеринга. Данный класс имеет статические методы *LoadBitmap* и *LoadAnimation*, загружающие изображение и последовательность изображений соответственно;

- *Scene* – абстрактный класс, содержащий 3 списка игровых объектов: текущие игровые объекты рендеринга, объекты для добавления в следующем кадре на сцену и объекты для удаления в следующем кадре со сцены – и имеет следующие методы:

1. *CreateGameObjectsOnScene* – абстрактный метод создания игровых объектов перед запуском рендеринга;

2. *DrawScene* – метод рендеринга игровых объектов на сцене;

3. *AddRenderGameObjects* и *RemoveRenderGameObjects* – методы для добавления и удаления игровых объектов в следующем кадре;

4. *EndScene* – виртуальный метод, оповещающий об окончании рендеринга сцены.

- *GameObject* – класс, описывающий каждую игровую сущность в данном проекте;

- *TransformComponent* – класс, описывающий положение и размер игрового объекта на сцене, имеет методы передвижения, осуществляет добавление гравитации к игровым объектам;

- *SpriteComponent* – хранит текущее изображение игрового объекта для рендеринга и методы для управления анимацией;

- *Animation* – хранит последовательность изображений, выполняющихся в зависимости от времени проигрывания и наличия заикленности;

- *ColliderComponent* – рассчитывает пересечения игровых объектов, обладающих данным компонентом, между собой на основе размера и смещения границ твердого тела, обновляемых методом *UpdateBounds* каждый кадр. Также имеет перегруженный метод *CheckIntersection*,

позволяющий проверять пересечения игровых объектов, имеющие некоторые имена игровых объектов или определённый сценарий выполнения. Методы *GetNormal* и *GetProjection* создают нормали и проекции для проверки пересечений;

- *ObjectScript* – абстрактный класс, представляющий собой сценарий выполнения игрового объекта на сцене. Имеет абстрактный метод *Start*, инициализирующий игровой объект при создании, а также абстрактный метод *Update*, описывающий поведение данного объекта в каждом кадре.

Таким образом, получившийся проект *EngineLibrary* позволяет решать задачу создания сцены и игровых объектов на ней. Для реализации игровой логики используется компонент *ObjectScript*.

На рисунке 2.5 представлена схема проекта *EngineLibrary*.

