МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ  
«ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ  
УНИВЕРСИТЕТ ИМЕНИ П.О. СУХОГО»

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту

по дисциплине «Объектно-ориентированное проектирование

и программирование»

на тему: «Приложение, реализующее игру «Битва на воздушных шарах» с

использованием *Windows* *Form* и графики *OpenGL*»

Исполнитель: студент группы ИТИ-21

Громыко И. В.

Руководитель: доцент

Курочка К. С.

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

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

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

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

Подписи членов комиссии

По защите курсового проекта: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Гомель 2023

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc130375192)

[1 Аналитический обзор средств для создания игрового приложения 4](#_Toc130375193)

[1.1 *OpenGL* как средство для создания графики 4](#_Toc130375194)

[1.2 *DirectX* как средство для создания графики 7](#_Toc130375195)

[1.3 Сравнение *OpenGL* и *DirectX* 10](#_Toc130375196)

[1.4 Требования к проектируемому программному обеспечению 11](#_Toc130375197)

[Список используемых источников 13](#_Toc130375198)

# **ВВЕДЕНИЕ**

Сегодня существует масса примеров различных игровых приложений, которые написаны на различных языках программирования и созданы для различных платформ и целевых аудиторий. Сейчас игры используются не только для развлечения, но и в различных других сферах: обучении, торговле и здравоохранении. Количество времени, которое проводят люди за игрой, с каждым годом растёт, в связи с чем растёт и потребность в создании новых приложений, так как старые игры интересуют их уже намного меньше.

Несмотря на большое количество различных игровых проектов разработчики не выпускают много игровых приложений, в которые могут соревноваться два игрока на одном компьютере, так как сейчас индустрия нацелена, в первую очередь, на развитие многопользовательских игр с соединением по сети.

Описанные факторы обуславливают потребность в создании игрового приложения, в котором двое пользователей могут проводить время вместе за игрой на одном компьютере. Преимуществом такого приложения будет то, что для запуска и полноценной игры от игроков будет требоваться только одно устройство без подключения к Интернету.

Целью данной работы будет создание игрового приложения «Битва на воздушных шарах» для двух игроков. Игра будет уникальной в своём роде, так как на рынке практически не существует аналогов, кроме небольших браузерных проектов, где игра рассчитана на одного игрока. Также игра будет снабжена механиками, которые заинтересуют пользователя. Кроме того, одна из главных целей проекта – простота в использовании, это делается для того, чтобы игроки могли зайти в игру и сразу, без больших затрат времени, наслаждаться процессом.

Для достижения данной цели будет необходимо решить ряд задач:

– исследовать существующие решения на рынке игровой индустрии и определить основные преимущества и недостатки представленных проектов;

– провести аналитический обзор средств для создания игрового приложения;

– по результатам аналитического обзора определить средства для достижения поставленной цели;

– спроектировать игровое приложение с использованием паттернов проектирования;

– реализовать и протестировать итоговое приложение.

**1 АНАЛИТИЧЕСКИЙ ОБЗОР СРЕДСТВ ДЛЯ СОЗДАНИЯ ИГРОВОГО ПРИЛОЖЕНИЯ**

* 1. ***OpenGL* как средство для создания графики**

В наше время для создания графики для игровых приложений существует масса подходов и вариантов решений, большинство из которых сводится к определению графического *API* (интерфейса прикладного программирования) или графических библиотек, которые предоставляют программистам доступ к графическому оборудованию. Самыми популярными *API* на данный момент являются *DirectX* и *OpenGL*.

*OpenGL* (*Open Graphics Library*)– современный кроссплатформенный *API*, который позволяет работать как с двумерной, так и с трёхмерной графикой. Данный интерфейс существует уже более 20 лет и поддерживается на множестве платформ: от мобильных устройств до мощных стационарных станций. На *OpenGL* за свою историю существования было написано множество игровых приложений различных величин: от небольших проектов до одних из самых крупных в индустрии. [1] Библиотеки для работы с *OpenGL* существуют практически для всех языков программирования, включая и *C*#.

*OpenGL* представляет собой программный интерфейс к графическому процессору (*GPU*). Ключевой идеей *OpenGL* является то, что он основан на модели клиент-сервер. В роли клиента выступает приложение, а в роли сервера – драйвер и графический процессор. Все запросы от клиента поступают в очередь, откуда они со временем извлекаются сервером и выполняются. Аналогично ответ от сервера также помещается в очередь, откуда он со временем будет извлечён клиентом.

Графический процессор работает асинхронно, то есть мы передаём запрос на сервер и сразу же получаем управление обратно. Однако в *OpenGL* предусмотрены функции для явной синхронизации, но их вызов фактически останавливает конвейер, и это значительно сказывается на быстродействии.

Для работы с *OpenGL* необходимо создать специальный контекст, который связан с окном, куда будет производиться рендеринг (отрисовка). Серверное состояние хранится в этом контексте. Также данный контекст привязан к текущему потоку, поэтому если приложение выполняется сразу на нескольких потоках, то созданный контекст будет валиден только для того потока, где он был создан.

Изначально интерфейс задумывался как не зависящий от аппаратного обеспечения *API*, то есть он может быть реализован на различных платформах. Именно поэтому в *OpenGL* не включены команды для работы с окнами и получения данных после ввода пользователя. Также у *OpenGL* нет команд для описания различных трёхмерных моделей, сам интерфейс может лишь выводить примитивы точек, отрезков и треугольников. Остальные модели строятся поверх *OpenGL* на основании данных примитивов.

Рендеринг геометрии в *OpenGL* осуществляется в фреймбуфер. Фреймбуфер – это буфер кадра, который содержит информацию о каждом пикселе окна и который состоит из набора различных буферов: буфера цвета, глубины и трафарета.

Буфер цвета содержит цвет каждого пикселя. Цвет задаётся в формате *RGB* или *RGBA*, где четвёртая компонента – это показатель непрозрачности.

Буфер глубины храни для каждого пикселя соответствующее ему значение глубины, то есть показатель удалённости элемента изображения.

Буфер трафарета позволяет для каждого пикселя задать несколько бит для дальнейших взаимодействий. Буфер трафарета используются при создании различных спецэффектов, например, для создания тени или отражения.

Для построения изображения на экран *OpenGL* использует конвейер рендеринга, который состоит из последовательности стадий, через которые проходят данные при их обработке.

Изначально *OpenGL* составляет списки отображения для текущего или последующего использования. Далее все геометрические примитивы описываются вершинами, которые поступают на вход конвейера, в вершинный шейдер. Вершинный шейдер – программа, которая выполняется на *GPU* и которая обрабатывает каждую вершину независимо от всех остальных вершин.

