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 .*NET* 6

1.3 Язык программирования *C#* 7

1.4 Платформа пользовательского интерфейса *WFA* 9

1.5 Графическая библиотека *DirectX*  11

1.6 Отличительные особенности технологии *DirectX* в сравнении с *OpenGl........................* 12

1.7 *SharpDX* как оболочка *DirectX* 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, поигрывают 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, c. 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*,* итипы значений кортежей*.* Ссылочные типы в *C*# подразделяютсянатипы классов*,*типы интерфейсов*,*типы массивовитипы делегатов*.*

Основными понятиями организации в *C*# являются **программы**,**пространстваимен**, **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.

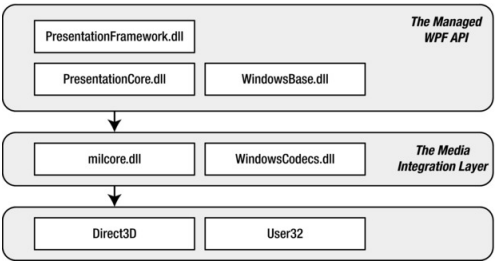


Рисунок 1.1 – Архитектура *WPF*

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 оборудования 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.

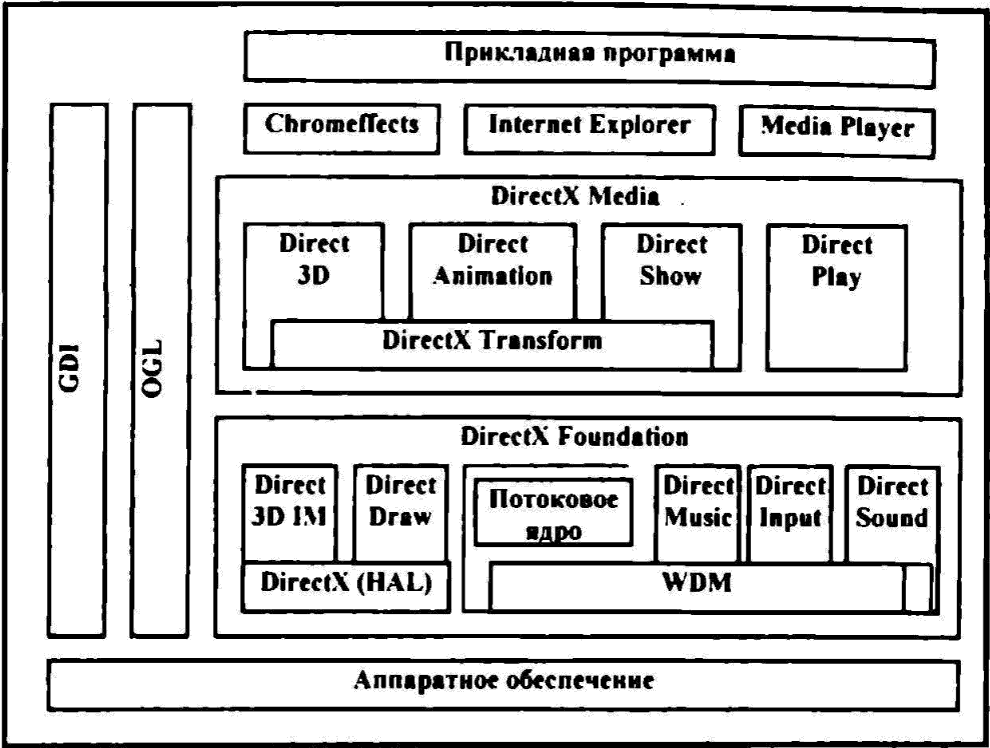
****

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](https://en.wikipedia.org/wiki/Unix)*-системы, такие как *Mac[OS](https://en.wikipedia.org/wiki/OS_X)*, *[Linux](https://en.wikipedia.org/wiki/Linux)*. *Nintendo* и *Sony* разработали свои собственные библиотеки, которые похожи, но не идентичны *OpenGL*. *OpenGL* было выбрано в качестве основной графической библиотеки для *[Android](https://en.wikipedia.org/wiki/Android_(operating_system))*, *[BlackBerry](https://en.wikipedia.org/wiki/BlackBerry)*, *[iOS](https://en.wikipedia.org/wiki/IOS)* и *[Symbian](https://en.wikipedia.org/wiki/Symbian)* в форме *[OpenGL ES](https://en.wikipedia.org/wiki/OpenGL_ES)*.

*DirectX* основан на технологии *COM,* одним и преимуществ которой является то, что *API* можно использовать на любом *COM*-совместимом языке, в частности: *[Object Pascal](https://en.wikipedia.org/wiki/Object_Pascal)* (*[Delphi](https://en.wikipedia.org/wiki/Delphi_(programming_language))*), *С++*, *[C#](https://en.wikipedia.org/wiki/C_Sharp_(programming_language))* и др.

*OpenGL* – спецификация, реализованная на [языке программирования](https://en.wikipedia.org/wiki/Programming_language) *[C](https://en.wikipedia.org/wiki/C_(programming_language))*, 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 производительностьво время игрового процесса.

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

*SharpDX* является низкоуровневой оболочкой *DirectX*. Большая часть функциональнасти *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 счетчит 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 выполнился 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 компьютерах 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;

## 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.

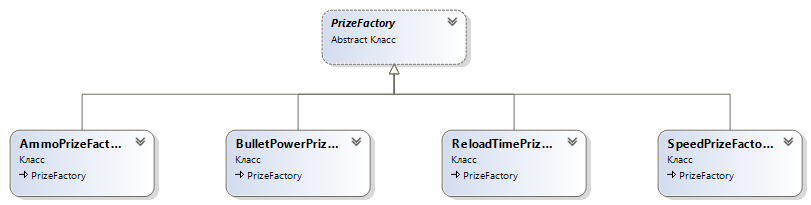


Рисунок 2.1 – Схема реализации паттерна «фабричный метод»

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, реализуя 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.

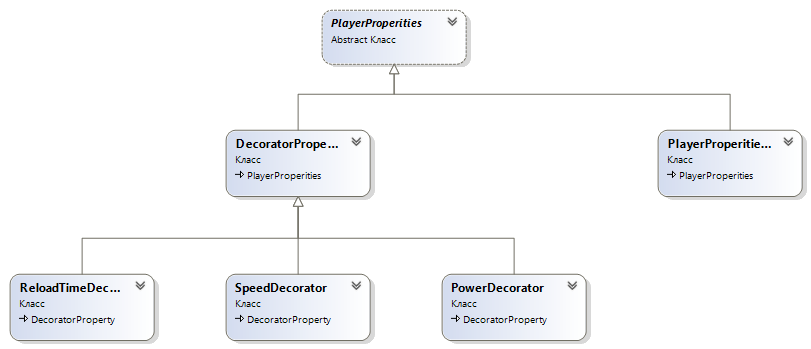


Рисунок 2.2 – Схема реализации паттерна «декоратор»

## 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.

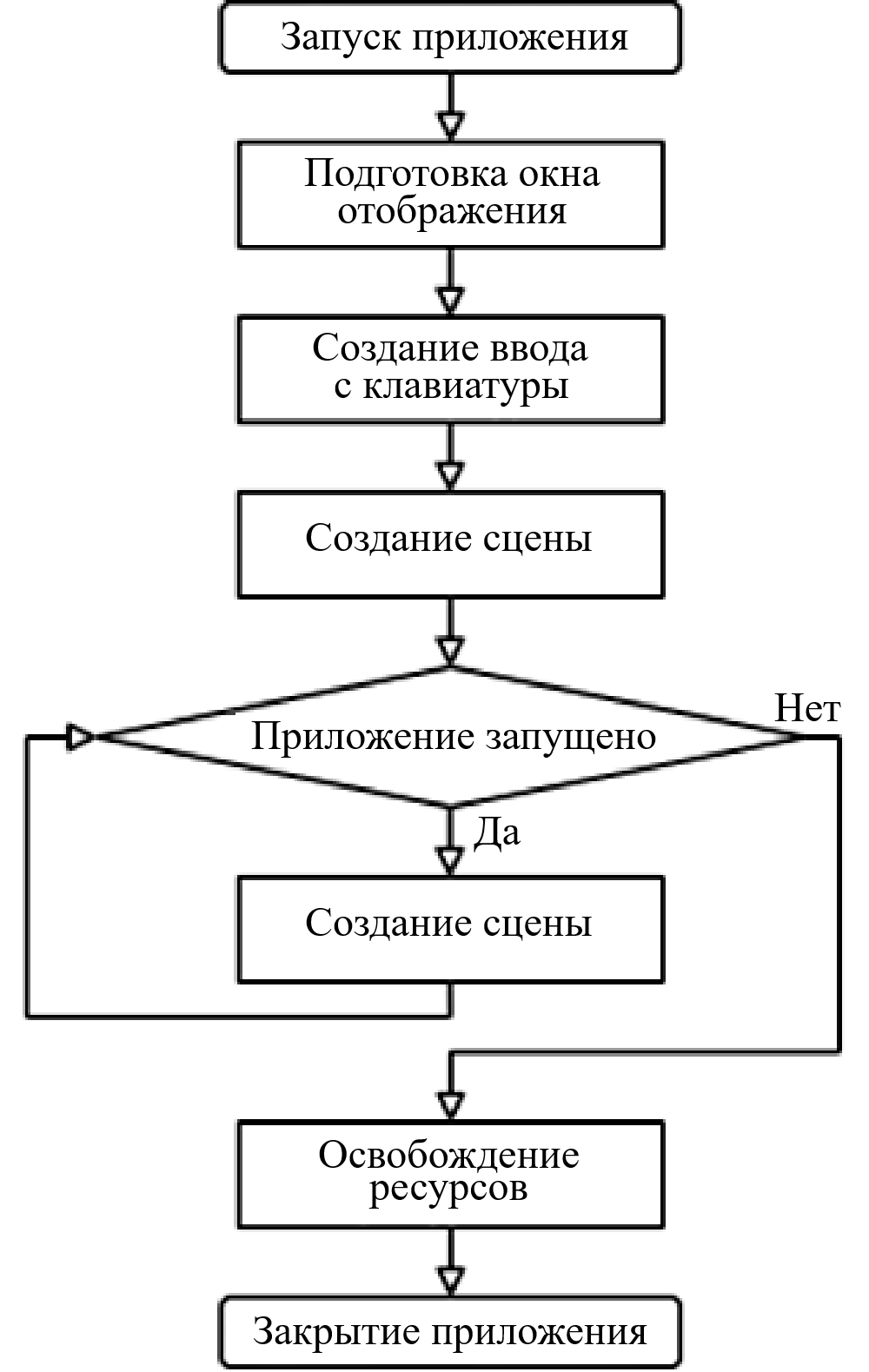


Рисунок 2.3 – Алгоритм работы игрового движка

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.

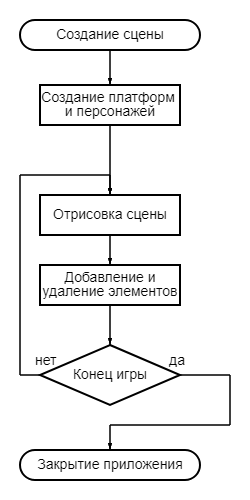


Рисунок 2.4 – Алгоритм работы проекта игровой логики

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, подобранное оружие, количество патронов в подобранном 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 зацикленности;
* *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.*

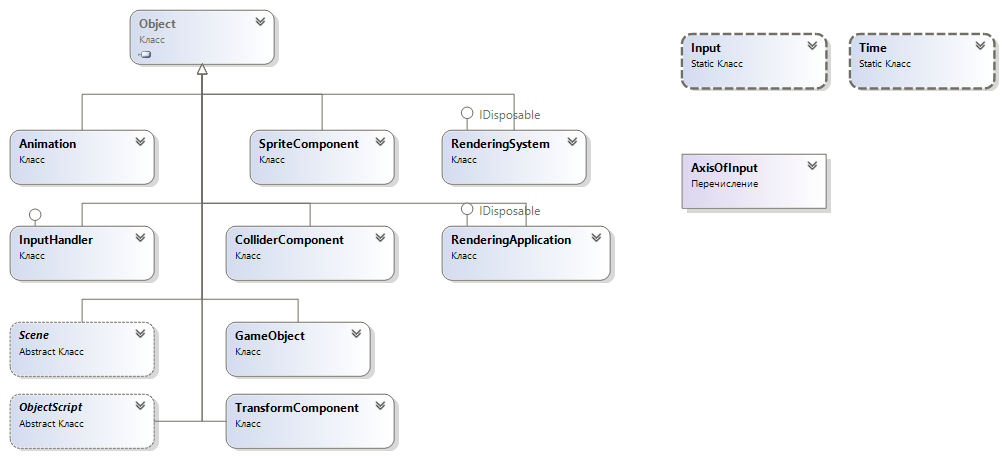


Рисунок 2.5 – Схема проекта *EngineLibrary*

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;
* *PlayerProperities* – 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 реализацию у наследуемых классов, по истечении этого самого времени;
* *PlayerProperitiesStandart* – класс, наследуемый от *PlayerProperities*, содержит сведения о базовых характеристиках игрока;
* *DecoratorProperty* – класс, наследуемый от *PlayerProperities*, 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 определённом 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.*

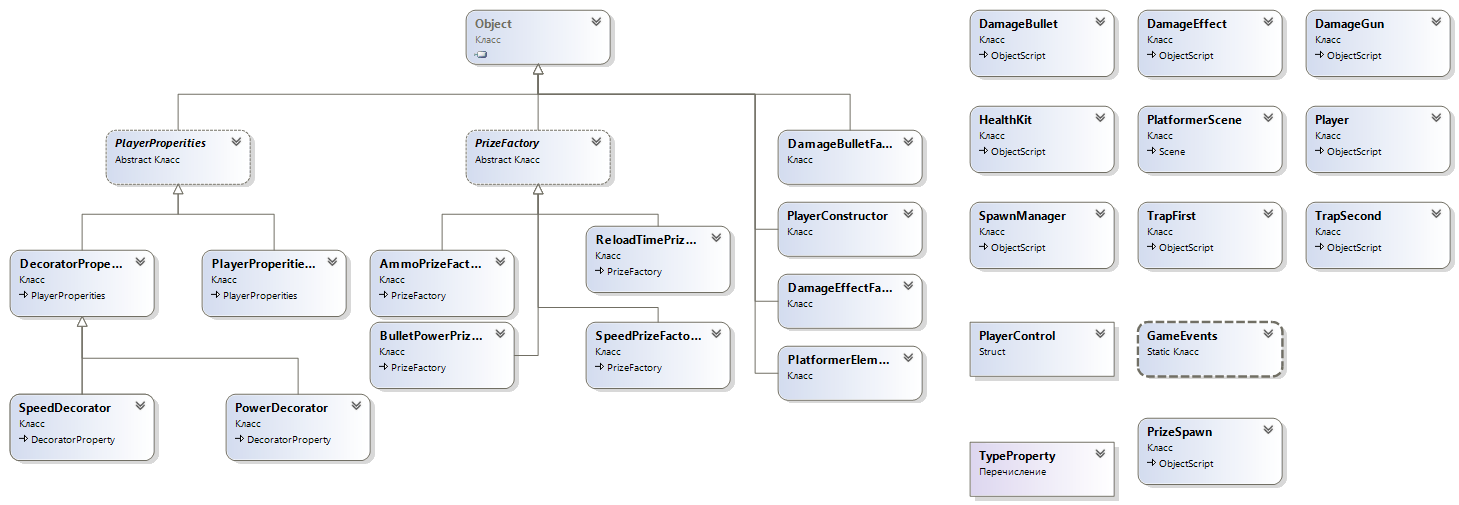


Рисунок 2.6 – Схема проекта *GameLibrary*

1. 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-платформер, где цвет каждого пикселя представляет собой элемент игровой сцены. Игровые объекты экшен-платформер создаются в экземпляре класса *PlatformerElementsFactory*. Также сцена сохраняет пустые ячейки в списке для последующего генерирования призов в планформере 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 аптечек. 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 *TestCollide*r;

– *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.

