МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования

«Гомельский государственный технический университет

имени П.О.Сухого»

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

направление специальности 1-40 05 01-12 Информационные системы и технологии (в игровой индустрии)

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту

по дисциплине «Объектно-ориентированное программирование»

на тему: «Игра «Пляжный волейбол», реализованная средствами WPF с использованием графики OpenGL»

Исполнитель: студент группы ИТИ-22

Алексеенко Е.Н.

Руководитель: зав. кафедрой ИТ

Курочка К.С.

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

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

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

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

Подписи членов комиссии

по защите курсового проекта: \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Гомель 2021

**Оглавление**

ВВЕДЕНИЕ...……………………………………………………………….5

1. Средства разработки игровых приложений……………………….…..6
   1. Графическая библиотека DirectX…………………………………..6
   2. Графическая библиотека OpenGL………………………………….7
   3. Преимущества и недостатки графических библиотек OpenGL и DirectX……………………………………………………………….8
   4. Основные понятия…………………………………………………..8
   5. Разработка компьютерных игр в настоящее время………………10
   6. Положение современного аркадного направления………………11
2. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ..……………….15
   1. Описание условий для решения задачи…………………………..15
   2. Подготовка исходных данных…………………………………….15
   3. Проектирование структуры ПО………………………………...…16
   4. Описание работы классов………………………………………....19
3. РЕАЛИЗАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ «ВОЛЕЙБОЛ».…….23
   1. Обзор готовой программы………………………………………...23
   2. Верификация игрового приложения……………………………...23
   3. Пользовательский интерфейс…………………………………..…25

ЗАКЛЮЧЕНИЕ…………………………………………………………...28

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ29

ПРИЛОЖЕНИЕ А30

ПРИЛОЖЕНИЕ Б56

ПРИЛОЖЕНИЕ В57

ПРИЛОЖЕНИЕ Г58

ПРИЛОЖЕНИЕ Д59

**ВВЕДЕНИЕ**

Во все времена люди любили игры и развлечения: такие простые как до­гонялки и сложные как шахматы. В наше время было придумано и реализо­вано бесчисленное множество разных игр. И теперь каждый человек может найти себе занятие по вкусу.

Стоит отметить, что существуют игры не только развлекательные, но и обучающие. К примеру: ребенку легче объяснить азы математики в формате игры, так как это больше увлечет и покажет наглядно, чем просто цифрами на бумаге. Так же с помощью компьютерных игр можно реализовать симуляции более сложных процессов, что позволяет сформировать теоретические сведе­ния.

Сейчас игры стали самым популярным развлечением. Это связано с их разнообразностью и доступностью. Они доступны не только на персональных компьютерах, но и на телефонах, консолях, автоматах, телевизорах и так да­лее. В игру можно играть как одному, так и в компании, но каждый найдет в ней что-то, что ему нравится.

Из-за возросшей популярности игр стали появляться новые профессии, которые хорошо оплачиваются. Но при этом это сложная и кропотливая ра­бота, которая занимает не один месяц.

Основная цель данного курсового проекта – разработка аркадного при­ложения «Пляжный волейбол» с использованием графики OpenGL. Само при­ложение не является обучающим, но при этом тренирует внимание и реакцию.

1. **СРЕДСТВА РАЗРАБОТКИ ПРИЛОЖЕНИЙ**

**1.1 Графическая библиотека DirectX**

DirectX – мощный программный интерфейс, который позволяет разра­ботчикам компьютерной графики писать быстродействующие приложения, используя стандартный набор интерфейсов. Игры, поддерживающие DirectX, используют встроенные в оборудование функции акселерации мультимедиа более эффективно, что позволяет многократно повысить производительность выполнения мультимедийных задач.

Первая версия DirectX 9, то есть первый выпуск управляемого DirectX, произошел 20 декабря 2002 года, но при этом остался в разработке еще на ка­кое-то время. Вначале предполагалось, что Управляемый DirectX будет столь же мощным, как и API. Ранее пользователи DirectX для Visual Basic не мог­ли создавать приложения, сравнимые по качеству с приложениями DirectX API. Частично это было связано непосредственно со временем выполне­ния проце­дур на Visual Basic, а частично с нехваткой ресурсов библиотек DLL.

После выхода DirectX 8 оказалось, что интерфейс API неудобен в ис­пользовании. Разработчики, использующие Visual Basic, посчитали API слиш­ком сложным, а разработчики на С++ не видели выгоды переходить на API, так как для них это было неудобно и невыгодно. Какое-то время это интересно только домашним разработчикам, а студии по созданию компьютерных игр об этом даже не знали. Поэтому работа была направлена на достижение быстро­действие и удобство использования, что повлияло на переход из С++ кода на С#. На рисунке 1.1 изображена конвейерная архитектура DirectX:

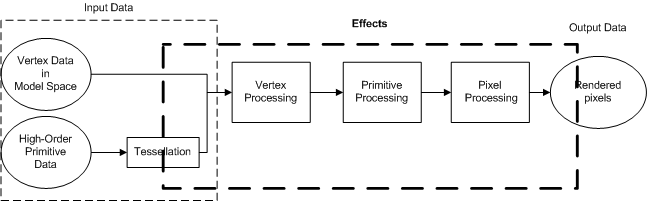


Рисунок 1.1 – Конвейерная архитектура DirectX

Сам DirectX был создан компанией Microsoft для операционной системы Windows. Также эта технология используется на игровых консолях Microsoft Xbox, но для других платформ она недоступна.

Во время работы с приложениями, созданными с использованием DirectX, применяются указатели на интерфейсы объектов. Работа с объектами осуществляется благодаря вызову различных методов его интерфейса. Созда­ние объектов происходил посредством вызова методов интерфейса кон­текстного окна [1, c.287].

**1.2 Графическая библиотека OpenGL**

OpenGL – спецификация, созданная для приложений, использующих двумерную и трехмерную графику. Ее основные области применения – видео­игры, поддержка виртуальной реальности, системы автоматизи- рованного проектирования и визуализация различных проектов.

