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](#_Toc133525898)

[1 Аналитический обзор средств для создания игрового приложения 4](#_Toc133525899)

[1.1 *OpenGL* как средство для создания графики 4](#_Toc133525900)

[1.2 *DirectX* как средство для создания графики 7](#_Toc133525901)

[1.3 Сравнение *OpenGL* и *DirectX* 10](#_Toc133525902)

[1.4 Требования к проектируемому программному обеспечению 11](#_Toc133525903)

[2 Архитектура и структура проекта игрового приложения 13](#_Toc133525904)

[2.1 Проектирование архитектуры проекта 13](#_Toc133525905)

[2.2 Реализация шаблона проектирования «Декоратор» 14](#_Toc133525906)

[2.3 Реализация шаблона проектирования «Фабричный метод» 16](#_Toc133525907)

[2.4 Описание работы классов и их взаимодействие 17](#_Toc133525908)

[3 Тестирование, верификация и валидация проекта 22](#_Toc133525909)

[3.1 Функциональное тестирование приложения 22](#_Toc133525910)

[3.2 Модульное тестирование 24](#_Toc133525911)

[3.3 Опытная эксплуатация приложения 26](#_Toc133525912)

[Заключение 31](#_Toc133525913)

[Список используемых источников 32](#_Toc133525914)

[Приложение А Листинг приложения «Битва на воздушных шарах» 33](#_Toc133525915)

[Приложение Б Изображения интерфейса программы 77](#_Toc133525916)

[Приложение В Инструкция пользователя 79](#_Toc133525917)

[Приложение Г Руководство программиста 80](#_Toc133525918)

[Приложение Д Руководство системного программиста 81](#_Toc133525919)

[Приложение Е Схема использования паттерна «декоратор» 83](#_Toc133525920)

# **ВВЕДЕНИЕ**

Сегодня существует масса примеров различных игровых приложений, которые написаны на разных языках программирования и созданы для различных платформ и целевых аудиторий. Сейчас игры используются не только для развлечения, но и в массе других сфер: в обучении, торговли 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 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 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 для 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, он состоит из набора различных буферов: 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 хранит 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 используется 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.

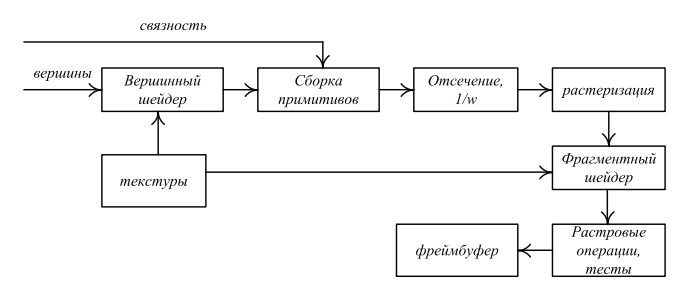


Рисунок 1.1 – Схема конвейера в *OpenGL* 3.3

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 её детализацию. [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 каждому пользователю 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 операционной системы семейства *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 необходимости в игровом приложении. Также *DirectX* поддерживает функцию добавления и обновления компонентов.

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 др. 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, в которую включаются новые функции, доступные для *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-го 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 будет использоваться 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», будет использоваться спрайтовая графика 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 приложение 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 «Битва на воздушных шарах» разрабатывалась на языке программирования *C*# с использованием *Windows* *Forms* и графики *OpenGL.* Для взаимодействия с *OpenGL* на *C*# использовалась библиотека *OpenTK*.

Для того, чтобы приложение работало с графикой *OpenGL* и *Windows* *Forms* использовался элемент управления *GLControl,* который представляет собой элемент для взаимодействия с графикой *OpenGL* на форме. Для создания и изменения спрайтов и текстур использовались графические редакторы, в том числе *Adobe* *Photoshop*.

Чтобы определить, какие сущности стоит создавать, первоначально следует определиться с тем, как будет работать приложение. Наглядно это можно сделать с помощью функциональной схемы приложения, приведённой на рисунке 2.1.



Рисунок 2.1 – Функциональная схема приложения

После составления функциональной схемы приложения намного легче определиться с разделением компонентов программы на библиотеки классов и сущности, что обеспечит приложению гибкую архитектуру.

**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 данных снарядов. Все классы, относящиеся к снарядам, будут реализованы в библиотеке классов *AmmoLibrary*.

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

Для изменения характеристик снарядов первоначально следует создать абстрактный класс снаряда, *Ammo* (приложение А, с. 33), от которого наследуются три вида снарядов:

– *ExplosiveAmmo (*приложение А, с.35);

– *SupersonicAmmo* (приложение А, с. 36)*;*

– *PiercingAmmo* (приложение А, с. 36).

Каждый вид снарядов отличается спрайтом и характеристиками: радиусом действия, дальностью полёта и скоростью.

Далее создаётся абстрактный класс *AmmoDecorator* (приложение А, с. 37)*,* который принимает в качестве поля объект *Ammo*. От абстрактного класса наследуются три вида декораторов: *RadiusDecorator* (приложение А, с. 37), *SpeedDecorator* (приложение А, с. 38) и *DistanceDecorator* (приложение А, с. 38), которые изменяют, соответственно, радиус действия, скорость и дальность полёта снарядов (рисунок 2.2).

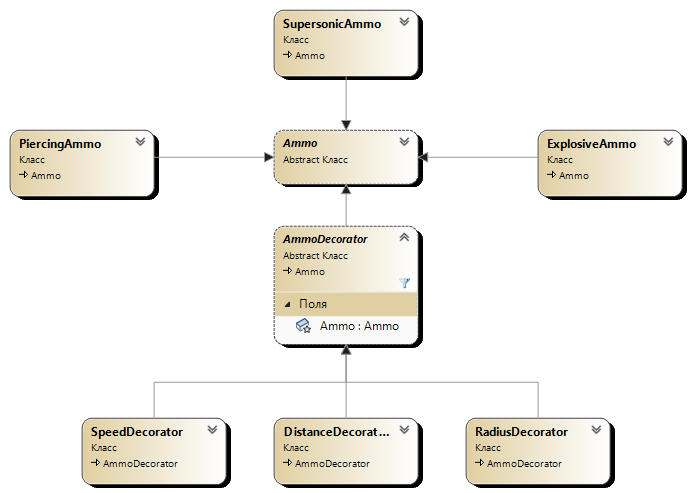


Рисунок 2.2 – Реализация шаблона «Декоратор»

Также класс *Ammo* содержит следующие методы:

– *Draw,* используемый для отрисовки его на экране;

– *GetCollider,* который возвращает прямоугольник, обозначающий границы объекта;

– *Spawn,* который используется при задании ему центра позиции и скорости перед отображением на экран;

– *Update,* используемый для изменения позиции и уменьшения оставшейся дистанции полёта снаряда;

– *UpdatePosition,* обновляющий координаты краёв объекта для дальнейшего получения коллайдера.

У *Ammo* присутствует конструктор, который копирует данные передаваемого в параметре объекта. Это сделано для того, чтобы создавать объекты с характеристиками снарядов с учётом подбора игроками призов на улучшение характеристик. В ином случае копируется ссылка на объект, а не сам объект.

**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 библиотека классов *PrizeLibrary,* где содержатся классы для работы с призами.

В данной библиотеке создаётся абстрактный класс *Prize* (приложение А, с. 38),а также четыре производных от него класса, которые представляют собой конкретные призы: *AmmoPrize* (приложение А, с. 40), *ArmourPrize* (приложение А, с. 41), *FuelPrize* (приложение А, с. 41) и *HealthPrize* (приложение А, с. 42), при подборе которых у игрока улучшаются соответственно характеристики снарядов, брони, топлива и здоровья.

Также библиотека классов *PrizeLibrary* содержит класс *PrizeGenerator* (приложение А, с. 42), 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 2.3).

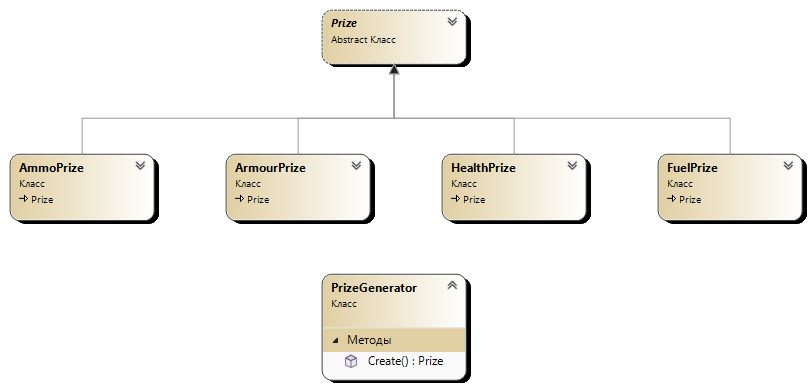


Рисунок 2.3 – Реализация шаблона «Фабричный метод»

Класс *Prize* содержит ряд методов, которые нужны для реализации логики игры:

– *GetSpeed,* возвращающий значение скорости полёта приза;

– *Draw,* используемый для отрисовки объекта приза на экране;

– *GetCollider,* возвращающий границы приза в виде прямоугольника;

– *GetPosition,* возвращающий массив из четырёх точек, которые определяют позицию объекта на экране;

– *Update,* предназначенный для обновления позиции приза при его перемещении.

**2.4 Описание работы классов и их взаимодействие**

Так как в нашем случае *GLControl* является посредником между *OpenGL* и *WinForms*, то можно было бы не разделять их на сущности, однако в будущем может появиться потребность в переносе игры, например, на *DirectX* или другую библиотеку для работы с графикой. В связи с этим 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 *CoordinatesConverter* (приложение А, с. 43), который преобразовывает точки из системы координат *OpenGL* в *Windows Forms*, так как системы координат *Windows* *Forms* и *OpenGL* сильно отличаются. В *WinForms* точка (0,0) находится в левом верхнем углу экрана и увеличивает свои координаты по *X* при движении вправо, а по *Y* при движении вниз. В *OpenGL* точкой (0,0) является центр экрана, а значения координат могут быть как отрицательными, так и положительными, варьирующимися в отрезке [-1;1] как по вертикали, так и по горизонтали в пределах экрана.

Класс *TextureLoader* (приложение А, с. 44)*,* находящийся в библиотеке *GraphicsOpenGL,* позволяет загружать текстуру с помощью функций библиотеки *OpenTK*. После загрузки текстуры в память и получения её уникального идентификатора её можно накладывать на объекты для отрисовки. Текстура задаётся таким образом, чтобы в процессе работы приложения она могла изменять свои размеры в зависимости от размера окна. Это происходит благодаря заданию параметров текстуры, которые позволяет указывать *OpenGL*.

В классе *ObjectDrawer* (приложение А, с. 45)сгенерированная в *TextureLoader* текстура накладывается на объект и изображается на экране с помощью метода *Draw*. Метод *Draw* принимает в качестве параметра логическую переменную *isFlipped,* которая при значении *true* загружает координаты текстуры с правой стороны, а при значении *false* – с левой, благодаря чему при необходимости можно отразить картинку по вертикали только средствами *OpenGL*.

Для классов, непосредственно связанных с логикой игры, была разработана библиотека классов *GameLibrary*, в которой содержатся классы движка (*BattleGame*), где прописана вся логика игры, воздушного шара (*Balloon*) и взрывов (*Explode*), которые происходят во время столкновения границ воздушных шаров и снарядов или при достижении максимальной величины дальности полёта снарядов.

При запуске приложения вызывается форма *GameForm* (приложение А, с. 46)*,* которая инициализирует объект *GLControl* из библиотеки *OpenTK,* который служит для того, чтобы взаимодействовать с *OpenGL* внутри формы *Windows* *Forms.* Для этого при загрузке данного объекта следует вызвать метод *MakeCurrent,* который делает контекст данного *GLControl* текущим для вызывающего потока. После вызова данного метода все команды, связанные с библиотекой *OpenGL,* будут интерпретироваться с этим контекстом.

Также данная форма реализует таймеры для обновления сцены, ветра и призов и используется для обновления текста с текущей информацией об игроках.

При каждом вызове таймера обновления игры (каждые 10 миллисекунд) вызывается метод *Update* созданного объекта *BattleGame* (приложение А, с. 50)*,* в котором движок игры первоначально обновляет положение игроков, вызывая метод *Update* объектов класса *Balloon* (приложение А, с. 61)*,* 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 действует 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*. Также в нём происходит проверка на то, достигло ли значение здоровья воздушного шара нуля или нет. 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, не столкнулись ли они 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 достижении максимальной дистанции полёта снаряда. Аналогичным образом обновляется позиция приза, который при его столкновении с игроком улучшает его характеристики. После этого метод *Update* возвращает значение ноль.

Вся логика, связанная с анимацией взрыва, происходит в классе *Explode* (приложение А, с. 67)*,* она работает таким образом, что при создании объекта в список *animation* загружается несколько последовательных изображений взрыва. Далее, при вызове метода *Draw* для отрисовки взрыва, происходит проверка переменной *Count,* отвечающей за текущее изображение анимации,на равенство длине (количеству отдельных изображений) анимации. Если переменная равна нулю, то метод возвращает значение *false,* которое сообщает о том, что анимация взрыва завершилась, в обратном случае метод изображает на экране текущее изображение и увеличивает значение *Count*. В *BattleGame* метод *Draw* взрыва вызывается до тех пор, пока данный метод не вернёт значение *false*.

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 позиций, 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 в пределах минимального и максимального значения. 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 закрытый метод *UpdateInput,* вызывающийся при обновлении сцены.

На рисунке 2.4 показана иерархия классов приложения.

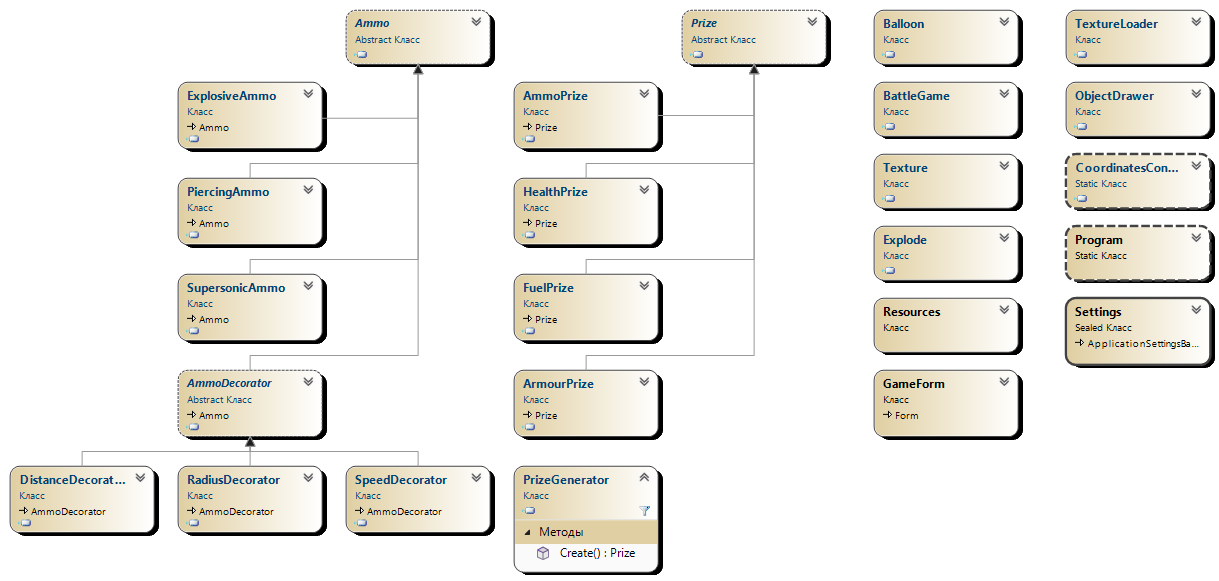


Рисунок 2.4 – Иерархия классов приложения

Таким образом, для игрового приложения «Битва на воздушных шарах» была разработана архитектура, которая позволит в будущем проводить рефакторинг кода без серьёзных затрат. Также проект спроектирован таким образом, что в будущем, при необходимости перенести проект на другое, например, графическое решение, будет затронута лишь определённая часть кода, которая чётко структурирована в рамках библиотеки классов.