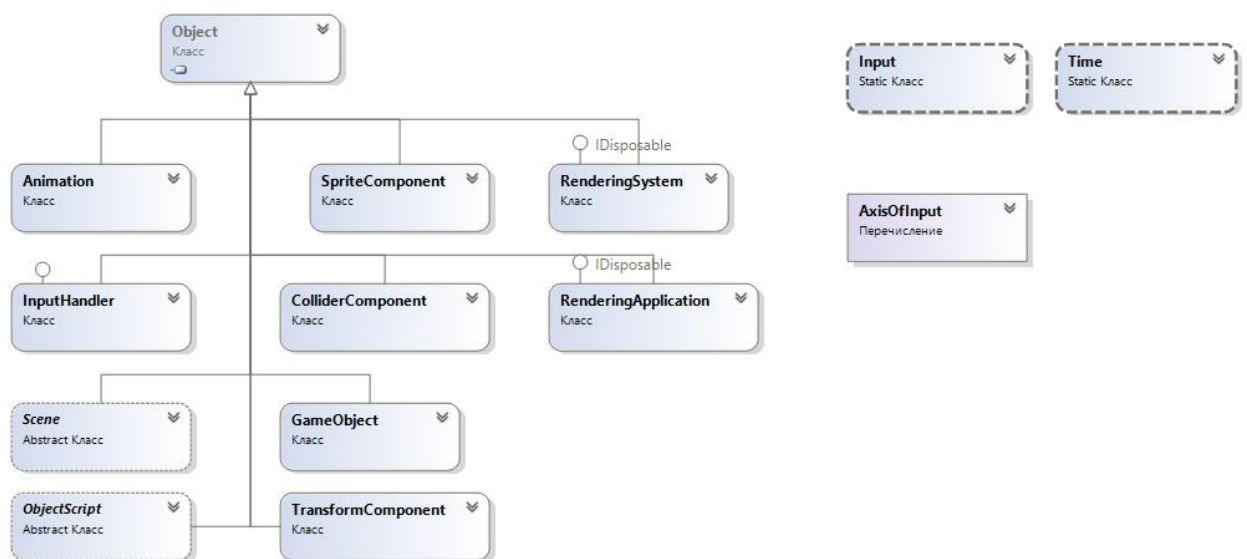


Рисунок 2.5 – Схема проекта *EngineLibrary*

Проект *GameLibrary* содержит следующие классы:

- *PlatformerScene* – класс, описывающий сцену с платформой. Он гарантирует, что экземпляр класса единственный, и позволяет взять ссылку на сцену в любом месте проекта;

- *SpawnManager* – класс, наследуемый от *ObjectScript*, отвечает за генерацию призов. Создание различных видов призов, отвечающих за изменение основных характеристик игрока, реализуется с помощью шаблона проектирования «фабричный метод»;

- *Player* – класс, наследуемый от *ObjectScript*, позволяет управлять игровым объектом персонажа. Он также содержит информацию о текущем количестве очков, сборе аптечек, призов. В методе *Update* вычисляется передвижение игрового объекта персонажа по сцене, а также просчитываются столкновения со стенами, взаимодействие с лестницами и платформами;

- *PlayerConstructor* – базовый класс, содержащий в себе конструктор, который в свою очередь позволяет создать игрока с оружием в руках;
- *PlayerProperties* – абстрактный класс, содержащий логику характеристик игрока. Содержит виртуальный метод *SetProperty*, позволяющий задать изменение характеристикам игрока, виртуальный метод *UpdateTime*, который отслеживает время действия эффектов и деактивирует их с помощью абстрактного метода *DeactivateProperties*, требующего реализации у наследуемых классов, по истечении этого самого времени;
- *PlayerPropertiesStandart* – класс, наследуемый от *PlayerProperties*, содержит сведения о базовых характеристиках игрока;
- *DecoratorProperty* – класс, наследуемый от *PlayerProperties*, декорирующий характеристики игрока. Его классы-наследники *PowerDecorator*, *ReloadTimeDecorator*, *SpeedDecorator* задают величины изменений соответствующих названиям классов характеристик;
- *TypeProperty* – перечисление всех существующих характеристик игрока;
- *DamageGun* – реализует абстрактный класс *Gun*. Появление оружия реализовано с помощью паттерна «фабричный метод»;
- *Bullet* – абстрактный класс, который задает и просчитывает перемещение и столкновение пули соответственно в методе *Update*. При столкновении с игровым объектом стены вызывается метод их разрушения, что позволяет проходить сквозь них, а при столкновении с персонажем вызывается метод *PlayerInteraction*, реализуемый в классах наследниках;
- *DamageBullet* – реализует абстрактный класс *Bullet*. Появление пуль реализовано с помощью паттерна «фабричный метод»;
- *Effect* – абстрактный класс, который описывает общую логику игрового объекта в методе *Update*, деактивируя эффект по истечении его времени действия. Класс имеет абстрактные методы *ActivateEffect*, активирующий эффект на определенном персонаже, и *DeactivateEffect*, который отключает его действие на персонажа и удаляет со сцены. *BehaviorOnScene* – абстрактный метод, который расширяет логику поведения игрового объекта на сцене;
- *DamageEffect* – класс, реализующий абстрактный класс *Effect*, наносящий урон, а затем уничтожающий персонажа, который возрождается спустя время. Срабатывание эффекта реализовано с помощью паттерна «фабричный метод».

Оповещения игрового приложения о полученных очках, собранных аптечках, подобранном оружии, действующем эффекте и конце игры осуществляется делегатами, созданными в статическом классе *GameEvents*. Таким образом, мы можем добавить собственные методы отображения игровой статистики в делегаты, вызов которых осуществляется в проекте игровой логики. Делегаты обеспечивают возможность последовательного вызова нескольких методов, а также вызова как статических, так и экземплярных методов.