Следующей стадией обработки является сбор примитивов из выданных вершинным шейдером вершин и информации о связности, то есть информации о том, как из вершин собирать отдельные примитивы.

Далее происходит отсечение геометрии – удаление частей объектов, которые выходят за границы полупространства, определенного плоскостью. Результатом этого этапа являются законченные графические примитивы, вершины которых преобразованы и отсечены.

После происходит процесс растеризации примитива, который превращает его в набор отдельных фрагментов. Фрагмент – то, что в конце конвейера может стать пикселем, если одна из последующих стадий не решит его отбросить. С каждым фрагментом связаны значения цвета и глубины.

После этого каждый фрагмент обрабатывается фрагментным шейдером, и для него выполняются различные тесты и растровые операции, которые могут изменить и отбросить фрагменты.

В конце сформированные фрагменты рисуются в соответствующем буфере, где из них формируются пиксели и где определяется их окончательное местоположение.

На рисунке 1.1 показана схема конвейера рендеринга в *OpenGL* 3.3.

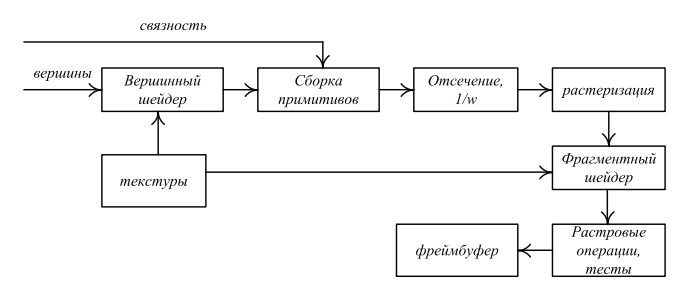


Рисунок 1.1 – Схема конвейера в *OpenGL* 3.3

В версии *OpenGL* 4.0 одним из самых масштабных добавлений стало добавление в конвейер этапа тесселяции. В процессе тесселяции каждый из многоугольников разбивается на заданное число более мелких многоугольников, которые выстраиваются в соответствии с общим направлением поверхности модели. Так можно создать сначала простую модель, а затем быстро и просто повысить детализацию модели. [2]

В течение времени спецификация *OpenGL* изменялась, в связи с чем выпускались новые версии спецификации, которые также включали поддержку *Shading Language* – шейдерного языка, который был разработан для выполнения математики в процессе растеризации. На *OpenGL Shading Language* (*GLSL*) частично написан программный конвейер *OpenGL*. [3]

Весь *API Open GL* основан на языке программирования *C* и представляет собой набор функций и констант. Поскольку в языке *С* нет пространств имён, то для того, чтобы избежать возможных конфликтов по именам, все функции и константы *OpenGL* начинаются со специальных префиксов: функции – с префикса *gl*, а константы – с *GL\_*. [4]

Также в библиотеку заложен механизм расширений, благодаря которому производители аппаратного обеспечения, в первую очередь, видеокарт, могут выпускать расширения *OpenGL* для поддержки новых возможностей, которые не были включены в текущую версию библиотеки.

Платформа *Microsoft .NET Framework* не имеет встроенных средств поддержки библиотеки *OpenGL*, однако существует несколько способов применения *OpenGL* на данной платформе. Одной из таких реализаций является подключение динамической библиотеки *Opengl.dll*, которая является реализацией *OpenGL* в операционной системе *Windows*.

Также на данный момент существует несколько библиотек, в которых выполнена необходимая реализация вызовов функций из динамической библиотеки и которые предоставляют дополнительные возможности, позволяющие упростить использование *OpenGL* на платформе *Microsoft .NET Framework*.

Среди таких библиотек можно выделить:

– *Tao Framework*;

– *Open Toolkit Library* (*OpenTK*);

– *SharpGL*;

– *OpenGL.NET*.

Библиотека *Open Toolkit Library* (*OpenTK*) имеет наиболее удобный интерфейс вызова функций библиотеки *OpenGL*, который позволяет избежать большого числа ошибок при её использовании. *Open Toolkit Library* имеет поддержку как для *.NET Framework*, так и для платформы *.NET*.

* 1. ***DirectX* как средство для создания графики**

До создания *DirectX* производители игровых приложений боролись с проблемами, связанными с несовместимостью оборудования, что практически не позволяло всем устанавливать и пользоваться игровыми приложениями из-за огромного количества существующих аппаратных конфигураций.

В версиях *MS-DOS* разработчикам предоставлялся прямой доступ к различным частям системы, в том числе к видеокарте. Однако в связи с необходимостью стандартизации *Microsoft* в своей системе *Windows* 95 предоставляла более унифицированный, но в то же время ограниченный доступ к устройствам. Так как разработчикам был необходим расширенный инструментарий для работы с графикой, корпорация выпустила *Windows Game SDK* для *Windows* 95, что стало первой версией *DirectX*. [5]

*DirectX* так же, как и *OpenGL*, является набором низкоуровневых интерфейсов программирования приложений (*API*)для создания игровых и других высокопроизводительных мультимедийных приложений.

*DirectX* предоставляет разработчикам интерфейс для аппаратного обеспечения компьютера, работающего под управлением операционных систем на базе операционной системы *Windows*. Каждый компонент данного *API* обеспечивает доступ к различным аспектам оборудования, включая графику, звук и устройства ввода, через стандартный интерфейс. Таким образом, *DirectX* позволяет не использовать несколько интерфейсов для различного оборудования, а реализовывать всё через один набор взаимосвязанных интерфейсов. Также *DirectX* поддерживается на игровых консолях *Xbox*.

*API DirectX* разделён на множество компонентов, каждый из которых представляет отдельный аспект системы. Каждый *API* можно использовать независимо друг от друга, что позволяет разработчику добавлять только те функции, которые ему нужно в игровом приложении. Также *DirectX* поддерживает функцию добавления и обновления компонентов по мере необходимости.

Основными компонентами *DirectX* являются:

– *Direct2D*;

– *Direct3D*;

– *DirectCompute*;

– *XACT3*;

– *XAudio2*;

– *DirectInput*.

*Direct3D* и *Direct2D* используются непосредственно для вывода трёхмерной и двумерной графики соответственно. *DirectCompute* позволяет выполнять многопоточные вычисления с использованием графического процессора. *XACT3* и *XAudio2* являются *API* обработки звука. *XAudio2* является низкоуровневым *API*, а *XACT3* – высокоуровневым *API*, созданным поверх *XAudio2*. *DirectInput* используется для взаимодействия с клавиатурой, мышью или джойстиком.