**3 ТЕСТИРОВАНИЕ, ВЕРИФИКАЦИЯ И ВАЛИДАЦИЯ ПРОЕКТА**

**3.1 Функциональное тестирование приложения**

Для обеспечения корректности выполнения программы, а также для соответствия разработанного проекта первоначальным требованиям, приложение должно быть протестировано. Причём, тестирование должно проводиться как с просмотром кода, так и только с конечным продуктом, который будет доступен пользователям.

Важным видом тестирования является функциональное тестирование, суть которого 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 данной программы.

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 | 2 | 3 |
| 1 | Запуск приложения на компьютере операционной системы *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 |

Продолжение таблицы 3.1

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 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 пополнять 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. Всё управление воздушными шарами игроки выполняют с помощью клавиатуры (таблица 3.2).

Таблица 3.2 – Управление воздушными шарами

|  |  |  |
| --- | --- | --- |
| **Действие** | **Клавиша для игрока 1** | **Клавиша для игрока 2** |
| Перемещение вверх | *W* | *I* |
| Перемещение вниз | *S* | *K* |
| Стрельба вправо | *D* | *L* |
| Стрельба влево | *A* | *J* |
| Переключить вид снаряда | *X* | *M* |

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 игрока 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 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 важных 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 снятия пунктов здоровья как при отсутствии пунктов брони, так и при их наличии были разработаны модульные тесты на языке *C*# (приложение А, с. 68).

В связи с тем, что игра заканчивается, когда пункты здоровья одного из игроков достигают нуля, были разработаны тесты для проверки корректности выполнения метода *CheckAlive,* который вызывается для проверки, достиг его уровень здоровья нуля или нет.

Также для приложения были разработаны *Unit*-тесты для проверки корректности повышения основных характеристик воздушных шаров и снарядов (реализуется с помощью шаблона «Декоратор»), а также для метода преобразования точек из системы координат *OpenGL* в *Windows* *Forms*.

Все разработанные тесты прошли проверки и показали ожидаемые результаты (рисунок 3.1).

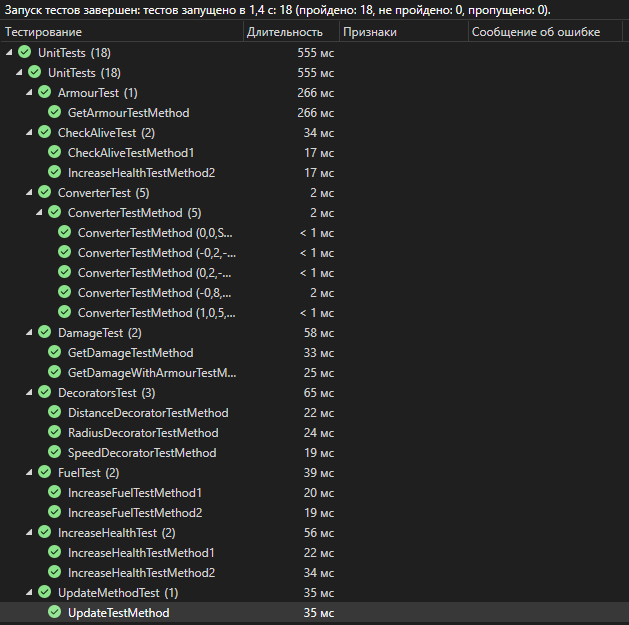


Рисунок 3.1 – Результаты выполнения модульных тестов

**3.3 Опытная эксплуатация приложения**

После запуска приложения и загрузки всех основных компонентов игровой сцены начинается непосредственно игровой процесс (рисунок 3.2). Двое игроков могут управлять воздушным шаром, а также выпускать снаряды.

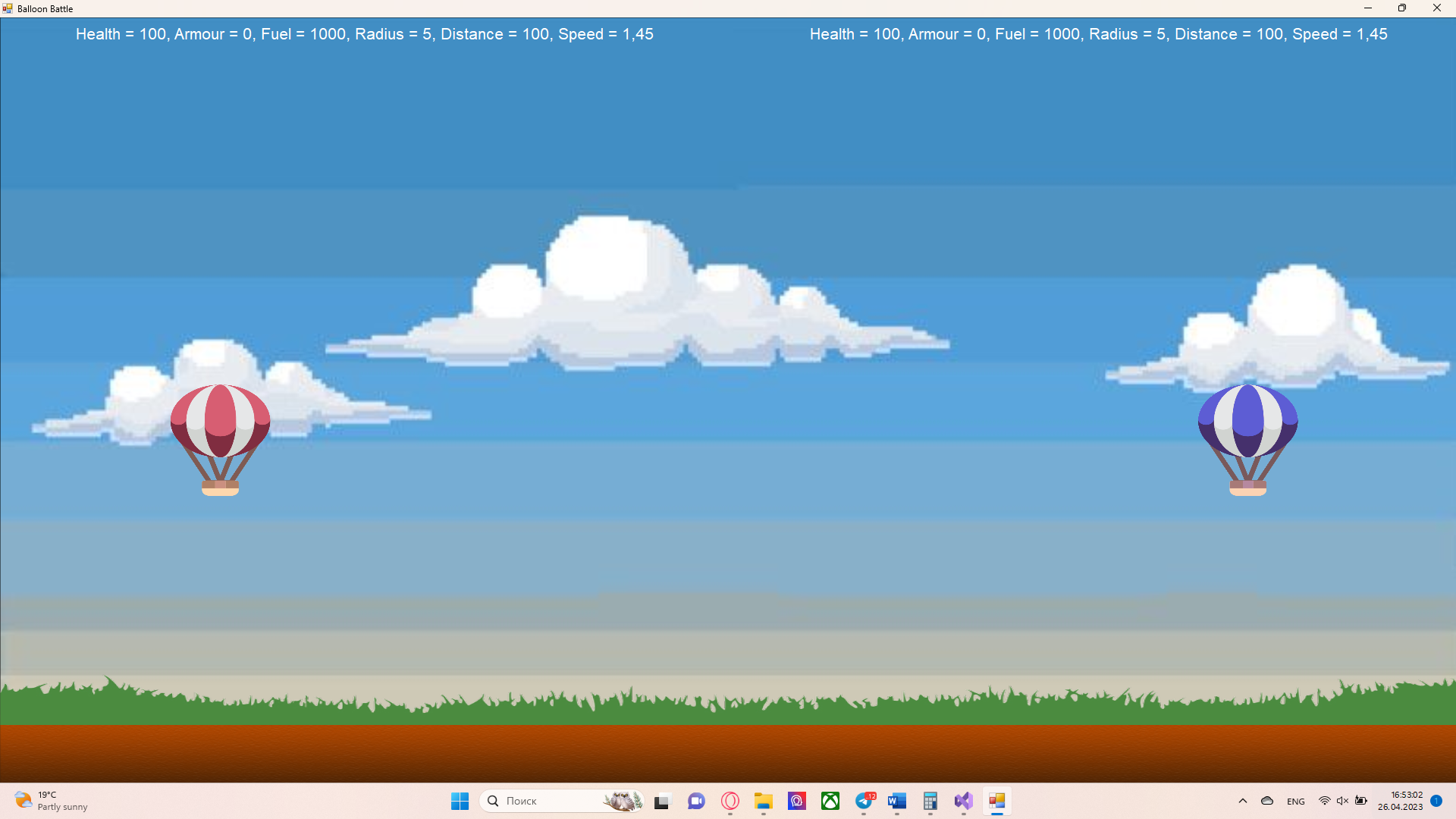


Рисунок 3.2 – Окно после запуска приложения

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

Кроме изображаемых объектов, в верхнем краю экрана демонстрируются основные характеристики воздушного шара: пункты здоровья, брони и запасы топлива, а также радиус действия, дистанция и скорость полёта текущего снаряда игрока.

Для того, чтобы сохранялась визуальная стилистика игры, а также для отличия игроков было решено создать два идентичных спрайта воздушных шаров с различным цветом (рисунок 3.3).

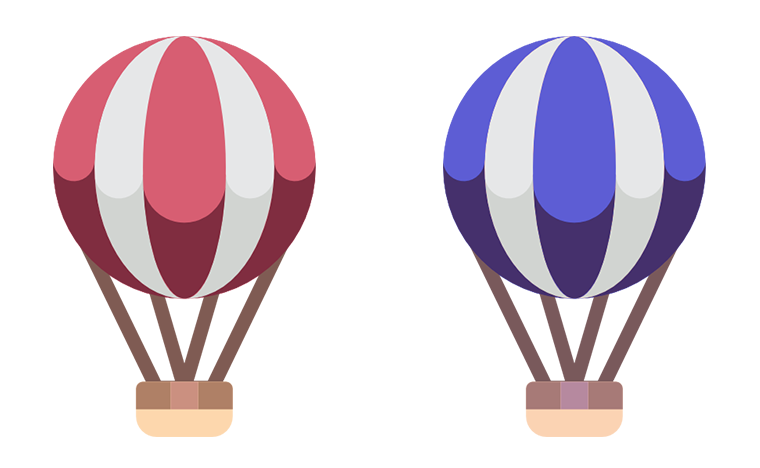


Рисунок 3.3 – Спрайты воздушных шаров

Каждый игрок имеет по три вида снарядов. Так как каждый шар может стрелять как влево, так и вправо, было принято решение создать только три спрайта снарядов и отражать их по вертикали при отрисовке объекта с помощью функций *OpenGL*. Игра разработана таким образом, что игрок может выпускать снаряды не чаще, чем раз в полсекунды для предотвращения постоянного появления снарядов на игровой сцене.

Каждый из видов снарядов отличается своими характеристиками: скоростью полёта, дальностью полёта и радиусом действия. Сверхзвуковой снаряд отличается высокой скоростью, низким радиусом и дистанцией полёта. Бронебойный снаряд имеет среднюю скорость, средний радиус действия и высокую дальность. Фугасный снаряд отличается высоким радиусом действия, средней дистанцией и низкой скоростью. На рисунке 3.4 показаны спрайты трёх видов снарядов.

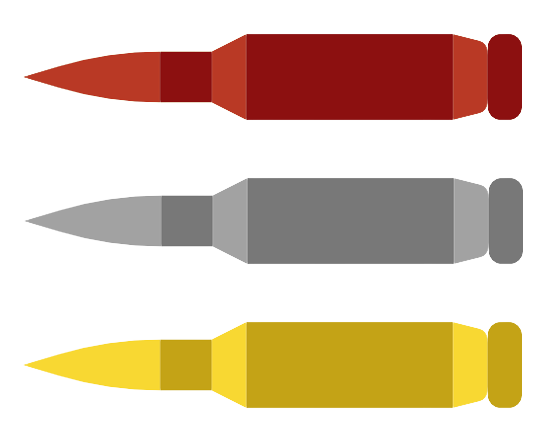


Рисунок 3.4 – Спрайты выпускаемых снарядов

После попадания снаряда в другого игрока или при столкновении шаров друг с другом или с землёй на экране отображается анимация взрыва, состоящая из пяти последовательных кадров (рисунок 3.5).

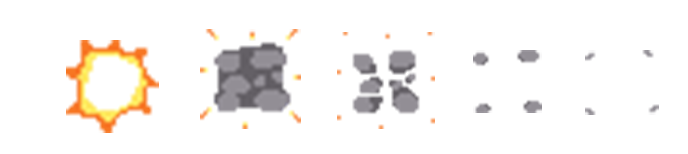


Рисунок 3.5 – Спрайты анимации взрывов

В процессе игры генерируются призы, которые перемещаются в горизонтальном направлении по сцене. В игре присутствуют четыре вида призов, которые увеличивают:

– пункты здоровья игрока;

– пункты брони игрока;

– запасы топлива воздушного шара;

– характеристики снарядов.

При подборе игроком приза увеличения здоровья (рисунок 3.5), запасы здоровья воздушного шара увеличиваются на 20 пунктов, однако пункты здоровья не могут превышать 100.

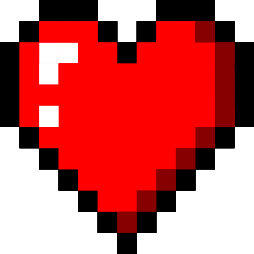


Рисунок 3.5 – Спрайт приза увеличения здоровья

При подборе приза увеличения брони игрока (рисунок 3.6) ему добавляются дополнительные 20 пунктов брони, которые снимаются при попадании снаряда в игрока. После достижения нуля в показателе пунктов брони при попадании снарядом в воздушный шар пункты начинают сниматься с показателя здоровья.



Рисунок 3.6 – Спрайт приза увеличения пунктов брони

Когда игрок сталкивается с призом добавления топлива (рисунок 3.7), воздушному шару, которым он управляет, прибавляется 300 пунктов топлива. При этом запасы топлива не могут превышать 1000 пунктов.



Рисунок 3.7 – Спрайт приза увеличения запасов топлива

При подборе игроком приза, улучшающего характеристики снарядов (рисунок 3.8), он случайным образом улучшает скорость, радиус действия или дальность полёта всех снарядов игрока.

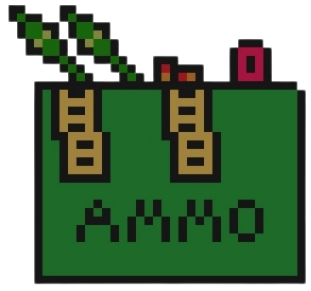


Рисунок 3.8 – Спрайт приза улучшения характеристик снарядов

Окончание игры происходит в одном из трёх случаев:

– количество здоровья одного из игроков достигло нуля;

– один из игроков столкнулся с землёй;

– игроки столкнулись друг с другом.

После завершения игры отображается диалоговое окно с сообщением о результате игры и предложением сыграть игрокам вновь (рисунок 3.8). При нажатии на кнопку «Да» игра будет запущена заново, при нажатии на кнопку «Нет» приложение будет закрыто.