На рисунке 2.6 представлена схема проекта *GameLibrary*.

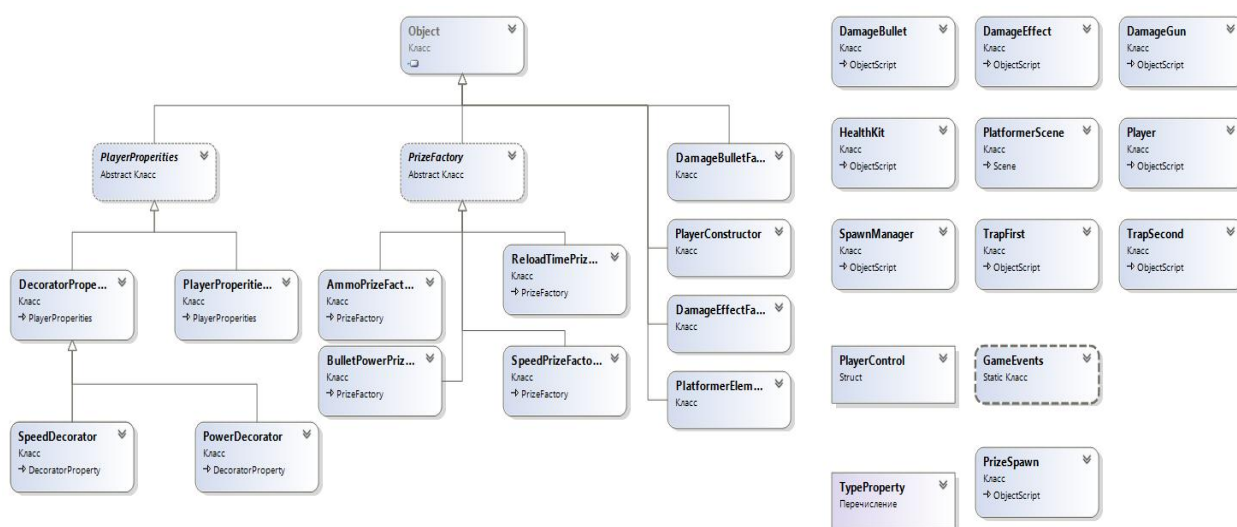


Рисунок 2.6 – Схема проекта *GameLibrary*

### 3. ВЕРИФИКАЦИЯ И АПРОБАЦИЯ ИГРЫ «Экшен-платформер»

#### 3.1 Принцип работы приложения

При запуске игрового приложения открывается проект пользовательского интерфейса *Windows Forms*. В классе *GameWindow* инициализируются компоненты интерфейса и создаются экземпляры классов игрового движка *RenderingApplication* и *PlatformerScene*, которые описывают игровую сцену платформера. Методы класса для отображения статистики подписываются на делегаты проекта игровой логики. Элементы *Image* и *TextBlock* отображают статистику в игре, а элемент *Button* позволяет запустить игру. Экземпляр окна визуализации *RenderForm* из класса *RenderingApplication* транслируется в пользовательский интерфейс с помощью элемента *WindowsFormsHost*, который позволяет использовать элементы *Windows Forms*.

Нажатие кнопки «*Play*» в пользовательском интерфейсе устанавливает и инициализирует игровую сцену в классе *RenderingApplication*. Во время инициализации создаются игровые персонажи, бонусы и уровень. Платформер создается из выбранного файла *.bmp*, который содержит структуру экшен-платформера, где цвет каждого пикселя представляет собой элемент игровой сцены. Игровые объекты экшен-платформера создаются в экземпляре класса *PlatformerElementsFactory*. Также сцена сохраняет пустые ячейки в списке для последующего генерирования призов в платформере и имеет методы для добавления и удаления игровых объектов со сцены. Когда бонус удаляется со сцены, его позиция снова добавляется в список генерируемых призов.