*Direct3D* так же, как и *OpenGL*, имеет конвейер рендеринга изображений, который также состоит из нескольких стадий, во время которых происходит преобразование изображений. [6]

Первый этап конвейера *Direct3D* 11 называется стадией сборщика входных данных, целью которого является считывание примитивов из буферов, заполненных пользователем. *Direct3D* позволяет работать не только с точками, линиями и треугольниками, но и с другими типами примитивов: списки линий, полосы треугольников и др.

Вторым этапом конвейера является этап вершинного шейдера, в процессе которого обрабатываются вершины после первого этапа. Вершинный шейдер выполняет операции с каждой вершиной, такие как преобразование, освещение и другие. Шейдер принимает на вход одну вершину и отправляет на выход также одну вершину.

Третий, четвёртый и пятый этапы являются необязательными этапами, которые связаны с тесселяцией, в процессе которой преобразуются поверхности с низкой детализацией в примитивы с более высоким уровнем детализации.

Шестой этап конвейера, этап геометрического шейдера, также является необязательной стадией. Так как, если выполняется процесс тесселяции, в геометрическом шейдере нет смысла. Если этапов тесселяции нет, то геометрический шейдер может создавать и разрушать геометрию по мере необходимости, например, на этом этапе могут накладываться тени или создаваться частицы для создания эффектов, таких как дождь или взрывы.

После этапа геометрического шейдера начинается этап потокового вывода, целью которого является непрерывный вывод данных вершин из этапа геометрического шейдера в один или несколько буферов в памяти.

Далее происходит этап растеризатора, в процессе которого векторные данные, состоящие из фигур или примитивов, преобразуются в изображение, состоящее из пикселей, для отображения трёхмерной графики в режиме реального времени.

Девятым этапом является этап шейдера пикселей, который получает геометрические данные со всех предыдущих этапов и позволяет использовать широкие методы заливки, такие как освещение для каждого пикселя и постобработка. Шейдер пикселей объединяет данные текстур, константные переменные и другие данные для получения отдельных пикселей.

Последним этапом конвейера является этап слияния выходных данных, который берёт все части выходных данных с других этапов конвейера и создаёт окончательный отрисованный цвет пикселей. Это последний шаг для определения видимых пикселей и смешивания окончательных цветов пикселей.

В *DirectX* двумерная графика является частью 3*D*-конвейера, в связи с чем для разработки двумерных игровых приложений используется *Direct3D* совместно с *Direct2D*, который применяется в первую очередь для отрисовки растровых изображений и текста, однако также может использоваться и для основной графики двумерного игрового приложения. [7]

Различные релизы *Windows* содержат и поддерживают различные версии *DirectX*, что позволяет более новым версиям оперативной системы продолжать работать с приложениями, разработанными для более ранних версий, пока эти версии не будет постепенно заменены новыми *API*, драйверами и оборудованием.

В 2002 году *Microsoft* выпустила версию *DirectX*, совместимую с *Microsoft* .*NET* *Framework*, что позволило программистам использовать функциональность *DirectX* в приложениях с использованием .*NET* *Framework* (через *Common* *Language* *Runtime*), в том числе с использованием языка *C*#. Данный *API* был назван *Managed* *DirectX*, поддержка которого завершилась в 2006, после того как *Microsoft* представили новую управляемую версию *DirectX* – *XNA* *Framework*, которая также позволяла использовать *DirectX* в приложениях, написанных на *C*#. Сейчас ни один продукт *Microsoft*, включая последние выпуски *XNA* не поддерживает версии *DirectX* 10 и новее для .*NET* *Framework*.

Другой подход для создания приложений на языке программирования *C*# с использованием *DirectX* заключается в использовании сторонних библиотек. Сейчас самыми популярными библиотеками для работы в .*NET* *Framework* являются *SlimDX* и *SharpDX*.

И *SharpDX*, и *SlimDX* являются *API* с открытым исходным кодом для создания приложений с использованием *DirectX* на .*NET* *Framework*, оба *API* предоставляют функциональность для создания как двумерной, так и трёхмерной графики.

**1.3 Сравнение *OpenGL* и *DirectX***

*DirectX* является разработкой *Microsoft*, созданной специально под приложения на операционных системах *Windows* и *Microsoft* *Xbox*. *OpenGL* является открытым стандартом некоммерческой организации *Khronos* *Group*. Стандарт доступен на большом количестве платформ, в том числе на *Linux*, *Mac* *OS* и мобильных устройствах.

Компонентом для вывода трёхмерной графики в *DirectX* является *Direct3D*, который основан на технологии *Component* *Object* *Model*, которая является стандартом для создания программного обеспечения на основе взаимодействующих компонентов, которые могут использоваться во многих программах одновременно. *COM* позволяет работать с объектно-ориентированной концепцией из любого языка, который поддерживает данный стандарт. В приложении на *Direct3D* работа с объектом осуществляется путём вызова методов его интерфейса.

*OpenGL* при этом использует обычные функции языка *C*, в связи с чем он может быть использован практически в любом языке программирования. В *OpenGL* результат вызовов функций зависит от внутреннего состояния, которое стандартможет менять. Поэтому для получения доступа к конкретному объекту нужно сначала выбрать его в качестве текущего определённой функцией, а затем уже изменять его характеристики.

Также стоит отметить, что *DirectX* включает в себя работу с вводом данных с контроллеров, работу со звуком и функции для создания окна, тогда как *OpenGL* предоставляет разработчикам только работу с графикой. При этом работа с графикой в двух *API* реализуется простейшими примитивами: точками, линиями и треугольниками. *DirectX* также может объединять эти примитивы для создания более сложных, например, списков линий.

Серьёзным различием является то, что *DirectX* фиксирован в пределах версий, которые выпускаются *Microsoft*, тогда как *OpenGL* поддерживает функцию расширений к основной спецификации. Если производителям графических процессоров или разработчикам нужно добавить какой-либо функционал, они могут реализовать его в своём драйвере и задокументировать доступное расширение в *OpenGL*. Приложения могут использовать новые функции, не дожидаясь включения в официальную спецификацию. Самые важные и глобальные расширения со временем становятся частью основной спецификации.

*DirectX* предоставляет разработчику управление аппаратным обеспечением, что даёт ему возможность эффективно распределять ресурсы по мере необходимости. *OpenGL* сам управляет аппаратным обеспечением, что упрощает работу программисту, однако в ряде случаев, когда разработчику необходимо самому указать, каким образом будут распределяться ресурсы, это является недостатком.

*DirectX* и *OpenGL* предлагают регулярно обновляемую документацию, которая обновляются регулярно с добавлением новых функций в *API*.