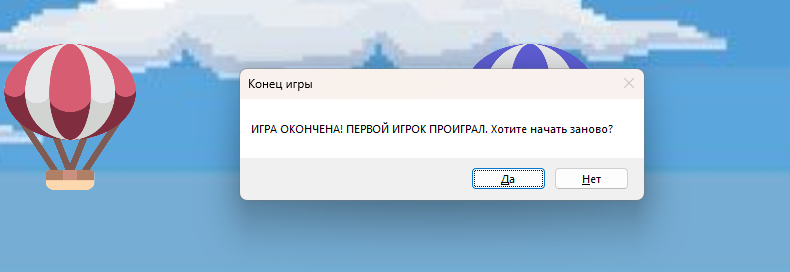


Рисунок 3.8 – Окно приложения после окончания игры

Таким образом, в процессе и после завершения разработки приложения «Битва на воздушных шарах» были проведены различные виды тестирования, в ходе которых была проведена проверка на соответствие первоначальным требованиям к программному обеспечению и на корректность работы различных модулей и всей программы в целом. Все тесты показали ожидаемые результаты и были успешно пройдены, что говорит о корректности выполнения поставленной задачи.

**ЗАКЛЮЧЕНИЕ**

В результате работы было разработано игровое приложение «Битва на воздушных шарах» на языке программирования *C*# с использованием *Windows* *Forms* и графики *OpenGL* с помощью средств интегрированной среды разработки *Visual* *Studio* 2022.

Для работы с графикой *OpenGL* на языке *C*# использовалась библиотека *OpenTK* версии 3.3.3, которая предоставляет доступ ко множеству средств библиотеки *OpenGL.*

При проектировании проекта был проведён анализ рынка компьютерных игр на предмет существования проектов в подобном жанре, в процессе которого были определены основные преимущества и недостатки подобных игр, благодаря чему было разработано приложение, которое бы сохранило все преимущества, устранило недостатки, присутствующие в существующих решениях, а также добавило что-то оригинальное.

При разработке использовалась спрайтовая графика, обрабатываемая с помощью средств графических редакторов. Также при проектировании использовались схемы, которые создавались с помощью средств *Visual* *Studio* и графического редактора *Inkscape.*

В процессе выполнения работы также были рассмотрены другие методы решения данной задачи, в том числе реализация игрового приложения на языке *C*# с помощью графики *DirectX.* Также были изучены различные подходы и шаблоны при проектировании приложения, такие как «фабричный метод» и «декоратор». Была разработана гибкая и адаптивная структура проекта и иерархия классов, внесение изменений в которую не повлекут за собой серьёзных затрат на разработку.

В данную игру одновременно могут играть два игрока на одном компьютере, управляя воздушными шарами с помощью клавиатуры. Приложение рассчитано на широкую аудиторию и, предназначено, в первую очередь, для развлечения, однако оно также может развивать скорость реакции, внимательность и концентрацию.

Курсовая работа выполнена самостоятельно, проверена в системе «Антиплагиат». Процент оригинальности составляет 98,9%. Цитирования обозначены ссылками на публикации, указанные в «Списке использованных источников».

**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/*](https://learn.microsoft.com/ru-ru/). – Дата доступа: 19.03.2023.

**ПРИЛОЖЕНИЕ А**

(обязательное)

**Листинг приложения «Битва на воздушных шарах»**

**Листинг класса *Ammo*.*cs*:**

using GraphicsOpenGL;

using OpenTK;

using System.Drawing;

namespace AmmoLibrary

{

/// <summary>

/// Абстрактный класс выпускаемого снаряда

/// </summary>

public abstract class Ammo

{

/// <summary>

/// Текстура снаряда

/// </summary>

public Texture Sprite;

/// <summary>

/// Центр позиции снаряда

/// </summary>

public Vector2 PositionCenter;

/// <summary>

/// Показатель, отвечающий за направление полёта пули (true - влево, false - вправо)

/// </summary>

public bool IsLeft;

/// <summary>

/// Скорость полёта пули

/// </summary>

public Vector2 Speed { get; set; }

/// <summary>

/// Дистанция полёта пули

/// </summary>

public float Distance { get; set; }

/// <summary>

/// Радиус действия взрыва снаряда

/// </summary>

public float Radius { get; set; }

/// <summary>

/// Позиция объекта

/// </summary>

public Vector2[] Position { get; set; }

/// <summary>

/// Создание объекта приза с помощью копирования существующего объекта

/// </summary>

/// <param name="clone">Копируемый объект</param>

public Ammo(Ammo clone)

{

this.Sprite = clone.Sprite;

PositionCenter = clone.PositionCenter;

this.IsLeft = clone.IsLeft;

Speed = clone.Speed;

Distance = clone.Distance;

Radius = clone.Radius;

Position = clone.Position;

}

/// <summary>

/// Конструктор для создания объекта

/// </summary>

public Ammo()

{

}

/// <summary>

/// Отображение снаряда на экран

/// </summary>

public void Draw()

{

UpdatePosition(false);

if (IsLeft)

{

ObjectDrawer.Draw(Sprite, Position, false);

}

else

ObjectDrawer.Draw(Sprite, Position, true);

}

/// <summary>

/// Получение границ объекта снаряда

/// </summary>

/// <param name="isExploding">Показатель, отвечающий за то, взрывается снаряд в данный момент или нет (true - взрывается, false - не взрывается)</param>

/// <returns>Коллайдер объекта снаряда</returns>

public RectangleF GetCollider(bool isExploding = false)

{

UpdatePosition(isExploding);

Vector2[] colliderPosition = Position;

float colliderWidth = (colliderPosition[2].X - colliderPosition[3].X) / 2.0f;

float colliderHeight = (colliderPosition[3].Y - colliderPosition[0].Y) / 2.0f;

float[] convertedLeftTop = CoordinatesConverter.Convert(colliderPosition[3].X, colliderPosition[3].Y);

RectangleF collider = new RectangleF(convertedLeftTop[0], convertedLeftTop[1], colliderWidth, colliderHeight);

return collider;

}

/// <summary>

/// Генерация позиции и направления полёта снаряда

/// </summary>

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

/// <param name="isLeft">Показатель, отвечающий за направление полёта пули (true - влево, false - вправо)</param>

public void Spawn(Vector2 position, bool isLeft)

{

this.PositionCenter = position;

this.IsLeft = isLeft;

}

/// <summary>

/// Обновление состояния снаряда

/// </summary>

public void Update()

{

if (IsLeft) // для проверки, в какую сторону летит снаряд

PositionCenter -= Speed;

else

PositionCenter += Speed;

Distance -= Speed.X;

}

/// <summary>

/// Обновление текущей позиции снаряда

/// </summary>

/// <param name="isExploding">Показатель, отвечающий за то, взрывается снаряд в данный момент или нет (true - взрывается, false - не взрывается)</param>

public virtual void UpdatePosition(bool isExploding)

{

float spriteWidth = 0.03f;

float spriteHeight = 0.0125f;

Position = new Vector2[4]

{

PositionCenter + new Vector2(-spriteWidth, -spriteHeight),

PositionCenter + new Vector2(spriteWidth, -spriteHeight),

PositionCenter + new Vector2(spriteWidth, spriteHeight),

PositionCenter + new Vector2(-spriteWidth, spriteHeight),

};

if (isExploding)

{

Position[0] += new Vector2(-Radius, -Radius);

Position[1] += new Vector2(Radius, -Radius);

Position[2] += new Vector2(Radius, Radius);

Position[3] += new Vector2(-Radius, Radius);

}

}

}

}

**Листинг класса *ExplosiveAmmo*.*cs*:**

using GraphicsOpenGL;

using OpenTK;

namespace AmmoLibrary

{

/// <summary>

/// Фугасный снаряд, отличается высоким радиусом действия, средней дистанцией и низкой скоростью

/// </summary>

public class ExplosiveAmmo : Ammo

{

/// <summary>

/// Конструктор для создания объекта фугасного снаряда

/// </summary>

public ExplosiveAmmo() : base()

{

this.Sprite = TextureLoader.LoadTexure("explosiveAmmo\_2.png");

this.Distance = 1.2f;

this.Radius = 0.2f;

this.Speed = new Vector2(0.009f, 0.0f);

}

/// <summary>

/// Конструктор для копирования объекта уже существующего снаряда

/// </summary>

/// <param name="clone">Копируемый объект снаряда</param>

public ExplosiveAmmo(Ammo clone) : base(clone)

{

}

}

}

**Листинг класса *SupersonicAmmo*.*cs*:**

using GraphicsOpenGL;

using OpenTK;

namespace AmmoLibrary

{

/// <summary>

/// Сверхзвуковой снаряд, отличается высокой скоростью, низким радиусом и дистанцией

/// </summary>

public class SupersonicAmmo : Ammo // сверхзвуковой

{

/// <summary>

/// Конструктор для создания объекта сверхзвукового снаряда

/// </summary>

public SupersonicAmmo() : base()

{

this.Sprite = TextureLoader.LoadTexure("supersonicAmmo.png");

this.Distance = 1.0f;

this.Radius = 0.05f;

this.Speed = new Vector2(0.0145f, 0.0f);

}

/// <summary>

/// Конструктор для копирования объекта уже существующего снаряда

/// </summary>

/// <param name="clone">Копируемый объект снаряда</param>

public SupersonicAmmo(Ammo clone) : base(clone) // для копирования объектов, а не ссылок

{

}

}

}

**Листинг класса *PiercingAmmo*.*cs*:**

using GraphicsOpenGL;

using OpenTK;

namespace AmmoLibrary

{

/// <summary>

/// Бронебойный снаряд, отличающийся высокой дальностью полёта, средней скоростью и радиусом действия

/// </summary>

public class PiercingAmmo : Ammo

{

/// <summary>

/// Конструктор для создания объекта

/// </summary>

public PiercingAmmo() : base()

{

this.Sprite = TextureLoader.LoadTexure("piercingAmmo.png");

this.Distance = 1.8f;

this.Radius = 0.1f;

this.Speed = new Vector2(0.012f, 0.0f);

}

/// <summary>

/// Конструктор для копирования объекта уже существующего снаряда

/// </summary>

/// <param name="clone">Копируемый объект снаряда</param>

public PiercingAmmo(Ammo clone) : base(clone) // для копирования объектов, а не ссылок

{

}

}

}

**Листинг класса *AmmoDecorator*.*cs*:**

namespace AmmoLibrary

{

/// <summary>

/// Декоратор для улучшения характеристик снаряда

/// </summary>

public abstract class AmmoDecorator : Ammo

{

/// <summary>

/// Улучшаемый снаряд

/// </summary>

protected Ammo Ammo;

/// <summary>

/// Конструктор создания декоратора для улучшения характеристик снаряда

/// </summary>

/// <param name="ammo">Улучшаемый снаряд</param>

public AmmoDecorator(Ammo ammo) : base(ammo)

{

this.Ammo = ammo;

}

}

}

**Листинг класса RadiusDecorator.cs:**

namespace AmmoLibrary

{

/// <summary>

/// Декоратор для увеличения радиуса действия пули

/// </summary>

public class RadiusDecorator : AmmoDecorator

{

/// <summary>

/// Конструктор для создания декоратора для увеличения радиуса действия снаряда

/// </summary>

/// <param name="ammo">Улучшаемый снаряд</param>

public RadiusDecorator(Ammo ammo) : base(ammo)

{

this.Sprite = ammo.Sprite;

this.Distance = ammo.Distance;

this.Radius = ammo.Radius \* 1.2f;

this.Speed = ammo.Speed;

}

}

}

**Листинг класса *SpeedDecorator*.*cs*:**

namespace AmmoLibrary

{

/// <summary>

/// Декоратор для увеличения скорости полёта снаряда

/// </summary>

public class SpeedDecorator : AmmoDecorator

{

/// <summary>

/// Конструктор для создания декоратора на увеличение скорости полёта снаряда

/// </summary>

/// <param name="ammo">Улучшаемый снаряд</param>

public SpeedDecorator(Ammo ammo) : base(ammo)

{

this.Sprite = ammo.Sprite;

this.Distance = ammo.Distance;

this.Radius = ammo.Radius;

this.Speed = ammo.Speed \* 1.3f;

}

}

}

**Листинг класса *DistanceDecorator*.*cs*:**

namespace AmmoLibrary

{

/// <summary>

/// Декоратор для увеличения дальности полёта пули

/// </summary>

public class DistanceDecorator : AmmoDecorator

{

/// <summary>

/// Конструктор для создания декоратора на увеличение дальности полёта пули

/// </summary>

/// <param name="ammo">Улучшаемый снаряд</param>

public DistanceDecorator(Ammo ammo) : base(ammo)

{

this.Sprite = ammo.Sprite;

this.Distance = ammo.Distance \* 1.25f;

this.Radius = ammo.Radius;

this.Speed = ammo.Speed;

}

}

}

**Листинг класса *Prize*.*cs*:**

using GraphicsOpenGL;

using OpenTK;

using System.Drawing;

namespace PrizesLibrary

{

/// <summary>

/// Класс создаваемого на сцене приза

/// </summary>

public abstract class Prize

{

/// <summary>

/// Центр позиции приза

/// </summary>

protected Vector2 CenterPosition;

/// <summary>

/// Показатель, отвечающий за направление полёта приза (true - летит влево, false - вправо)

/// </summary>

protected bool IsLeft;

/// <summary>

/// Текстура приза

/// </summary>

protected Texture Sprite;

/// <summary>

/// Получение скорости полёта приза

/// </summary>

/// <returns>Скорость полёта приза</returns>

protected abstract Vector2 GetSpeed();

/// <summary>

/// Отрисовка приза

/// </summary>

/// <param name="isFlipped">Показатель, отвечающий за то, следует отразить по горизонтали объект или нет

/// (true - следует отразить объект, false - не следует)\

/// </param>

public void Draw(bool isFlipped)

{

ObjectDrawer.Draw(Sprite, GetPosition(), isFlipped);

}

/// <summary>

/// Получение границ объекта приза

/// </summary>

/// <returns>Коллайдер приза</returns>

public RectangleF GetCollider()

{

Vector2[] colliderPosition = GetPosition();

float colliderWidth = (colliderPosition[2].X - colliderPosition[3].X) / 2.0f;

float colliderHeight = (colliderPosition[3].Y - colliderPosition[0].Y) / 2.0f;

float[] convertedLeftTop = CoordinatesConverter.Convert(colliderPosition[3].X, colliderPosition[3].Y);

RectangleF collider = new RectangleF(convertedLeftTop[0], convertedLeftTop[1], colliderWidth, colliderHeight);

return collider;

}

/// <summary>

/// Получение позиции приза

/// </summary>

/// <returns>Массив из четырёх крайних точек объекта</returns>

protected Vector2[] GetPosition()

{

return new Vector2[4]

{

CenterPosition + new Vector2(-0.06f, -0.06f),

CenterPosition + new Vector2(0.06f, -0.06f),

CenterPosition + new Vector2(0.06f, 0.06f),

CenterPosition + new Vector2(-0.06f, 0.06f),

};

}

/// <summary>

/// Обновление позиции приза

/// </summary>

public void Update()

{

if (IsLeft)

CenterPosition -= GetSpeed();

else

CenterPosition += GetSpeed();

}

/// <summary>

/// Конструктор приза при его генерации на экран

/// </summary>

/// <param name="centerPosition">Начальная позиция приза</param>

/// <param name="isLeft">Направление полёта приза (true - влево, false - вправо)</param>

public Prize(Vector2 centerPosition, bool isLeft)

{

this.CenterPosition = centerPosition;

this.IsLeft = isLeft;