После инициализации сцены запускается зацикленная инфраструктура рендеринга, которая визуализирует каждый кадр игрового окна с помощью соответствующего метода *RenderCallback*. Этот метод позволяет игровому приложению обновлять состояние клавиатуры и компонентов времени каждый кадр, а затем отображать текущий кадр путем рендеринга игровых объектов в сцене. Игровые объекты должны быть инициализированы компонентами, необходимыми для их назначения, после чего игровой объект отображается в каждом кадре с использованием метода *Draw*, пока игровой объект существует на сцене. Рендеринг объекта состоит из создания матрицы перемещения, поворота, размера и рисования изображения на основе этой матрицы.

Класс *InputHandler*, экземпляр которого создается в *RenderingApplication*, обрабатывает ответ событий ввода с клавиатуры. Каждый вызов *RenderCallback* обновляет состояние клавиатуры каждый кадр, а нажатия клавиш отслеживаются с помощью свойства *KeyboardState*. Статический класс *Input* упрощает доступ к клавиатуре, но ограничивает функциональность класса *InputHandler*.



Статический класс *Time* позволяет работать со временем. Этот класс хранит текущее время *Time* с момента начала игры и время *DeltaTime*, которое потребовалось для визуализации последнего кадра. Каждый вызов *RenderCallback* обновляет значение свойств *CurrentTime* и *DeltaTime*. Использование *DeltaTime* в качестве множителя при вычитании или сложении значений каждого кадра делает приложение независимым от частоты кадров.

Движение игровых персонажей осуществляется как по горизонтали, так и по вертикали при условии, что игрок столкнулся с лианой, путем изменения координат игрового объекта в зависимости от нажатой клавиши перемещения. Анимация меняется в зависимости от состояния игрового объекта.

Одна из игровых механик – сбор аптек. Класс *HealthKit*, унаследованный от *ObjectScript*, проверяет наличие столкновений с игровыми персонажами в методе *Update*, после чего игровой объект удаляется из списка рендеринга, а значение здоровья игрока увеличивается на единицу, и уведомляет систему об изменении с помощью соответствующий делегат.

При генерации бонусов проверяется возможность их создания в зависимости от времени, прошедшего с момента последней генерации, и если это время прошло, то на сцене создается игровой объект со случайным бонусом. У каждого типа бонусов есть свои шансы на появление.

У игрока есть возможность стрелять, нажимая клавишу огня на клавиатуре. Игровой объект персонажа устанавливается родительским объектом и позволяет оружию перемещаться по сцене относительно персонажа. При стрельбе генерируются пули, которые движутся по горизонтали и имеют методы проверки столкновения с игровыми объектами. Если объект - персонаж игрока, то он получает урон, количество его очков здоровья уменьшается на единицу. В любом случае при столкновении с объектом игровой объект пули удаляется из списка визуализации сцены.

Когда игра заканчивается, элемент управления *WindowsFormsHost* с окном рендеринга закрывается, рендеринг игровых объектов в сцене отключается, и отображаются результаты игры. Закрытие приложения или непредвиденная ошибка высвобождают ресурсы проекта игрового движка.

### 3.2 Результаты тестирования приложения

Решение включает в себя проект под названием *UnitTestGame*, который предназначен для тестирования разработанных библиотек игрового приложения. Он состоит из нескольких тестовых классов, речь о которых пойдет подробнее ниже. В свою очередь, каждый из тестовых классов содержит статические методы класса *Assert*.