Концепция OpenGL была разработана в 1982 году в университете Стэн­форда. Прототип технологии впервые реализовала компания Silicon Graphics, создавшая конвейер для рендеринга. Ее разработки стали основой библиотек OpenGL. В отличии от DirectX, OpenGL доступен на всех платформах, так же является основополагающим компонентом при взаимодействии с GPU в опе­рационных системах Mac и Linux. На рисунке 1.2 представлена графическая схема конвейерной архитектуры OpenGL:

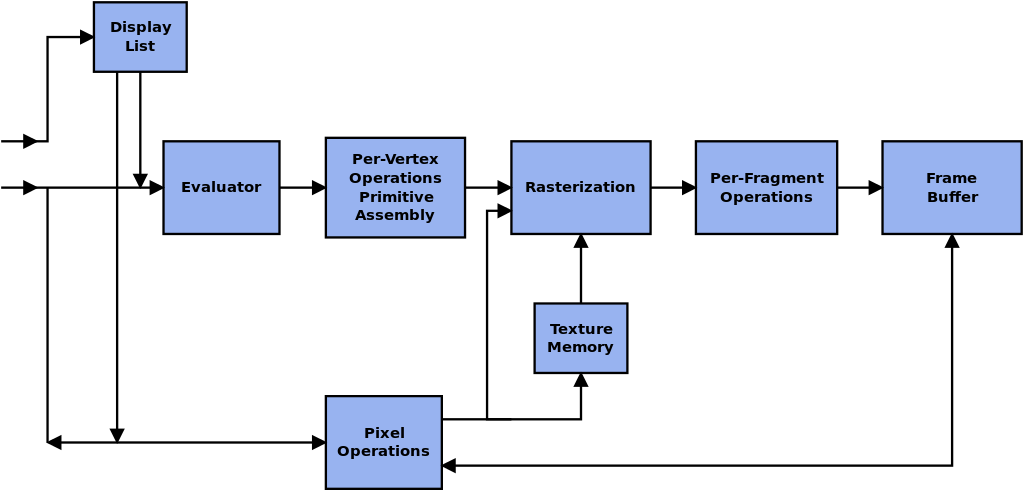


Рисунок 1.2 – Конвейерная архитектура OpenGL

В настоящее время язык GLSL является фундаментальной и неотъемлемой

частью OpenGL API. Любая программа, написанная с привлечением OpenGL API, внутри использует одну или более программ на языке GLSL. Такие «мини-программы» часто называют шейдерными программами. Обычно шей­дерная программа состоит из нескольких компонентов, называемых шейде­рами. Каждый шейдер выполняется в рамках отдельного этапа в общем кон­вейере OpenGL. Каждый шейдер выполняется на GPU и, как можно заключить из названия1, обычно реализует алгоритм, так или иначе связанный с эффектами освещения и затенения в изображении. Од­нако шейдеры способны на большее, чем просто воспроизводить эффекты освещения и затенения.

## 1.3 Преимущества и недостатки графических библиотек *OpenGL* и *DirectX*

Для выбора наиболее подходящей библиотеки для решения задачи необ­ходимо рассмотреть их достоинства и недостатки.

Преимущества DirectX:

– библиотека способна работать с двумерной и трехмерной графикой;

– поставляется с драйверами для операционной системы Windows;

– компания Microsoft постоянно обновляет и совершенствует API.

Также DirectX имеет ряд недостатков:

– используется исключительно на операционной системе Windows и консолях Xbox;

– не является библиотекой высокого уровня, так как предоставляет пользова­телю только средства для работы с графикой;

– для того, чтобы использовать все возможности новой версии, возможно при­дется обновить код программы.

По аналогии следует рассмотреть достоинства OpenGL:

– библиотека также предоставляет возможность использовать двумерную и трехмерную графику;

– может быть использована на любой платформе;

– основная спецификация постоянно обновляется;

– библиотека предоставляет возможность для использования современного функционала с помощью специальных расширений, что позволяет не ждать включения их в основную спецификацию.

Несмотря на все преимущества, OpenGL имеет и ряд недостатков:

– драйвер необходимо установить самостоятельно с сайта;

– также не является библиотекой высокого уровня.

Рассмотрев все достоинства и недостатки обеих библиотек, можно ска­зать, что для разработки приложения библиотека OpenGL подходит больше, чем DirectX. После того, как было выбрано средство для разработки игрового графического приложения, следует рассмотреть игровой жанр, которому бу­дет соответствовать разработанное приложение.

**1.4 Основные понятия**

Для начала необходимо сказать, что такое игра.

Игра – деятельность с правилами. Форма забавы, не всегда, но часто включающая в себя либо конфликт с другими персонажами, либо с самой иг­ровой системой, либо со случайностью, удачей или судьбой.

Видеоигра – игра с использованием цифровой графики и устройства, на котором она реализована.

Игры можно классифицировать по-разному. Рассмотрим классификацию по жанрам:

* platformers – это игры, где герой путешествует по миру переходя с плат­формы на платформу, чаще всего они выполнены в двумерном пространстве с видом сбоку, так же эти игры часто называют бродилками;
* RPG – Role Playing Game – ролевые игры, где игрок отыгрывает роль какого-либо персонажа, в таких играх события развиваются ход за ходом;
* simulator – игры, главная цель которых сымитировать что-то;
* fighting – игры, где игрок вступает в схватку с виртуальным противником, который управляется либо человеком, либо компьютером;
* arcade – игры для игровых автоматов, которые имеют интенсивный игровой процесс за короткое время.

Рассмотрим получше жанр «аркада». Изначально такие игры были напи­саны для автоматов, из-за чего они должны быть динамичными и с простым сюжетом. Такие автоматы были популярны до начала 2000-х. но потребность в них сокращалась из-за появления персональных компьютеров. Но сам по себе жанр не исчез и перешел на компьютеры. Помимо перевода старых игр с автоматов, появлялись и новые. При этом их особенности – графика и интер­фейс остались неизменными. Зал с аркадными игровыми автоматами изобра­жен на рисунке 1.1.



Рисунок 1.4 – Зал аркадных автоматов

В настоящем игры занимают больше место в жизни людей. С каждым годом интерес человечества к играм только растёт так как это один из самых доступных видов развлечения. За последнее время индустрия игр развилась настолько, что понемногу обгоняет шоубизнес. В наше время индустрия игр стала очень прибыльной. Но не стоит забывать, что это задумывалось все не как стабильный заработок, а в первую очередь развлечения. Также с каждым годом она будет только расти и развиваться.