}

}

}

**Листинг класса *AmmoPrize*.*cs*:**

using OpenTK;

using GraphicsOpenGL;

namespace PrizesLibrary

{

/// <summary>

/// Приз, отвечающие за увеличение характеристик выпускаемого снаряда

/// </summary>

public class AmmoPrize : Prize

{

/// <summary>

/// Конструктор приза при её генерации на экран

/// </summary>

/// <param name="centerPosition">Начальная позиция приза</param>

/// <param name="isLeft">Направление полёта приза (true - влево, false - вправо)</param>

public AmmoPrize(Vector2 centerPosition, bool isLeft) : base(centerPosition, isLeft)

{

this.CenterPosition = centerPosition;

this.IsLeft = isLeft;

this.Sprite = TextureLoader.LoadTexure("ammoPrize.png");

}

/// <summary>

/// Получение значения скорости полёта приза

/// </summary>

/// <returns>Скорость полёта приза</returns>

protected override Vector2 GetSpeed()

{

return new Vector2(0.005f, 0.0f);

}

}

}

**Листинг класса *ArmourPrize*.*cs*:**

using GraphicsOpenGL;

using OpenTK;

namespace PrizesLibrary

{

/// <summary>

/// Приз, отвечающий за увеличение показателя брони воздушного шара

/// </summary>

public class ArmourPrize : Prize

{

/// <summary>

/// Конструктор приза при её генерации на экран

/// </summary>

/// <param name="centerPosition">Начальная позиция приза</param>

/// <param name="isLeft">Направление полёта приза (true - влево, false - вправо)</param>

public ArmourPrize(Vector2 centerPosition, bool isLeft) : base(centerPosition, isLeft)

{

this.CenterPosition = centerPosition;

this.IsLeft = isLeft;

this.Sprite = TextureLoader.LoadTexure("armorPrize.png");

}

/// <summary>

/// Получение значения скорости приза

/// </summary>

/// <returns>Скорость приза</returns>

protected override Vector2 GetSpeed()

{

return new Vector2(0.005f, 0.0f);

}

}

}

**Листинг класса *FuelPrize*.*cs*:**

using GraphicsOpenGL;

using OpenTK;

namespace PrizesLibrary

{

/// <summary>

/// Приз, отвечающий за увеличение запасов топлива

/// </summary>

public class FuelPrize : Prize

{

/// <summary>

/// Конструктор приза при её генерации на экран

/// </summary>

/// <param name="centerPosition">Начальная позиция приза</param>

/// <param name="isLeft">Направление полёта приза (true - влево, false - вправо)</param>

public FuelPrize(Vector2 centerPosition, bool isLeft) : base(centerPosition, isLeft)

{

this.CenterPosition = centerPosition;

this.IsLeft = isLeft;

this.Sprite = TextureLoader.LoadTexure("fuelPrize.png");

}

/// <summary>

/// Получение скорости полёта приза

/// </summary>

/// <returns>Скорость полёта приза</returns>

protected override Vector2 GetSpeed()

{

return new Vector2(0.005f, 0.0f);

}

}

}

**Листинг класса *HealthPrize*.*cs*:**

using GraphicsOpenGL;

using OpenTK;

namespace PrizesLibrary

{

/// <summary>

/// Приз, увеличивающий показатель здоровья игрока

/// </summary>

public class HealthPrize : Prize

{

/// <summary>

/// Конструктор приза при её генерации на экран

/// </summary>

/// <param name="centerPosition">Начальная позиция приза</param>

/// <param name="isLeft">Направление полёта приза (true - влево, false - вправо)</param>

public HealthPrize(Vector2 centerPosition, bool isLeft) : base(centerPosition, isLeft)

{

this.CenterPosition = centerPosition;

this.IsLeft = isLeft;

this.Sprite = TextureLoader.LoadTexure("healthPrize.png");

}

/// <summary>

/// Получение скорости полёта приза

/// </summary>

/// <returns>Скорость полёта приза</returns>

protected override Vector2 GetSpeed()

{

return new Vector2(0.005f, 0.0f);

}

}

}

**Листинг класса *PrizeGenerator*.*cs*:**

using OpenTK;

using System;

namespace PrizesLibrary

{

/// <summary>

/// Генератор призов, реализующий шаблон "фабричный метод"

/// </summary>

public class PrizeGenerator

{

/// <summary>

/// Создание объекта приза

/// </summary>

/// <param name="height">Высота экрана</param>

/// <returns>Созданный генератором приз</returns>

public Prize Create(int height)

{

Random random = new Random();

Prize newPrize=null;

float prizePozitionX;

bool isLeft;

int prizeSpawnSide = random.Next(0, 2);

if (prizeSpawnSide == 0)

{

isLeft = false;

prizePozitionX = -1.05f;

}

else

{

isLeft = true;

prizePozitionX = 1.05f;

}

float prizePozitionY = (float)(random.Next((int)(-0.6f \* height), (int)(0.7f \* height))) / (float)height;

int prizeType = random.Next(0, 4);

switch (prizeType)

{

case 0:

newPrize = new AmmoPrize(new Vector2(prizePozitionX, prizePozitionY), isLeft);

break;

case 1:

newPrize = new ArmourPrize(new Vector2(prizePozitionX, prizePozitionY), isLeft);

break;

case 2:

newPrize = new FuelPrize(new Vector2(prizePozitionX, prizePozitionY), isLeft);

break;

case 3:

newPrize = new HealthPrize(new Vector2(prizePozitionX, prizePozitionY), isLeft);

break;

}

return newPrize;

}

}

}

**Листинг класса *CoordinatesConverter*.*cs*:**

namespace GraphicsOpenGL

{

/// <summary>

/// Класс, предназначенный для конвертации из системы координат OpenGL в WinForms

/// </summary>

public static class CoordinatesConverter

{

/// <summary>

/// Метод конвертации точки из системы координат OpenGL в WinForms

/// </summary>

/// <param name="pointX">Координата X заданной точки в системе координат OpenGL</param>

/// <param name="pointY">Координата Y заданной точки в системе координат OpenGL</param>

/// <returns>Точка в системе координат WInForms</returns>

public static float[] Convert(float pointX, float pointY)

{

decimal centralPointX = 0.5M;

decimal centralPointY = 0.5M;

float[] resultPoint = new float[2];

resultPoint[0] = (float)(centralPointX + (decimal)(pointX / 2.0f));

resultPoint[1] = (float)(centralPointY - (decimal)(pointY / 2.0f));

return resultPoint;

}

}

}

**Листинг класса *TextureLoader*.*cs*:**

using OpenTK.Graphics.OpenGL;

using System.Drawing;

using System.Drawing.Imaging;

namespace GraphicsOpenGL

{

/// <summary>

/// Класс, предназначенный для загрузки и генерации текстуры

/// </summary>

public class TextureLoader

{

/// <summary>

/// Загрузка и генерация текстуры

/// </summary>

/// <param name="path">Путь к файлу с текстурой</param>

/// <returns>Загруженная и сгенерированная текстура</returns>

public static Texture LoadTexure(string path)

{

int textureId = GL.GenTexture();

GL.BindTexture(TextureTarget.Texture2D, textureId);

Bitmap bitmap = new Bitmap("Assets/" + path);

BitmapData data = bitmap.LockBits(new Rectangle(0, 0, bitmap.Width, bitmap.Height), ImageLockMode.ReadOnly,

System.Drawing.Imaging.PixelFormat.Format32bppArgb);

GL.PixelStore(PixelStoreParameter.UnpackAlignment, 1);

GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, data.Width, data.Height, 0,

OpenTK.Graphics.OpenGL.PixelFormat.Bgra, PixelType.UnsignedByte, data.Scan0);

bitmap.UnlockBits(data);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapS, (int)TextureWrapMode.Clamp);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapT, (int)TextureWrapMode.Clamp);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Linear);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter, (int)TextureMinFilter.Linear);

return new Texture(textureId, bitmap.Width, bitmap.Height);

}

}

}

**Листинг класса *ObjectDrawer*.*cs*:**

using OpenTK;

using OpenTK.Graphics.OpenGL;

namespace GraphicsOpenGL

{

/// <summary>

/// Класс для отрисовки объектов на экран

/// </summary>

public class ObjectDrawer

{

/// <summary>

/// Отрисовка объектов на экран

/// </summary>

/// <param name="texture">Текстура отрисовываемого объекта</param>

/// <param name="position">Позиция дла отрисовки объекта</param>

/// <param name="isFlipped">Показатель, отвечающий за то, нужно ли отражать по вертикали объект

/// (true - объект следует отражать, false - не следует)</param>

public static void Draw(Texture texture, Vector2[] position, bool isFlipped)

{

Start();

Vector2[] vertices;

if (isFlipped)

{

vertices = new Vector2[4] // вершины спрайта

{

new Vector2(1.0f,1.0f), // правый низ

new Vector2(0.0f,1.0f), // правый верх

new Vector2(0.0f,0.0f), // левый верх

new Vector2(1.0f,0.0f), // левый низ

};

}

else

{

vertices = new Vector2[4] // вершины спрайта

{

new Vector2(0.0f,1.0f), // левый низ

new Vector2(1.0f,1.0f), // правый низ

new Vector2(1.0f,0.0f), // правый верх

new Vector2(0.0f,0.0f), // левый верх

};

}

GL.Enable(EnableCap.Texture2D);

GL.BindTexture(TextureTarget.Texture2D, texture.Id);

GL.Begin(PrimitiveType.Quads);

for (int i = 0; i < 4; i++)

{

GL.TexCoord2(vertices[i]);

GL.Vertex2(position[i]);

}

GL.End();

}

/// <summary>

/// Подготовка OpenGL к отрисовке объектов

/// </summary>

private static void Start()

{

GL.MatrixMode(MatrixMode.Projection);

GL.LoadIdentity();

GL.Ortho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);

GL.MatrixMode(MatrixMode.Modelview);

GL.LoadIdentity();

}

}

}

**Листинг класса *Texture*.*cs*:**

namespace GraphicsOpenGL

{

/// <summary>

/// Класс текстуры

/// </summary>

public class Texture

{

/// <summary>

/// Идентификатор, генерируемый OpenGL

/// </summary>

public int Id;

/// <summary>

/// Ширина текстуры

/// </summary>

public int Width;

/// <summary>

/// Высота текстуры

/// </summary>

public int Height;

/// <summary>

/// Конструктор создания объекта текстуры

/// </summary>

/// <param name="id">Идентификатор текстуры</param>

/// <param name="width">Ширина текстуры</param>

/// <param name="height">Высота текстуры</param>

public Texture(int id, int width, int height)

{

Id = id;

Width = width;

Height = height;

}

}

}

**Листинг класса *GameForm.cs*:**

using System;

using System.Drawing;

using System.Windows.Forms;

using OpenTK.Graphics.OpenGL;

using GameLibrary;

namespace Ballon\_Battle

{

public partial class GameForm : Form

{

/// <summary>

/// Объект игрового движка

/// </summary>

BattleGame gameEngine;

/// <summary>

/// Label для отображения текущего состояния первого игрока

/// </summary>

Label firstPlayerInfo;

/// <summary>

/// Label для отображения текущего состояния второго игрока

/// </summary>

Label secondPlayerInfo; // label для отображения текущего состояния второго игрока

/// <summary>

/// Конструктор формы

/// </summary>

public GameForm()

{

InitializeComponent();

CenterToScreen();

glControl.Size = this.Size;

glControl.Visible = true;

glTimer.Start();

prizeTimer.Start();

windTimer.Start();

gameEngine = new BattleGame();

}

/// <summary>

/// Обработчик события загрузки GLControl

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void glControl\_Load(object sender, EventArgs e)

{

glControl.MakeCurrent();

gameEngine.LoadGLControl();

gameEngine.LoadObjects();

glControl.SendToBack();

this.WindowState = FormWindowState.Maximized; // для открытия окна в полном экране

}

/// <summary>

/// Обработчик события для отрисовки GLControl

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void glControl\_Paint(object sender, PaintEventArgs e)

{

GL.Clear(ClearBufferMask.DepthBufferBit | ClearBufferMask.ColorBufferBit); // ?

gameEngine.Draw();

glControl.SwapBuffers();

}

/// <summary>

/// Обновление информации об игроках в Label

/// </summary>

private void updateInfo()

{

firstPlayerInfo.SetBounds((int)(0.05 \* Width), (int)(0.01 \* Height), (int)(0.45 \* Width), (int)(0.03 \* Height)); // информация первого игрока (здоровье, топливо, броня)

firstPlayerInfo.Font = new Font("Arial", 0.008f \* Width);

firstPlayerInfo.Text = gameEngine.GetFirstPlayerInfo();

secondPlayerInfo.SetBounds((int)(0.55 \* Width), (int)(0.01 \* Height), (int)(0.45 \* Width), (int)(0.03 \* Height)); // информация первого игрока (здоровье, топливо, броня)

secondPlayerInfo.Font = new Font("Arial", 0.008f \* Width);

secondPlayerInfo.Text = gameEngine.GetSecondPlayerInfo();

}

/// <summary>

/// Завершение игры

/// </summary>

/// <param name="message">Сообщение, выводимое при завершении игры</param>

private void endGame(string message)

{

glTimer.Stop();

DialogResult result = MessageBox.Show(message, "Конец игры", MessageBoxButtons.YesNo);

if (result == DialogResult.Yes)

{

Application.Restart();

Environment.Exit(0);

}

else

this.Close();

}

/// <summary>

/// Обработчик события тика таймера игры перед окончанием игры в случае ничьей

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void glTimer\_DrawTick(object sender, EventArgs e)

{

glControl.Refresh();

if (gameEngine.GetExplodesCount() <= 0)

{

endGame("ИГРА ОКОНЧЕНА! НИЧЬЯ! Хотите начать заново?");

}

}

/// <summary>

/// Обработчик события тика таймера игры перед окончанием игры в случае поражения первого игрока

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void glTimer\_FirstPlayerLooseTick(object sender, EventArgs e)

{

glControl.Refresh();

if (gameEngine.GetExplodesCount() <= 0)

{

endGame("ИГРА ОКОНЧЕНА! ПЕРВОЙ ИГРОК ПРОИГРАЛ. Хотите начать заново?");

}

}

/// <summary>

/// Обработчик события тика таймера игры перед окончанием игры в случае поражения второго игрока

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void glTimer\_SecondPlayerLooseTick(object sender, EventArgs e)

{

glControl.Refresh();

if (gameEngine.GetExplodesCount() <= 0)

{

endGame("ИГРА ОКОНЧЕНА! ВТОРОЙ ИГРОК ПРОИГРАЛ. Хотите начать заново?");

}

}

/// <summary>

/// Обработчик события тика таймера, отвечающего за появление ветра

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void windTimer\_Tick(object sender, EventArgs e)

{

gameEngine.UpdateWind();

}

/// <summary>

/// Обработчик события для обновления картинки каждые 10 миллисекунд

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void glTimer\_Tick(object sender, EventArgs e)

{

int resultCode = gameEngine.Update();

switch(resultCode)

{

case 0:

break;

case 1:

glTimer.Stop();

prizeTimer.Stop();

windTimer.Stop();

glTimer.Tick -= glTimer\_Tick;

glTimer.Tick += glTimer\_FirstPlayerLooseTick;

glTimer.Start();

break;

case 2:

glTimer.Stop();

prizeTimer.Stop();

windTimer.Stop();

glTimer.Tick -= glTimer\_Tick;

glTimer.Tick += glTimer\_SecondPlayerLooseTick;

glTimer.Start();

break;

case 3:

glTimer.Stop();

prizeTimer.Stop();

windTimer.Stop();

glTimer.Tick -= glTimer\_Tick;

glTimer.Tick += glTimer\_DrawTick;

glTimer.Start();

break;