И *DirectX*, и *OpenGL* используют похожие конвейеры рендеринга, в связи с чем сейчас за исключением нескольких незначительных функциональных различий, два *API* обеспечивают почти одинаковый уровень функциональности.

Для работы в .*NET* *Framework* *OpenGL* является более удобным стандартом, так как его реализации доступны в ряде постоянно обновляющихся библиотек, таких как *SharpGL*, *OpenTK*, *Tao* *Framework* и другие. *DirectX* представлен двумя основными библиотеками: *SharpDX* и *SlimDX*. *SharpDX* не поддерживается разработчиками с 2019 года, *SlimDX* последний раз получала обновления в начале 2012-го.

**1.4 Требования к проектируемому программному обеспечению**

Целью разработки является создание уникального игрового приложения «Битва на воздушных шарах» на языке программирования *C*#. Игра будет рассчитана на двух игроков, каждый из которых управляет своим воздушным шаром.

Воздушные шары самостоятельно могут перемещаться только по вертикальной оси, а перемещение в горизонтальной плоскости будет происходить под влиянием ветра, который задаётся случайным образом для каждого игрока.

Под шарами находится земля, при столкновении с которой шар уничтожается, и игра заканчивается. Воздушные шары обладают бронёй и жизнью. Если запасы жизни заканчиваются, то шар также разрушается и игра заканчивается.

Для перемещения по вертикальной плоскости воздушному шару требуется топливо. Шар может стрелять различными видами снарядов и собирать призы, которые позволяют улучшить характеристики снаряда: скорость полёта, убойную силу и радиус поражения. Также призы могут пополнять запасы жизни, брони и топлива.

Для реализации генерации призов будет использован шаблон «фабричный метод». Для изменения характеристик снарядов – «декоратор».

В результате работы будет разработано игровое приложение, содержащее весь перечисленный выше функционал. Для запуска приложения потребуется компьютер на операционной системе семейства *Microsoft Windows.*

Для создания графического интерфейса пользователя на языке *C*# существует два основных решения: *Windows Presentation Foundation* (*WPF*) и *Windows Forms*. Так как технология *Windows Forms* создавалась намного раньше, чем *WPF*, то она является более испытанной и протестированной. Также *Windows Forms* является более простой и лёгкой в разработке, в связи с чем для разработки графического интерфейса пользователя игрового приложения будем использовать именно *WinForms*.

Так как *OpenGL* является более простым для работы с графикой в связи с тем, что стандарт сам управляет аппаратным обеспечением, а также представлен большим выбором библиотек для языка программирования *C#* и поддерживает разработку на *Windows* *Forms* без сильных потерь в производительности в сравнении с *DirectX*, то для приложения, реализующего игру «Битву на воздушных шарах», будем использовать спрайтовую графику и средства *OpenGL* с использованием библиотеки *OpenTK*.

В процессе аналитического обзора средств для создания игрового приложения выделили преимущества и недостатки различных подходов, по функционалу и требованиям задачи определили средства, которые будут использоваться при разработке. По результатам аналитического обзора определили основные направления и методы для создания итогового приложения.

**2 АРХИТЕКТУРА И СТРУКТУРА ПРОЕКТА ИГРОВОГО ПРИЛОЖЕНИЯ**

**2.1 Проектирование архитектуры проекта**

Для того, чтобы иметь возможность модифицировать приложения без серьёзных затрат и усилий, первоначально следует разработать гибкую архитектуру проекта. Это позволит легко добавлять новый функционал или, при необходимости, перенести проект на другие средства, например, создания графики. Однако первоначально следует спроектировать структуру компонентов архитектуры.

Игровое приложение, как было рассмотрено ранее, разрабатывалось на языке программирования *C*# с использованием *Windows Forms* и графики *OpenGL*. Так как в будущем может появиться потребность в переносе игры, например, на *DirectX,* было принято решение расположить классы, напрямую связанные с графикой (например, загрузка текстур, их наложение на объекты), в отдельную библиотеку классов *GraphicsOpenGL*. Там же располагается и класс, который преобразовывает точки из системы координат *OpenGL* в *Windows Forms*, так как в первом случае точка (0, 0) соответствует центру экрана, а во втором случае – левому верхнему углу. На рисунке 2.1 показана структура классов библиотеки *GraphicsOpenGL*.

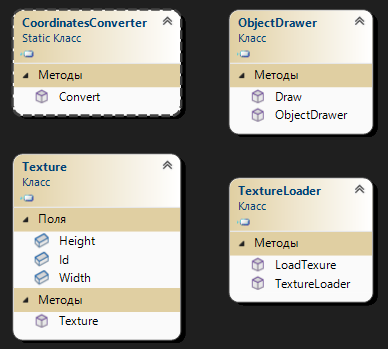


Рисунок 2.1 – Структура классов *GraphicsOpenGL*

Для того, чтобы разграничить компоненты, созданные непосредственно для создания логики игры, они были распределены на три библиотеки классов:

- *GameLibrary*;

- *AmmoLibrary*;

- *PrizesLibrary*.

В *GameLibrary* (рисунок 2.2) содержатся классы движка игры, где прописана вся логика игры, игрока и взрыва, который происходит во время столкновения границ объектов или при достижении максимальной величины дальности полёта снарядов.

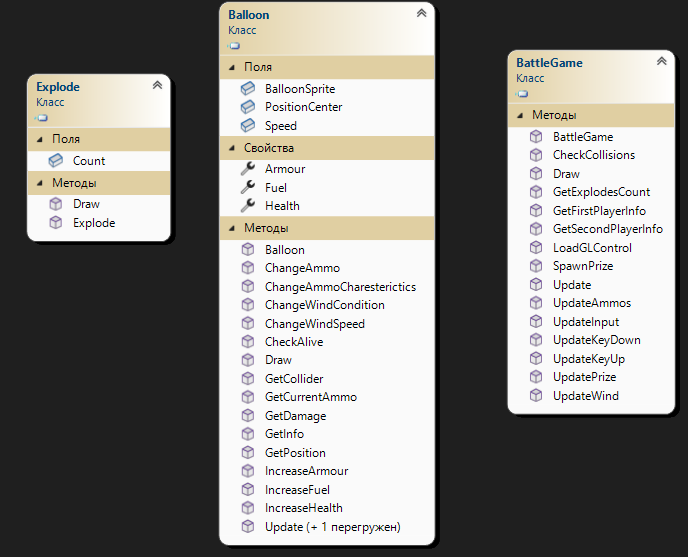


Рисунок 2.2 – Структура классов *GameLibrary*

**2.2 Реализация шаблона проектирования «Декоратор»**

Согласно требованиям проекта, в игре «Битва на воздушных шарах» присутствуют несколько видов снарядов, а также используется шаблон проектирования «Декоратор», который изменяет характеристики снарядов.