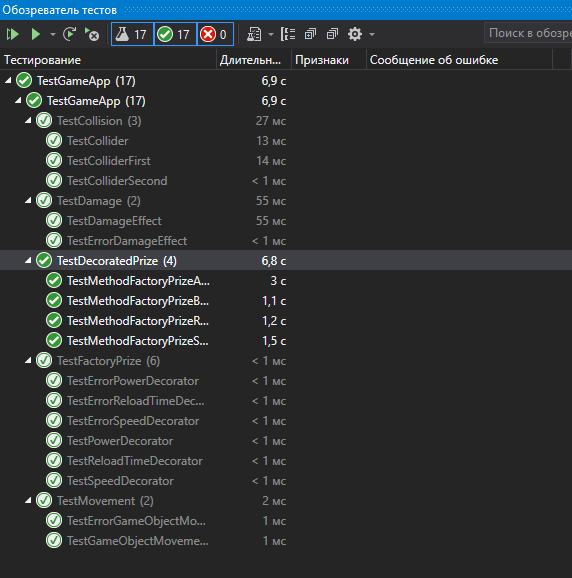


Рисунок 3.1 – Результаты тестирования классов разработанного игрового приложения

## 3.3 Результаты верификации приложения

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

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

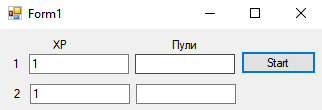


Рисунок 3.2 – Интерфейс игрового приложения

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

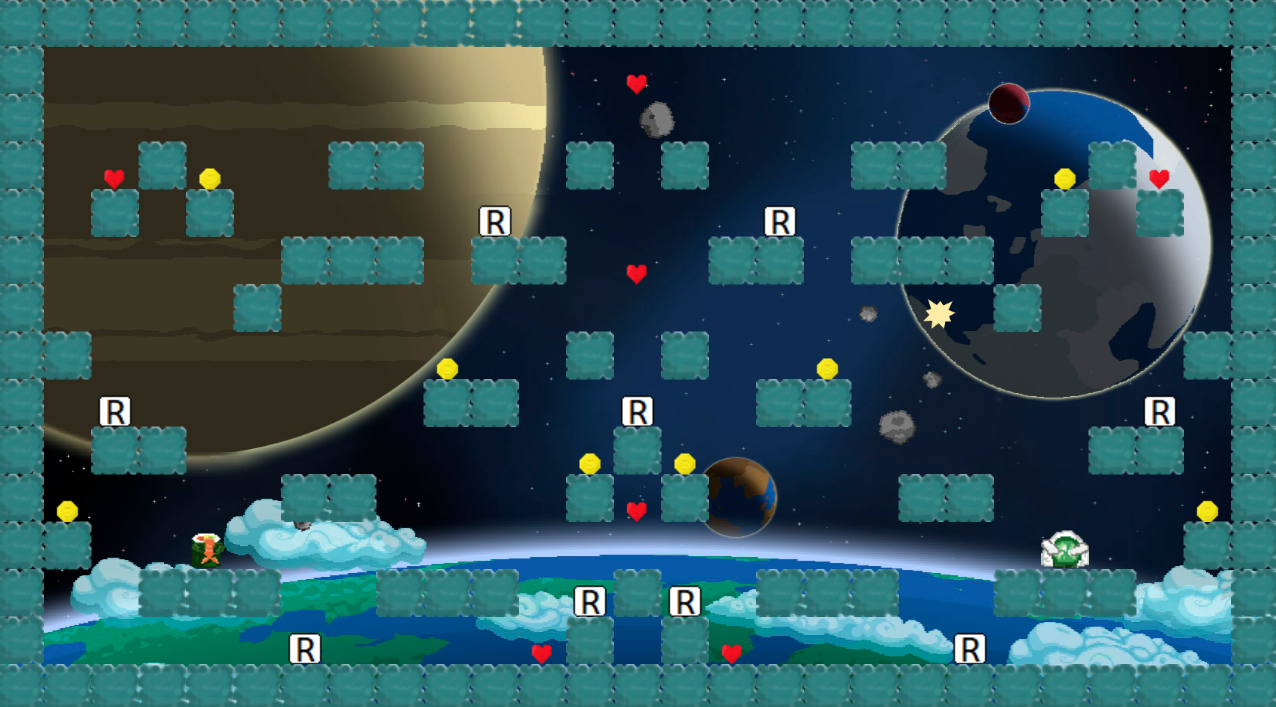


Рисунок 3.3 – Изображение игры в действии

На рисунке 3.4 приведена информация об управлении игровыми персонажами.



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 вероятность 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.

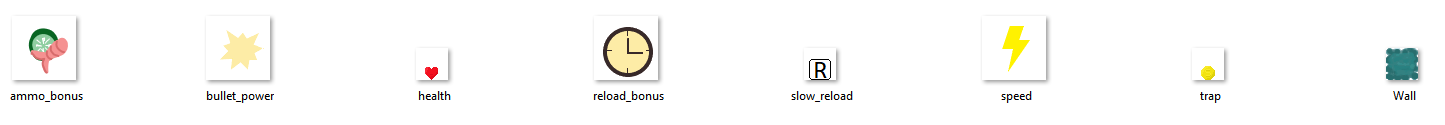


Рисунок 3.5 – Спрайты элементов экшен-платформер, с которыми можно взаимодействовать

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.

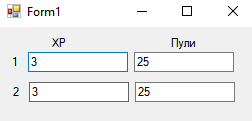


Рисунок 3.7 – Пример статистики игрока из пользовательского интерфейса во время игры

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.

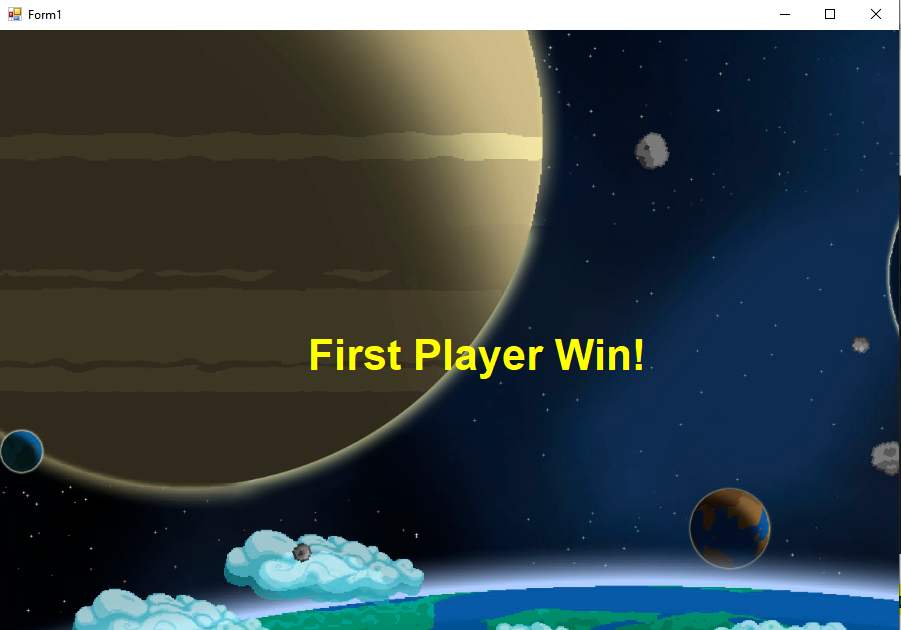


Рисунок 3.8 – Пользовательский интерфейс приложения после окончания игры

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. Последующееиспользование 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/ru-ru/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 c.
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 *С*++. – 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 c.

# ПРИЛОЖЕНИЕ А

(обязательное)

**Листинг программы «Экшен-платформер»**

**Код программы для *Input.cs*:**

using SharpDX.DirectInput;

using System;

namespace EngineLibrary.EngineComponents

{

/// <summary>

/// Класс, позволяющий управлять вводом с клавиатуры

/// </summary>

public static class Input

{

private static InputHandler inputHandler;

public static AxisOfInput AxisOfInput

{

get => default;

set

{

}

}

/// <summary>

/// Установка обработчика ввода с клавиатуры

/// </summary>

/// <param name="input">Обработчик ввода с клавиатуры</param>

public static void SetupInputHandler(InputHandler input)

{

inputHandler = input;

}

/// <summary>

/// Метод, возращающий значение ввода основных осей направления

/// </summary>

/// <param name="axis">Ось направления</param>

/// <returns>Положительное или отрицательное значение оси</returns>

public static int GetAxis(AxisOfInput axis)

{

if (inputHandler == null) return 0;

int move = 0;

if (inputHandler.KeyboardUpdated)

{

switch(axis)

{

case AxisOfInput.Horizontal:

if (inputHandler.KeyboardState.IsPressed(Key.D)) move++;

if (inputHandler.KeyboardState.IsPressed(Key.A)) move--;

break;

case AxisOfInput.Vertical:

if (inputHandler.KeyboardState.IsPressed(Key.W)) move++;

if (inputHandler.KeyboardState.IsPressed(Key.S)) move--;

break;

case AxisOfInput.AlternativeHorizontal:

if (inputHandler.KeyboardState.IsPressed(Key.Right)) move++;

if (inputHandler.KeyboardState.IsPressed(Key.Left)) move--;

break;

case AxisOfInput.AlternativeVertical:

if (inputHandler.KeyboardState.IsPressed(Key.Up)) move++;

if (inputHandler.KeyboardState.IsPressed(Key.Down)) move--;

break;

}

}

return move;

}

/// <summary>

/// Метод, возращающий реакцию на нажатие клавиши ввода

/// </summary>

/// <param name="key">Клавиша ввода</param>

/// <returns>Реакция true или false</returns>

public static bool GetButtonDawn(Key key)

{

if (inputHandler == null) return false;

if (inputHandler.KeyboardUpdated)

{

return inputHandler.KeyboardState.IsPressed(key);

}

return false;

}

public static int GetAxis(object verticalAxis)

{

throw new NotImplementedException();

}

}

/// <summary>

/// Ось направления ввода

/// </summary>

public enum AxisOfInput

{

/// <summary>

/// Горизонтальная ось

/// </summary>

Horizontal = 0,

/// <summary>

/// Вертикальная ось

/// </summary>

Vertical = 1,

/// <summary>

/// Альтернативная горизонтальная ось

/// </summary>

AlternativeHorizontal = 2,

/// <summary>

/// Альтернативная вертикальная ось

/// </summary>

AlternativeVertical = 3,

}

}

**Код программы для *InputHandler.cs*:**

using System;

using SharpDX;

using SharpDX.DirectInput;

using SharpDX.Windows;

namespace EngineLibrary.EngineComponents

{

/// <summary>

/// Обработчик ввода с клавиатуры

/// </summary>

public class InputHandler : IDisposable

{

private DirectInput directInput;

private Keyboard keyboard;

/// <summary>

/// Состояние клавиатуры

/// </summary>

public KeyboardState KeyboardState { get => keyboardState; }

private KeyboardState keyboardState;

/// <summary>

/// Удалось ли обновить состоние клавиатуры

/// </summary>

public bool KeyboardUpdated { get; private set; }

private bool keyboardAcquired;

/// <summary>

/// Конструктор класса, инциализирующий компоненты обработчика

/// </summary>

/// <param name="renderForm">Form отрисовки</param>

public InputHandler(RenderForm renderForm)

{

directInput = new DirectInput();

keyboard = new Keyboard(directInput);

keyboard.Properties.BufferSize = 16;

AcquireKeyboard();

keyboardState = new KeyboardState();

}

/// <summary>

/// Попытка получить доступ к клавиатуре

/// </summary>

private void AcquireKeyboard()

{

try

{

keyboard.Acquire();

keyboardAcquired = true;

}

catch (SharpDXException e)

{

if (e.ResultCode.Failure)

keyboardAcquired = false;

}

}

/// <summary>

/// Обновление состония клавиатуры

/// </summary>

public void UpdateKeyboardState()

{

if (!keyboardAcquired) AcquireKeyboard();

ResultDescriptor resultCode = ResultCode.Ok;

try

{

keyboard.GetCurrentState(ref keyboardState);

KeyboardUpdated = true;

}

catch (SharpDXException e)

{

resultCode = e.Descriptor;

KeyboardUpdated = false;

}

if (resultCode == ResultCode.InputLost || resultCode == ResultCode.NotAcquired)

keyboardAcquired = false;

}

/// <summary>

/// Освобождение ресурсов

/// </summary>

public void Dispose()

{

Utilities.Dispose(ref keyboard);

Utilities.Dispose(ref directInput);

}

}

}

**Код программы для *RenderingApplication.cs*:**

using System;

using System.Drawing;

using SharpDX.Windows;

using SharpDX.Direct2D1;

using System.Windows.Forms;

namespace EngineLibrary.EngineComponents

{

/// <summary>

/// Класс отрисовки и обновление всех процессов

/// </summary>

public class RenderingApplication : IDisposable

{

private const float unitsPerHeight = 15f;

private const int widthOfApplication = 1280;

private const int heightOfApplication = 720;

private readonly WindowRenderTarget renderTarget;

private readonly RenderingSystem rendering;

private readonly InputHandler input;

private Scene scene;

/// <summary>

/// Форма отрисовки

/// </summary>

public RenderForm RenderForm { get; set; }

public InputHandler InputHandler

{

get => default;

set

{

}

}

public RenderingSystem RenderingSystem

{

get => default;

set

{

}

}

public Scene Scene

{

get => default;

set

{

}

}

/// <summary>

/// Конструктор класса, инциализирующий его компоненты

/// </summary>

public RenderingApplication()