}

updateInfo();

glControl.Refresh();

}

/// <summary>

/// Обработчик события тика таймера, отвечающего за генерацию призов

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void prizeTimer\_Tick(object sender, EventArgs e)

{

gameEngine.SpawnPrize(this.Height);

}

/// <summary>

/// Отвечает за изменение размера GLControl при изменении размера окна

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void GameForm\_Resize(object sender, EventArgs e)

{

glControl.Size = this.Size;

GL.Viewport(0, 0, Width, Height);

}

/// <summary>

/// Обработчик события отпускания кнопки клавиатуры

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void glControl\_KeyUp(object sender, KeyEventArgs e)

{

gameEngine.UpdateKeyUp(e);

}

/// <summary>

/// Обработчки события нажатия кнопки клавиатуры

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void glControl\_KeyDown(object sender, KeyEventArgs e)

{

gameEngine.UpdateKeyDown(e);

}

}

}

**Листинг класса *BattleGame*.*cs*:**

using AmmoLibrary;

using GraphicsOpenGL;

using OpenTK;

using PrizesLibrary;

using System;

using System.Collections.Generic;

using System.Drawing;

using OpenTK.Graphics.OpenGL;

using System.Windows.Forms;

namespace GameLibrary

{

/// <summary>

/// Движок игры, содержащий всю основную логику

/// </summary>

public class BattleGame

{

/// <summary>

/// Текстура заднего фона сцены

/// </summary>

Texture backgroundTexture;

/// <summary>

/// Текстура земли

/// </summary>

Texture landTexture;

/// <summary>

/// Текстура травы

/// </summary>

Texture grassTexture;

/// <summary>

/// Объект первого игрока

/// </summary>

Balloon firstPlayer;

/// <summary>

/// Объект второго игрока

/// </summary>

Balloon secondPlayer;

/// <summary>

/// Границы земли

/// </summary>

RectangleF landCollider;

/// <summary>

/// Границы экрана

/// </summary>

RectangleF screenCollider;

/// <summary>

/// Границы объекта первого игрока

/// </summary>

RectangleF firstPlayerCollider;

/// <summary>

/// Границы объекта второго игрока

/// </summary>

RectangleF secondPlayerCollider;

/// <summary>

/// Границы объекта приза, находящегося на сцене

/// </summary>

RectangleF currentPrizeCollider;

/// <summary>

/// Список с выпущенными первым игроком снарядами, находящимися на сцене

/// </summary>

List<Ammo> firstAmmos;

/// <summary>

/// Список с выпущенными вторым игроком снарядами, находящимися на сцене

/// </summary>

List<Ammo> secondAmmos;

/// <summary>

/// Список с взрывами, происходящими на сцене

/// </summary>

List<Explode> explodes;

/// <summary>

/// Объект для генерации случайных чисел

/// </summary>

Random random = new Random();

/// <summary>

/// Объект текущего приза

/// </summary>

Prize currentPrize = null;

/// <summary>

/// Максимально возможная скорость ветра, умноженная на 10000

/// </summary>

int maxWindSpeed = 20;

/// <summary>

/// Минимально возможная скорость ветра, умноженная на 10000

/// </summary>

int minWindSpeed = 5;

/// <summary>

/// Количество тиков таймера ветра

/// </summary>

int windTicks = 0;

/// <summary>

/// Переменная, отвечающая за направление ветра для первого игркоа (true - ветер дует налево, false - направо)

/// </summary>

bool isFirstPlayerWindLeft = false;

/// <summary>

/// Переменная, отвечающая за направление ветра для второго игркоа (true - ветер дует налево, false - направо)

/// </summary>

bool isSecondPlayerWindLeft = false;

/// <summary>

/// Список для проверки нажатия кнопок игроками (W, S, I, K, J, D, A, L, X, M)

/// </summary>

List<bool> keysDown;

/// <summary>

/// Количество тиков выстрелов второго игрока, отвечает за выпуск снарядов не ранее 50 тиков

/// </summary>

int secondPlayerTicks = 50;

/// <summary>

/// Количество тиков выстрелов первого игрока, отвечает за выпуск снарядов не ранее 50 тиков

/// </summary>

int firstPlayerTicks = 50;

/// <summary>

/// Загружает объекты, необходимые для процесса игры

/// </summary>

public void LoadObjects()

{

backgroundTexture = TextureLoader.LoadTexure("clouds.jpg");

Texture firstPlayerTexture = TextureLoader.LoadTexure("firstPlayerBalloon.png");

firstPlayer = new Balloon(new Vector2(-0.7f, 0.0f), firstPlayerTexture);

Texture secondPlayerTexture = TextureLoader.LoadTexure("secondPlayerBalloon.png");

secondPlayer = new Balloon(new Vector2(0.7f, 0.0f), secondPlayerTexture);

landTexture = TextureLoader.LoadTexure("testLandNew.png");

grassTexture = TextureLoader.LoadTexure("grasstexture.png");

firstAmmos = new List<Ammo>();

secondAmmos = new List<Ammo>();

screenCollider = new RectangleF(0.0f, 0.05f, 1.0f, 0.825f);

landCollider = new RectangleF(0.0f, 0.89f, 1.0f, 0.12f);

explodes = new List<Explode>();

keysDown = new List<bool>();

for (int i = 0; i < 8; i++)

keysDown.Add(false);

}

/// <summary>

/// Включение, необходимых для функционирования OpenGL, функций

/// </summary>

public void LoadGLControl()

{

GL.Enable(EnableCap.Texture2D);

GL.Enable(EnableCap.Blend);

GL.BlendFunc(BlendingFactor.SrcAlpha, BlendingFactor.OneMinusSrcAlpha);

}

/// <summary>

/// Отрисовывает объекты (фон, траву, землю, игроков, снаряды, взрывы и приз)

/// </summary>

public void Draw()

{

ObjectDrawer.Draw(backgroundTexture, new Vector2[4]

{

new Vector2(-1.0f, -1.0f),

new Vector2(1.0f, -1.0f),

new Vector2(1.0f, 1.0f),

new Vector2(-1.0f, 1.0f),

}, false);

ObjectDrawer.Draw(grassTexture, new Vector2[4]

{

new Vector2(-1.0f, -0.8f),

new Vector2(1.0f, -0.8f),

new Vector2(1.0f, -0.55f),

new Vector2(-1.0f, -0.55f),

}, false);

ObjectDrawer.Draw(landTexture, new Vector2[4]

{

new Vector2(-1.0f, -1.0f),

new Vector2(1.0f, -1.0f),

new Vector2(1.0f, -0.78f),

new Vector2(-1.0f, -0.78f),

}, false);

firstPlayer.Draw(false);

secondPlayer.Draw(false);

foreach (var item in firstAmmos)

{

item.Draw();

}

foreach (var item in secondAmmos)

{

item.Draw();

}

for (int i = 0; i < explodes.Count; i++)

{

int thisCount = explodes.Count;

if (!explodes[i].Draw(false))

{

explodes.RemoveAt(i);

}

if (thisCount != explodes.Count)

i--;

}

if (currentPrize != null)

currentPrize.Draw(false);

}

/// <summary>

/// Обновление игры

/// </summary>

/// <returns>Код завершения программы (0 - продолжение игры,

/// 1 - заверешение игры поражением первого игрока,

/// 2 - завершение игры поражением второго игрока,

/// 3 - завершение игры ничьей)

/// </returns>

public int Update()

{

firstPlayerTicks++;

secondPlayerTicks++;

firstPlayer.Update();

secondPlayer.Update();

firstPlayerCollider = firstPlayer.GetCollider();

secondPlayerCollider = secondPlayer.GetCollider();

UpdateInput();

int codeResult = CheckCollisions();

if (codeResult != 0)

{

return codeResult;

}

UpdateAmmos();

UpdatePrize();

return 0;

}

/// <summary>

/// Обновление нажатия кнопок на клавиатуре игроками

/// </summary>

private void UpdateInput()

{

if (keysDown[0] && (firstPlayerCollider.Y > screenCollider.Y))

firstPlayer.Update(new Vector2(0f, 0.01f));

if (keysDown[1])

firstPlayer.Update(new Vector2(0f, -0.01f));

if (keysDown[2] && (secondPlayerCollider.Y > screenCollider.Y))

secondPlayer.Update(new Vector2(0f, 0.01f));

if (keysDown[3])

secondPlayer.Update(new Vector2(0f, -0.01f));

if ((keysDown[4] || keysDown[7]) && secondPlayerTicks >= 50)

{

secondPlayerTicks = 0;

Ammo newAmmo = null;

if (keysDown[4])

newAmmo = secondPlayer.GetCurrentAmmo(true);

else if (keysDown[7])

newAmmo = secondPlayer.GetCurrentAmmo(false);

// Debug.WriteLine($"Distance={newAmmo.Distance}, Radius={newAmmo.Radius}, Speed={newAmmo.Speed.X}");

secondAmmos.Add(newAmmo);

}

if ((keysDown[5] || keysDown[6]) && firstPlayerTicks >= 50)

{

firstPlayerTicks = 0;

Ammo newAmmo = null;

if (keysDown[5])

newAmmo = firstPlayer.GetCurrentAmmo(false);

else if (keysDown[6])

newAmmo = firstPlayer.GetCurrentAmmo(true);

// Debug.WriteLine($"Distance={newAmmo.Distance}, Radius={newAmmo.Radius}, Speed={newAmmo.Speed.X}");

firstAmmos.Add(newAmmo);

}

}

/// <summary>

/// Обработка столкновения коллайдеров и проверка на то, больше ли 0 значение здоровья шаров

/// </summary>

/// <returns>Код завершения метода (0 - продолжение игры,

/// 1 - заверешение игры поражением первого игрока,

/// 2 - завершение игры поражением второго игрока,

/// 3 - завершение игры ничьей)

/// </returns></returns>

private int CheckCollisions ()

{

if ((firstPlayerCollider.X <= screenCollider.X) && isFirstPlayerWindLeft) // ?

{

firstPlayer.ChangeWindCondition(false);

}

else if ((firstPlayerCollider.X + secondPlayerCollider.Width >= screenCollider.X + screenCollider.Width) && !isFirstPlayerWindLeft)

{

firstPlayer.ChangeWindCondition(false);

}

else

firstPlayer.ChangeWindCondition(true);

if ((secondPlayerCollider.X + secondPlayerCollider.Width >= screenCollider.X + screenCollider.Width) && !isSecondPlayerWindLeft) // ||

{

secondPlayer.ChangeWindCondition(false);

}

else if ((secondPlayerCollider.X <= screenCollider.X) && isSecondPlayerWindLeft)

{

secondPlayer.ChangeWindCondition(false);

}

else

secondPlayer.ChangeWindCondition(true);

if (landCollider.IntersectsWith(firstPlayerCollider) || !firstPlayer.CheckAlive())

{

explodes.Add(new Explode(firstPlayer.GetPosition()));

return 1;

}

if (landCollider.IntersectsWith(secondPlayerCollider) || !secondPlayer.CheckAlive())

{

explodes.Add(new Explode(secondPlayer.GetPosition()));

return 2;

}

if (firstPlayerCollider.IntersectsWith(secondPlayerCollider))

{

explodes.Add(new Explode(firstPlayer.GetPosition()));

explodes.Add(new Explode(secondPlayer.GetPosition()));

return 3;

}

return 0;

}

/// <summary>

/// Обновление состояния выпущенных снарядов

/// </summary>

private void UpdateAmmos()

{

for (int i = 0; i < firstAmmos.Count; i++)

{

int thisCount = firstAmmos.Count;

RectangleF ammoCollider = firstAmmos[i].GetCollider(false);

firstAmmos[i].Update();

if (secondPlayerCollider.IntersectsWith(ammoCollider))

{

firstAmmos[i].UpdatePosition(true);

explodes.Add(new Explode(firstAmmos[i].Position));

firstAmmos.RemoveAt(i);

secondPlayer.GetDamage();

}

else if (!ammoCollider.IntersectsWith(screenCollider))

{

firstAmmos.RemoveAt(i);

}

else if (firstAmmos[i].Distance <= 0)

{

RectangleF ammoExplode = firstAmmos[i].GetCollider(true);

explodes.Add(new Explode(firstAmmos[i].Position));

firstAmmos.RemoveAt(i);

if (ammoExplode.IntersectsWith(secondPlayerCollider))

{

secondPlayer.GetDamage();

}

}

if (thisCount != firstAmmos.Count)

i--;

}

for (int i = 0; i < secondAmmos.Count; i++)

{

int thisCount = secondAmmos.Count;

RectangleF ammoCollider = secondAmmos[i].GetCollider(false);

secondAmmos[i].Update();

if (firstPlayerCollider.IntersectsWith(ammoCollider))

{

secondAmmos[i].UpdatePosition(true);

explodes.Add(new Explode(secondAmmos[i].Position));

secondAmmos.RemoveAt(i);

firstPlayer.GetDamage();

}

else if (!ammoCollider.IntersectsWith(screenCollider))

{

secondAmmos.RemoveAt(i);

}

else if (secondAmmos[i].Distance <= 0)

{

RectangleF ammoExplode = secondAmmos[i].GetCollider(true);

explodes.Add(new Explode(secondAmmos[i].Position));

secondAmmos.RemoveAt(i);

if (ammoExplode.IntersectsWith(firstPlayerCollider))

{

firstPlayer.GetDamage();

}

}

if (thisCount != secondAmmos.Count)

i--;

}

}

/// <summary>

/// Обновление состояния текущего приза и добавление эффектов от него

/// </summary>

private void UpdatePrize()

{

if (currentPrize != null)

{

currentPrizeCollider = currentPrize.GetCollider();

currentPrize.Update();

if (firstPlayerCollider.IntersectsWith(currentPrizeCollider))

{

if (currentPrize is AmmoPrize)

{

firstPlayer.ChangeAmmoCharesterictics();

currentPrize = null;

}

else if (currentPrize is ArmourPrize)

{

firstPlayer.IncreaseArmour();

currentPrize = null;

}

else if (currentPrize is FuelPrize)

{

firstPlayer.IncreaseFuel();

currentPrize = null;

}

else if (currentPrize is HealthPrize)

{

firstPlayer.IncreaseHealth();

currentPrize = null;

}

}

if (currentPrize != null && secondPlayerCollider.IntersectsWith(currentPrizeCollider))

{

if (currentPrize is AmmoPrize)

{

secondPlayer.ChangeAmmoCharesterictics();

currentPrize = null;

}

else if (currentPrize is ArmourPrize)

{

secondPlayer.IncreaseArmour();

currentPrize = null;

}

else if (currentPrize is FuelPrize)

{

secondPlayer.IncreaseFuel();

currentPrize = null;

}

else if (currentPrize is HealthPrize)

{

secondPlayer.IncreaseHealth();

currentPrize = null;

}

}

if (currentPrize != null && !screenCollider.IntersectsWith(currentPrizeCollider))

{

currentPrize = null;

}

}

}

/// <summary>

/// Обновление состояния ветра

/// </summary>

public void UpdateWind()

{

if (windTicks == 1)

{

firstPlayer.ChangeWindCondition(false);

secondPlayer.ChangeWindCondition(false);