## 1.5 Разработка компьютерных игр в настоящее время

В условиях пандемии все больше людей развлекают себя компьютер­ными играми, что способствует развитию этой сферы. За последнее время ин­терфейс игр очень поменялся. Из простейшего геймплей стал похож эволюци­онировал во что-то невероятное.

Рассмотрим игру Pac-Man. Она была разработана японской компанией Namco в 1980 году. Игрок управляет смайлом, который передвигается по ла­биринту. Цель игры – собрать все точки избегая встречи с приведениями. С повышением уровня лабиринт меняется, повышая при этом сложность. Всего в игре было 255 уровней. В отличие от родины видеоигры Японии, в Америка приняла игру с большим интересом. Игра привлекла внимание отсутствием насилия, что сделало ее культовой. На рисунке 1.5 представлен фрагмент этой игры.



Рисунок 1.5 – Интерфейс игры Pac-Man

Современные технологии вывели графику видеоигр на совершенно иной уровень. Сейчас не так просто отличить видеосъемку от графики. Как пример современной игры рассмотрим Genshin Impact.

Genshin Impact – [видеоигра](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%B8%D0%B3%D1%80%D0%B0) в жанре экшн с [открытым миром](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BA%D1%80%D1%8B%D1%82%D1%8B%D0%B9_%D0%BC%D0%B8%D1%80) и элемен­тами [RPG](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%80%D0%BE%D0%BB%D0%B5%D0%B2%D0%B0%D1%8F_%D0%B8%D0%B3%D1%80%D0%B0), разработанная китайской компанией [miHoYo Limited](https://ru.wikipedia.org/w/index.php?title=MiHoYo_Limited&action=edit&redlink=1) в 2019 году. Действие происходит в мире Тэйват, в котором игроку предстоит отыскать своего брата или сестру в зависимости от выбора персонажа. Игра прослави­лась благодаря своему сюжету и красивой графике. На рисунке 1.6 представ­лен фрагмент игры Genshin Impact.



Рисунок 1.6 – Интерфейс игры Genshin Impact

Но не смотря на значительное превосходство современных технологий, многие люди остаются преданными жанру аркадных игр и их графике. В про­цессе исследования игровых приложений и средств для их разработки было решено, что жанр «аркадная игра» станет основным для разрабатываемого приложения, а в качестве средства обработки графики была выбрана графиче­ская библиотека OpenGL.

**1.6 Положение современного аркадного направления**

Существуют игровые приложения, которые значительно повлияли на всю игровую индустрию. В большинстве своём они подтвердили теорию об актуальности современного рынка компьютерных и мобильных развлечений. Яркими примерами для обзора популярнейшего мобильного гейминга можно назвать следующие игры:

Doodle Jump (2009 г.) – игра-платформер, которая стала прорывом для мобильных платформ. Основная концепция игры заключается в том, что необходимо помочь персонажу подняться как можно выше, балансировать между платформами. Успех столь примитивной идеи заключается в необычной на то время графике и понятном исполнении, вкупе с более чем простым управлением. Оригинальный стиль интерфейса заложил будущий успех приложения.

Subway Surfers (2012 г.) – главным героем является хулиган, которого на месте преступления обнаруживает охранник с собакой. Персонажу необходимо бежать, справляться с преградами на пути, параллельно собирая всяческие бонусы, помогающие в прохождении некоторых участков, сборе монет, за которые затем могут быть куплены другие бонусы или интересные костюмы для персонажа, а также выбирать наиболее оптимальный путь побега, дабы не заманить себя в ловушку.



Рисунок 1.7 – Скриншоты игр Doodle Jump и Subway Surfers

Также очень много аркадных игр строятся именно на спорте. Такие игры были популярны все время и вряд ли когда-нибудь утратят свою популярность. Связано это с тем, что люди всегда интересовались спортом. Такие игры позволяют получить желаемые ощущения, не выходя из дома, например, из-за плохой погоды. Сами по себе они несложные в понимании и игровых механиках. Яркими примерами таких игр в наше время являются такие разросшиеся брэнды как FIFA (разновидность футбол европейского), NBA (баскетбол), NHL (хоккей), MADDEN (разновидность американского футбола) и др. В основе данного жанра видеоигр лежит имитация практики спорта. В играх, упомянутых ранее, делается акцент на реальных занятиях спортом, но существуют также и игры несколько другого плана, в которых акцент ставится на стратегии и спортивном менеджменте (например, Football Manager и Out of the Park Baseball).



Рисунок 1.8 – Скриншот игры FIFA 17

Спортивные игры включают в себя физические и тактические задачи и проверяют точность и аккуратность игрока. Большинство спортивных игр пытаются смоделировать спортивные характеристики, необходимые для этого вида спорта, включая скорость, силу, ускорение, точность и так далее. Как и в случае с соответствующими видами спорта, эти игры проходят на стадионе или арене с четкими границами. Спортивные игры часто содержат пошаговые и цветные комментарии с использованием записанного звука. В спортивных играх иногда используются разные режимы для разных частей игры. Это особенно верно в играх об американском футболе, таких как серия Madden NFL, где для выполнения паса требуется шесть различных режимов игры в течение примерно 45 секунд. Иногда другие спортивные игры предлагают меню, в котором игроки могут выбрать стратегию, пока игра временно приостановлена.

Футбольные видеоигры ассоциации иногда меняют режимы игры, когда игроку пора выполнить пенальти, свободный удар по воротам с пенальти, сделанный одним игроком. Некоторые спортивные игры также требуют от игроков смены ролей между спортсменами и тренером или менеджером. Эти переключатели режимов более интуитивно понятны, чем другие игровые жанры, потому что они отражают реальный спорт.

В более старых двухмерных спортивных играх иногда использовался нереалистичный графический масштаб, когда спортсмены казались довольно крупными, чтобы игрок мог их видеть. По мере развития спортивных игр игроки привыкли ожидать реалистичного графического масштаба с высокой степенью правдоподобия.

Также рассмотрим еще одну спортивную игру, но сделанную по подобию волейбола. Volleyball Fever Flat – это игра в жанре экшен и аркада, разрабатываемая Neuston AB для платформы PC. Окружение в игре относится к cтилистике спорт, а выделить особенности, можно следующие: экшен, казуальная игра, спортивная игра, аркада. Так же в ней игроку доступны такие игровые режимы как «Для одного игрока». В таком режиме человек будет соревноваться с компьютером за победу. Интерфейс игры представляет собой популярную с то время графику игры Minecraft.