{

RenderForm = new RenderForm("Game");

RenderForm.ClientSize = new Size(widthOfApplication, heightOfApplication);

RenderForm.TopLevel = false;

RenderForm.AllowUserResizing = false;

RenderForm.FormBorderStyle = System.Windows.Forms.FormBorderStyle.None;

rendering = new RenderingSystem(RenderForm);

renderTarget = rendering.RenderTarget;

input = new InputHandler(RenderForm);

Input.SetupInputHandler(input);

}

/// <summary>

/// Установка игровой сцены

/// </summary>

/// <param name="targetScene"></param>

public void SetScene(Scene targetScene)

{

scene = targetScene;

scene.InitializeScene(renderTarget, RenderForm.ClientSize.Height / unitsPerHeight);

}

/// <summary>

/// Запуск приложения

/// </summary>

public void Run()

{

RenderLoop.Run(RenderForm, RenderCallback);

}

/// <summary>

/// Формирование каждого кадра

/// </summary>

private void RenderCallback()

{

input.UpdateKeyboardState();

Time.UpdateTime();

renderTarget.BeginDraw();

renderTarget.Clear(SharpDX.Color.Black);

scene.DrawScene();

renderTarget.EndDraw();

if (!scene.IsDrawScene)

{

Dispose();

}

}

/// <summary>

/// Освобождение ресурсов

/// </summary>

public void Dispose()

{

RenderForm.Dispose();

rendering.Dispose();

input.Dispose();

}

}

}

**Код программы для *RenderingSystem.cs*:**

using System;

using System.Collections.Generic;

using SharpDX;

using SharpDX.Windows;

using SharpDX.Direct2D1;

using SharpDX.WIC;

using Direct2D1 = SharpDX.Direct2D1;

using WIC = SharpDX.WIC;

using RenderTargetFactory = SharpDX.Direct2D1.Factory;

namespace EngineLibrary.EngineComponents

{

/// <summary>

/// Класс создания системы отрисовки

/// </summary>

public class RenderingSystem : IDisposable

{

private RenderTargetFactory renderTargetFactory;

/// <summary>

/// Окно отрисовки

/// </summary>

public WindowRenderTarget RenderTarget { get => renderTarget; }

private WindowRenderTarget renderTarget;

private static ImagingFactory imageFactory;

private static WindowRenderTarget staticRenderTarget;

/// <summary>

/// Коснтруктор, создаюший систему отрисовки в форме

/// </summary>

/// <param name="form">Форма для отрисовки</param>

public RenderingSystem(RenderForm form)

{

renderTargetFactory = new RenderTargetFactory();

imageFactory = new ImagingFactory();

RenderTargetProperties renderProperties = new RenderTargetProperties()

{

DpiX = 0,

DpiY = 0,

MinLevel = FeatureLevel.Level\_10,

PixelFormat = new Direct2D1.PixelFormat(SharpDX.DXGI.Format.B8G8R8A8\_UNorm, AlphaMode.Premultiplied),

Type = RenderTargetType.Hardware,

Usage = RenderTargetUsage.None

};

HwndRenderTargetProperties winProperties = new HwndRenderTargetProperties()

{

Hwnd = form.Handle,

PixelSize = new Size2(form.ClientSize.Width, form.ClientSize.Height),

PresentOptions = PresentOptions.None

};

renderTarget = new WindowRenderTarget(renderTargetFactory, renderProperties, winProperties);

staticRenderTarget = renderTarget;

}

/// <summary>

/// Загрузка изображения из файла

/// </summary>

/// <param name="imageFileName">Путь к файлу</param>

/// <returns>Изображение</returns>

public static Direct2D1.Bitmap LoadBitmap(string imageFileName)

{

BitmapDecoder decoder = new BitmapDecoder(imageFactory, imageFileName, DecodeOptions.CacheOnDemand);

BitmapFrameDecode frame = decoder.GetFrame(0);

FormatConverter converter = new FormatConverter(imageFactory);

converter.Initialize(frame, WIC.PixelFormat.Format32bppPRGBA, BitmapDitherType.Ordered4x4, null, 0.0, BitmapPaletteType.Custom);

Direct2D1.Bitmap bitmap = Direct2D1.Bitmap.FromWicBitmap(staticRenderTarget, converter);

Utilities.Dispose(ref converter);

Utilities.Dispose(ref frame);

Utilities.Dispose(ref decoder);

return bitmap;

}

/// <summary>

/// Загрузка последовательности изображений для анимации

/// </summary>

/// <param name="pathToFiles">Путь к файлам</param>

/// <param name="countOfBitmaps">Количество файлов для загрузки</param>

/// <returns>Список изображений</returns>

public static List<Direct2D1.Bitmap> LoadAnimation(string pathToFiles, int countOfBitmaps)

{

List<Direct2D1.Bitmap> bitmaps = new List<Direct2D1.Bitmap>();

for(int i = 1; i <= countOfBitmaps; i++)

{

string fileName = pathToFiles + i + ".png";

bitmaps.Add(LoadBitmap(fileName));

}

return bitmaps;

}

/// <summary>

/// Освобождение ресурсов

/// </summary>

public void Dispose()

{

Utilities.Dispose(ref renderTarget);

Utilities.Dispose(ref imageFactory);

Utilities.Dispose(ref renderTargetFactory);

}

}

}

**Код программы для *Scene.cs*:**

using System.Collections.Generic;

using EngineLibrary.ObjectComponents;

using SharpDX.Direct2D1;

namespace EngineLibrary.EngineComponents

{

/// <summary>

/// Абстрактный класс сцены

/// </summary>

public abstract class Scene

{

private WindowRenderTarget renderTarget;

private float worldScale;

/// <summary>

/// Список текущих игровых объектов для отрисовки

/// </summary>

public List<GameObject> gameObjects = new List<GameObject>();

/// <summary>

/// Список игровых объектов для добалвения в список отрисовки

/// </summary>

protected List<GameObject> gameObjectsToAdd = new List<GameObject>();

/// <summary>

/// Список игровых объектов для удаления их списка отрисовки

/// </summary>

protected List<GameObject> gameObjectsToRemove = new List<GameObject>();

/// <summary>

/// Состояния отрисовки сцены

/// </summary>

public bool IsDrawScene = true;

/// <summary>

/// Инициализация сцены и игровых объектов

/// </summary>

/// <param name="target">Окно отрисовки</param>

/// <param name="scale">Относительный размер объектов</param>

public void InitializeScene(WindowRenderTarget target, float scale)

{

renderTarget = target;

worldScale = scale;

CreateGameObjectsOnScene();

InitializeEngineSettingsToGameObjects(gameObjects);

}

/// <summary>

/// Установка опций рендеринга игровым объектам

/// </summary>

private void InitializeEngineSettingsToGameObjects(List<GameObject> gameObjects)

{

foreach (GameObject gameObject in gameObjects)

gameObject.InitalizeEngineSettings(renderTarget, worldScale);

}

/// <summary>

/// Создание игровых объектов на сцене

/// </summary>

protected abstract void CreateGameObjectsOnScene();

/// <summary>

/// Отрисовка сцены (игровых объектов)

/// </summary>

public void DrawScene()

{

foreach (GameObject gameObject in gameObjects)

gameObject.Draw();

AddRenderGameObjects();

RemoveRenderGameObjects();

if (!IsDrawScene)

gameObjects.Clear();

}

/// <summary>

/// Добавление игрового объекта для отрисовки

/// </summary>

private void AddRenderGameObjects()

{

InitializeEngineSettingsToGameObjects(gameObjectsToAdd);

gameObjects.AddRange(gameObjectsToAdd);

gameObjectsToAdd.Clear();

}

/// <summary>

/// Удаление игрового объекта из отрисовки

/// </summary>

private void RemoveRenderGameObjects()

{

foreach(GameObject removeGameObject in gameObjectsToRemove)

{

gameObjects.Remove(removeGameObject);

}

gameObjectsToRemove.Clear();

}

/// <summary>

/// Поведения при окончании сцены

/// </summary>

protected virtual void EndScene()

{

IsDrawScene = false;

}

}

}

**Код программы для *Time.cs*:**

using System;

using System.Diagnostics;

namespace EngineLibrary.EngineComponents

{

/// <summary>

/// Статический класс управления временем

/// </summary>

public static class Time

{

private readonly static Stopwatch watch;

private static long previousTicks;

/// <summary>

/// Текущее время с запуска приложения

/// </summary>

public static float CurrentTime { get; private set; }

/// <summary>

/// Разница во времени между кадрами

/// </summary>

public static float DeltaTime { get; private set; }

/// <summary>

/// Конструктори статического класса

/// </summary>

static Time()

{

watch = new Stopwatch();

Reset();

}

/// <summary>

/// Обновление подсчитанных значений

/// </summary>

public static void UpdateTime()

{

long ticks = watch.Elapsed.Ticks;

CurrentTime = (float)ticks / TimeSpan.TicksPerSecond;

DeltaTime = (float)(ticks - previousTicks) / TimeSpan.TicksPerSecond;

previousTicks = ticks;

}

/// <summary>

/// Сброс таймера и счетчика

/// </summary>

public static void Reset()

{

watch.Reset();

watch.Start();

previousTicks = watch.Elapsed.Ticks;

}

}

}

**Код программы для *Animation.cs*:**

using System.Collections.Generic;

using EngineLibrary.EngineComponents;

using SharpDX.Direct2D1;

namespace EngineLibrary.ObjectComponents

{

/// <summary>

/// Класс анимации

/// </summary>

public class Animation

{

private readonly List<Bitmap> sprites;

private int currentIndexInAnimation;

private float changeTime;

private readonly float deltaTimeAnimation;

private readonly bool isLoop;

/// <summary>

/// Конструктор

/// </summary>

/// <param name="sprites">Список спрайтов</param>

/// <param name="changeTime">Время смены каждого спрайта</param>

/// <param name="isLoop">Зацикленностб анимации</param>

public Animation(List<Bitmap> sprites, float changeTime, bool isLoop)

{

this.sprites = sprites;

this.changeTime = changeTime;

this.isLoop = isLoop;

deltaTimeAnimation = changeTime;

currentIndexInAnimation = 0;

}

/// <summary>

/// Сброс анимации

/// </summary>

public void ResetAnimation()

{

currentIndexInAnimation = 0;

}

/// <summary>

/// Возращает текущее изображение анимации

/// </summary>

/// <returns>Текущее изображение в анимации</returns>

public Bitmap GetSpriteFromAnimation()

{

if (sprites == null)

return null;

if (changeTime < Time.CurrentTime)

{

currentIndexInAnimation++;

changeTime = Time.CurrentTime + deltaTimeAnimation;

}

if (currentIndexInAnimation >= sprites.Count)

if (!isLoop)

return null;

else

currentIndexInAnimation = 0;

return sprites[currentIndexInAnimation];

}

}

}

**Код программы для *ColliderComponent.cs*:**

using SharpDX;

using System.Collections.Generic;

namespace EngineLibrary.ObjectComponents

{

/// <summary>

/// Класс компонента, описывающий твердое тело

/// </summary>

public class ColliderComponent

{

private static List<GameObject> collidersOfGameObjects;

private readonly Vector2[] boundCorners;

private readonly GameObject gameObject;

private Size2F colliderScale;

private Vector2 offsetCollider;

/// <summary>

/// Неактивность элемента

/// </summary>

public bool IsInactive { get; set; }

/// <summary>

/// Конструктор класса

/// </summary>

/// <param name="gameObject">Игровой объект, которому принадлежит компонент</param>

/// <param name="scale">Размер коллайдера</param>

/// <param name="offset">Cмещения коллайдера от центра</param>

public ColliderComponent(GameObject gameObject, Size2F scale, Vector2 offset = new Vector2())

{

if (collidersOfGameObjects == null)

collidersOfGameObjects = new List<GameObject>();

collidersOfGameObjects.Add(gameObject);

this.gameObject = gameObject;

boundCorners = new Vector2[4];

colliderScale = scale;

offsetCollider = offset;

IsInactive = false;

}

/// <summary>

/// Обновление границ твердого тела

/// </summary>

private void UpdateBounds()

{

Vector2 position = gameObject.Transform.Position;

float offsetWidth = colliderScale.Width / 2;

float offsetHeight = colliderScale.Height / 2;

boundCorners[0] = new Vector2(position.X + offsetCollider.X - offsetWidth, position.Y + offsetCollider.Y - offsetHeight);

boundCorners[1] = new Vector2(position.X + offsetCollider.X - offsetWidth, position.Y + offsetCollider.Y + offsetHeight);

boundCorners[2] = new Vector2(position.X + offsetCollider.X + offsetWidth, position.Y + offsetCollider.Y + offsetHeight);

boundCorners[3] = new Vector2(position.X + offsetCollider.X + offsetWidth, position.Y + offsetCollider.Y - offsetHeight);

}

/// <summary>

/// Проверка на пересечние компонента твердого тела с другими компонентами твердого тела, имеющие тег у игрового объекта

/// </summary>

/// <param name="tagNames">Теги игровых объектов, с которыми ожидается столкновение</param>

/// <returns>Реакция на проверку</returns>

public bool CheckIntersection(params string[] tagNames)

{

foreach (GameObject otherGameObject in collidersOfGameObjects)

{

if (otherGameObject == gameObject || otherGameObject.Collider.IsInactive) continue;

bool hasTag = false;

for (int i = 0; i < tagNames.Length && !hasTag; i++)

{

hasTag = otherGameObject.GameObjectTag == tagNames[i];

}

if(hasTag)

{

if (CheckGameObjectIntersection(otherGameObject))

{

return true;

}

}

}

return false;

}

/// <summary>

/// Проверка на пересечние компонента твердого тела с другими компонентами твердого тела, имеющие тег у игрового объекта

/// </summary>

/// <param name="intersecredGameObject">Пересаемый объект</param>

/// <param name="tagNames">Теги игровых объектов, с которыми ожидается столкновение</param>

/// <returns>Реакция на проверку</returns>

public bool CheckIntersection(out GameObject intersecredGameObject,params string[] tagNames)

{

foreach (GameObject otherGameObject in collidersOfGameObjects)

{

if (otherGameObject == gameObject || otherGameObject.Collider.IsInactive) continue;

bool hasTag = false;

for (int i = 0; i < tagNames.Length && !hasTag; i++)

{

hasTag = otherGameObject.GameObjectTag == tagNames[i];

}

if (hasTag)

{

if (otherGameObject.GameObjectTag != null && CheckGameObjectIntersection(otherGameObject))

{

intersecredGameObject = otherGameObject;

return true;

}

}

}

intersecredGameObject = null;

return false;

}

/// <summary>

/// Проверка на пересечение компонента твердого тела с другими компонентами твердого тела, имеющие конкретный сценарий выполнения T

/// </summary>

/// <typeparam name="T">Конкретный сценарий выполнения</typeparam>

/// <param name="objectScript">Сценарий выполнения игрового объекта, с которым столкнулся</param>