}

else

{

int windDirection = random.Next(0, 2);

float windSpeed = random.Next(minWindSpeed, maxWindSpeed + 1) / 10000f;

switch (windDirection)

{

case 0:

firstPlayer.ChangeWindSpeed(new Vector2(-windSpeed, 0.0f));

isFirstPlayerWindLeft = true;

break;

case 1:

firstPlayer.ChangeWindSpeed(new Vector2(windSpeed, 0.0f));

isFirstPlayerWindLeft = false;

break;

}

firstPlayer.ChangeWindCondition(true);

windDirection = random.Next(0, 2);

windSpeed = random.Next(minWindSpeed, maxWindSpeed + 1) / 10000f;

switch (windDirection)

{

case 0:

secondPlayer.ChangeWindSpeed(new Vector2(-windSpeed, 0.0f));

isSecondPlayerWindLeft = true;

break;

case 1:

secondPlayer.ChangeWindSpeed(new Vector2(windSpeed, 0.0f));

isSecondPlayerWindLeft = false;

break;

}

secondPlayer.ChangeWindCondition(true);

}

windTicks++;

if (windTicks >= 2)

windTicks = 0;

}

/// <summary>

/// Генерация приза в случае отсутствия на сцене приза

/// </summary>

/// <param name="height">Текущая высота экрана</param>

public void SpawnPrize(int height)

{

if (currentPrize != null)

return;

PrizeGenerator prizeGenerator = new PrizeGenerator();

Prize newPrize = prizeGenerator.Create(height);

currentPrize = newPrize;

}

/// <summary>

/// Обновление списка с нажатыми кнопками после события нажатия клавиши

/// </summary>

/// <param name="e">Нажатая клавиша</param>

public void UpdateKeyDown(KeyEventArgs e)

{

switch (e.KeyCode)

{

case Keys.W:

keysDown[0] = true;

break;

case Keys.S:

keysDown[1] = true;

break;

case Keys.I:

keysDown[2] = true;

break;

case Keys.K:

keysDown[3] = true;

break;

case Keys.J:

keysDown[4] = true;

break;

case Keys.D:

keysDown[5] = true;

break;

case Keys.A:

keysDown[6] = true;

break;

case Keys.L:

keysDown[7] = true;

break;

}

}

/// <summary>

/// Обновление списка с нажатыми кнопками после события отпускания клавиши

/// </summary>

/// <param name="e">Отжатая клавиша</param>

public void UpdateKeyUp(KeyEventArgs e)

{

switch (e.KeyCode)

{

case Keys.W:

keysDown[0] = false;

break;

case Keys.S:

keysDown[1] = false;

break;

case Keys.I:

keysDown[2] = false;

break;

case Keys.K:

keysDown[3] = false;

break;

case Keys.J:

keysDown[4] = false;

break;

case Keys.D:

keysDown[5] = false;

break;

case Keys.A:

keysDown[6] = false;

break;

case Keys.L:

keysDown[7] = false;

break;

case Keys.M:

secondPlayer.ChangeAmmo();

break;

case Keys.X:

firstPlayer.ChangeAmmo();

break;

}

}

/// <summary>

/// Получение информации о первом игроке

/// </summary>

/// <returns>Информация о первом игроке</returns>

public string GetFirstPlayerInfo()

{

return firstPlayer.GetInfo();

}

/// <summary>

/// Получение информации о втором игркое

/// </summary>

/// <returns>Информация о втором игроке</returns>

public string GetSecondPlayerInfo()

{

return secondPlayer.GetInfo();

}

/// <summary>

/// Получение значения количества взрывов на экране

/// </summary>

/// <returns>Количество взрывов на экране</returns>

public int GetExplodesCount()

{

return explodes.Count;

}

}

}

**Листинг класса *Balloon*.*cs*:**

using System;

using System.Collections.Generic;

using OpenTK;

using System.Drawing;

using System.Diagnostics;

using GraphicsOpenGL;

using AmmoLibrary;

namespace GameLibrary

{

/// <summary>

/// Класс воздушного шара (игрока)

/// </summary>

public class Balloon

{

/// <summary>

/// Центр позиции шара

/// </summary>

public Vector2 PositionCenter;

/// <summary>

/// Скорость шара

/// </summary>

public Vector2 Speed;

/// <summary>

/// Спрайт воздушного шара

/// </summary>

public Texture BalloonSprite;

/// <summary>

/// Список с текущими снарядамиы игрока

/// </summary>

private List<Ammo> ammos;

/// <summary>

/// Показатель, отвечающий за то, какой сейчас снаряд выбран у игрока

/// </summary>

private int currentAmmo=0;

/// <summary>

/// Скорость ветра игрока

/// </summary>

private Vector2 windSpeed = new Vector2(0.0f, 0.0f);

/// <summary>

/// Показатель, отвечающий за то, работает ветер или нет (true - работает, false - нет)

/// </summary>

private bool isWindOn = false;

/// <summary>

/// Конструктор создания шара

/// </summary>

/// <param name="startPosition">Начальная позиция</param>

/// <param name="baloonSprite">Спрайт игрока</param>

public Balloon(Vector2 startPosition, Texture baloonSprite)

{

this.PositionCenter = startPosition;

this.BalloonSprite = baloonSprite;

this.Speed = new Vector2(0, -0.001f);

this.currentAmmo = 0;

this.ammos = new List<Ammo>()

{

new SupersonicAmmo(),

new PiercingAmmo(),

new ExplosiveAmmo(),

};

}

/// <summary>

/// Показатель брони

/// </summary>

public int Armour { get; set; } = 0;

/// <summary>

/// Показатель здоровья

/// </summary>

public int Health { get; set; } = 100;

/// <summary>

/// Показатель топлива

/// </summary>

public int Fuel { get; set; } = 1000;

/// <summary>

/// Проверка на то, равняется ли показатель здоровья игрока нулю

/// </summary>

/// <returns>true - показатель здоровья игрока выше нуля, false - показатель здоровья игрока равно нулю</returns>

public bool CheckAlive()

{

if (Health <= 0)

return false;

else

return true;

}

/// <summary>

/// Обновление позиции шара при его движении

/// </summary>

/// <param name="movement">Показатель, отвечающий за направление движение игрока</param>

public void Update(Vector2 movement)

{

if (Fuel <= 0)

return;

PositionCenter += movement;

Fuel--;

if (isWindOn)

PositionCenter += windSpeed;

}

/// <summary>

/// Обновление падения игрока вниз при отсутствии нажатия клавиш

/// </summary>

public void Update()

{

PositionCenter += Speed;

if(isWindOn)

PositionCenter += windSpeed;

}

/// <summary>

/// Получение урона при столкновении со снарядами

/// </summary>

public void GetDamage()

{

int damage = 15;

if(Armour>0)

{

if(Armour>damage)

{

Armour -= damage;

}

else

{

int remainder = damage - Armour;

Armour = 0;

Health -= remainder;

}

}

else

Health -= damage;

if (Health < 0)

Health = 0;

}

/// <summary>

/// Повышение показателя здоровья

/// </summary>

public void IncreaseHealth()

{

int extraHealth = 20;

Health += extraHealth;

if (Health > 100)

Health = 100;

}

/// <summary>

/// Повышение показателя брони

/// </summary>

public void IncreaseArmour()

{

int extraArmour = 20;

Armour += extraArmour;

if (Armour > 100)

Armour = 100;

}

/// <summary>

/// Повышение запасов топлива

/// </summary>

public void IncreaseFuel()

{

int extraFuel = 350;

Fuel += extraFuel;

if (Fuel > 1000)

Fuel = 1000;

}

/// <summary>

/// Изменение скорости ветра

/// </summary>

/// <param name="windSpeed">Новая скорость ветра</param>

public void ChangeWindSpeed(Vector2 windSpeed)

{

this.windSpeed = windSpeed;

}

/// <summary>

/// Изменение состояния ветра

/// </summary>

/// <param name="isWindOn">Состояние ветра (true - работает, false - не рабоатет)</param>

public void ChangeWindCondition(bool isWindOn)

{

this.isWindOn = isWindOn;

}

/// <summary>

/// Получение границ объекта игрока

/// </summary>

/// <returns>Коллайдер игрока</returns>

public RectangleF GetCollider()

{

Vector2[] colliderPosition = GetPosition();

colliderPosition[3].X += 0.02f; // делаем это для более точного коллайдера; т.к. модель вытянута, будем считать касание о шар дальше его крайней точки

colliderPosition[2].X -= 0.02f;

float colliderWidth = (colliderPosition[2].X - colliderPosition[3].X)/2.0f;

float colliderHeight = (colliderPosition[3].Y - colliderPosition[0].Y)/2.0f;

float[] convertedLeftTop = CoordinatesConverter.Convert(colliderPosition[3].X, colliderPosition[3].Y);

RectangleF collider = new RectangleF(convertedLeftTop[0], convertedLeftTop[1], colliderWidth, colliderHeight);

return collider;

}

/// <summary>

/// Отрисовка игрока

/// </summary>

/// <param name="isFlipped">Показатель, отвечающий за отражение по вертикали объекта (true - нужно отражать, false - не нужно)</param>

public void Draw(bool isFlipped)

{

ObjectDrawer.Draw(BalloonSprite, GetPosition(), isFlipped);

}

/// <summary>

/// Замена текущего снаряда

/// </summary>

public void ChangeAmmo()

{

currentAmmo++;

if (currentAmmo >= ammos.Count)

currentAmmo = 0;

}

/// <summary>

/// Получение текущего снаряда

/// </summary>

/// <param name="isLeft">Показатель, отвечающий за выпуск снаряда влево (true - выпускается влево, false - вправо)</param>

/// <returns>Текущий снаряд игрока</returns>

public Ammo GetCurrentAmmo(bool isLeft)

{

Ammo newAmmo = null;

ammos[currentAmmo].Spawn(PositionCenter-new Vector2(0.01f, 0.07f), isLeft);

switch(currentAmmo)

{

case 0:

newAmmo = new SupersonicAmmo(ammos[currentAmmo]);

break;

case 1:

newAmmo = new PiercingAmmo(ammos[currentAmmo]);

break;

case 2:

newAmmo = new ExplosiveAmmo(ammos[currentAmmo]);

break;

}

return newAmmo;

}

/// <summary>

/// Изменение характеристик снарядов с помощью "декоратора"

/// </summary>

public void ChangeAmmoCharesterictics()

{

Random random = new Random();

int decoratorType = random.Next(0, 3);

switch(decoratorType)

{

case 0:

for (int i = 0; i < ammos.Count; i++)

{

ammos[i] = new DistanceDecorator(ammos[i]);

Debug.WriteLine("Distance Decorator");

}

break;

case 1:

for (int i = 0; i < ammos.Count; i++)

{

ammos[i] = new RadiusDecorator(ammos[i]);

Debug.WriteLine("Radius Decorator");

}

break;

case 2:

for (int i = 0; i < ammos.Count; i++)

{

ammos[i] = new SpeedDecorator(ammos[i]);

Debug.WriteLine("Speed Decorator");

}

break;

}

}

/// <summary>

/// Получение текущей позиции игрока

/// </summary>

/// <returns>Массив из четырёх элементов, содержащий края позиции игрока</returns>

public Vector2[] GetPosition()

{

float spriteWidth = 0.07f;

float spriteHeight = 0.14f;

return new Vector2[4]

{

PositionCenter + new Vector2(-spriteWidth, -spriteHeight),

PositionCenter + new Vector2(spriteWidth, -spriteHeight),

PositionCenter + new Vector2(spriteWidth, spriteHeight),

PositionCenter + new Vector2(-spriteWidth, spriteHeight),

};

}

/// <summary>

/// Получение информации об игроке

/// </summary>

/// <returns>Информация об игроке</returns>

public string GetInfo()

{

return $"Health = {Health}, Armour = {Armour}, Fuel = {Fuel}, Radius = {100\*ammos[currentAmmo].Radius}, Distance = {100\*ammos[currentAmmo].Distance}, Speed = {100\*ammos[currentAmmo].Speed.X}";

}

}

}

**Листинг класса *Explode*.*cs*:**

using OpenTK;

using System.Collections.Generic;

using GraphicsOpenGL;

namespace GameLibrary

{

/// <summary>

/// Взрыв после столкновения коллайдеров снарядов или игроков

/// </summary>

public class Explode

{

/// <summary>

/// Список текстур для анимации взрыва

/// </summary>

List<Texture> animation;

/// <summary>

/// Позиция для отрисовки взрыва

/// </summary>

Vector2[] position;

/// <summary>

/// Номер анимации в списке

/// </summary>

public int Count;

/// <summary>

/// Конструктор создания взрыва

/// </summary>

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

public Explode(Vector2[] position)

{

this.position = position;

this.Count = 0;

this.animation = new List<Texture>()

{

TextureLoader.LoadTexure("Animation/1.png"),

TextureLoader.LoadTexure("Animation/2.png"),

TextureLoader.LoadTexure("Animation/3.png"),

TextureLoader.LoadTexure("Animation/4.png"),

TextureLoader.LoadTexure("Animation/5.png"),

};

}

/// <summary>

/// Отрисовка взрыва

/// </summary>

/// <param name="isFlipped">Значение, отвечающее за то, следует ли отразить текстуру объекта или нет

/// (true - текстура отражается, false - не отражается</param>

/// <returns>Показатель, отвечающий за то, закончилась ли анимация или нет

/// (true - не закончилась, false - закончилась)</returns>

public bool Draw(bool isFlipped)

{

if(Count >= animation.Count)

{

return false;

}

ObjectDrawer.Draw(animation[Count], position, isFlipped);

Count++;

return true;

}

}

}

**Листинг класса *ArmourTest*.*cs*:**

using GameLibrary;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using OpenTK.Graphics;

using OpenTK;

namespace UnitTests

{

/// <summary>

/// Тесты для проверки добавления пунктов брони

/// </summary>

[TestClass]

public class ArmourTest

{

/// <summary>

/// Тест для проверки корректности выполнения добавления пунктов брони

/// </summary>

[TestMethod]

public void GetArmourTestMethod()

{

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

game.LoadGLControl();

Balloon balloon = new Balloon(Vector2.Zero, null);

int expectedArmour = 20;

int actualArmour;

// Act

balloon.IncreaseArmour();

actualArmour = balloon.Armour;

// Assert

Assert.AreEqual(expectedArmour, actualArmour);

}

}

}

**Листинг класса *CheckAliveTest*.*cs*:**

using GameLibrary;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using OpenTK.Graphics;

using OpenTK;

namespace UnitTests

{

/// <summary>

/// Проверка на корректность выполнения метода CheckAlive у игрока

/// </summary>

[TestClass]

public class CheckAliveTest

{

/// <summary>

/// Метод для проверки корректности выполнения CheckAlive при значении здоровья равным нулю

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void CheckAliveTestMethod1()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

game.LoadGLControl();

Balloon balloon = new Balloon(Vector2.Zero, null);

balloon.Health = 0;

bool expectedResult = false;

bool actualResult;

// Act

actualResult = balloon.CheckAlive();

// Assert

Assert.AreEqual(expectedResult, actualResult);

}

/// <summary>

/// Метод для проверки корректности выполнения CheckAlive при значении здоровья больше нуля

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void IncreaseHealthTestMethod2()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

game.LoadGLControl();