Суть декоратора заключается в том, что он позволяет динамически добавлять объектам новую функциональность, при этом, он независим от добавления новых производных от базового классов.

Для изменения характеристик снарядов первоначально следует создать абстрактный класс снаряда, *Ammo*, от которого наследуются три вида снарядов (*Explosive*, *Supersonic* и *Piercing*), которые отличаются спрайтом и характеристиками: радиусом действия, дальностью полёта и скоростью.

Далее, создаётся абстрактный класс *AmmoDecorator,* который принимает в качестве поля объект *Ammo*. От абстрактного класса наследуются три вида декораторов: *RadiusDecorator*, *SpeedDecorator* и *DistanceDecorator*, которые изменяют, соответственно, радиус действия, скорость и дальность полёта.

В связи с чем в *AmmoLibrary* содержится три вида снарядов (*Explosive*, *Supersonic* и *Piercing*), которые наследуются от абстрактного класса *Ammo,* а также три декоратора для изменения скорости, радиуса действия и дальности полёта снарядов (рисунок 2.3).

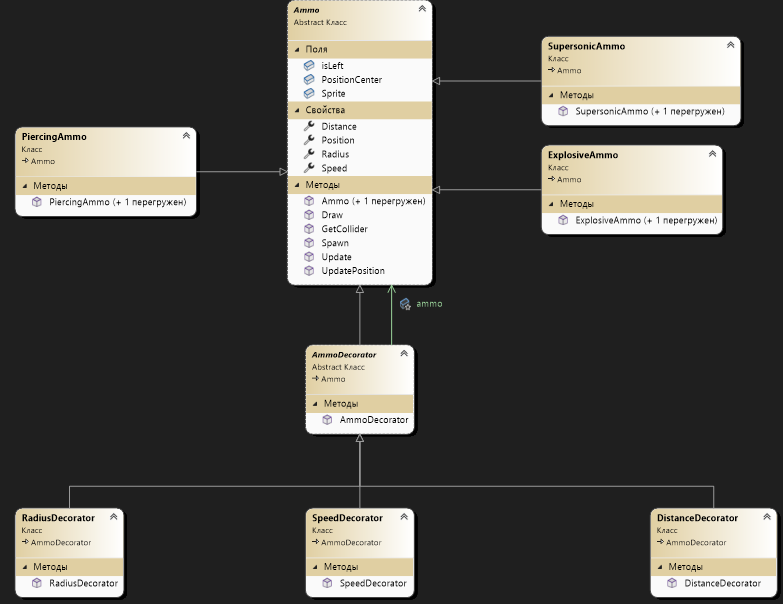


Рисунок 2.3 – Реализация шаблона «Декоратор»

**2.3 Реализация шаблона проектирования «Фабричный метод»**

Для генерации призов в игровом приложении используется шаблон «Фабричный метод», суть которого заключается в том, что реализация создания объекта лежит не в самом классе объекта, а в классе, который содержит метод создания данного объекта.

В игровом приложении «Битва на воздушных шарах» для работы с призами создаётся абстрактный класс *Prize*,а также четыре производных от него класса, которые представляют собой конкретные призы: *AmmoPrize*, *ArmourPrize*, *FuelPrize* и *HealthPrize*, при подборе которых у игрока улучшаются соответственно характеристики снарядов, брони, топлива и здоровья.

Также библиотека классов *PrizeLibrary* содержит класс *PrizeGenerator*, который содержит реализацию «Фабричного метода». В методе *Create* случайным образом выбирается точка генерации приза и направление его полёта, а также сам тип приза, после чего созданный приз отдаётся возвращаемым значением (рисунок 2.4).

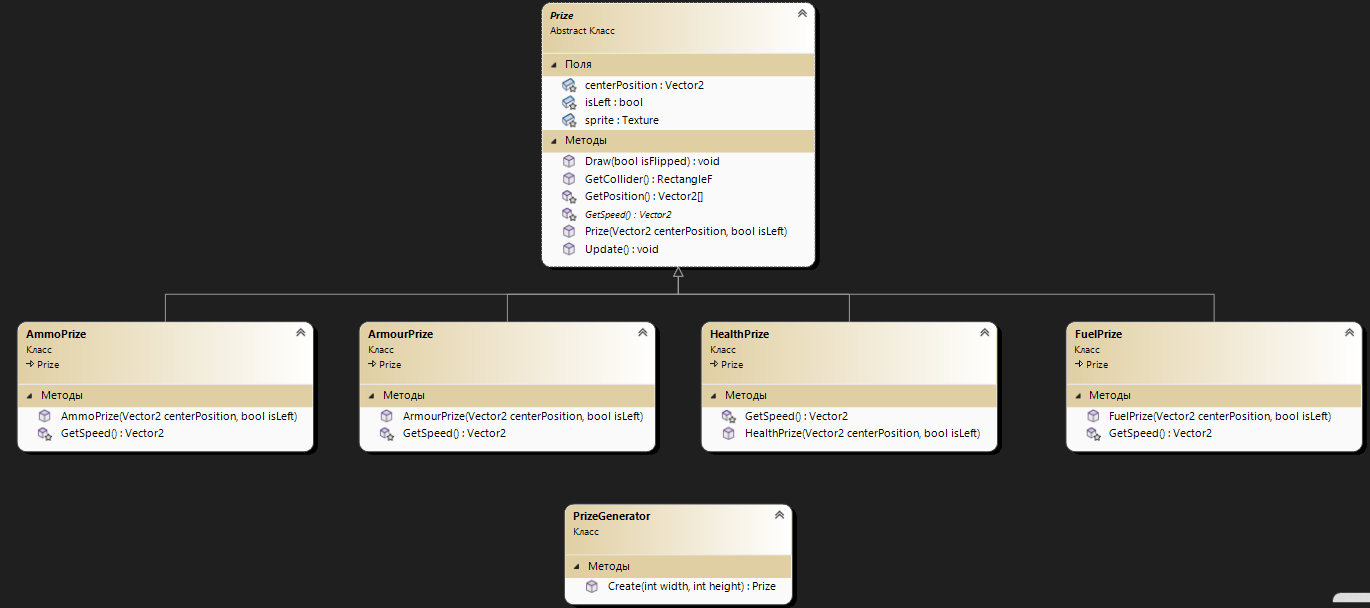


Рисунок 2.4 – Реализация шаблона «Фабричный метод»

**2.4 Реализация графических объектов**

В проекте присутствует ряд объектов, которые графически отображаются на экране пользователя. Это два воздушных шара, снаряды, призы, земля и задний фон, а также анимация взрывов. Для всех графических объектов использовалась спрайтовая графика.