/// <returns>Реакция на столкновение</returns>

public bool CheckIntersection<T>(out T objectScript)

where T : ObjectScript

{

foreach (GameObject otherGameObject in collidersOfGameObjects)

{

if (otherGameObject == gameObject || otherGameObject.Script == null || otherGameObject.Collider.IsInactive) continue;

if (otherGameObject.Script is T)

{

if (CheckGameObjectIntersection(otherGameObject))

{

objectScript = (T)otherGameObject.Script;

return true;

}

}

}

objectScript = null;

return false;

}

/// <summary>

/// Проверка на пересечение компонента твердого тела с другим компонентам твердого тела

/// </summary>

/// <param name="otherGameObject">Игровой объект с компонетом твердого тела</param>

/// <returns>Реакция на столкновение</returns>

private bool CheckGameObjectIntersection(GameObject otherGameObject)

{

UpdateBounds();

otherGameObject.Collider.UpdateBounds();

ColliderComponent collider = otherGameObject.Collider;

int count = boundCorners.Length + collider.boundCorners.Length;

Vector2[] allCorners = new Vector2[count];

boundCorners.CopyTo(allCorners, 0);

collider.boundCorners.CopyTo(allCorners, boundCorners.Length);

Vector2 normal;

bool isInteresect = false;

for (int i = 0; i < count && !isInteresect; i++)

{

normal = GetNormal(allCorners, i);

Vector2 currentProjection = GetProjection(normal);

Vector2 otherProjection = collider.GetProjection(normal);

if (currentProjection.X < otherProjection.Y || otherProjection.X < currentProjection.Y)

{

return false;

}

}

return true;

}

/// <summary>

/// Создание нормали

/// </summary>

/// <param name="corners">Углы двух компонетов</param>

/// <param name="index">Номер угла</param>

/// <returns>Нормаль</returns>

private Vector2 GetNormal(Vector2[] corners, int index)

{

int next = index + 1;

next = next == corners.Length ? 0 : next;

Vector2 firstPoint = corners[index];

Vector2 secondPoint = corners[next];

Vector2 edge = new Vector2(secondPoint.X - firstPoint.X, secondPoint.Y - firstPoint.Y);

return new Vector2(-edge.Y, edge.X);

}

/// <summary>

/// Создание проекции

/// </summary>

/// <param name="normal">Нормаль</param>

/// <returns>Проекцию</returns>

private Vector2 GetProjection(Vector2 normal)

{

Vector2 result = new Vector2();

bool isNull = true;

foreach(Vector2 current in boundCorners)

{

float projection = normal.X \* current.X + normal.Y \* current.Y;

if(isNull)

{

result = new Vector2(projection, projection);

isNull = false;

}

if (projection > result.X)

result.X = projection;

if (projection < result.Y)

result.Y = projection;

}

return result;

}

}

}

**Код программы для *GameObject.cs*:**

using SharpDX;

using SharpDX.Direct2D1;

namespace EngineLibrary.ObjectComponents

{

/// <summary>

/// Игровой объект

/// </summary>

public class GameObject

{

private WindowRenderTarget renderTarget;

private float worldScale;

/// <summary>

/// Родительский игровой объект

/// </summary>

public GameObject ParentGameObject { get; set; }

/// <summary>

/// Тэг игрового объекта

/// </summary>

public string GameObjectTag { get; set; }

/// <summary>

/// Активность игрового объекта

/// </summary>

public bool IsActive { get; set; } = true;

/// <summary>

/// Компонент перемещения

/// </summary>

public TransformComponent Transform { get; private set; }

/// <summary>

/// Компонент спрайта

/// </summary>

public SpriteComponent Sprite { get; private set; }

/// <summary>

/// Компонент твердого тела

/// </summary>

public ColliderComponent Collider { get; private set; }

/// <summary>

/// Сценарий выполения

/// </summary>

public ObjectScript Script { get; private set; }

public TransformComponent TransformComponent

{

get => default;

set

{

}

}

public SpriteComponent SpriteComponent

{

get => default;

set

{

}

}

public ColliderComponent ColliderComponent

{

get => default;

set

{

}

}

public ObjectScript ObjectScript

{

get => default;

set

{

}

}

/// <summary>

/// Инциализация опций рендеринга игрового объекта

/// </summary>

/// <param name="target">Окно отрисовки</param>

/// <param name="scale">Относительный размер</param>

internal void InitalizeEngineSettings(WindowRenderTarget target, float scale)

{

renderTarget = target;

worldScale = scale;

}

/// <summary>

/// Инциализация компонетов игрового объекта

/// </summary>

/// <param name="component">Компонент игровго объекта</param>

public void InitializeObjectComponent(object component)

{

switch(component)

{

case TransformComponent transformComponent:

Transform = transformComponent;

break;

case SpriteComponent spriteComponent:

Sprite = spriteComponent;

break;

case ColliderComponent colliderComponent:

Collider = colliderComponent;

break;

}

}

/// <summary>

/// Инициализация сценария поведения объекта

/// </summary>

/// <param name="objectScript">Сценарий игрового объекта</param>

public void InitializeObjectScript(ObjectScript objectScript)

{

Script = objectScript;

Script.Start(this);

}

/// <summary>

/// Метод отрисовки игрового объекта

/// </summary>

public void Draw()

{

if (Collider != null)

Collider.IsInactive = !IsActive;

if (Script != null)

Script.Update(this);

if (!IsActive || (ParentGameObject != null && !ParentGameObject.IsActive)) return;

if (Sprite == null) return;

Vector2 translation = new Vector2(0,0);

if (ParentGameObject != null)

{

translation.X = ParentGameObject.Transform.Position.X \* ParentGameObject.Transform.Scale.Width;

translation.Y = ParentGameObject.Transform.Position.Y \* ParentGameObject.Transform.Scale.Height;

}

translation.X += Transform.Position.X \* Transform.Scale.Width;

translation.Y += Transform.Position.Y \* Transform.Scale.Height;

Matrix3x2 matrix = Matrix3x2.Rotation(0, translation);

matrix \*= Matrix3x2.Scaling(Transform.Scale.Width \* worldScale / Sprite.WidthOfSprite, Transform.Scale.Height \* worldScale / Sprite.HeightOfSprite, translation);

matrix \*= Matrix3x2.Translation(translation \* worldScale);

renderTarget.Transform = matrix;

Sprite.PlayAnimation();

if (Sprite.Bitmap != null)

renderTarget.DrawBitmap(Sprite.Bitmap, 1f, BitmapInterpolationMode.Linear);

}

}

}

**Код программы для *ObjectScript.cs*:**

namespace EngineLibrary.ObjectComponents

{

/// <summary>

/// Абстрактный класс сценария поведения игрового объекта

/// </summary>

public abstract class ObjectScript

{

/// <summary>

/// Поведение на момент создание игрового объекта

/// </summary>

public abstract void Start(GameObject gameObject = null);

/// <summary>

/// Обновление игрового объекта

/// </summary>

public abstract void Update(GameObject gameObject);

}

}

**Код программы для *SpriteComponent.cs*:**

using System.Collections.Generic;

using SharpDX.Direct2D1;

namespace EngineLibrary.ObjectComponents

{

/// <summary>

/// Класс спрайта игровйо объекта

/// </summary>

public class SpriteComponent

{

/// <summary>

/// Изображение игрового объекта

/// </summary>

public Bitmap Bitmap { get; set; }

/// <summary>

/// Ширина спрайта

/// </summary>

public float WidthOfSprite { get; private set; }

/// <summary>

/// Высота спрайта

/// </summary>

public float HeightOfSprite { get; private set; }

/// <summary>

/// Инверсия по оси X

/// </summary>

public bool IsFlipX { get; set; }

public Animation Animation

{

get => default;

set

{

}

}

private readonly Dictionary<string, Animation> animations;

private string currentAnimation = "";

/// <summary>

/// Конструктор класса спрайта

/// </summary>

/// <param name="sprite">Изображение</param>

public SpriteComponent(Bitmap sprite)

{

Bitmap = sprite;

WidthOfSprite = sprite.Size.Width;

HeightOfSprite = sprite.Size.Height;

animations = new Dictionary<string, Animation>();

animations.Add("inactive", new Animation(null, 1f, true));

}

/// <summary>

/// Добавление анимации

/// </summary>

/// <param name="name">Ключ анимации</param>

/// <param name="animation">Анимация</param>

public void AddAnimation(string name, Animation animation)

{

animations.Add(name, animation);

}

/// <summary>

/// Проигрывание анимации

/// </summary>

public void PlayAnimation()

{

if (currentAnimation == "") return;

Bitmap = animations[currentAnimation].GetSpriteFromAnimation();

}

/// <summary>

/// Установка текущей анимации

/// </summary>

/// <param name="name">Ключ анимации</param>

public void SetAnimation(string name)

{

if (currentAnimation == name) return;

if(currentAnimation != "")

animations[currentAnimation].ResetAnimation();

currentAnimation = name;

}

}

}

**Код программы для *TransformComponent.cs*:**

using SharpDX;

namespace EngineLibrary.ObjectComponents

{

/// <summary>

/// Компонент перемещения игрового объекта

/// </summary>

public class TransformComponent

{

private const float accelerationOfGravity = 0.0981f;

/// <summary>

/// Позиция игрового объекта

/// </summary>

public Vector2 Position { get => position; set => position = value; }

private Vector2 position;

/// <summary>

/// Размер игрового объекта

/// </summary>

public Size2F Scale { get => scale; set => scale = value; }

private Size2F scale;

/// <summary>

/// Использование гравитации

/// </summary>

public bool IsUseGravitation { get; set; } = true;

private Vector2 movementInCurrentFrame;

/// <summary>

/// Конструктор компонента

/// </summary>

/// <param name="position">начальная позиция</param>

/// <param name="scale">Начальный размер</param>

public TransformComponent(Vector2 position, Size2F scale)

{

movementInCurrentFrame = Vector2.Zero;

this.position = position;

this.scale = scale;

}

/// <summary>

/// Перемещение объкта

/// </summary>

/// <param name="movement">Вектор перемещения</param>

public void SetMovement(Vector2 movement)

{

movementInCurrentFrame = movement;

position.X += movement.X;

position.Y -= movement.Y;

}

/// <summary>

/// Возврат позиция в этом кадре

/// </summary>

public void ResetMovement()

{

position.X -= movementInCurrentFrame.X;

if(!IsUseGravitation)

position.Y += movementInCurrentFrame.Y;

}

/// <summary>

/// Добавление силы тяжести

/// </summary>

public void AddGravitation()

{

if(IsUseGravitation)

position.Y += accelerationOfGravity;

}

/// <summary>

/// Возрат силы тяжести

/// </summary>

public void ResetGravitation()

{

if (IsUseGravitation)

position.Y -= accelerationOfGravity;

}

}

}

**Код программы для *BulletFactory.cs*:**

using EngineLibrary.ObjectComponents;

using SharpDX;

namespace GameLibrary.Bullets.BulletFactories

{

/// <summary>

/// Абстрактный класс фабрики создания пуль

/// </summary>

public abstract class BulletFactory

{

/// <summary>

/// Создание игрового объекта пули

/// </summary>

/// <param name="position">Позиция появления пули</param>

/// <param name="direction">Направление пули</param>

/// <param name="tag">Тег игрового объекта, создающий пулю</param>

/// <returns>Игровой объект</returns>

public abstract GameObject CreateBullet(Vector2 position, Vector2 direction, string tag, float power = 1);

}

}

**Код программы для *DamageBulletFactory.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using SharpDX;

namespace GameLibrary.Bullets.BulletFactories

{

/// <summary>

/// Класс фабрики создания пули урона

/// </summary>

public class DamageBulletFactory : BulletFactory

{

/// <summary>

/// Создание игрового объекта пули, которая убивает

/// </summary>

/// <param name="position">Позиция появления пули</param>

/// <param name="direction">Направление пули</param>

/// <param name="tag">Тег игрового объекта, создающий пулю</param>

/// <returns>Игровой объект</returns>

public override GameObject CreateBullet(Vector2 position, Vector2 direction, string tag, float power = 1)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(position, new Size2F(1, 1)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/Bullets/damage bullet.png")));

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(0.4f, 0.4f)));

gameObject.GameObjectTag = "Bullet";

DamageBullet bullet = new DamageBullet();

bullet.SetSettings(direction, tag, power);

gameObject.InitializeObjectScript(bullet);

return gameObject;

}

}

}

**Код программы для *Bullet.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Platformer;

using SharpDX;

namespace GameLibrary.Bullets

{

/// <summary>

/// Абстрактный класс пули

/// </summary>

public abstract class Bullet : ObjectScript

{

/// <summary>

/// Скорость пули

/// </summary>

public abstract float Speed { get; }

/// <summary>

/// Экземпляр сцены игры

/// </summary>

protected PlatformerScene Platformer;

private Vector2 flyDirection;

private string[] interactionTag = new string[2];

protected string tag;

protected float power;

/// <summary>

/// Установление направления полета пули

/// </summary>

/// <param name="direction">Вектор направления</param>

/// <param name="tag">Тег игрового объекта, создающий пулю</param>

public void SetSettings(Vector2 direction, string tag, float power)

{

this.tag = tag;

flyDirection = direction;

this.power = power;

if (tag == "Blue Player")

interactionTag[0] = "Red Player";

else if(tag == "Red Player")

interactionTag[0] = "Blue Player";

else

{

interactionTag[0] = "Blue Player";

interactionTag[1] = "Red Player";

}

}

/// <summary>

/// Поведение на момент создание игрового объекта

/// </summary>

public override void Start(GameObject gameObject = null)

{

Platformer = PlatformerScene.instance;

}

/// <summary>

/// Обновление игрового объекта

/// </summary>

public override void Update(GameObject gameObject)

{

Vector2 movement = new Vector2();

movement = flyDirection \* Speed \* Time.DeltaTime;

gameObject.Transform.SetMovement(movement);

if (gameObject.Collider.CheckIntersection("Wall"))

{

Platformer.RemoveObjectFromScene(gameObject);

}

if (gameObject.Collider.CheckIntersection(out GameObject wall, "BreakWall"))

{

Platformer.RemoveObjectFromScene(gameObject);

(wall.Script as BreakWall).DestroyWall(wall);

}

if (gameObject.Collider.CheckIntersection(out GameObject playerGameObject, interactionTag))

{

PlayerInteraction(playerGameObject);

Platformer.RemoveObjectFromScene(gameObject);

}

if (gameObject.Collider.CheckIntersection(out GameObject monsterGameObject, "Monster"))

{

if(tag != monsterGameObject.GameObjectTag)

{

PlayerInteraction(monsterGameObject);

Platformer.RemoveObjectFromScene(gameObject);

}

}

}

/// <summary>

/// Взаимодействие с игроком

/// </summary>

public abstract void PlayerInteraction(GameObject playerGameObject);