Перечисление классов, их назначение и методы:

- *CollisionTest* – класс, предназначенный для тестирования различных видов столкновений. В качестве примера рассмотрен случай, в котором мы

создаем два объекта, описываем их масштабы и изначальное местоположение, и, отталкиваясь от их местоположения и наименования тэга, определяем наличие столкновения. Всё вышеперечисленное осуществляется в методе *TestCollider*;

– *DecoratorTest* – класс, предназначенный для тестирования работы шаблона «декоратор». В нашем игровом приложении данный шаблон применяется для изменения характеристик персонажа. Таким образом, методы *TestSpeedDecorator*, *TestPowerDecorator*, *TestReloadTimeDecorator* осуществляют проверку на соответствие значений одноименных характеристик ожидаемым значениям;

– *MovementTest* – класс, предназначенный для тестирования осуществления движения игрового объекта. Данное тестирование осуществляется на следующем примере: мы создаем игровой объект, задаем его изначальное местоположение, задаем офсет для перемещения, осуществляем перемещение, сравниваем полученные координаты объекта с ожидаемыми. Всё вышеперечисленное осуществляется в методе *TestGameObjectMovement*.

На рисунке 3.1 представлен результат тестирования вышеописанных классов разработанного игрового приложения.

Тестирование	Длительн...	Признаки	Сообщение об ошибке
TestGameApp (17)	6,9 с		
TestGameApp (17)	6,9 с		
TestCollision (3)	27 мс		
TestCollider	13 мс		
TestColliderFirst	14 мс		
TestColliderSecond	< 1 мс		
TestDamage (2)	55 мс		
TestDamageEffect	55 мс		
TestErrorDamageEffect	< 1 мс		
TestDecoratedPrize (4)	6,8 с		
TestMethodFactoryPrizeA...	3 с		
TestMethodFactoryPrizeB...	1,1 с		
TestMethodFactoryPrizeR...	1,2 с		
TestMethodFactoryPrizeS...	1,5 с		
TestFactoryPrize (6)	< 1 мс		
TestErrorPowerDecorator	< 1 мс		
TestErrorReloadTimeDec...	< 1 мс		
TestErrorSpeedDecorator	< 1 мс		
TestPowerDecorator	< 1 мс		
TestReloadTimeDecorator	< 1 мс		
TestSpeedDecorator	< 1 мс		
TestMovement (2)	2 мс		
TestErrorGameObjectMo...	1 мс		
TestGameObjectMoveme...	1 мс		

Рисунок 3.1 – Результаты тестирования классов разработанного игрового приложения

### 3.3 Результаты верификации приложения

При запуске *WFA*-проекта открывается интерфейс игрового приложения. Игроки появляются на сцене в местах, заданных в хранимой структуре экшн-платформер формата *.bmp*.

На рисунке 3.2 изображен интерфейс игрового приложения после запуска программы (до начала игры).

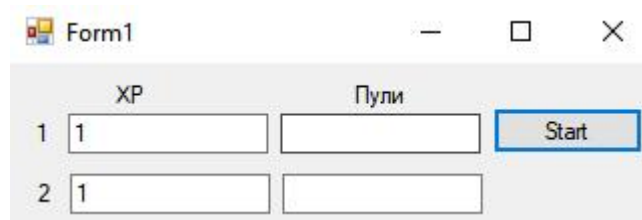


Рисунок 3.2 – Интерфейс игрового приложения

На рисунке 3.3 изображена игра после нажатия на кнопку «Play».