Во всем мире игра Volleyball Fever Flat распространяется по модели разовая покупка издателем Neuston AB. На данный момент стадия игры – запущена, а дата её выхода – 04.12.2019. Бесплатно скачать Volleyball Fever Flat, в том числе и через торрент, нельзя, поскольку игра распространяется по модели разовая покупка.



Рисунок 1.9 – Скриншот игры Volleyball Fever Flat

Таким образом, для выполнения курсового проекта была выбрана спортивная аркадная игра «Пляжный волейбол», так как в ней будут простые для понимания механики. Цель игры – забить противнику как можно больше мячей, не роняя его на своей стороне.

# **2 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ**

## 2.1 Описание условий для решения задачи

В процессе разработки игрового приложения «Волейбол» для игры на одном экране двух пользователей друг против друга необходимо реализовать все основные возможности игр жанра «Аркадная игра». Каждый игрок должен управлять человечком, который может перемещаться по экрану вправо и влево, а также может изменять угол отскока мяча. Каждый игрок должен оста­ваться на своей территории. Вначале игры каждый игрок получает количество очков, равное нулю, которое будет увеличиваться в течении игры. Также на карте должны появляться бонусы, которые увеличивают скорость передвиже­ния и силу удара.

Для реализации приложения необходимо использовать средства языка программирования C#, технологию Windows Presentation Foundation (WPF) и шаблон проектирования «декоратор» – для создания боеприпасов с различ­ными характеристиками.

Для отображения игровых объектов на поле необходимо использовать спрайтовую графику и средства OpenGL.

## 2.2 Подготовка исходных данных

В процессе разработки программного обеспечения использовалась среда разработки Visual Studio 2019. Для обеспечения возможности работы с функ­циями OpenGL в приложении WPF были загружены необходимые графиче­ские библиотеки. В связи с тем, что код программы пишется на языке C#, для отображения графики используются библиотеки OpenTK, позволяющая рабо­тать с OpenGL на языке C#, и GlWpfControl, позволяющая отображать графику OpenGL в специальном компоненте WPF. Библиотека OpenTK содержит реа­лизацию всех основных функций OpenGL, а также содержит ряд типов и пере­числений, требуемых для работы с графикой. Библиотека GlWpfControl содер­жит компонент для WPF-приложения, который способен отображать в себе графику, созданную средствами OpenGL.

Игровое приложение «Волейбол» должно иметь имеет различные тек­стуры для отображения на экране. Для создания текстур танков, ракет, горного хребта и заднего фона была использована программа Adobe Photoshop CC 2020. Созданные изображения были сохранены в формате bmp и добавлены в проект Visual Studio.

Для реализации паттерна проектирования «декоратор», позволяющего изменять скорости игрока и мяча, помимо основной красной текстуры ракеты были созданы голубая и фиолетовая текстуры.

## 2.3 Проектирование структуры ПО

В ходе создания архитектуры игрового приложения использовался про­цесс декомпозиции. Декомпозиция представляет из себя универсальный науч­ный метод решения сложных задач посредством разбиения сложной задачи на несколько более простых подзадач. Каждая подзадача была спроектированная и разработана, после чего все подзадачи были объединены в общее решение. Уникальность каждого элемента декомпозиции позволяет сформировать и от­личить различные составляющие объекта друг от друга. Элементы, получен­ные в результате декомпозиции, связаны с функциями отдельных подзадач единой сложной задачи.

Таким образом, перед началом разработки программного обеспечения при помощи процесса декомпозиции была создана обобщенная функциональ­ная схема информационного приложения, которая описывает последователь­ность работы его функций. Обобщенная функциональная схема приложения представлена на рисунке 2.1:

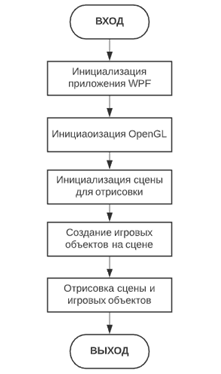


Рисунок 2.1 – Обобщенная функциональная схема приложения

Использование процесса декомпозиции позволяет добиться решения за­дачи с минимальными затратами и максимальной гибкостью. Каждая подси­стема была протестирована во время разработки, что позволило устранить ряд потенциальных ошибок на начальных этапах написания игры.

Для возможности в будущем легко расширить функционал программы были использованы паттерны (шаблоны) проектирования. Паттерны проекти­рования позволяют оптимально решать часто встречающиеся проблемы в про­ектировании программ. Существует большое количество различных паттер­нов проектирования, которые решают разные задачи проектирования.

В дан­ном курсовом проекте используются структурный паттерн проектирования «декоратор». Шаблон проектирования «декоратор» нужен в случае, необходимости динамически добавлять новую функциональность для объекта или для класса объектов во время выполнения программы. Данный шаблон проектирования является альтернативой наследования, так как он позволяет решать те же за­дачи, но использование наследования для добавления функциональности объ­екту является менее гибким решением, так как данный подход не позволит до­бавлять функциональность для целого класса объектов, а лишь для конкрет­ного объекта. В данном случае использовать такой подход не представляется возможным, поскольку это решение затрудняет комбинирование различной функциональности на объекте. «Декоратор» с задачей комбинирования функ­циональности справляется намного лучше, так как данный паттерн проектиро­вания позволяет динамически декорировать уже декорированные классы. В игровом приложении «Волейбол» декорируемым объектом является класс компонент. Также, благодаря использованию паттерна «Декоратор», в прило­жение можно легко добавить новые эффекты, а также присутствует возмож­ность комбинировать старые эффекты для получения увеличения динамики игры.

На рисунке 2.2 представлена схема реализации паттерна «декоратор».