Для того, чтобы сохранялась визуальная стилистика игры, а также для отличия игроков было решено создать два спрайта идентичных спрайта воздушных шаров с различным цветом (рисунок 2.5).

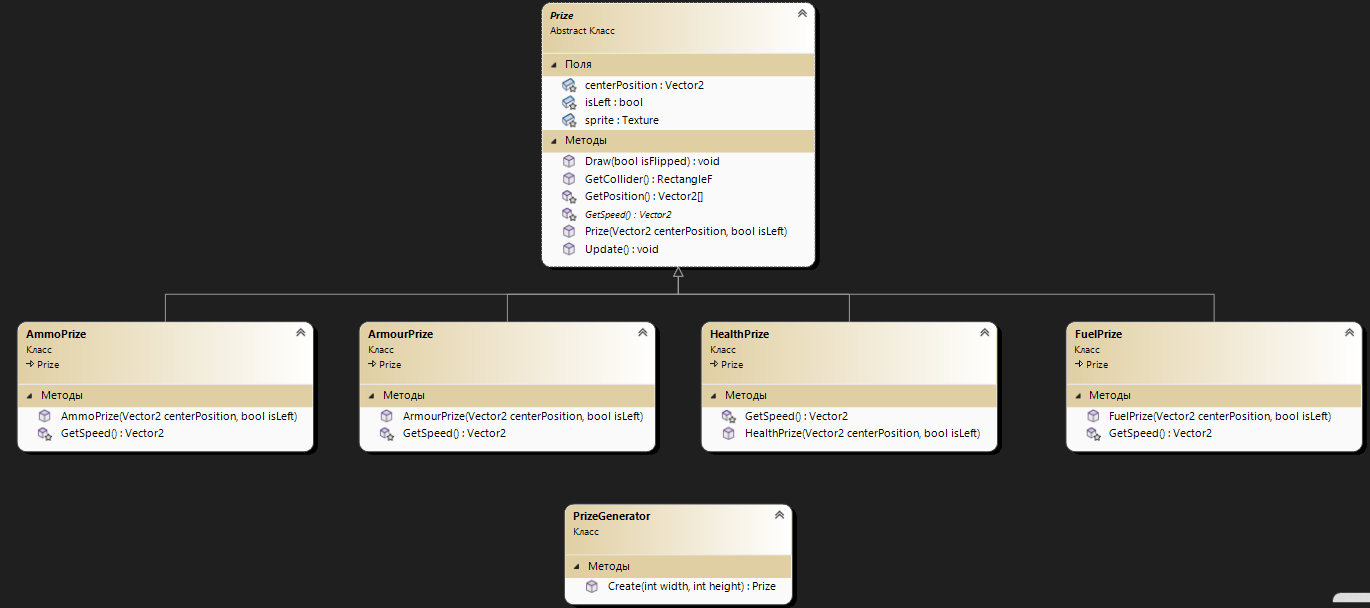


Рисунок 2.5 – Спрайты воздушных шаров

Каждый игрок имеет по три вида снарядов (рисунок 2.6). Так как каждый игрок может стрелять как влево, так и вправо, было принято решение создать только три вида снарядов и отражать их по вертикали при отрисовке объекта с помощью функций *OpenGL*.

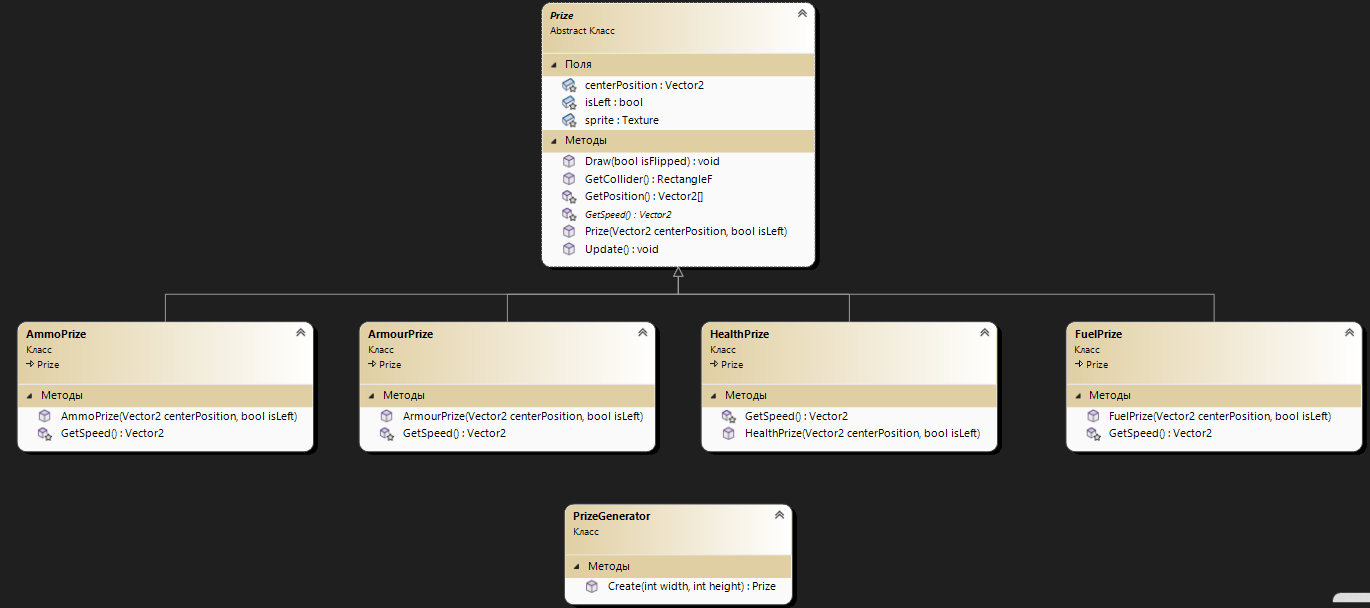


Рисунок 2.6 – Спрайты выпускаемых снарядов

На рисунке 2.7 продемонстрированы спрайты объектов призов.