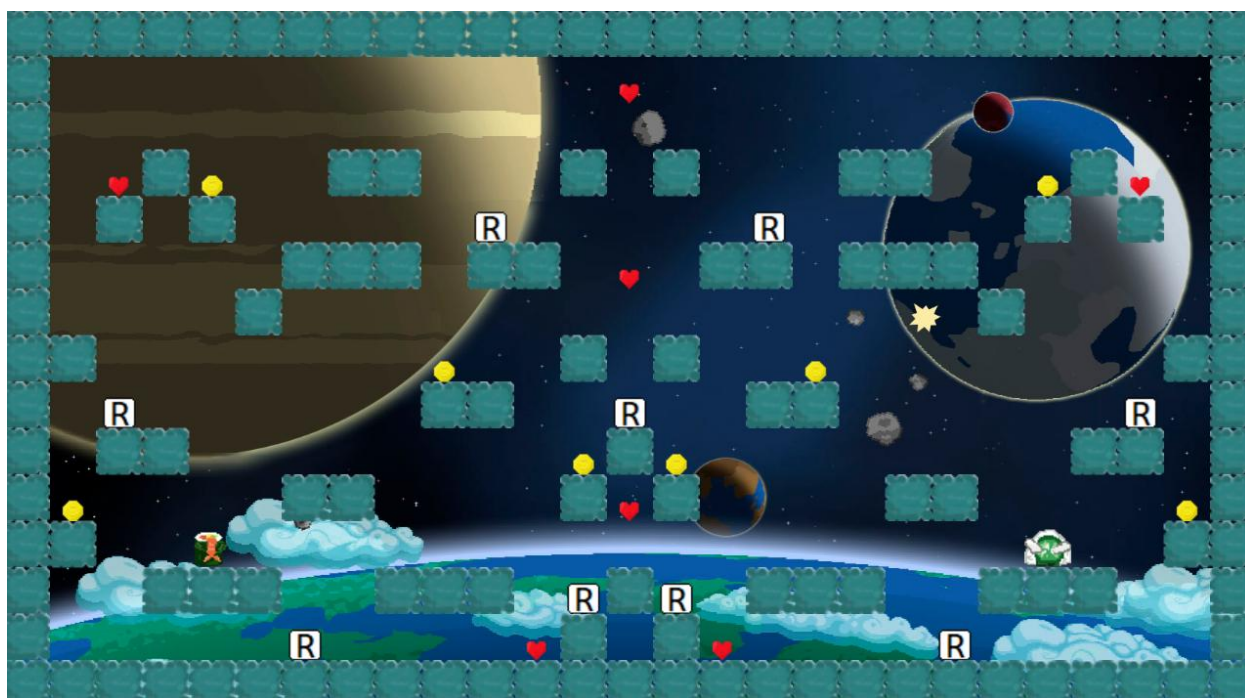


Рисунок 3.3 – Изображение игры в действии

На рисунке 3.4 приведена информация об управлении игровыми персонажами.

Игровой персонаж	Красный игрок	Синий игрок
Передвижение по горизонтали	<i>A/D</i>	<i>KeyLeft/KeyRight</i>
Передвижение по вертикали	<i>W/S</i>	<i>KeyUp/KeyDown</i>
Стрельба	<i>Space</i>	<i>RightShift</i>
Подбор приза	<i>C</i>	<i>LeftSlash</i>

Рисунок 3.4 – Информация об управлении игровыми персонажами

Первый игрок управляет синим персонажем, а второй игрок – красным. Каждый игровой объект персонажа имеет анимации покоя и бега в обоих направлениях. Цель каждого из игроков является убийство друг друга. В процессе игры можно передвигаться по горизонтали и вертикали. Если подойти к бонусу и нажать соответствующую клавишу, персонаж игрока приобретет соответствующие бонусу характеристики, а сам бонус исчезнет из экшен-платформер.

Подробное описание генерирующихся в случайном порядке бонусов на карте:

- с вероятностью в 20 процентов в свободной ячейке экшен-платформер появляется бонус, увеличивающий скорость передвижения персонажа в 2 раза по сравнению с базовой в течение 2-х секунд;
- с вероятностью в 20 процентов в свободной ячейке экшен-платформер появляется бонус, который восстанавливает количество патронов персонажа до начального максимального уровня (25 шт.);
- с вероятностью в 30 процентов в свободной ячейке экшен-платформер появляется бонус, увеличивающий скорость перезарядки персонажа в 2 раза по сравнению с базовой в течение 2-х секунд;
- с вероятностью в 30 процентов в свободной ячейке экшен-платформер появляется бонус, увеличивающий убойную силу выстрела в 2 раза по сравнению с базовой в течение 2-х секунд.

На рисунке 3.5 представлены спрайты элементов экшен-платформер, с которыми можно взаимодействовать.