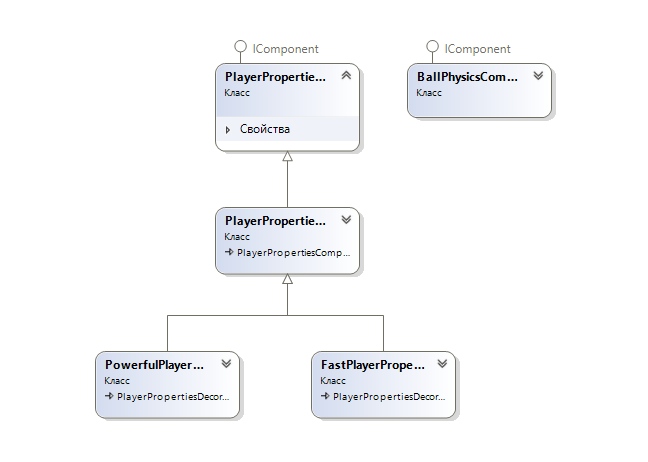


Рисунок 2.2 – Декоратор на игрока

Применение паттерна «декоратор» повышает гибкость разрабатывае­мого программного обеспечения и позволяет динамически подключать новую функциональность, что предоставляет возможность добавлять новые виды бо­нусов без написания с нуля большей части кода. Данный подход сильно упро­щает разработку программного продукта, так как позволяет повторно исполь­зовать уже написанные классы для решения новых задач.

Паттерн проектирования «Фабричный метод» является порождающим паттерном, определяющий интерфейс для создания объекта, но оставляет подклассам задачу решения о создании конкретного класса [7, с. 112]. Данный паттерн использует механизм полиморфизма, при котором классы всех конечных типов наследуются от одного базового класса.

В игре паттерн применим к классам, которые динамически добавляются и удаляются со сцены, что требует постоянного создания новых объектов. Такими классами в игре являются бонусы, накладываемые на персонажей. Поэтому в реализации необходимо создать абстрактные классы фабрик и их наследников с реализацией создания каждого объекта. Такая структура позволит вносить изменения в проект без возникновения проблем.

На рисунке 2.3 представлена схема реализации паттерна «Фабричный метод».



Рисунок 2.3 – Схема реализации паттерна «Фабричный метод»

## 2.4 Описание работы классов

В процессе проектирования архитектуры игрового приложения «Пере­стрелка» было принято решение разбить основное приложение на 3 части – игровой движок, игровую логику и пользовательский интерфейс. Подобное разделение позволило отделить логику отображения игры от логики взаимо­действия объектов друг с другом. Данный подход позволяет легко заменять различные элементы интерфейса или подменять алгоритмы работы игровых объектов. Классы каждой части были помещены в отдельные проекты.

Проект с игровым движком содержит в себе базовые классы для работы игрового приложения. Наиболее важным классом данного приложения явля­ется класс GameObject, который состоит из компонентов, представленных ин­терфейсом IComponent и исполняемых скриптов, которые представлены аб­страктным классом Script. Под компонентами в данном случае понимаются классы, реализующие интерфейс IComponent, которые хранят в себе какие-либо данные и содержат методы для обработки этих данных. Под исполняе­мыми скриптами следует понимать классы, наследующиеся от абстрактного класса Script, которые предназначены для исполнения каких-либо действий в каждом кадре, который отрисовывается в игровом приложении. Данные классы могут брать данные из компонентов игрового объекта и обрабатывать их в соответствии с нуждами системы.

В проекте с игровым движком по умолчанию реализовано несколько классов-компонентов, которые необходимы большинству игровых объектов, а именно классы Texture2D, Animation2D, который наследуется от класса тек­стуры, Transform и Collider. Классы Texture2D и Animation2D хранят в себе данные для последующей отрисовки объекта на сцене посредством класса Renderer. Класс Transform хранит всю информацию о положении объекта на сцене, такую как позицию на сцене, угол поворота объекта. Класс Collider со­держит информацию о вершинах фигуры, с которой может случиться столк­новение.

Также в проекте с игровым движком содержится класс Scene, который хранит набор объектов, отрисовываемых на сцене. Каждый кадр состояние объектов на сцене обновляется посредством циклического вызова метода об­новления скриптов для всех объектов на сцене. Объекты, которые должны быть добавлены или удалены из набора помещаются в отдельный список, ко­торый обрабатывает сцена после обновления состояний объектов. Данный подход позволяет избежать ошибок при переборе коллекций. После обновле­ния всех объектов на сцене, обновленные объекты отрисовываются на сцене при помощи класса Renderer. Класс Renderer способен брать текстуры для от­рисовки у каждого объекта на сцене и отрисовать их посредством вызовов функций библиотеки OpenGL. Текстуры объектов представлены классами Tex­ture2D и Animation2D, которые хранят информацию о размерах текстуры и о самой текстуре, загруженной из файла. Animation2D хранит информацию о не­скольких текстурах в виде массива, а также данный класс способен опреде­лять, какую именно текстуру из массива нужно отобразить в данный момент времени.

Также в классе Scene содержится ссылка на окно WPF приложения, в котором будет реализовываться игровая сцена. Данная ссылка нужна для того, чтобы сцена могла подстроиться под размеры окна, а также использовать ка­кие-либо элементы интерфейса, разработанные средствами WPF, такие как кнопки, полосы прогресса, элементы с текстом и так далее.

После разработки движка игрового приложения была разработана струк­тура проекта с игровой логикой приложения. Данный проект использует объ­екты из игрового движка для своей работы, но при этом расширяет их и до­полняет новыми сущностями. Для того, чтобы описать корректное поведение мяча, были созданы скрипты, позволяющие добавить объекту физику и скрипты.

Для повышения гибкости описанной структуры был добавлен класс Bat­tleSceneSettings, который хранит конфигурацию боевой сцены, что позволяет настраивать боевую сцену во время выполнения программного обеспечения.

На рисунке 2.4 представлена схема данных проекта *GameEngineLibrary*.