}

}

**Код программы для *DamageBullet.cs*:**

using EngineLibrary.ObjectComponents;

using GameLibrary.Effects.EffectFactories;

namespace GameLibrary.Bullets

{

/// <summary>

/// Класс пули урона

/// </summary>

public class DamageBullet : Bullet

{

/// <summary>

/// Скорость пули

/// </summary>

public override float Speed => 10f;

/// <summary>

/// Взаимодействие с игроком

/// </summary>

public override void PlayerInteraction(GameObject playerGameObject)

{

DamageEffectFactory factory = new DamageEffectFactory();

Platformer.AddObjectOnScene(factory.CreateEffect(playerGameObject, tag, power));

}

}

}

**Код программы для *DamageEffectFactory*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Game;

using GameLibrary.Monsters;

using SharpDX;

namespace GameLibrary.Effects.EffectFactories

{

/// <summary>

/// Фабрика создания эффекта смерти

/// </summary>

public class DamageEffectFactory : EffectFactory

{

/// <summary>

/// Создание могилы

/// </summary>

/// <param name="player">Игровой объект игрока</param>

/// <returns>Игровой объект</returns>

public override GameObject CreateEffect(GameObject gameObj, string tag = null, float power = 1)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(gameObj.Transform.Position, new Size2F(1f, 1f)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/PlatformerElements/Effects/damage idle 1.png")));

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(0.8f, 0.8f)));

gameObject.GameObjectTag = "Effect";

DamageEffect damageEffect = new DamageEffect();

damageEffect.ActivateEffect(gameObj, tag, power);

switch (gameObj.Script)

{

case Monster scriptMonster:

if(scriptMonster.Health > 0)

gameObject.Sprite.Bitmap = null;

break;

case Player scriptPlayer:

if (scriptPlayer.Property.Health > 0)

gameObject.Sprite.Bitmap = null;

break;

}

gameObject.InitializeObjectScript(damageEffect);

return gameObject;

}

}

}

**Код программы для *EffectFactory.cs*:**

using EngineLibrary.ObjectComponents;

namespace GameLibrary.Effects.EffectFactories

{

/// <summary>

/// Абстрактная фабрика создания эффектов

/// </summary>

public abstract class EffectFactory

{

/// <summary>

/// Создание эффекта

/// </summary>

/// <param name="player">Игровой объект игрока</param>

/// <returns>Игровой объект</returns>

public abstract GameObject CreateEffect(GameObject player, string tag = null, float power = 1);

}

}

**Код программы для *DamageEffect.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Game;

using GameLibrary.Monsters;

namespace GameLibrary.Effects

{

/// <summary>

/// Эффект после урона

/// </summary>

public class DamageEffect : Effect

{

/// <summary>

/// Время дейтсвия эффекта

/// </summary>

public override float EffectTime => 0.5f;

private int storedCoins;

//private int storedHealth;

/// <summary>

/// Активация эффекта

/// </summary>

/// <param name="player">Игровой объект, на который будет наложен эффект</param>

public override void ActivateEffect(GameObject player, string tag = null, float power = 1)

{

playerGameObject = player;

playerGameObject.IsActive = true;

if(playerGameObject.Script is Player)

{

if ((playerGameObject.Script as Player).Property.Health > 0)

{

(playerGameObject.Script as Player).ChangeStatsValue(player, -power);

}

if ((playerGameObject.Script as Player).Property.Health <= 0)

{

playerGameObject.IsActive = false;

if (tag != null)

{

//Player.SetCoins(tag, 0);

if (tag == "Red Player")

Player.SetCoins("Blue Player", -10);

else if (tag == "Blue Player")

Player.SetCoins("Red Player", -10);

}

//GameEvents.ChangeCoins?.Invoke(playerGameObject.GameObjectTag, storedCoins);

if (playerGameObject.GameObjectTag == "Red Player" && Player.RPCoins != 1)

(playerGameObject.Script as Player).ChangeStatsValue(player,(-Player.RPCoins / 2), "Death");

else if (playerGameObject.GameObjectTag == "Blue Player" && Player.BPCoins != 1)

(playerGameObject.Script as Player).ChangeStatsValue(player, (-Player.BPCoins / 2), "Death");

else

(playerGameObject.Script as Player).ChangeStatsValue(player, -power, "Death");

GameEvents.ChangeEffect?.Invoke(playerGameObject.GameObjectTag, "Death");

}

}

if (playerGameObject.Script is Monster)

{

if ((playerGameObject.Script as Monster).Health > 0)

{

(playerGameObject.Script as Monster).ChangeStatsValue(player, -power);

playerGameObject.Sprite.Bitmap = null;

}

if ((playerGameObject.Script as Monster).Health <= 0)

{

playerGameObject.IsActive = false;

playerGameObject.Collider.IsInactive = true;

if(tag != null)

{

Player.SetCoins(tag, 10);

GameEvents.ChangeCoins?.Invoke(tag, 10);

}

GameEvents.ChangeEffect?.Invoke(playerGameObject.GameObjectTag, "Death");

}

}

}

/// <summary>

/// Инициализация эффекта

/// </summary>

protected override void Initialize() { }

/// <summary>

/// Деактивация эффекта

/// </summary>

protected override void DeactivateEffect(GameObject gameObject)

{

if (playerGameObject.IsActive != true)

{

if (playerGameObject.GameObjectTag == "Blue Player")

playerGameObject.Transform.Position = Platformer.BluePlayerFactory.StartPosition;

else if (playerGameObject.GameObjectTag == "Red Player")

playerGameObject.Transform.Position = Platformer.RedPlayerFactory.StartPosition;

playerGameObject.IsActive = true;

}

Platformer.RemoveObjectFromScene(gameObject);

GameEvents.ChangeEffect?.Invoke(playerGameObject.GameObjectTag, "");

}

/// <summary>

/// Поведение на сцене

/// </summary>

protected override void BehaviorOnScene(GameObject gameObject)

{

//if (gameObject.Sprite.Bitmap == null)

//{

// gameObject.Sprite.SetAnimation("idle");

//}

if (gameObject.Collider.CheckIntersection(out GameObject player) && gameObject.IsActive)

{

if (playerGameObject.GameObjectTag == player.GameObjectTag)

gameObject.IsActive = false;

//player.ChangeStatsValue(storedHealth);

//player.ChangeStatsValue(storedCoins);

}

if (playerGameObject.IsActive)

{

Platformer.RemoveObjectFromScene(gameObject);

}

}

}

}

**Код программы для *Effect.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Game;

using GameLibrary.Platformer;

namespace GameLibrary.Effects

{

/// <summary>

/// Асбтрактный класс эффекта

/// </summary>

public abstract class Effect : ObjectScript

{

/// <summary>

/// Время дейтсвия эффекта

/// </summary>

public abstract float EffectTime { get; }

/// <summary>

/// Экземпляр сцены игры

/// </summary>

protected PlatformerScene Platformer;

/// <summary>

/// Ссылка на сценарий игрока, на которого действует эффект

/// </summary>

protected Player playerScript;

/// <summary>

/// Ссылка на игровой объект, на которого действует эффект

/// </summary>

protected GameObject playerGameObject;

protected float currentEffectTime;

/// <summary>

/// Поведение на момент создание игрового объекта

/// </summary>

public override void Start(GameObject gameObject = null)

{

Platformer = PlatformerScene.instance;

currentEffectTime = Time.CurrentTime + EffectTime;

Initialize();

}

/// <summary>

/// Обновление игрового объекта

/// </summary>

public override void Update(GameObject gameObject)

{

if (currentEffectTime < Time.CurrentTime)

DeactivateEffect(gameObject);

BehaviorOnScene(gameObject);

}

/// <summary>

/// Активация эффекта

/// </summary>

/// <param name="player">Игровой объект, на который будет наложен эффект</param>

public abstract void ActivateEffect(GameObject player, string tag = null, float power = 1);

/// <summary>

/// Инициализация эффекта

/// </summary>

protected abstract void Initialize();

/// <summary>

/// Деактивация эффекта

/// </summary>

protected abstract void DeactivateEffect(GameObject gameObject);

/// <summary>

/// Поведение на сцене

/// </summary>

protected abstract void BehaviorOnScene(GameObject gameObject);

}

}

**Код программы для *DecoratorProperty.cs*:**

namespace GameLibrary.Game

{

/// <summary>

///

/// </summary>

/// <seealso cref="GameLibrary.Game.PlayerProperities" />

public class DecoratorProperty : PlayerProperities

{

/// <summary>

/// The player properities

/// </summary>

protected PlayerProperitiesStandart playerProperities;

/// <summary>

/// Конструктор объекта

/// </summary>

/// <param name="playerProperities"></param>

public DecoratorProperty(PlayerProperitiesStandart playerProperities)

{

this.playerProperities = playerProperities;

}

/// <summary>

/// Запас здоровья игрока

/// </summary>

public override int Health { get => playerProperities.Health; protected set => playerProperities.SetProperty( TypeProperty.Health,value); }

/// <summary>

/// Боезапас

/// </summary>

public override int Ammo { get => playerProperities.Ammo; protected set => playerProperities.SetProperty(TypeProperty.Ammo, value); }

/// <summary>

/// Скорость

/// </summary>

public override float Speed { get => playerProperities.Speed; }

/// <summary>

/// Время перезарядки оружия

/// </summary>

public override float ReloadTime { get => playerProperities.ReloadTime; }

/// <summary>

/// Power

/// </summary>

public override float Power { get => playerProperities.Power; }

/// <summary>

/// Deactivates the properities.

/// </summary>

/// <param name="player">The player.</param>

protected override void DeactivateProperities(Player player)

{

player.SetProperty(new PlayerProperitiesStandart());

}

}

}

**Код программы для *GameEvents.cs*:**

namespace GameLibrary.Game

{

/// <summary>

/// Статический класс событий игры

/// </summary>

public static class GameEvents

{

/// <summary>

/// Делегат события изменения количества здоровья

/// </summary>

/// <param name="tagPlayer">Тег игрового объекта игрока</param>

/// <param name="value">Значение собранных монет</param>

public delegate void HealthDelegate(string tagPlayer, int value);

/// <summary>

/// Событие изменения количества здоровья

/// </summary>

public static HealthDelegate ChangeHealth { get; set; }

/// <summary>

/// Делегат события изменения количества монет

/// </summary>

/// <param name="tagPlayer">Тег игрового объекта игрока</param>

/// <param name="value">Значение собранных монет</param>

public delegate void CoinsDelegate(string tagPlayer, int value);

/// <summary>

/// Событие изменения количества монет

/// </summary>

///

public static CoinsDelegate ChangeCoins { get; set; }

/// <summary>

/// Делегат события окончания игры

/// </summary>

public delegate void EndGameDelegate(string winPlayer);

/// <summary>

/// Событие окончания игры

/// </summary>

public static EndGameDelegate EndGame { get; set; }

/// <summary>

/// Делегат события получения эффекта игроком

/// </summary>

/// <param name="tagPlayer">Тег игрового объекта игрока</param>

/// <param name="effectName">Название эффекта</param>

public delegate void EffectDelegate(string tagPlayer, string effectName);

/// <summary>

/// Событие получения эффекта игроком

/// </summary>

public static EffectDelegate ChangeEffect { get; set; }

/// <summary>

/// Делегат события получения оружия игроком

/// </summary>

/// <param name="tagPlayer">Тег игрового объекта игрока</param>

/// <param name="gunName">Название оружия</param>

public delegate void GunDelegate(string tagPlayer, string gunName);

/// <summary>

/// Событие получения оружия игроком

/// </summary>

public static GunDelegate ChangeGun { get; set; }

/// <summary>

/// Делегат события получения пуль игроком

/// </summary>

/// <param name="tagPlayer">Тег игрового объекта игрока</param>

public delegate void CountBullets(string tagPlayer, int count);

/// <summary>

/// Событие получения пуль игроком

/// </summary>

public static CountBullets ChangeCount { get; set; }

}

}

**Код программы для *Player.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using SharpDX;

using SharpDX.DirectInput;

using GameLibrary.Platformer;

namespace GameLibrary.Game

{

/// <summary>

/// Класс, описывающий сценарий поведения игрока

/// </summary>

public class Player : ObjectScript

{

/// <summary>

/// Управление игроком

/// </summary>

public PlayerControl Control { get; private set; }

/// <summary>

/// Возможность двигаться у игрока

/// </summary>

public bool IsCanMove { get; set; } = true;

/// <summary>

/// Полученные очки

/// </summary>

/// <value>

/// Gets the rp coins.

/// </value>

public static int RPCoins { get; private set; } = 0;

/// <value>

/// Gets the bp coins.

/// </value>

public static int BPCoins { get; private set; } = 0;

/// <summary>

/// The child game object

/// </summary>

private GameObject childGameObject;

/// <summary>

/// Gets the property.

/// </summary>

public PlayerProperities Property { get; private set; }

/// <summary>

/// Sets the property.