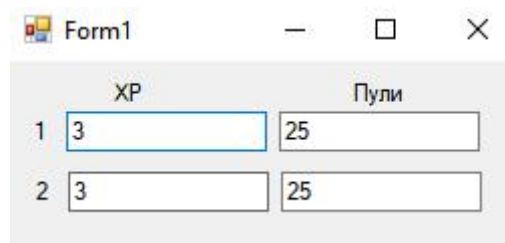


Рисунок 3.5 – Спрайты элементов экшен-платформер, с которыми можно взаимодействовать

Информационная статистика выводится в *WFA*-приложение с помощью делегатов. При столкновении, к примеру, с аптечкой изменяется значение строкового свойства *Text* элемента *TextBlock*, связанного с конкретным игроком. В случае смерти игрового персонажа на него накладывается

соответствующий эффект, данной информацией заполняется свойство *Text* элемента *TextBlock*. После окончания действия эффекта в поле *Text* устанавливается пустая строка. Аналогичными способами выводится и остальная информация.

На рисунке 3.7 представлен пример статистики игрока из пользовательского интерфейса во время игры.



	XP	Пули
1	3	25
2	3	25

Рисунок 3.7 – Пример статистики игрока из пользовательского интерфейса во время игры

Игра заканчивается после убийства оппонента. Победителем является игрок оставшийся в живых и имеющий запас здоровья. Окно рендеринга закрывается, а в окне *WFA*-приложения показывается победитель.

На рисунке 3.8 представлен пользовательский интерфейс приложения после окончания игры.

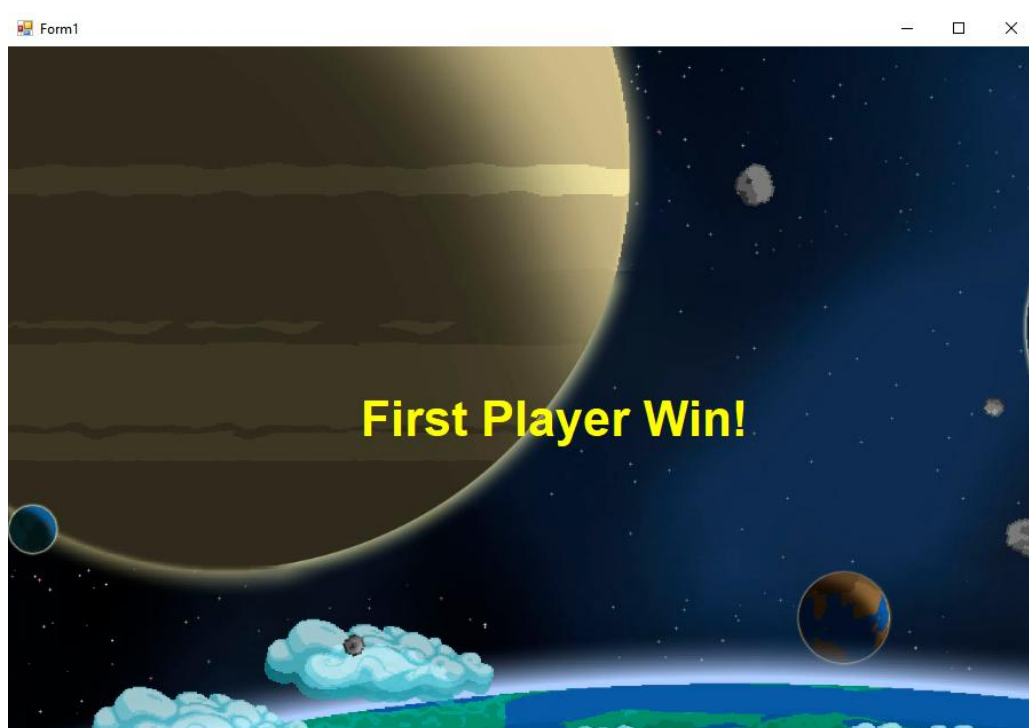


Рисунок 3.8 – Пользовательский интерфейс приложения после окончания игры

Как показано на рисунке выше, после завершения игры пользователи продолжают видеть на информационных панелях свою статистику на момент окончания игры. Для того чтобы повторно сыграть в игру, пользователю необходимо перезапустить приложение.