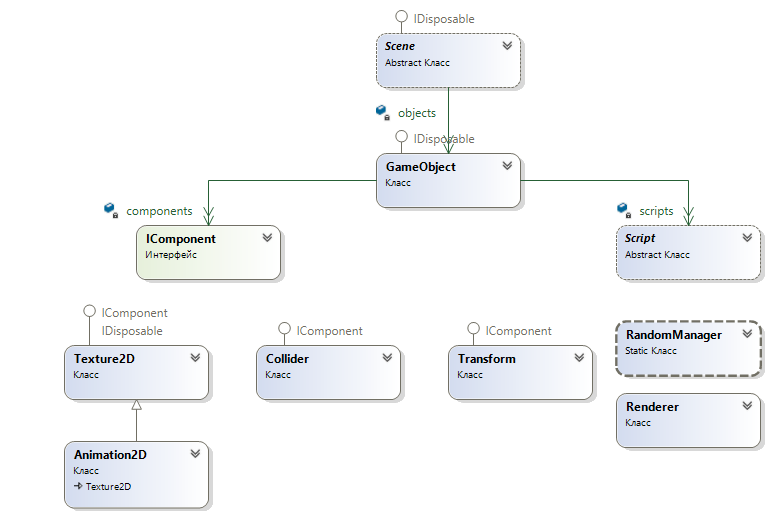


Рисунок 2.4 – Cхема данных проекта GameEngineLibrary

В процессе проектирование архитектуры игры «Волейбол» была создана схема данных, которая применяется для объектно-ориентированного анализа и проектирования. Данную диаграмму можно использовать для конструирова­ния визуализации и документирования программных систем. Сама диаграмма включает в себя несколько вдов строительных блоков, а именно диаграмму, сущности и связи. Сущности являются основными элементами модели, а связи соединяют сущности между собой, диаграммы в свою очередь группируют нужные наборы сущностей. Таким образом, схема данных показывает набор классов и интерфейсов, и связи между ними.

На рисунке 2.5 представлена схема данных проекта GameLibrary.

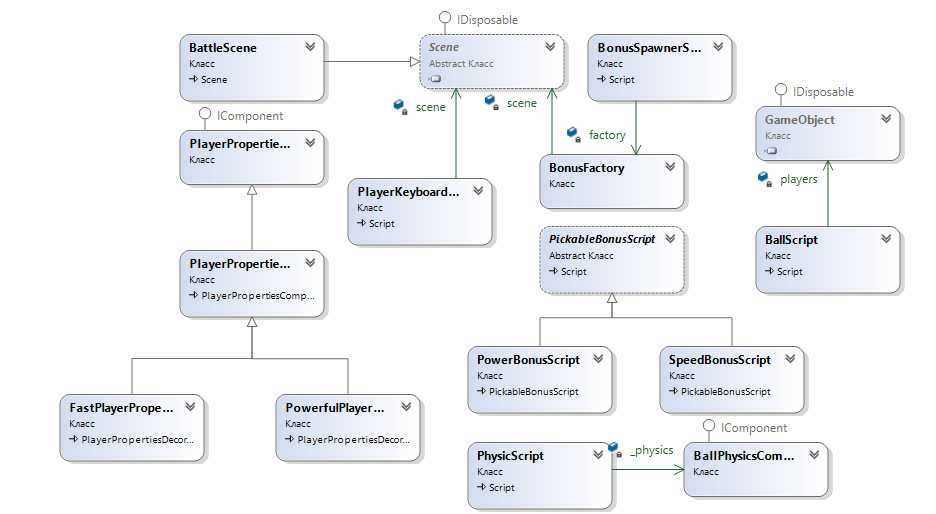
****

Рисунок 2.5 – Cхема данных проекта GameLibrary

Проект *GameLibrary* содержит класс поведения мяча в классе *BallScript*(Приложение А, код программы *BallScript.cs*), наследуемый от Script. В котором в методе *Update*() проверяется пересечении мяча центра игрового поля или выход мяча из карты и начисление игру очков.

Класс PhysicScript (Приложение А, код программы PhysicScript.cs), наследуемый от Script, описывающий физику объекта, на который действуют различные силы.

Класс *BonusFactory* (Приложение А, код программы *BonusFactory.cs*) это Фабрика создания бонусов.

Класс *PlayerPropertiesComponent* (Приложение А, код программы *PlayerPropertiesComponent.cs*), который описывает базовые характеристики игрока, сила удара и скорость передвижения. При столкновении с бонусов скорости передвижения или силы удара с помощью декоратора *PlayerPropertiesDecorator* (Приложение А, код программы *PlayerPropertiesDecorator*.cs) навсегда увеличиваются характеристики.

Класс *FastPlayerPropertiesDecorator* (Приложение А, код программы *FastPlayerPropertiesDecorator.cs*) декорирует свойство Speed.

Класс *PowerfulPlayerPropertiesDecorator* (Приложение А, код программы *PowerfulPlayerPropertiesDecorator.cs*) декорирует свойство *Power*.

Класс *BallPhysicsComponent* (Приложение А, код программы *BallPhysicsComponent.cs*) для придания физики мячу. Класс обладает конструктором который принимает аргументы как Сила гравитации, Высота уровня земли, Объект мяча, обладает методом *UpdateObjectPosition* который обновляет позицию мяча, *Resetdirection*() Обнуление направления физических сил, *AddImpulse*(*Vector2* *impulse*) Добавление импульса к физическому телу.

Класс *BattleScene* (Приложение А, код программы *BattleScene.cs*) для создания игровой сцены. Класс создает игровые объекты игроков, устанавливает границы игрового поля, загружает текстуры бонусов, игроков, мяча. Методом *СreateBackground()* который создает фон игрового поля.

**3 РЕАЛИЗАЦИЯ ИТОГОВОГО ПРИЛОЖЕНИЯ** **«ВОЛЕЙБОЛ»**

* 1. **Обзор итоговой программы**

Итоговым результатом работы является игровое приложения «Пляжный волейбол», в котором используются алгоритмы «Фабричный метод» и «Деко­ратор». Основные программные действия, такие как описание объектов, отри­совка объектов, а также некоторые проверки, проводятся в классе But­tleScene.cs (Приложение А, листинг класса ButtleScene.cs). Объектами для опи­сания данного класса являются такие игровые объекты, как мяч, игроки (пер­вый и второй), игровые бонусы, увеличивающие скорость передвижения и силу удара, а также счётчики очков игроков.

Рассмотрим поближе объекты игровой сцены, создающиеся после ини­циализации основных элементов:

1. Объект beach служит в качестве фонового изображения;
2. Два объекта player. Первый и второй игроки располагаются в левой и правой частях экрана соответственно. Для объектов данного класса указываются параметры коллайдеров. Каждый игрок имеет отдель­ный счётчик забитых мячей. Программно реализованы списки, содер­жащие значения клавиш, используемых для управления аватарами первого и второго игроков;
3. Объект ball (Приложение А, листинг класса BallScript.cs). Объект дан­ного класса располагается над первым игроком.