Balloon balloon = new Balloon(Vector2.Zero, null);

balloon.Health = 2;

bool expectedResult = true;

bool actualResult;

// Act

actualResult = balloon.CheckAlive();

// Assert

Assert.AreEqual(expectedResult, actualResult);

}

}

}

**Листинг класса *ConverterTest*.*cs*:**

using Microsoft.VisualStudio.TestTools.UnitTesting;

using GraphicsOpenGL;

namespace UnitTests

{

/// <summary>

/// Класс для проверки корректности преобразования координат из OpenGL в WinForms

/// </summary>

[TestClass]

public class ConverterTest

{

/// <summary>

///

/// </summary>

/// <param name="pointX">Координата точки по X</param>

/// <param name="pointY">Координата точки по Y</param>

/// <param name="expected">Ожидаемое значение точки</param>

[DataTestMethod]

[DataRow(0.2f, -0.4f, new float[] { 0.6f, 0.7f})]

[DataRow(0.0f, 0.0f, new float[] { 0.5f, 0.5f })]

[DataRow(-0.2f, -0.3f, new float[] { 0.4f, 0.65f })]

[DataRow(1.0f, 0.5f, new float[] { 1.0f, 0.25f })]

[DataRow(-0.8f, 0.4f, new float[] { 0.1f, 0.3f })]

public void ConverterTestMethod(float pointX, float pointY, float[] expected)

{

// Arrange

float[] actual;

// Act

actual = CoordinatesConverter.Convert(pointX, pointY);

// Assert

Assert.AreEqual(expected[0], actual[0]);

Assert.AreEqual(expected[1], actual[1]);

}

}

}

**Листинг класса *DamageTest*.*cs*:**

using GameLibrary;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using OpenTK;

using OpenTK.Graphics;

namespace UnitTests

{

/// <summary>

/// Класс для проверки корректности выполнения метода GetDamage у игрока

/// </summary>

[TestClass]

public class DamageTest

{

/// <summary>

/// Проверка корректности выполнения метода GetDamage без наличия брони у игрока

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void GetDamageTestMethod()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

game.LoadGLControl();

Balloon balloon = new Balloon(Vector2.Zero, null);

int expectedHealth = 85;

int actualHealth;

// Act

balloon.GetDamage();

actualHealth = balloon.Health;

// Assert

Assert.AreEqual(expectedHealth, actualHealth);

}

/// <summary>

/// Проверка корректности выполнения метода GetDamage с наличием брони у игрока

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void GetDamageWithArmourTestMethod()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

game.LoadGLControl();

Balloon balloon = new Balloon(Vector2.Zero, null); // отрисовка и позиция для проверки нам не требуется

int expectedHealth = 90;

int expectedArmour = 0;

int actualHealth;

int actualArmour;

// Act

balloon.IncreaseArmour();

balloon.GetDamage();

balloon.GetDamage();

actualHealth = balloon.Health;

actualArmour = balloon.Armour;

// Assert

Assert.AreEqual(expectedHealth, actualHealth);

Assert.AreEqual(expectedArmour, actualArmour);

}

}

}

**Листинг класса *DecoratorsTest*.*cs*:**

using GameLibrary;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using OpenTK.Graphics;

using OpenTK;

using System;

using AmmoLibrary;

namespace UnitTests

{

/// <summary>

/// Класс для проверки корректности функционирования декораторов

/// </summary>

[TestClass]

public class DecoratorsTest

{

/// <summary>

/// Проверка корректности выполнения декоратора на увеличение дистанции полёта снаряда

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void DistanceDecoratorTestMethod()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

Ammo ammo;

game.LoadGLControl();

ammo = new ExplosiveAmmo();

float actualDistance;

float expectedDistance=1.5f;

// Act

ammo = new DistanceDecorator(ammo);

actualDistance = ammo.Distance;

// Assert

Assert.AreEqual(actualDistance, expectedDistance);

}

/// <summary>

/// Проверка корректности выполнения декоратора на увеличение радиуса действия снаряда

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void RadiusDecoratorTestMethod()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

Ammo ammo;

game.LoadGLControl();

ammo = new SupersonicAmmo();

float actualRadius;

float expectedRadius = 0.06f;

// Act

ammo = new RadiusDecorator(ammo);

actualRadius = (float)Math.Round(ammo.Radius, 2); // округление из-за неточности float

// Assert

Assert.AreEqual(actualRadius, expectedRadius);

}

/// <summary>

/// Проверка корректности выполнения декоратора на увеличение скорости полёта снаряда

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void SpeedDecoratorTestMethod()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

Ammo ammo;

game.LoadGLControl();

ammo = new PiercingAmmo();

float actualSpeed;

float expectedSpeed = 0.0156f;

// Act

ammo = new SpeedDecorator(ammo);

actualSpeed = ammo.Speed.X;

// Assert

Assert.AreEqual(actualSpeed, expectedSpeed);

}

}

}

**Листинг класса *FuelTest*.*cs*:**

using GameLibrary;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using OpenTK.Graphics;

using OpenTK;

namespace UnitTests

{

/// <summary>

/// Проверка корректности выполнения метода IncreaseFuel у игрока

/// </summary>

[TestClass]

public class FuelTest

{

/// <summary>

/// Проверка корректности выполнения увеличения запасов топлива у игрока при значении ниже 700

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void IncreaseFuelTestMethod1()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

game.LoadGLControl();

Balloon balloon = new Balloon(Vector2.Zero, null);

balloon.Fuel = 200;

int expectedFuel = 550;

int actualFuel;

// Act

balloon.IncreaseFuel();

actualFuel = balloon.Fuel;

// Assert

Assert.AreEqual(expectedFuel, actualFuel);

}

/// <summary>

/// Проверка корректности выполнения увеличения запасов топлива у игрока при значении выше 700

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void IncreaseFuelTestMethod2()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

game.LoadGLControl();

Balloon balloon = new Balloon(Vector2.Zero, null);

balloon.Fuel = 800;

int expectedFuel = 1000;

int actualFuel;

// Act

balloon.IncreaseFuel();

actualFuel = balloon.Fuel;

// Assert

Assert.AreEqual(expectedFuel, actualFuel);

}

}

}

**Листинг класса *IncreaseHealthTest*.*cs*:**

using GameLibrary;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using OpenTK.Graphics;

using OpenTK;

namespace UnitTests

{

/// <summary>

/// Класс для тестирования увеличения здоровья у игрока

/// </summary>

[TestClass]

public class IncreaseHealthTest

{

/// <summary>

/// Проверка корректности выполнения увеличения запасов здоровья у игрока при текущем значении ниже 80

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void IncreaseHealthTestMethod1()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

game.LoadGLControl();

Balloon balloon = new Balloon(Vector2.Zero, null); // отрисовка и позиция для проверки нам не требуется

balloon.Health = 50;

int expectedHealth = 70;

int actualHealth;

// Act

balloon.IncreaseHealth();

actualHealth = balloon.Health;

// Assert

Assert.AreEqual(expectedHealth, actualHealth);

}

/// <summary>

/// Проверка корректности выполнения увеличения запасов здоровья у игрока при текущем значении выше 80

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void IncreaseHealthTestMethod2()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

game.LoadGLControl();

Balloon balloon = new Balloon(Vector2.Zero, null); // отрисовка и позиция для проверки нам не требуется

balloon.Health = 90;

int expectedHealth = 100;

int actualHealth;

// Act

balloon.IncreaseHealth();

actualHealth = balloon.Health;

// Assert

Assert.AreEqual(expectedHealth, actualHealth);

}

}

}

**Листинг класса *UpdateMethodTest*.*cs*:**

using GameLibrary;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using OpenTK.Graphics;

using OpenTK;

namespace UnitTests

{

/// <summary>

/// Класс для проверки корректности выполнения метода Update у игрока

/// </summary>

[TestClass]

public class UpdateMethodTest

{

/// <summary>

/// Проверка корректности выполнения уменьшения значения запасов топлива при вызове метода Update

/// (создаётся окно GameWindow для загрузки необходимых для выполнения OpenGL функций)

/// </summary>

[TestMethod]

public void UpdateTestMethod()

{

// Arrange

GameWindow window = new GameWindow(1, 1, GraphicsMode.Default, "", GameWindowFlags.Default, DisplayDevice.Default, 3, 0, GraphicsContextFlags.Default);

window.Visible = false;

BattleGame game = new BattleGame();

game.LoadGLControl();

Balloon balloon = new Balloon(Vector2.Zero, null);

int expectedFuel = 999;

int actualFuel;

// Act

balloon.Update(Vector2.Zero);

actualFuel = balloon.Fuel;

// Assert

Assert.AreEqual(expectedFuel, actualFuel);

}

}

}

**Листинг класса *Program*.*cs:***

using System;

using System.Windows.Forms;

namespace Ballon\_Battle

{

internal static class Program

{

/// <summary>

/// Главная точка входа для приложения.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new GameForm());

}

}

}

**ПРИЛОЖЕНИЕ Б**

(обязательное)

**Изображения интерфейса программы**

На рисунках Б.1-Б.4 продемонстрированы изображения интерфейса разработанного приложения «Битва на воздушных шарах».