## ЗАКЛЮЧЕНИЕ

Результат проделанной работы – разработанное игровое приложение для игры двух пользователей на одном экране. Каждый игрок управляет персонажем на платформах с клавиатуры, и их задача – убить друг друга.

Для программной реализации приложения использовался язык программирования *C#* с графической библиотекой *DirectX*. Также были изучены основные принципы работы с *WFA*-приложением. Последующее использование паттернов фабричного метода и декоратора для перспективной расширяемости приложения позволило эффективно реализовать его функционал.

В процессе разработки игрового приложения проводился аналитический обзор игр, относящихся к разным жанрам, например, в которых может подразумеваться наличие платформеров, а также инструментов разработки, достаточных для освоения заданной темы. Затем с помощью декомпозиции определяется структура и алгоритм разработки программного обеспечения. Основными этапами были разработка игрового движка для запуска игры и игровой логики. Разработанные классы были протестированы юнит-тестами, и выявленные по результатам тестирования ошибки были устранены, что гарантирует корректную работу приложения и минимизирует вероятность получения ошибки пользователем. Заключительным этапом стало создание пользовательского интерфейса, который запускает игру, получает и отображает соответствующую статистику, а также интегрирует в неё ранее разработанные проекты.

Игра имеет минимальный порог вхождения, интуитивно понятна и, как следствие, не требует особых навыков. Соревновательные элементы игры гарантируют полное погружение в игровой процесс, получение интересного опыта, оказание положительного влияния на такие спектры развития, как логическое мышление и внимательность, что в совокупности повышает значимость и ценность игры как программного продукта.

Приложение прошло пробную эксплуатацию с положительным результатом – ошибки и недочеты учтены и устранены.

Авторские права на программную часть проекта принадлежат автору курсовой работы.



## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Troelsen, Pro C# 9 with .NET 5: Foundational Principles and Practices in Programming. 10th edition / Andrew Troelsen, Phillip Japikse. – Apress Media, LLC, 2021. – 1383 p.
2. Windows Forms in C# [Электронный ресурс]. – URL: <https://docs.microsoft.com/ruru/dotnet/desktop/winforms/overview/?view=netdesktop-5.0> (дата обращения: 21.03.2022).
3. Петровичев, Е.И. Компьютерная графика / Петровичев Е.И. – М.: Издательство Московского государственного горного университета, 2003. – 207 с.
4. Использование SharpDX [Электронный ресурс]. – URL: <http://sharpdx.org/wiki/usage/> (дата обращения: 21.03.2022).
5. Обзор C# [Электронный ресурс]. – URL: <https://docs.microsoft.com/ruru/dotnet/csharp/tour-of-csharp/> (дата обращения: 21.03.2022).
6. Ч. Петсольд. – *Microsoft Windows Presentation Foundation*, пер. с англ. Матвеев Е. А. – Москва: Русская Редакция; Санкт-Петербург: Питер, 2008 – 944 с.
7. Флёнов М.Е. – *DirectX и C++*. Искусство программирования [2006, *DjVu, RUS*]. Страницы: 1. ... Искусство программирования. Год выпуска: 2006. Автор: Фленов М.Е. Издательство: БХВ-Петербург ISBN: 5-94157-831-8. Формат: *DjVu*.
8. Попов А. А. П58 *DirectX 10* – это просто. Программируем графику на C++. – СПб.: БХВ-Петербург, 2008. – 464 с.: ил. + *CD-ROM* – (Внесерийная) ISBN 978-5-9775-0139-2.
9. Джим Адамс. Программирование ролевых игр с *DirectX* (2-е издание), *Thomson Course Technology PTR*, 2004 ISBN: 1-59200-315-X.
10. Рихтер, Дж. *CLR via C#*. Программирование на платформе *Microsoft .NET Framework 4.5* на языке C#. 4-е издание. / Дж. Рихтер. – СПб: Питер, 2019. – 896 с.