/// </summary>

/// <param name="property">The property.</param>

public void SetProperty(PlayerProperities property)

{

property.SetProperty(TypeProperty.Health, Property.Health);

property.SetProperty(TypeProperty.Ammo, Property.Ammo);

Property = property;

}

/// <summary>

/// Поведение на момент создание игрового объекта

/// </summary>

public override void Start(GameObject gameObject = null)

{

Animation animation;

Property = new PlayerProperitiesStandart();

Property.SetProperty(TypeProperty.Health, 10);

Property.SetProperty(TypeProperty.Ammo, 10);

if (gameObject.GameObjectTag == "Blue Player")

{

animation = new Animation(RenderingSystem.LoadAnimation("Resources/Blue Player/left idle ", 2), 0.2f, true);

gameObject.Sprite.AddAnimation("idleLeft", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/Blue Player/left run ", 4), 0.2f, true);

gameObject.Sprite.AddAnimation("runLeft", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/Blue Player/right idle ", 2), 0.2f, true);

gameObject.Sprite.AddAnimation("idleRight", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/Blue Player/right run ", 4), 0.2f, true);

gameObject.Sprite.AddAnimation("runRight", animation);

Control = new PlayerControl(AxisOfInput.Horizontal, AxisOfInput.Vertical, Key.Space, Key.C);

GameEvents.ChangeHealth?.Invoke(gameObject.GameObjectTag, Property.Health);

GameEvents.ChangeCoins?.Invoke(gameObject.GameObjectTag, BPCoins);

}

else

{

animation = new Animation(RenderingSystem.LoadAnimation("Resources/Red Player/left idle ", 2), 0.2f, true);

gameObject.Sprite.AddAnimation("idleLeft", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/Red Player/left run ", 4), 0.2f, true);

gameObject.Sprite.AddAnimation("runLeft", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/Red Player/right idle ", 2), 0.2f, true);

gameObject.Sprite.AddAnimation("idleRight", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/Red Player/right run ", 4), 0.2f, true);

gameObject.Sprite.AddAnimation("runRight", animation);

Control = new PlayerControl(AxisOfInput.AlternativeHorizontal, AxisOfInput.AlternativeVertical, Key.RightShift, Key.Backslash);

GameEvents.ChangeHealth?.Invoke(gameObject.GameObjectTag, Property.Health);

GameEvents.ChangeCoins?.Invoke(gameObject.GameObjectTag, RPCoins);

}

gameObject.Sprite.SetAnimation("idleLeft");

}

/// <summary>

/// Установка дочернего объекта

/// </summary>

/// <param name="gameObject">Дочерний объект</param>

public void SetChildGameObject(GameObject gameObject)

{

if (childGameObject != null)

PlatformerScene.instance.RemoveObjectFromScene(childGameObject);

childGameObject = gameObject;

}

/// <summary>

/// Обновление игрового объекта

/// </summary>

public override void Update(GameObject gameObject)

{

if (gameObject.IsActive && IsCanMove)

Move(gameObject);

if (Property != null)

Property.UpdateTime(this);

}

/// <summary>

/// Изменение значения характеристик игрока

/// </summary>

/// <param name="value">Значение, которое прибавляется к текущему значению монет</param>

public void ChangeStatsValue(GameObject gameObject, float value)

{

if (gameObject.Collider.CheckIntersection("Bullet") || Property.Health <= 10)

{

Property.SetProperty(TypeProperty.Health, Property.Health + value);

GameEvents.ChangeHealth?.Invoke(gameObject.GameObjectTag, Property.Health);

}

}

public void ChangeStatsValue(GameObject gameObject, float value, string gametag)

{

if (gametag == "Death")

{

Property.SetProperty(TypeProperty.Health, 10);

//Coins += value;

GameEvents.ChangeHealth?.Invoke(gameObject.GameObjectTag, Property.Health);

//GameEvents.ChangeCoins?.Invoke(gameObject.GameObjectTag, Coins);

}

}

/// <summary>

/// Метод движения игрока

/// </summary>

private void Move(GameObject gameObject)

{

int directionX = 0, directionY = 0;

directionX = Input.GetAxis(Control.HorizontalAxis);

if (gameObject.Collider.CheckIntersection("Stair"))

{

directionY = Input.GetAxis(Control.VerticalAxis);

gameObject.Transform.IsUseGravitation = gameObject.Collider.CheckIntersection("Wall");

}

else

{

gameObject.Transform.IsUseGravitation = true;

}

Vector2 direction;

if (directionX == 0)

{

if(childGameObject != null)

childGameObject.Sprite.IsFlipX = gameObject.Sprite.IsFlipX;

if (childGameObject != null && childGameObject.Sprite.IsFlipX)

childGameObject.Sprite.SetAnimation("idleLeft");

else if (childGameObject != null)

childGameObject.Sprite.SetAnimation("idleRight");

if (gameObject.Sprite.IsFlipX)

gameObject.Sprite.SetAnimation("idleLeft");

else

gameObject.Sprite.SetAnimation("idleRight");

direction = new Vector2(0, directionY);

}

else

{

gameObject.Sprite.IsFlipX = directionX < 0;

if (childGameObject != null)

childGameObject.Sprite.IsFlipX = gameObject.Sprite.IsFlipX;

if (childGameObject != null && childGameObject.Sprite.IsFlipX)

childGameObject.Sprite.SetAnimation("runLeft");

else if (childGameObject != null)

childGameObject.Sprite.SetAnimation("runRight");

if (gameObject.Sprite.IsFlipX)

gameObject.Sprite.SetAnimation("runLeft");

else

gameObject.Sprite.SetAnimation("runRight");

direction = new Vector2(directionX, 0);

}

Vector2 movement = direction \* Property.Speed \* Time.DeltaTime;

gameObject.Transform.SetMovement(movement);

DetectCollision(gameObject);

}

/// <summary>

/// Распознавание столкновений и реакция на них

/// </summary>

private void DetectCollision(GameObject gameObject)

{

if (gameObject.Collider.CheckIntersection("Wall","BreakWall"))

{

gameObject.Transform.ResetMovement();

}

if (gameObject.GameObjectTag == "Blue Player" || gameObject.GameObjectTag == "Red Player")

gameObject.Transform.AddGravitation();

string tag = (Input.GetAxis(Control.VerticalAxis) == -1) ? "" : "Platform";

if (gameObject.Collider.CheckIntersection("Wall", tag))

{

gameObject.Transform.ResetGravitation();

}

}

/// <summary>

/// Sets the points.

/// </summary>

/// <param name="tag">The tag.</param>

/// <param name="value">The value.</param>

public static void SetCoins(string tag, int value)

{

if (tag == "Red Player")

{

RPCoins += value;

GameEvents.ChangeCoins?.Invoke(tag, value);

}

else

{

BPCoins += value;

GameEvents.ChangeCoins?.Invoke(tag, value);

}

}

}

/// <summary>

/// Структура игрового управления персонажа

/// </summary>

public struct PlayerControl

{

/// <summary>

/// Горизонтальная ось передвижения

/// </summary>

public AxisOfInput HorizontalAxis { get; private set; }

/// <summary>

/// Вертикальная ось передвижения

/// </summary>

public AxisOfInput VerticalAxis { get; private set; }

/// <summary>

/// Кнопка стрельбы

/// </summary>

public Key ShootKey { get; private set; }

/// <summary>

/// Кнопка стрельбы

/// </summary>

public Key GetKey { get; private set; }

/// <summary>

/// Конструктор структуры

/// </summary>

/// <param name="horizontalAxis">Горизонтальная ось передвижения</param>

/// <param name="verticalAxis"> Вертикальная ось передвижения</param>

/// <param name="shootKey">Кнопка стрельбы</param>

public PlayerControl(AxisOfInput horizontalAxis, AxisOfInput verticalAxis, Key shootKey, Key getKey)

{

HorizontalAxis = horizontalAxis;

VerticalAxis = verticalAxis;

ShootKey = shootKey;

GetKey = getKey;

}

}

}

**Код программы для *PlayerConstructor.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Guns;

using SharpDX;

namespace GameLibrary.Game

{

/// <summary>

/// Класс фабрики создания персонажа

/// </summary>

public class PlayerConstructor

{

/// <summary>

/// Игровой Объект игрока

/// </summary>

public GameObject PlayerGameObject { get; private set; }

/// <summary>

/// Тег плеера

/// </summary>

public string PlayerTag { get; private set; }

/// <summary>

/// Начальная позиция игрока в платформере

/// </summary>

public Vector2 StartPosition { get; set; }

/// <summary>

/// Создание игрового объекта персонажа

/// </summary>

/// <returns>Игровой объект</returns>

public GameObject CreatePlayer(string path)

{

PlayerTag = path;

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(StartPosition, new Size2F(1f, 1f)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/" + PlayerTag + "/left idle 1.png")));

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(0.5f, 0.5f), new Vector2(0.2f, 0.2f)));

gameObject.GameObjectTag = PlayerTag;

Player playerScript = new Player();

gameObject.InitializeObjectScript(playerScript);

PlayerGameObject = gameObject;

return gameObject;

}

/// <summary>

/// Создание игрового объектиа оружия

/// </summary>

/// <returns>Игровой объект</returns>

public GameObject CreateGun(Gun gun, string nameOfgun)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(new Vector2(0f, 0f), new Size2F(1, 1)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/" + PlayerTag + "/" + nameOfgun + " Gun/" + nameOfgun + " left idle 1.png")));

gameObject.GameObjectTag = PlayerGameObject.GameObjectTag;

gameObject.ParentGameObject = PlayerGameObject;

gun.PlayerFactory = this;

gameObject.InitializeObjectScript(gun);

Player playerScript = PlayerGameObject.Script as Player;

playerScript.SetChildGameObject(gameObject);

GameEvents.ChangeGun?.Invoke(PlayerTag, nameOfgun);

return gameObject;

}

}

}

**Код программы для *PlayerProperties.cs*:**

using EngineLibrary.EngineComponents;

namespace GameLibrary.Game

{

/// <summary>

///

/// </summary>

public abstract class PlayerProperities

{

/// <summary>

/// The time deactivate

/// </summary>

protected float timeDeactivate = 2;

/// <summary>

/// The timer

/// </summary>

protected float timer = 0;

/// <summary>

/// Запас здоровья игрока

/// </summary>

public abstract int Health { get; protected set; }

/// <summary>

/// Боезапас

/// </summary>

public abstract int Ammo { get; protected set; }

/// <summary>

/// Скорость

/// </summary>

public abstract float Speed { get; }

/// <summary>

/// Время перезарядки оружия

/// </summary>

public abstract float ReloadTime { get; }

/// <summary>

/// Power

/// </summary>

public abstract float Power { get; }

/// <summary>

/// Sets the property.

/// </summary>

/// <param name="type">The type.</param>

/// <param name="value">The value.</param>

public virtual void SetProperty(TypeProperty type, float value)

{

switch (type)

{

case TypeProperty.Health:

Health = (int)value;

break;

case TypeProperty.Ammo:

Ammo = (int)value;

break;

}

}

/// <summary>

/// Updates the time.

/// </summary>

/// <param name="player">The player.</param>

public virtual void UpdateTime(Player player)

{

timer += Time.DeltaTime;

if (timer >= timeDeactivate)

{

timer = 0;

DeactivateProperities(player);

}

}

/// <summary>

/// Deactivates the properities.

/// </summary>

/// <param name="player">The player.</param>

protected abstract void DeactivateProperities(Player player);

}

}

**Код программы для *PlayerPropertiesStandart.cs*:**

namespace GameLibrary.Game

{

/// <summary>

/// Класс с характеристиками игрока

/// </summary>

public class PlayerProperitiesStandart : PlayerProperities

{

/// <summary>

/// Запас здоровья игрока

/// </summary>

public override int Health { get; protected set; }

/// <summary>

/// Боезапас

/// </summary>

public override int Ammo { get; protected set; }

/// <summary>

/// Скорость

/// </summary>

public override float Speed { get; } = 5;

/// <summary>

/// Время перезарядки оружия

/// </summary>

public override float ReloadTime { get; } = 0.5f;

/// <summary>

/// Power

/// </summary>

public override float Power { get; } = 1;

/// <summary>

/// Deactivates the properities.

/// </summary>

/// <param name="player">The player.</param>

protected override void DeactivateProperities(Player player)

{

}

}

}

**Код программы для *PowerDecorator.cs*:**

namespace GameLibrary.Game

{

/// <summary>

///

/// </summary>

/// <seealso cref="GameLibrary.Game.DecoratorProperty" />

public class PowerDecorator : DecoratorProperty

{

/// <summary>

/// Initializes a new instance of the <see cref="PowerDecorator"/> class.

/// </summary>

/// <param name="playerProperities"></param>

public PowerDecorator(PlayerProperitiesStandart playerProperities) : base(playerProperities)

{

}

/// <summary>

/// Power

/// </summary>

public override float Power { get => playerProperities.Power \* 2; }

}

}

**Код программы для *ReloadTimeDecorator.cs*:**

namespace GameLibrary.Game

{

/// <summary>

///

/// </summary>

/// <seealso cref="GameLibrary.Game.DecoratorProperty" />

public class ReloadTimeDecorator : DecoratorProperty

{

/// <summary>

/// Initializes a new instance of the <see cref="ReloadTimeDecorator"/> class.

/// </summary>

/// <param name="playerProperities"></param>

public ReloadTimeDecorator(PlayerProperitiesStandart playerProperities) : base(playerProperities)

{

}

/// <summary>

/// Время перезарядки

/// </summary>

public override float ReloadTime { get => playerProperities.ReloadTime / 2; }

}

}

**Код программы для *SpeedDecorator.cs*:**

namespace GameLibrary.Game

{

/// <summary>

///

/// </summary>

/// <seealso cref="GameLibrary.Game.DecoratorProperty" />

public class SpeedDecorator : DecoratorProperty

{

/// <summary>

/// Initializes a new instance of the <see cref="SpeedDecorator"/> class.

/// </summary>

/// <param name="playerProperities"></param>

public SpeedDecorator(PlayerProperitiesStandart playerProperities) : base(playerProperities)

{

}

/// <summary>

/// Скорость

/// </summary>

public override float Speed { get => playerProperities.Speed \* 1.5f; }

}

}

**Код программы для *TypeProperty.cs*:**

namespace GameLibrary

{

/// <summary>

///

/// </summary>

public enum TypeProperty

{

/// <summary>

/// The health

/// </summary>

Health,

/// <summary>

/// The ammo

/// </summary>

Ammo,

/// <summary>

/// The speed

/// </summary>

Speed,

/// <summary>

/// The reload time

/// </summary>

ReloadTime,

/// <summary>

/// The power

/// </summary>

Power

}

}

**Код программы для *DamageGun.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Bullets.BulletFactories;

using SharpDX;

namespace GameLibrary.Guns

{

/// <summary>

/// Класс оружия урона

/// </summary>

public class DamageGun : Gun

{

/// <summary>

/// Загрузка анимации оружия

/// </summary>

protected override void LoadAnimation(GameObject gameObject)

{

Animation animation;

if (PlayerFactory != null)

{

animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + PlayerFactory.PlayerTag + "/Damage Gun/damage left idle ", 2), 0.2f, true);

gameObject.Sprite.AddAnimation("idleLeft", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + PlayerFactory.PlayerTag + "/Damage Gun/damage left run ", 4), 0.2f, true);

gameObject.Sprite.AddAnimation("runLeft", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + PlayerFactory.PlayerTag + "/Damage Gun/damage right idle ", 2), 0.2f, true);

gameObject.Sprite.AddAnimation("idleRight", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + PlayerFactory.PlayerTag + "/Damage Gun/damage right run ", 4), 0.2f, true);

gameObject.Sprite.AddAnimation("runRight", animation);

gameObject.Sprite.SetAnimation("idleLeft");

}

else

{

animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + "Blue Player" + "/Damage Gun/damage left idle ", 2), 0.2f, true);

gameObject.Sprite.AddAnimation("idleLeft", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + "Blue Player" + "/Damage Gun/damage left run ", 4), 0.2f, true);

gameObject.Sprite.AddAnimation("runLeft", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + "Blue Player" + "/Damage Gun/damage right idle ", 2), 0.2f, true);

gameObject.Sprite.AddAnimation("idleRight", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + "Blue Player" + "/Damage Gun/damage right run ", 4), 0.2f, true);

gameObject.Sprite.AddAnimation("runRight", animation);

gameObject.Sprite.SetAnimation("idleLeft");

}

}

/// <summary>

/// Создание пули из фабрики

/// </summary>

/// <param name="position">Позиция создания</param>

/// <param name="direction">Направление пули</param>

protected override void SpawnBullet(GameObject gameObject, Vector2 position, Vector2 direction, float power = 1)