В момент, когда все размещаемые в игровом пространстве объекты находятся на своих местах, игра стартует. По ходу игровой сессии благодаря определённым флагам соблюдаются основные условия игры: взаимодействие коллайдеров игровых объектов, зачисление очков игроку за успешное попада­ние по зоне врага.

* 1. **Верификация игрового приложения**

Модульное тестирование – это программный процесс, который позво­ляет проверять отдельные части исходного кода на корректность. Это позво­ляет быстро проверить обновляемый или дополняемый код на наличие ошибок в уже оттестированных местах программы, а также в целом облегчают задачу оптимизации кода.

ColisionTests (Приложение А, код программы ColisionTests.cs), тестирующий класс Collider, который отслеживает столкновение игровых объектов, с помощью IsTrue и IsFalse, проверяющий булевое значение.

PowerDecorator (Приложение А, код программы PowerDecorator.cs), тестирующий класс PowerDecorator, описывает поведение игроков после усиления, с помощью метода AreEqual, который проверяет равенство двух значений.

Аналогично SpeedDecorator (Приложение А, код программы SpeedDecorator.cs), тестирующий класс SpeedDecorator, описывает поведение игроков после усиления, с помощью метода AreEqual, который проверяет равенство двух значений.

Модульное тестирование позже позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно. Это поощряет программистов к изменениям кода, поскольку достаточно легко проверить, что код работает и после изменений. Клиенты, которые не знают, как использовать данный класс, могут использовать юнит-тест в качестве примера.

Описанный класс модульного тестирования и соответствующие ему тестирующие методы, а также результаты тестирования представлены на рисунке 3.1.

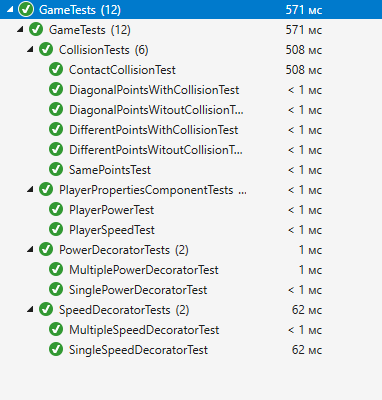
****

Рисунок 3.1 – Результаты тестирования классов разработанного игрового приложения

**3.3 Пользовательский интерфейс**

После запуска игрового приложения, пользователь наблюдает старто­вый экран (рисунок 3.1). Надпись на главном экране предлагает произвести непосредственный старт игры.



Рисунок 3.2 – Стартовое меню приложения «Волейбол»

Сразу после нажатия Play начинается игра, состоящая из перебрасыва­ния мячом. Если мяч касается поля или вылетает за его пределы, подававшему игроку засчитывается очко.

Таблица 3.1 – Управление аватаром игрока

|  |  |  |
| --- | --- | --- |
| Тип передвижения | Первый игрок | Второй игрок |
| Перемещение в ле­вую сторону | Клавиша «A» | Клавиша «Left» |
| Перемещение в пра­вую сторону | Клавиша «D» | Клавиша «Right» |
|  |  |  |



Рисунок 3.3 – Игровой аватар игрока

В ходе игры на поле появляются бонусы в случайном порядке. Бонус “Скорость” (рисунок 3.4) дает игроку, подобравшему его, ускорение на десять процентов.



Рисунок 3.4 – Игровой бонус “Скорость”

Так же может появиться игровой бонус “Усиление” (рисунок 3.5), кото­рый усиливает удар по мячу. Не исключено возникновение ситуации, в кото­рой на сторону одного игрока может упасть большое количество бонусов под­ряд.



Рисунок 3.5 – Игровой бонус “Усиление”

При поднятии “Скорость”, передвижения персонажа ускоряется на де­сять процентов. При поднятии “Усиление”, сила удара игрока увеличивается на десять процентов. Данные значения являются настраиваемыми сугубо про­граммно.

Чтобы добиться перебрасывания мяча через сетку, игроку необходимо дотронуться до волейбольного мяча (рисунок 3.5) при таком условии, чтобы его персонаж двигался в сторону удара. Мяч примет значения скорости, ими­тируя событие удара и летит в нужную игроку сторону. Благодаря тому, что скорость игрока может быть изменена по ходу игры с помощью бонусов, сила удара того же игрока будет меняться в зависимости от количества подобран­ных бонусов, поэтому их сбор является крайне приоритетным занятием. Игро­вая сессия занимает довольно небольшой временный промежуток, поэтому ве­роятность возникновения ситуаций, в которых игра проходит “в одни ворота” сведено к минимуму, без искусственного ограничения игровых характеристик аватаров.



Рисунок 3.6 – Окно игрового приложения.

После того, как мяч касается территории одного из игроков, он появля­ется над аватаром того игрока, который пропустил мяч. Забившему гол игроку начисляется один победный балл. Каждый игрок имеет право касаться мяча несколько раз, главное не уронить его. Но при постоянно увеличивающейся динамике игры, удерживать мяч у себя долго не получится.

Так как в «Пляжный волейбол» играют не для победы, а для развлече­ния, то ограничения по времени нет. Но из-за постоянно появляющихся бону­сов, игра будет постепенно увеличивать свою скорость, что является ее услож­нением.

Игра представляет собой простое в использовании приложение с про­стыми механиками. В него можно легко добавлять и убирать механики и объ­екты, улучшая приложение.

Приложение представляет из себя предельно простое в использовании и настройке приложение с предельно простыми механиками, описанными в предыдущих главах. Приложение пригодно для масштабирования, являясь до­вольно просторным, что даёт возможность для добавления новых механик и/или объектов.

**ЗАКЛЮЧЕНИЕ**

В ходе выполнения проекта было разработано игровое приложение «Пляжный волейбол». Это приложение подходит для аудитории любой воз­растной категории, так как в нем нет сцен насилия и кровопролития, а также просто в освоении управления. Аркадные игры не занимают много времени и сил, так как в них нет единой истории. Так же такие игры тренируют внима­тельность, глазомер и скорость реакции.

Сама структура приложения была разработана таким способом, что в приложение можно легко добавить новые функции. Такой эффект достигается паттернам “Декоратор”.