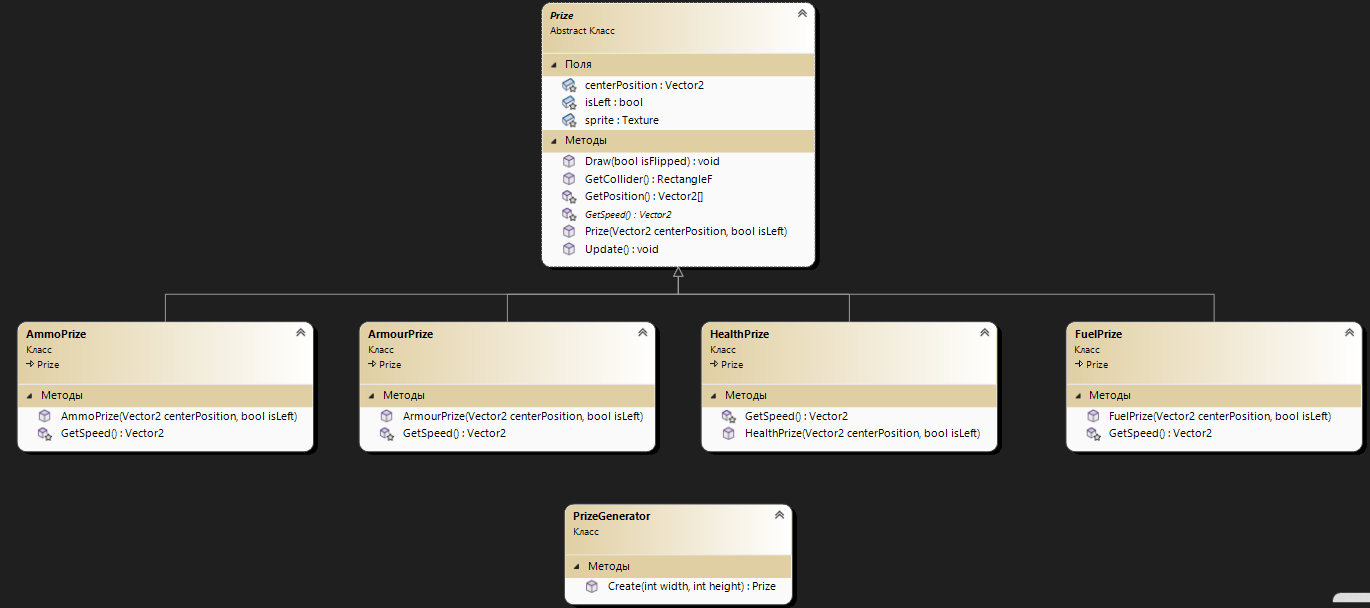


Рисунок 2.7 – Спрайты призов

**2.5 Основной алгоритм решения задачи**

При запуске приложения вызывается форма *GameForm,* которая реализует таймеры для обновления сцены, ветра и призов. Также форма используется для обновления текста с текущей информацией об игроках.

При каждом вызове таймера обновления игры (каждые 10 миллисекунд) вызывается метод *Update* созданного объекта *BattleGame,* в котором движок игры первоначально обновляет положение игроков, вызывая метод *Update* объектов класса *Ballon,* который формирует падение игрока вниз, если игрок не нажимает на кнопки движения, или движение в вертикальной плоскости в зависимости от нажатия. Также данный метод формирует движение ветра, если он работает в данный момент.

Далее проверяется нажатие игроками кнопок выпуска снарядов, которые могут выпускаться не чаще 50 тиков (500 миллисекунд) для того, чтобы сцена игры не была полностью заполнена снарядами.

После проверки необходимости выпуска снарядов игровой движок проверяет соприкосновение отрисованных объектов сцены на соприкосновение с другими объектами закрытым методом CheckCollisions. Данный метод возвращает целочисленное значение, которое является кодом выполнения метода. В таблице 2.1 приведены коды завершения метода и их условия.

Таблица 2.1 – Коды завершения метода *Update*

|  |  |
| --- | --- |
| **Значение кода** | **Условие** |
| 0 | Значение здоровья игроков выше нуля, ни один из них не столкнулся с землёй. |
| 1 | Первый игрок столкнулся с землёй или значение здоровья первого игрока равно нулю. |
| 2 | Второй игрок столкнулся с землёй или значение здоровья второго игрока равно нулю. |
| 3 | Первый игрок столкнулся со вторым игроком. |

Если код завершения метода *CheckCollisions* равен нулю, то далее игровой движок обновляет позиции снарядов и проверяет на их соприкосновение с игроками. Если игрок столкнулся со снарядом противника, то у данного воздушного шара уменьшается показатель здоровья, а на сцене на месте снаряда отображается анимация взрыва. Также она происходит при достижение максимальной дистанции полёта снаряда. Аналогичным образом обновляется позиция приза, который при его столкновении с игроком улучшает его характеристики. Далее, метод *Update* возвращает значение ноль.

Если код завершения метода *CheckCollisions* не равен нулю, то метод Update возвращает *GameForm* данное значение. Форма в свою очередь завершает игру путём отображения диалогового окна с сообщением о победителе игры и предлагает либо выйти из приложения, либо сыграть игрокам ещё раз.

В другом случае, программа обновляет текст с информацией об игроке, а также запускает отрисовку объектов с обновлёнными значениями позиции, вызывая метод *Draw* у объекта *BattleGame*, который сообщает *OpenGL* об отрисовке всех объектов, а также взрывов, если они должны быть произведены.

Таймер для обновления ветра вызывается каждые две секунды. Его обработчик вызывает метод *UpdateWind* объекта *BattleGame*. Ветер случайным образом задаётся для каждого из игроков. Случайным является как направление ветра, так и его скорость, которая задана с минимальным и максимальным значением. Каждую вторую итерацию ветер затихает, и игроки остаются на прежних позициях в горизонтальной плоскости.

Для генерации призов таймер *prizeTimer* каждые пять секунд вызывает метод *SpawnPrize* объекта игрового движка. Если на сцене уже присутствует объект приза, то выполнение функции завершается и программа ждёт следующего вызова метода. Если приз исчезает со сцены путём подбора его игроком или вылета за пределы экрана, то метод обращается к генератору призов, который, в свою очередь, возвращает новый приз.

Также на форме присутствует событие нажатия и отпускания кнопок игроками, обработка которых происходит в методах *UpdateKeyDown* и *UpdateKeyUp* соответственно. Данные методы проверяют, были ли нажаты кнопки, отвечающие за движение игроков, выпуск или смену снарядов, и записывают значения в список с кнопками, с которым взаимодействует метод закрытый *UpdateInput,* который вызывается при обновлении сцены.

На рисунке 2.8 показана схема алгоритма основной логики программы.