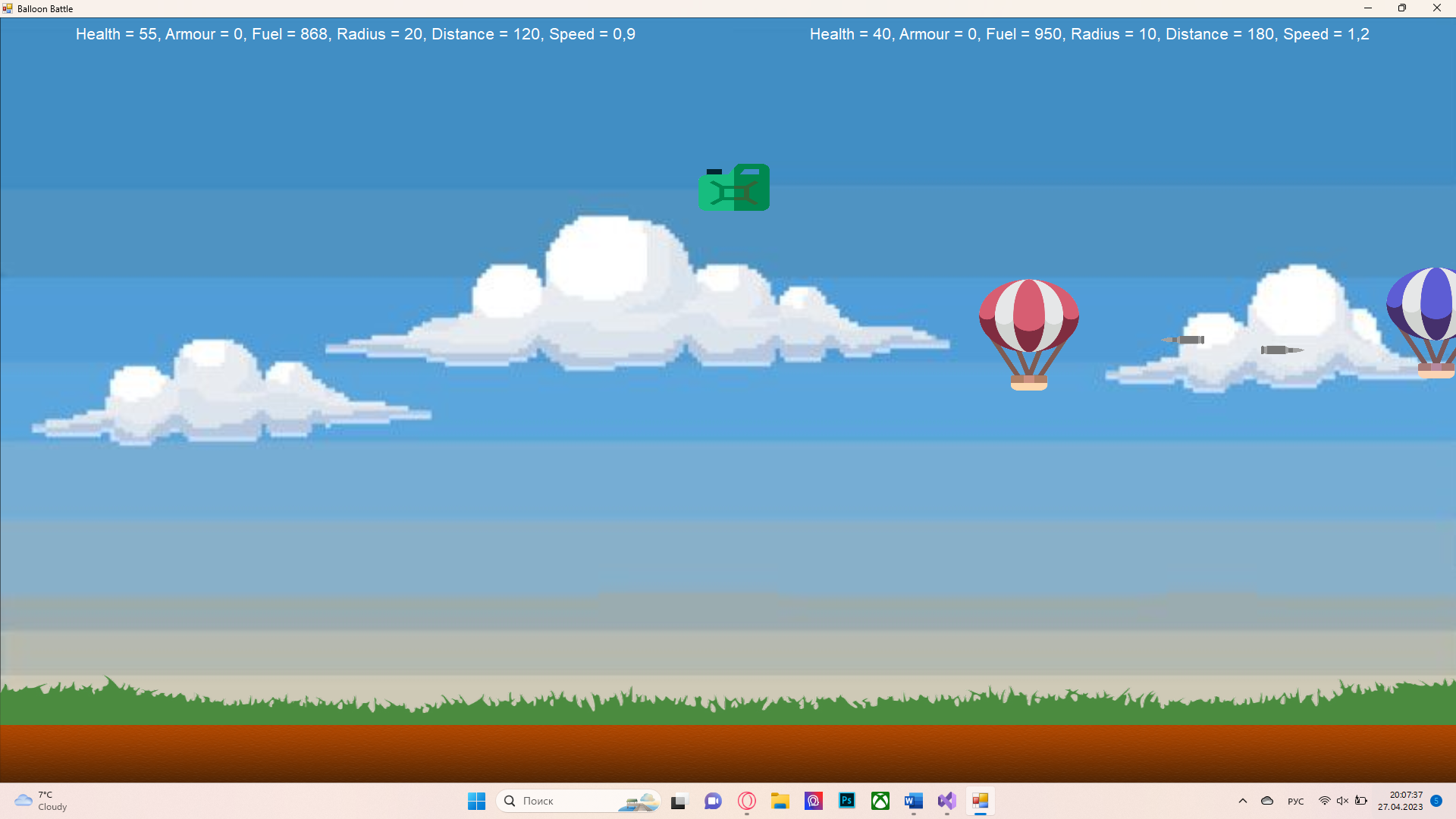


Рисунок Б.1 – Окно приложения в процессе игры

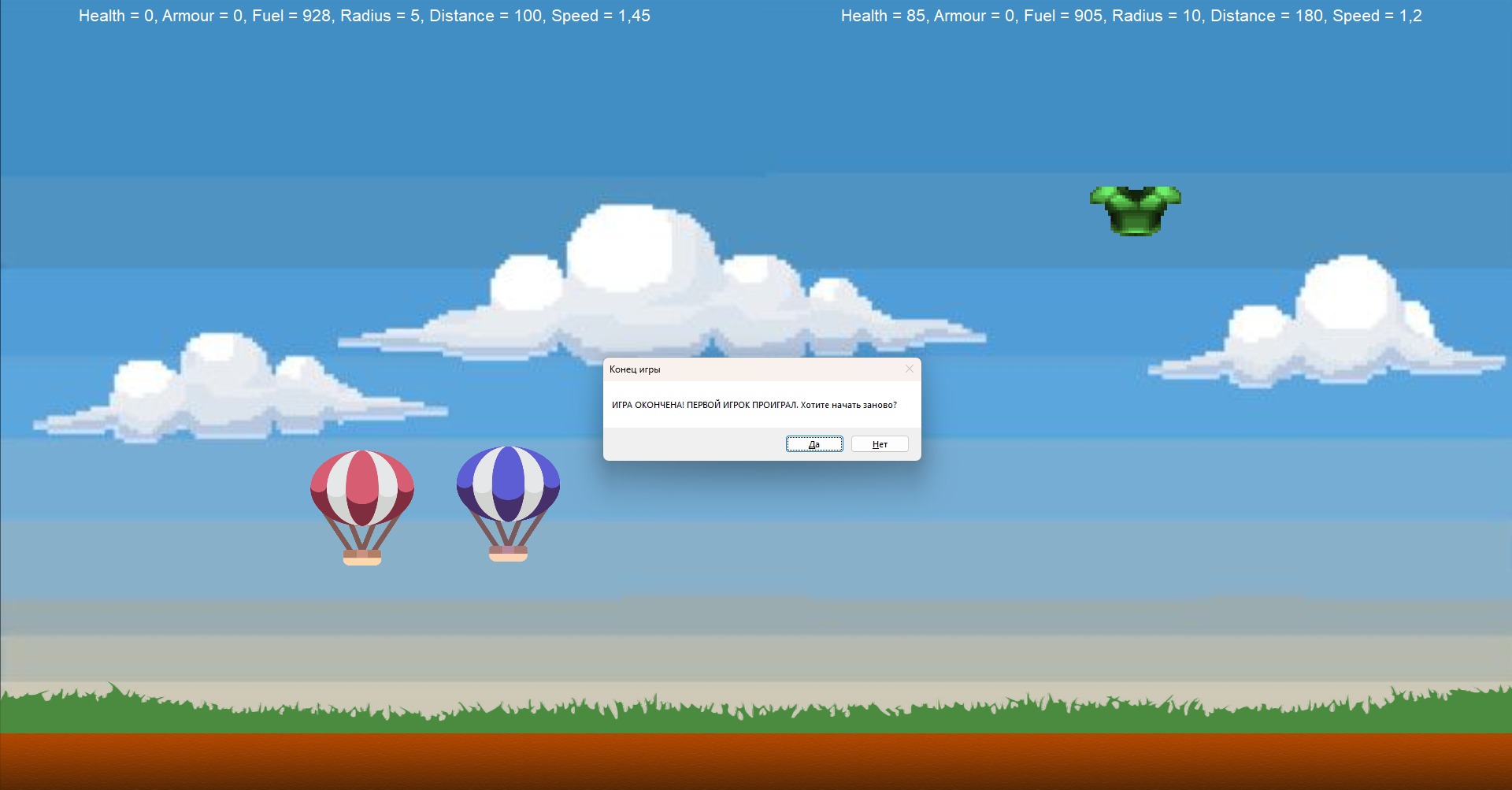


Рисунок Б.2 – Окно приложения после поражения первого игрока

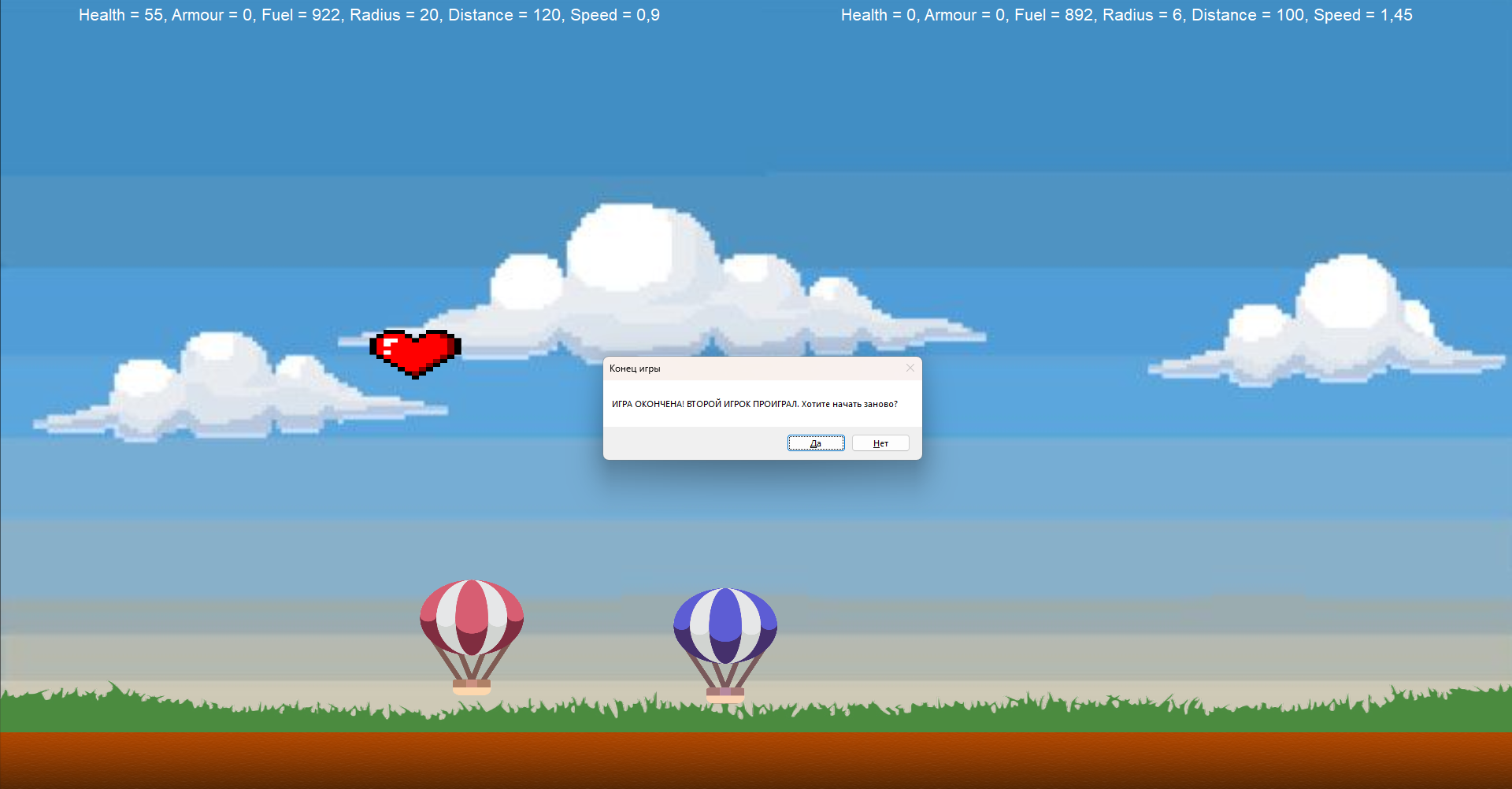


Рисунок Б.3 – Окно приложения после поражения второго игрока

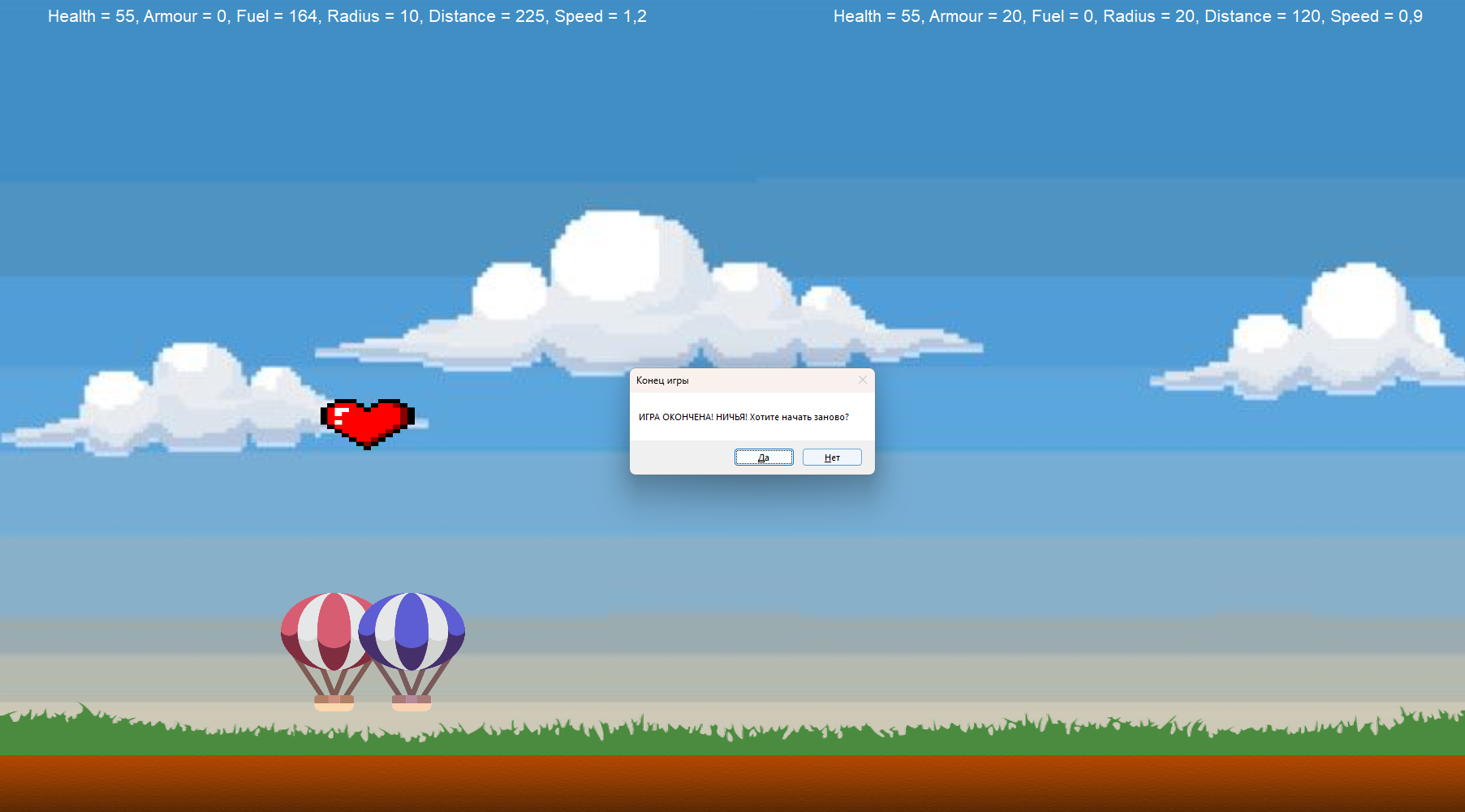


Рисунок Б.4 – Окно приложения после столкновения шаров

**ПРИЛОЖЕНИЕ В**

(обязательное)

**Инструкция пользователя**

1. Введение.

Разработанное приложение «Битва на воздушных шарах» представляет собой игру, в которую одновременно могут играть два игрока на одном компьютере.

Игра предназначена практически для любого пользователя, который обладает базовыми навыками владения компьютером.

1. Назначение и условия применения.

Данная игра разработана в первую очередь для развлечения, также она может развивать скорость реакции, внимательность, концентрацию и может способствовать развитию быстроты принятия решений.

Для запуска и корректной работы программы компьютер пользователя должен иметь определённую конфигурацию:

– операционная система *Windows*;

– наличие клавиатуры и монитора;

– не менее 150 Мбайт оперативной памяти.

3. Подготовка к работе.

Для запуска программы первоначально необходимо загрузить каталог *Release,* содержащий готовое приложение, на компьютер. Далее следует открыть файл *Balloon* *Battle*.*exe,* находящийся в каталоге игры. После этого программа будет запущена.

4. Описание операций

Два игрока могут управлять воздушным шаром в вертикальной плоскости клавишами *W*, *S* для первого игрока и клавишами *I*, *K* для второго игрока.

Также игроки могут выпускать снаряды в левую и правую сторону клавишами *A*, *D* (для первого игрока) и *J*, *L* (для второго игрока). Каждому игроку доступны три вида снарядов, которые можно переключить на кнопки *X* и *M* для первого и второго игрока соответственно.

5. Аварийные ситуации.

Если компьютер пользователя не соответствует определённым ранее минимальным системным требованиям, могут случиться непредвиденные сбои. Если программа не отвечает, то рекомендуется выключить её с помощью сочетания клавиш *Alt*+*F*4 или с помощью диспетчера задач.

6. Рекомендации по освоению.

Для успешного освоения и эксплуатации разработанного программного обеспечения необходимо лишь обладание базовыми навыками владения компьютером.

**ПРИЛОЖЕНИЕ Г**

(обязательное)

**Руководство программиста**

1. Назначение и условия применения программы.

Разработанное приложение «Битва на воздушных шарах» предназначено для игры двух игроков на одном экране. Программа представляет собой игру с двумерной графикой. Приложение разработано для развлечения, однако может способствовать развитию скорости реакции и внимательности.

Условия, необходимые для корректной работы программы:

– компьютер с операционной системой *Windows*;

– наличие клавиатуры и монитора;

– не менее 150 Мбайт оперативной памяти.

1. Характеристики программы.

Приложение имеет один режим работы. При открытии программы запускается игровая сцена, и начинается игра. Двум игрокам доступно управление двумя воздушными шарами. После завершения игры пользователям предлагается либо сыграть вновь, либо закрыть приложение.

1. Обращение к программе.

Вызов программы выполняется после запуска файла *Balloon* *Battle*.*exe* из каталога *bin*/*Release*, находящийся в основном каталоге приложения.

1. Входные и выходные данные.

В качестве входной информации программа принимает нажатия клавиш пользователями для движения (*W*, *S*, *I* и *K*), выпуска снарядов (*A*, *D*, *J* и *L*) и смены типа снарядов (*X* и *M*).

Выходными данными программы является отображение двух воздушных шаров, выпускаемых снарядов, призов и других объектов игровой сцены на экран. Это выполняется с помощью окна *Windows* *Forms* и графики *OpenGL.*

1. Сообщения.

При проигрыше игрока, которое происходит при его столкновении с землёй или достижении нуля значения пунктов здоровья воздушного шара, появляется сообщение о том, какой из игроков проиграл. Также пользователям предлагается либо продолжить игру нажатием на кнопку «Да», либо закрыть приложение кнопкой «Нет». Аналогичное сообщение о ничьей появляется после столкновения двух воздушных шаров.

**ПРИЛОЖЕНИЕ Д**

(обязательное)

**Руководство системного программиста**

1. Общие сведения о программе.

Разработанное приложение «Битва на воздушных шарах» представляет собой игру, в которую одновременно могут играть два игрока на одном компьютере. Игроки могут управлять движением воздушных шаров вверх или вниз, а также выпускать три вида снарядов влево и вправо. Игра завершается при окончании у одного из игроков пунктов здоровья, либо при столкновении шара с землёй, либо при столкновении двух шаров.

Минимальными системными требованиями приложения являются:

– компьютер с операционной системой *Windows*;

– наличие клавиатуры и монитора;

– не менее 150 Мбайт оперативной памяти.

1. Структура программы.

Программа разделена на несколько частей в зависимости от представляемого функционала: создание графики, логика игры и отображение созданной графики на экран.

Приложение содержит библиотеки, где содержатся все объекты, используемые в игре, а также прописана логика игры. При проектировании приложения использовались шаблоны «фабричный метод» для генерации призов и «декоратор» для улучшения характеристик снарядов.

Также отдельной частью приложения является отображение графики, которое было реализовано благодаря средствам *OpenGL* с использованием библиотеки *OpenTK*. Вывод созданной графики отображался благодаря окнам *Windows* *Forms*.

1. Настройка программы.

Для запуска приложения необходимо запустить файл *Balloon* *Battle*.*exe* из каталога *bin*/*Release*. Дальнейшая настройка программы не требуется.

1. Проверка программы.

Для проверки работоспособности приложения было выполнено функциональное тестирования с целью проверки соответствия конечного продукта первоначальным требованиям, определённым при создании приложения. Все тесты показали ожидаемые результаты и были успешно пройдены.

Также было проведено модульное тестирования благодаря функционалу языка программирования *C*# и среды разработки *Visual* *Studio*. В процессе данного тестирования была проведена проверка всех основных методов, связанных с логикой игрового приложения.

1. Сообщения системному программисту.

В процессе выполнения программы при завершении игры посредством окончания запасов здоровья игроков или их столкновении друг с другом или землёй, на экран выводится сообщение о результате данной игры, а также предложение повторной игры. При нажатии на кнопку «Да» приложение будет запущено вновь, при нажатии на кнопку «Нет» приложение будет закрыто.

**ПРИЛОЖЕНИЕ Е**

(обязательное)

**Схема использования паттерна «декоратор»**