{

DamageBulletFactory factory = new DamageBulletFactory();

Platformer.AddObjectOnScene(factory.CreateBullet(position, direction, gameObject.GameObjectTag, power));

}

}

}

**Код программы для *HealthKit.cs*:**

using EngineLibrary.ObjectComponents;

using GameLibrary.Game;

namespace GameLibrary.Platformer

{

/// <summary>

/// Класс аптечки

/// </summary>

public class HealthKit : ObjectScript

{

/// <summary>

/// The Platformer

/// </summary>

private PlatformerScene Platformer;

/// <summary>

/// Поведение на момент создание игрового объекта

/// </summary>

public override void Start(GameObject gameObject = null)

{

Platformer = PlatformerScene.instance;

}

/// <summary>

/// Обновление игрового объекта

/// </summary>

public override void Update(GameObject gameObject)

{

if (gameObject.Collider.CheckIntersection(out GameObject player))

{

if ((player.Script as Player).Property.Health < 10)

{

(player.Script as Player).ChangeStatsValue(player, 1);

Platformer.RemoveObjectFromScene(gameObject);

}

}

}

}

}

**Код программы для *PlatformerElementsFactory.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Guns;

using GameLibrary.Monsters;

using SharpDX;

namespace GameLibrary.Platformer

{

/// <summary>

/// Класс фабрики создания элементов экшен-платформер

/// </summary>

public class PlatformerElementsFactory

{

/// <summary>

/// Создает элемент экшен-платформер

/// </summary>

/// <param name="position">Позиция объекта на сцене</param>

/// <param name="TagName">Тег игрового объекта</param>

/// <returns>Созданный игровой объект</returns>

public GameObject CreatePlatformerElement(Vector2 position, string TagName)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(position, new Size2F(1f, 1f)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/PlatformerElements/" + TagName + ".png")));

if (TagName == "Platform")

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(1f, 0.1f), new Vector2(0, -0.5f)));

else if (TagName == "Stair")

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(1f, 0.5f)));

else if (TagName == "Wall")

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(1f, 1f)));

else

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(1f, 1f)));

gameObject.GameObjectTag = TagName;

if (TagName == "BreakWall")

gameObject.InitializeObjectScript(new BreakWall());

return gameObject;

}

/// <summary>

/// Создание монет в платформере

/// </summary>

/// <param name="position">Позиция объекта на сцене</param>

/// <returns>Игровой объект</returns>

public GameObject CreateHealthKit(Vector2 position)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(position, new Size2F(1f, 1f)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/PlatformerElements/HealthKit.png")));

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(1f, 1f)));

gameObject.GameObjectTag = "HealthKit";

gameObject.InitializeObjectScript(new HealthKit());

return gameObject;

}

/// <summary>

/// Создание монет в платформере

/// </summary>

/// <param name="position">Позиция объекта на сцене</param>

/// <returns>Игровой объект</returns>

public GameObject CreateMonsters(Vector2 position)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(position, new Size2F(1f, 1f)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/" + "Red Player" + "/left idle 1.png")));

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(0.5f, 0.5f), new Vector2(0, 0.2f)));

gameObject.GameObjectTag = "Monster";

var monsterScript = new Monster();

gameObject.InitializeObjectScript(monsterScript);

return gameObject;

}

/// <summary>

/// Создание игрового объекта рук

/// </summary>

/// <returns>Игровой объект</returns>

public GameObject CreateArms(GameObject monster)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(new Vector2(0f, 0f), new Size2F(1, 1)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/" + "Blue Player" + "/Arms/arms left idle 1.png")));

gameObject.GameObjectTag = "Arms";

Animation animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + "Blue Player" + "/Arms/arms left idle ", 2), 0.2f, true);

gameObject.Sprite.AddAnimation("idleLeft", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + "Blue Player" + "/Arms/arms left run ", 4), 0.2f, true);

gameObject.Sprite.AddAnimation("runLeft", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + "Blue Player" + "/Arms/arms right idle ", 2), 0.2f, true);

gameObject.Sprite.AddAnimation("idleRight", animation);

animation = new Animation(RenderingSystem.LoadAnimation("Resources/" + "Blue Player" + "/Arms/arms right run ", 4), 0.2f, true);

gameObject.Sprite.AddAnimation("runRight", animation);

gameObject.Sprite.SetAnimation("idleLeft");

Monster playerScript = monster.Script as Monster;

playerScript.SetChildGameObject(gameObject);

gameObject.ParentGameObject = monster;

return gameObject;

}

/// <summary>

/// Создание игрового объектиа оружия

/// </summary>

/// <returns>Игровой объект</returns>

public GameObject CreateGun(Gun gun, string nameOfgun, GameObject monster)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(new Vector2(0f, 0f), new Size2F(1, 1)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/" + "Blue Player" + "/" + nameOfgun + " Gun/" + nameOfgun + " left idle 1.png")));

gameObject.GameObjectTag = "Gun";

gameObject.ParentGameObject = monster;

gameObject.InitializeObjectScript(gun);

Monster playerScript = monster.Script as Monster;

playerScript.SetChildGameObject(gameObject);

return gameObject;

}

}

}

**Код программы для *PlatformerScene.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using SharpDX;

using System;

using System.Drawing;

using System.Collections.Generic;

using GameLibrary.Game;

using GameLibrary.Monsters;

using GameLibrary.Guns;

namespace GameLibrary.Platformer

{

/// <summary>

/// Класс экшен-платформер

/// </summary>

public class PlatformerScene : Scene

{

/// <summary>

/// Статическая ссылка на класс

/// </summary>

public static PlatformerScene instance = null;

/// <summary>

/// Фабрика создания элементов экшен-платформер

/// </summary>

public PlatformerElementsFactory ElementsFactory { get; private set; }

/// <summary>

/// Конструктор синего игрока

/// </summary>

public PlayerConstructor BluePlayerFactory { get; set; }

/// <summary>

/// Конструктор красного игрока

/// </summary>

public PlayerConstructor RedPlayerFactory { get; set; }

/// <summary>

/// The empty blocks

/// </summary>

private readonly List<Vector2> emptyBlocks = new List<Vector2>();

/// <summary>

/// The count of health kits

/// </summary>

public int countOfHealthKits = 0;

/// <summary>

/// Counts the empty blocks.

/// </summary>

/// <returns></returns>

public int CountEmptyBlocks()

{

return emptyBlocks.Count;

}

/// <summary>

/// Создание игровых объектов на сцене

/// </summary>

protected override void CreateGameObjectsOnScene()

{

if (instance == null)

instance = this;

ElementsFactory = new PlatformerElementsFactory();

BluePlayerFactory = new PlayerConstructor();

RedPlayerFactory = new PlayerConstructor();

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(new Vector2(0f, 0f), new Size2F(27, 15)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/Фон.png")));

gameObject.GameObjectTag = "Background";

gameObjects.Add(gameObject);

gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(new Vector2(0f, 0f), new Size2F(1, 1)));

gameObject.GameObjectTag = "GameManager";

gameObject.InitializeObjectScript(new SpawnManager());

gameObjects.Add(gameObject);

CreatePlatformer();

GameObject player = BluePlayerFactory.CreatePlayer("Blue Player");

gameObjects.Add(player);

gameObjects.Add(BluePlayerFactory.CreateGun(new DamageGun(), "damage"));

player = RedPlayerFactory.CreatePlayer("Red Player");

gameObjects.Add(player);

gameObjects.Add(RedPlayerFactory.CreateGun(new DamageGun(), "damage"));

}

/// <summary>

/// Метод создания экшен-платформер

/// </summary>

public void CreatePlatformer()

{

Random random = new Random();

string text = "Resources/Platformers/Platformer " + random.Next(3, 4).ToString() + ".bmp";

Bitmap bitmap = new Bitmap(text);

for (int i = 0; i < bitmap.Height; i++)

{

for (int j = 0; j < bitmap.Width; j++)

{

System.Drawing.Color color = bitmap.GetPixel(j, i);

GameObject gameObject = null;

if (color.R == 0 && color.G == 0 && color.B == 0)

gameObject = ElementsFactory.CreatePlatformerElement(new Vector2(j, i), "Wall");

else if (color.R == 255 && color.G == 0 && color.B == 0)

gameObject = ElementsFactory.CreatePlatformerElement(new Vector2(j, i), "BreakWall");

else if (color.R == 0 && color.G == 255 && color.B == 0)

gameObject = ElementsFactory.CreatePlatformerElement(new Vector2(j, i), "Platform");

else if (color.R == 0 && color.G == 0 && color.B == 255)

gameObject = ElementsFactory.CreatePlatformerElement(new Vector2(j, i), "Stair");

else if (color.R == 255 && color.G == 255 && color.B == 0)

{

gameObject = ElementsFactory.CreateHealthKit(new Vector2(j, i));

countOfHealthKits++;

}

else if (color.R == 0 && color.G == 255 && color.B == 255)

{

gameObject = ElementsFactory.CreateMonsters(new Vector2(j, i));

gameObjects.Add(ElementsFactory.CreateGun(new DamageGun(), "damage", gameObject));

}

else if (color.R == 125 && color.G == 0 && color.B == 0)

RedPlayerFactory.StartPosition = new Vector2(j, i);

else if (color.R == 0 && color.G == 0 && color.B == 125)

BluePlayerFactory.StartPosition = new Vector2(j, i);

else

emptyBlocks.Add(new Vector2(j, i));

if (gameObject != null)

gameObjects.Add(gameObject);

}

}

}

/// <summary>

/// Добавление объекта в лист отрисовки

/// </summary>

/// <param name="gameObject">Игровой объект</param>

public void AddObjectOnScene(GameObject gameObject)

{

gameObjectsToAdd.Add(gameObject);

}

/// <summary>

/// Удаления объекта из листа отрисовки

/// </summary>

/// <param name="gameObject">Игровой объект</param>

public void RemoveObjectFromScene(GameObject gameObject)

{

int count = 0;

if (gameObject.GameObjectTag == "Spawn")

emptyBlocks.Add(gameObject.Transform.Position);

gameObjectsToRemove.Add(gameObject);

foreach (var monsterObject in gameObjects)

{

if (monsterObject.Script is Monster)

{

count++;

}

}

if (count == 0)

{

EndScene();

}

}

/// <summary>

/// Рандомное место в платформере

/// </summary>

/// <returns>Позицию</returns>

public Vector2 GetRandomPosition()

{

Random random = new Random();

int index = random.Next(0, emptyBlocks.Count);

Vector2 position = emptyBlocks[index];

emptyBlocks.Remove(position);

return position;

}

/// <summary>

/// Поведение при завершении сцены

/// </summary>

protected override void EndScene()

{

base.EndScene();

string winPlayer;

if(Player.BPCoins < Player.RPCoins)

winPlayer = RedPlayerFactory.PlayerTag;

else

winPlayer = BluePlayerFactory.PlayerTag;

GameEvents.EndGame?.Invoke(winPlayer);

}

}

}

**Код программы для *PrizeSpawn.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Game;

namespace GameLibrary.Platformer

{

/// <summary>

/// Класс подбираемого оружия в платформере

/// </summary>

public class PrizeSpawn : ObjectScript

{

/// <summary>

/// The Platformer

/// </summary>

private PlatformerScene Platformer;

/// <summary>

/// The drop out prize

/// </summary>

private PlayerProperities dropOutPrize;

/// <summary>

/// The cuurent time to disappear

/// </summary>

private float cuurentTimeToDisappear;

/// <summary>

/// Инициализация места подбираемого оружия

/// </summary>

/// <param name="name">Название оружия</param>

/// <param name="gun">Подбираемое оружие</param>

/// <param name="disappearTime">Время исчезнование места</param>

public void InitializeGunSpawn(PlayerProperities playerProperities, float disappearTime)

{

dropOutPrize = playerProperities;

//nameOfgun = name;

cuurentTimeToDisappear = Time.CurrentTime + disappearTime;

}

/// <summary>

/// Инициализация места подбираемого оружия

/// </summary>

/// <param name="name">Название оружия</param>

/// <param name="gun">Подбираемое оружие</param>

/// <param name="disappearTime">Время исчезнование места</param>

public void InitializeGunSpawn(float disappearTime)

{

cuurentTimeToDisappear = Time.CurrentTime + disappearTime;

}

/// <summary>

/// Поведение на момент создание игрового объекта

/// </summary>

public override void Start(GameObject gameObject = null)

{

Platformer = PlatformerScene.instance;

}

/// <summary>

/// Обновление игрового объекта

/// </summary>

public override void Update(GameObject gameObject)

{

if(cuurentTimeToDisappear < Time.CurrentTime)

{

Platformer.RemoveObjectFromScene(gameObject);

}

if(gameObject.Collider.CheckIntersection(out GameObject player,"Blue Player","Red Player"))

{

if (player.GameObjectTag == "Blue Player" && Input.GetButtonDawn((player.Script as Player).Control.GetKey))

{

if (dropOutPrize == null)

{

(player.Script as Player).Property.SetProperty(TypeProperty.Ammo, 10);

if ((player.Script as Player).Property.Ammo >= 0)

GameEvents.ChangeCount?.Invoke(player.GameObjectTag, (player.Script as Player).Property.Ammo);

}

else

(player.Script as Player).SetProperty(dropOutPrize);

Platformer.RemoveObjectFromScene(gameObject);

}

else if (player.GameObjectTag == "Red Player" && Input.GetButtonDawn((player.Script as Player).Control.GetKey))

{

if (dropOutPrize == null)

{

(player.Script as Player).Property.SetProperty(TypeProperty.Ammo, 10);

}

else

(player.Script as Player).SetProperty(dropOutPrize);

Platformer.RemoveObjectFromScene(gameObject);

}

}

}

}

}

**Код программы для *SpawnManager.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using SharpDX;

using System;

namespace GameLibrary.Platformer

{

/// <summary>

/// Класс игрового менеджера

/// </summary>

public class SpawnManager : ObjectScript

{

/// <summary>

/// The time to spawn

/// </summary>

private const float timeToSpawn = 0.5f;

/// <summary>

/// The current time to spawn

/// </summary>

private float currentTimeToSpawn;

/// <summary>

/// The Platformer

/// </summary>

private PlatformerScene Platformer;

/// <summary>

/// The spawn factory

/// </summary>

private PrizeFactory spawnFactory;

/// <summary>

/// Поведение на момент создание игрового объекта

/// </summary>

public override void Start(GameObject gameObject = null)

{

Platformer = PlatformerScene.instance;

currentTimeToSpawn = Time.CurrentTime;

}

/// <summary>

/// Обновление игрового объекта

/// </summary>

public override void Update(GameObject gameObject)

{

if (currentTimeToSpawn < Time.CurrentTime)

{

Random random = new Random();

//int chance = random.Next(0, 101);

int chance = random.Next(0, 101);

if (Platformer.CountEmptyBlocks() == 0) return;