**3 ТЕСТИРОВАНИЕ, ВЕРИФИКАЦИЯ И ВАЛИДАЦИЯ ПРОЕКТА**

**3.1 Функциональное тестирование приложения**

Суть функционального тестирования приложения заключается в том, что данный проект проверяется на реализуемость требований, которые были выявлены перед проектированием приложения.

Ключевой особенностью данного вида тестирования является то, что тестирование проводится без знания внутренней работы кода, то есть, проверяется только то, что будет видеть пользователь в процессе эксплуатации данного программного обеспечения.

В процессе определения требований к игровому приложению «Битва на воздушных шарах» был определён функционал, который должна содержать в себе разработанная программа. В таблице 3.1 показаны этапы функционального тестирования.

Таблица 3.1 – Этапы функционального тестирования

|  |  |  |
| --- | --- | --- |
| **№** | **Название теста** | **Ожидаемый результат** |
| 1 | Запуск приложения на операционной системе *Windows* | Программа запустится и будет корректно работать на компьютере с операционной системой семейства *Windows* |
| 2 | Управление двумя игроками воздушными шарами в вертикальной плоскости | Приложение работает для двух игроков, которые могут придавать движение одному из воздушных шаров вверх/вниз |
| 3 | Передвижение шаров в горизонтальной плоскости с помощью ветра | Воздушные шары могут перемещаться вправо/влево с помощью ветра, который задаётся случайным образом. Игроки не могут самостоятельно управлять шарами в горизонтальной плоскости |
| 4 | Завершение игры после столкновения с землёй | Внизу шаров находится земля, при столкновении с которой, игрок проигрывает |
| 5 | Характеристики воздушных шаров | В игровом окне должны отображаться основные характеристики игроков: уровень топлива, здоровья и брони |
| 6 | Завершение игры после окончания запасов здоровья | Воздушный шар, запас здоровья которого достиг нуля, проигрывает, и игра завершается |
| 7 | Невозможность передвижения после окончания запасов топлива | Воздушный шар, запас топлива которого достиг нуля, не может передвигаться самостоятельно |
| 8 | Стрельба различными видами снарядов | Воздушные шары могут стрелять несколькими видами снарядов |
| 9 | Появление призов | На игровой сцене случайным образом появляются призы, которые могут увеличить скорость полёта снаряда, его убойную силу и радиус поражения, а также пополнить запасы топлива, брони и здоровья |

В первом этапе функционального тестирования сказано о запуске и корректной работе приложения на компьютере с операционной системой *Windows*. Работоспособность данного пункта была проверена на нескольких компьютерах различных операционных систем (*Windows* 11 и *Windows* 10). Результаты тестирования совпали с ожидаемыми.

После запуска приложения игроки могут управлять воздушными шарами в вертикальной плоскости. Первый игрок с помощью кнопок *W* и *S*, второй – с помощью *I* и *K*.

С течением времени появляется случайным образом задаваемый для каждого игрока ветер, благодаря которому воздушные шары передвигаются влево или вправо в зависимости от направления ветра. После определённого времени направление и сила ветра изменяется.

Внизу экрана находится земля, при столкновении шара с которой воздушный шар взрывается, и на экран высвечивается окно, которое сообщает о победителе.

Сверху экрана отображаются характеристики игрока: запасы топлива, здоровья и брони, а также характеристики снарядов. Если запасы здоровья игрока достигают нуля, то шар взрывается и появляется с соответствующее сообщение о завершении игры и проигрыше данного игрока. Если запасы топлива шара заканчиваются, то игрок не может перемещать воздушный шар, он двигается только вниз и по направлению ветра, пока либо не подберёт приз с добавлением запасов топлива, либо пока не достигнет земли.

Каждому из игроков доступны три вида снарядов, которые отличаются скоростью, убойной силой и радиусом поражения. Игрок может переключить текущий снаряд с помощью кнопки на клавиатуре.

С течением времени на сцене появляются призы, которые перемещаются параллельно земле в области полёта шаров, при их подборе у игрока увеличиваются характеристики шара или снарядов.

Таким образом, все этапы функционального тестирования показали ожидаемые результаты, следовательно, все требования, определённые при проектировании приложения, были выполнены.

**3.2 Модульное тестирование**

Модульное тестирование приложения является более углубленным типом тестирования, при котором тестируются отдельные модули программы. Цель данного тестирования заключается в том, чтобы определить, работает ли каждый компонент разработанного кода верно. Благодаря нему проверяется работоспособность определённых частей кода, а не всей работающей программы в целом, как в функциональном тестировании.

Одной из самых важнейших механик в игре являются изменение характеристик воздушных шаров: здоровья, топлива и брони. При этом данные показатели могут как повышаться, так и понижаться.

Изначально у игрока 100 пунктов здоровья, при столкновении со снарядом игрок получает урон в 20 пунктов. С помощью приза игрок может повысить значение здоровья, а также значение брони. Если у игрока есть броня, то у воздушного шара первоначально снимаются пункты со значения брони, а после, если значение брони равно нулю, снимаются оставшиеся пункты со здоровья воздушного шара.

Для проверки корректности выполнения данных методов были разработаны модульные тесты на языке *C*#. Для проверки корректности выполнения метода снятия пунктов здоровья были реализованы методы для проверки снятия здоровья при отсутствии и наличии пунктов брони (рисунок 3.1).



Рисунок 3.1 – Результаты модульных тестов на снятие пунктов здоровья

**Список используемых источников**

1. Гордон, С. *Computer Graphics Programming in OpenGL with C*++ / С. Гордон, Дж. Клевенгер. – *Mercury* *Learning* *and* *Information*, 2019. – 383 с.

2. Боресков, А. В. Программирование компьютерной графики. Современный *OpenGL*. – М. : ДМК Пресс, 2019. – 372 с.

3. Вольф, Д. *OpenGL* 4. Язык шейдеров. Книга рецептов / пер. с англ. А. Н. Киселева. – М. : ДМК Пресс, 2015. – 368 с.

4. Девис, Ву М. *OpenGL*. Руководство по программированию / Ву М. Девис, Дж. Нейдер, Д. Шрайнер. – 4-е изд. – СПб.: Питер, 2006. – 624 с.

5. Адамс, Дж. Программирование ролевых игр с *DirectX*. – *Thomson* *Course* *Technology* *PTR*, 2004. – 809 с.

6. Шеррод, А. *Begginning* *DirectX* 11 *Game* *Programming* / А. Шеррод, У. Джонс. – *Course* *Technology*, 2012. – 372 с.

7. Графический конвейер – *Win32* *apps*: *Microsoft*. – Электронн. Данные. – Режим доступа: <https://learn.microsoft.com/ru-ru/>. – Дата доступа: 19.03.2023.