Vector2 position = Platformer.GetRandomPosition();

if (chance < 20)

{

spawnFactory = new SpeedPrizeFactory();

}

else if (chance > 20 && chance <= 50)

{

spawnFactory = new BolletPowerPrizeFactory();

}

else if(chance > 50 && chance <= 80)

{

spawnFactory = new ReloadTimePrizeFactory();

}

else

spawnFactory = new AmmoPrizeFactory();

Platformer.AddObjectOnScene(spawnFactory.CreatePrize(position));

currentTimeToSpawn += timeToSpawn;

}

}

}

}

**Код программы для *AmmoPrizeFactory.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Platformer;

using SharpDX;

namespace GameLibrary

{

/// <summary>

///

/// </summary>

/// <seealso cref="GameLibrary.PrizeFactory" />

public class AmmoPrizeFactory : PrizeFactory

{

/// <summary>

/// </summary>

/// <param name="position">Позиция появления</param>

/// <returns>

/// Игровой объект

/// </returns>

public override GameObject CreatePrize(Vector2 position)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(position, new Size2F(1, 1)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/PlatformerElements/Prize/ammo.png")));

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(0.8f, 0.8f)));

gameObject.GameObjectTag = "Spawn";

PrizeSpawn speedPrize = new PrizeSpawn();

speedPrize.InitializeGunSpawn(5f);

gameObject.InitializeObjectScript(speedPrize);

return gameObject;

}

}

}

**Код программы для *BulletPowerPrizeFactory.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Game;

using GameLibrary.Platformer;

using SharpDX;

namespace GameLibrary

{

/// <summary>

///

/// </summary>

/// <seealso cref="GameLibrary.PrizeFactory" />

public class BolletPowerPrizeFactory : PrizeFactory

{

/// <summary>

/// </summary>

/// <param name="position">Позиция появления</param>

/// <returns>

/// Игровой объект

/// </returns>

public override GameObject CreatePrize(Vector2 position)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(position, new Size2F(1, 1)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/PlatformerElements/Prize/bullet\_power.png")));

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(0.8f, 0.8f)));

gameObject.GameObjectTag = "Spawn";

PrizeSpawn speedPrize = new PrizeSpawn();

speedPrize.InitializeGunSpawn(new PowerDecorator(new PlayerProperitiesStandart()), 5f);

gameObject.InitializeObjectScript(speedPrize);

return gameObject;

}

}

}

**Код программы для *PrizeFactory.cs*:**

using EngineLibrary.ObjectComponents;

using SharpDX;

namespace GameLibrary

{

/// <summary>

///

/// </summary>

public abstract class PrizeFactory

{

/// <summary>

/// Creates the prize.

/// </summary>

/// <param name="position">Позиция появления</param>

/// <returns>

/// Игровой объект

/// </returns>

public abstract GameObject CreatePrize(Vector2 position);

}

}

**Код программы для *ReloadTimePrizeFactory.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Game;

using GameLibrary.Platformer;

using SharpDX;

namespace GameLibrary

{

/// <summary>

///

/// </summary>

/// <seealso cref="GameLibrary.PrizeFactory" />

public class ReloadTimePrizeFactory : PrizeFactory

{

/// <summary>

/// Creates the prize.

/// </summary>

/// <param name="position">Позиция появления</param>

/// <returns>

/// Игровой объект

/// </returns>

public override GameObject CreatePrize(Vector2 position)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(position, new Size2F(1, 1)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/PlatformerElements/Prize/reduce\_reloadtime.png")));

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(0.8f, 0.8f)));

gameObject.GameObjectTag = "Spawn";

PrizeSpawn speedPrize = new PrizeSpawn();

speedPrize.InitializeGunSpawn(new ReloadTimeDecorator(new PlayerProperitiesStandart()), 5f);

gameObject.InitializeObjectScript(speedPrize);

return gameObject;

}

}

}

**Код программы для *SpeedPrizeFactory.cs*:**

using EngineLibrary.EngineComponents;

using EngineLibrary.ObjectComponents;

using GameLibrary.Game;

using GameLibrary.Platformer;

using SharpDX;

namespace GameLibrary

{

/// <summary>

///

/// </summary>

/// <seealso cref="GameLibrary.PrizeFactory" />

public class SpeedPrizeFactory : PrizeFactory

{

/// <summary>

/// Creates the prize.

/// </summary>

/// <param name="position">Позиция появления</param>

/// <returns>

/// Игровой объект

/// </returns>

public override GameObject CreatePrize(Vector2 position)

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(position, new Size2F(1, 1)));

gameObject.InitializeObjectComponent(new SpriteComponent(RenderingSystem.LoadBitmap("Resources/PlatformerElements/Prize/speed.png")));

gameObject.InitializeObjectComponent(new ColliderComponent(gameObject, new Size2F(0.8f, 0.8f)));

gameObject.GameObjectTag = "Spawn";

PrizeSpawn speedPrize = new PrizeSpawn();

speedPrize.InitializeGunSpawn(new SpeedDecorator(new PlayerProperitiesStandart()), 5f);

gameObject.InitializeObjectScript(speedPrize);

return gameObject;

}

}

}

**Код программы для *CollisionTest.cs*:**

using EngineLibrary.ObjectComponents;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using SharpDX;

using System;

namespace UnitTestGame

{

[TestClass]

public class UnitTest1

{

/// <summary>

/// Tests the collider.

/// </summary>

[TestMethod]

public void TestCollider()

{

GameObject firstObject = new GameObject();

firstObject.InitializeObjectComponent(new TransformComponent(new Vector2(1f, 1f), new Size2F(1, 1)));

firstObject.InitializeObjectComponent(new ColliderComponent(firstObject, new Size2F(1, 1)));

firstObject.GameObjectTag = "Test";

GameObject secondObject = new GameObject();

secondObject.InitializeObjectComponent(new TransformComponent(new Vector2(1f, 3f), new Size2F(1, 1)));

secondObject.InitializeObjectComponent(new ColliderComponent(secondObject, new Size2F(1, 1)));

secondObject.GameObjectTag = "Test";

Assert.IsFalse(firstObject.Collider.CheckIntersection("Test"));

secondObject.Transform.Position = new Vector2(1f, 1.5f);

Assert.IsTrue(firstObject.Collider.CheckIntersection("Test"));

secondObject.GameObjectTag = "";

Assert.IsFalse(firstObject.Collider.CheckIntersection("Test"));

}

}

}

**Код программы для *DecoratorTest.cs*:**

using GameLibrary;

using GameLibrary.Game;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System;

namespace UnitTestGame

{

[TestClass]

public class DecoratorTest

{

/// <summary>The player properities</summary>

PlayerProperitiesStandart playerProperities;

/// <summary>Initalizes the bullet.</summary>

[TestInitialize]

public void InitalizeBullet()

{

playerProperities = new PlayerProperitiesStandart();

playerProperities.SetProperty(TypeProperty.Health, 10);

playerProperities.SetProperty(TypeProperty.Ammo, 10);

}

/// <summary>

/// Tests the speed decorator.

/// </summary>

[TestMethod]

public void TestSpeedDecorator()

{

SpeedDecorator speedDecorator = new SpeedDecorator(playerProperities);

float expectedSpeed = playerProperities.Speed \* 1.5f;

Assert.AreEqual(expectedSpeed, speedDecorator.Speed);

}

/// <summary>

/// Tests the power decorator.

/// </summary>

[TestMethod]

public void TestPowerDecorator()

{

PowerDecorator powerDecorator = new PowerDecorator(playerProperities);

float expectedPower = playerProperities.Power \* 2;

Assert.AreEqual(expectedPower, powerDecorator.Power);

}

/// <summary>

/// Tests the reload time decorator.

/// </summary>

[TestMethod]

public void TestReloadTimeDecorator()

{

ReloadTimeDecorator reloadTimeDecorator = new ReloadTimeDecorator(playerProperities);

float expectedReloadTime = playerProperities.ReloadTime / 2;

Assert.AreEqual(expectedReloadTime, reloadTimeDecorator.ReloadTime);

}

}

}

**Код программы для *MovementTest.cs*:**

using EngineLibrary.ObjectComponents;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using SharpDX;

using System;

namespace UnitTestGame

{

[TestClass]

public class MovementTest

{

/// <summary>

/// Tests the game object movement.

/// </summary>

[TestMethod]

public void TestGameObjectMovement()

{

GameObject gameObject = new GameObject();

gameObject.InitializeObjectComponent(new TransformComponent(new Vector2(1f, 1f), new Size2F(1, 1)));

Vector2 offset = new Vector2(1f, 1f);

gameObject.Transform.SetMovement(offset);

Vector2 expected = new Vector2(2f, 0f);

Vector2 actual = gameObject.Transform.Position;

Assert.AreEqual(expected, actual);

}

}

}

# ПРИЛОЖЕНИЕ Б

(обязательное)

**Руководство пользователя**

1. Введение.

Разработанное программное приложение предназначено для запуска на ОС не ниже *Windows* 7. Игра обладает минимальным порогом вхождения для пользователей.

Игровое приложение обладает следующим функционалом:

* игровой уровень в виде экшен-платформер с элементами для взаимодействия и бонусами;
* передвижение игроков;
* генерация призов.

Для использования программного приложения пользователь должен быть ознакомлен со следующими документами:

* настоящим руководством пользователя;
* правилами использования ЭВМ.

1. Назначение и условия применения.

Разработанное программное приложение предназначено для игры двух игроков на одном экране. Разработанное игровое приложение предназначено для развития внимания и реакции, а также для развлечения. Также развивает концентрацию и внимание и значительно улучшает память, позволяя запоминать всё большие объёмы информации.

Для корректной работы приложения необходима следующая конфигурация технических средств аппаратного обеспечения:

* центральный процессор *Intel Core* 2 *Duo* c тактовой частотой 2.30 МГц или более;
* наличие клавиатуры, мыши и цветного монитора *SVGA* с разрешением не менее 1280x720;
* операционная система *Windows* 7 и выше;
* 1024 Мб оперативной памяти.

1. Подготовка к работе.

Приложение запускается путём открытия файла *GameApplication.exe*. Также на компьютере должны быть установлены драйвера для видеокарты. Если все инструкции соблюдены, и приложение не выдаёт никаких сообщений об ошибках, значит, программа работает исправно.

1. Описание операций.

При запуске приложения открывается интерфейс, который выводит изображение с информацией об управлении в игре и кнопкой запуска игры. Первый игрок передвигается с помощью клавиш *W*, *A*, *S*, *D*, стреляет при нажатии на клавишу *Space*, подбирает бонус при нажатии на клавишу *C*. Второй игрок передвигается с помощью клавиш *Up*, *Left*, *Down*, *Right*, стреляет при нажатии на клавишу *RightEnter*, подбирает бонус при нажатии на клавишу *LeftSlash*.

Первый игрок управляет синим персонажем, а второй игрок – красным. Цель каждого игрока заключается в убийстве друг друга. В процессе игры можно передвигаться по горизонтали и вертикали.

В процессе игры можно подбирать бонусы оружие четырёх типов:

* ящик с патронами;
* бонус на увеличение урона;
* бонус на увеличение скорости перезарядки;
* бонус на увеличение скорости передвижения.

1. Аварийные ситуации

Чтобы избежать ошибок при использовании программы, необходимо соблюдать порядок действий и условия пользования, описанные в пункте 3 данного руководства пользователя.

В случае непредвиденного «зависания» программы рекомендуется завершить процесс в диспетчере задач и запустить снова.

1. Рекомендации по освоению.

Заранее изучить работу с клавиатурой персонального компьютера. Запомнить расположение клавиш, отвечающих за игровое управление.

# ПРИЛОЖЕНИЕ В

(обязательное)

**Руководство** **программиста**

1. Назначения и условия применения программы.

Разработанное игровое приложение предназначено для игры между двумя игроками на одном компьютере. Разработанное игровое приложение предназначено для развития внимания и реакции, а также для развлечения. Также развивает концентрацию и внимание и значительно улучшает память, позволяя запоминать всё большие объёмы информации.

Для корректной работы приложения необходима следующая конфигурация технических средств аппаратного обеспечения:

* центральный процессор *Intel Core* 2 *Duo* c тактовой частотой 2.30 МГц или более;
* наличие клавиатуры, мыши и цветного монитора *SVGA* с разрешением не менее 1280x720;
* операционная система *Windows* 7 и выше;
* 1024 Мб оперативной памяти.

1. Характеристики программы.

Для запуска приложения не требуется никаких дополнительных настроек. Для запуска решения необходима среда разработки *Visual Studio* с установленными фреймворком .*NET* и библиотекой *SharpDX.*

1. Обращение к программе.

Приложение запускается путём открытия файла *GameApplication.exe*, находящегося в папке *game*.

1. Входные и выходные данные.

В данной программе в качестве входных данных используется ввод с клавиатуры кнопок управления игровым процессом. В качестве выходных выступает окно отображения игры.

1. Сообщения.

При окончании игры выводится победивший игрок. В процессе работы приложения игровая статистика выводится в информационные лейблы каждого игрока.

# ПРИЛОЖЕНИЕ Г

(обязательное)

**Руководство системного программиста**

1. Общие сведения о программе.

Разработанное игровое приложение предназначено для игры между двумя игроками на одном компьютере. Разработанное игровое приложение предназначено для развития внимания и реакции, а также для развлечения. Также развивает концентрацию и внимание и значительно улучшает память, позволяя запоминать всё большие объёмы информации.

Для корректной работы приложения необходима следующая конфигурация технических средств аппаратного обеспечения:

* центральный процессор *Intel Core* 2 *Duo* c тактовой частотой 2.30 МГц или более;
* наличие клавиатуры, мыши и цветного монитора *SVGA* с разрешением не менее 1280x720;
* операционная система *Windows* 7 и выше;
* 1024 Мб оперативной памяти.

1. Структура программы.

Игровое приложение логически можно разбить на несколько составляющих: игровой движок, содержащий средства работы с графикой, непосредственно логика игровых объектов и игрового процесса и графический интерфейс пользователя.

1. Настройка программы.

Для запуска приложения не требуется никаких дополнительных настроек. Для запуска решения необходима среда разработки *Visual Studio* с установленными фреймворком .*NET* и библиотекой *SharpDX.*

1. Проверка программы.

Для верификации программного средства были реализованы модульные тесты. Для запуска отладки в среде разработки *Visual Studio* необходимо запустить выполнение модульных тестов. По окончанию процесса отладки будет выдан отчет о результатах тестирования приложения.

1. Дополнительные возможности.

Приложение является узконаправленным и не имеет дополнительных возможностей.

1. Сообщение системному программисту.

Проект игрового движка можно использовать для разработки других игр. Проект игровой логики можно дополнить, добавив новые Экшен-платформер, бонусы и элементы экшен-платформер.

**ПРИЛОЖЕНИЕ Д**

(обязательное)

**Функциональная схема приложения**