Так же было проведено сравнение библиотек DirectX и OpenGL, в ходе которого OpenGL показало себя с лучшей стороны, поэтому эта библиотека была выбрана. Данный подход позволил сосредоточиться на реализации игро­вой логики, так как окно WPF имеет ряд созданных компонентов, вроде кно­пок, которые упростили взаимодействие пользователя с игровым приложе­нием.

# **СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ**

1. Миллер, Т. Managed DirectX 9 с управляемым кодом. Программиро­вание игр и графика / Т. Миллер. – М.:КомБук, 2005. – 401 с.
2. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# / Дж. Рихтер. – СПб.:Питер, 2020. – 896 с.
3. Флемов, М. Е. DirectX и C++. Искусство программирования /

М. Е. Флемов. – СПб.:БХВ-Петербург, 2017. – 384 с.

1. Селлерс, Г. Супербиблия OpenGL / Г. Селлерс. – Addison-Wesley Professional, 2015. – 880 с.
2. Вольф, Д. OpenGL 4. Язык Шейдеров. Книга рецептов / Д. Вольф. – М.:ДМК Пресс, 2015. – 368 с.
3. Боресков, А. В. Программирование компьютерной графики. Совре­менный OpenGL / А. В. Боресков. – М.:ДМК Пресс, 2019. – 372 с.
4. Приемы объектно-ориентированного проектирования. Паттерны про­ектирования / Э. Гамма [и др.]. – СПб.:Питер, 2020. – 368 с.

**ПРИЛОЖЕНИЕ А**

(обязательное)

**Листинг игрового приложения «Пляжный волейбол»**

**Листинг программного класса Animation2D.cs:**

using OpenTK.Graphics.OpenGL;

using System;

using System.Drawing;

using System.Drawing.Imaging;

namespace GameEngineLibrary

{

/// <summary>

/// Класс, отвечающий за анимированные 2d изображения.

/// </summary>

public class Animation2D : Texture2D

{

/// <summary>

/// Массив идентификаторов текстур анимации.

/// </summary>

private int[] animationId;

/// <summary>

/// Индекс текущей текстуры.

/// </summary>

private int index;

/// <summary>

/// Время, прошедшее с начала анимации.

/// </summary>

private int currentTime;

/// <summary>

/// Уникальный идентификатор текстуры,

/// которая должна отображаться в текущий

/// момент анимации.

/// </summary>

public override int ID

{

get

{

return animationId[index];

}

}

/// <summary>

/// Время анимации.

/// </summary>

public int AnimationTime { get; set; }

/// <summary>

/// Создание анимации.

/// </summary>

/// <param name="animationId">Массив идентификаторов изображений в анимации.</param>

/// <param name="width">Размер одного изображения.</param>

/// <param name="height">Высота одного изображения.</param>

public Animation2D(int[] animationId, int width, int height)

: base(animationId[0], width, height)

{

index = 0;

AnimationTime = 1;

currentTime = 0;

this.animationId = animationId;

}

/// <summary>

/// Конструктор копирования анимации.

/// Его стоит использовать для передачи анимаций объектам,

/// чтобы не захватывать лишние неуправляемые ресурсы.

/// </summary>

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

public Animation2D(Animation2D animation)

: this(animation.animationId, animation.Width, animation.Height)

{

disposed = true;

AnimationTime = animation.AnimationTime;

}

/// <summary>

/// Обновить состояние анимации.

/// </summary>

/// <param name="delta">Время, прошедшее с предыдущего кадра.</param>

public void Update(TimeSpan delta)

{

currentTime += delta.Milliseconds;

if (currentTime >= AnimationTime)

{

currentTime = 0;

}

double deltaTime = (double)currentTime / AnimationTime;

index = (int)(animationId.Length \* deltaTime);

}

/// <summary>

/// Загрузить анимацию из файла.

/// </summary>

/// <param name="path">Путь к файлу.</param>

/// <param name="columns">Количество колонок со спрайтами.</param>

/// <param name="rows">Количество рядов со спрайтами.</param>

/// <returns>Новая анимация.</returns>

public static Animation2D LoadAnimation(string path, int columns = 1, int rows = 1)

{

int length = columns \* rows;

int[] idArr = new int[length];

Bitmap bitmap = new Bitmap(path);

int spriteWidth = bitmap.Width / columns;

int spriteHeight = bitmap.Height / rows;

Rectangle rectangle = new Rectangle(0, 0, spriteWidth, spriteHeight);

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

{

rectangle.X = spriteWidth \* (i % columns);

rectangle.Y = spriteHeight \*(i / columns);

int id = GL.GenTexture();

GL.BindTexture(TextureTarget.Texture2D, id);

Bitmap sprite = bitmap.Clone(rectangle,

System.Drawing.Imaging.PixelFormat.Format32bppArgb);

BitmapData data = sprite.LockBits(

new Rectangle(0, 0, sprite.Width, sprite.Height),

ImageLockMode.ReadOnly,

System.Drawing.Imaging.PixelFormat.Format32bppArgb);

GL.TexImage2D(TextureTarget.Texture2D, 0,

PixelInternalFormat.Rgba, data.Width, data.Height, 0,

OpenTK.Graphics.OpenGL.PixelFormat.Bgra,

PixelType.UnsignedByte, data.Scan0);

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.Nearest);

GL.TexParameter(TextureTarget.Texture2D,

TextureParameterName.TextureMagFilter,

(int)TextureMagFilter.Nearest);

sprite.UnlockBits(data);

idArr[i] = id;

}

return new Animation2D(idArr, spriteWidth, spriteHeight);

}

/// <summary>

/// Уничтожение анимации.

/// </summary>

public override void Dispose()

{

if (disposed)

return;

foreach (int id in animationId)

{

GL.DeleteTexture(id);

}

disposed = true;

}

}

}

**Листинг программного класса Collider.cs:**

using OpenTK;

using System;

namespace GameEngineLibrary

{

/// <summary>

/// Класс, описывающий выпуклую фигуру,

/// которая способна совершать проверки на

/// пересечение с другими выпуклыми фигурами.

/// </summary>

public class Collider : IComponent

{

/// <summary>

/// Центр коллизии.

/// </summary>

private Vector2 center;

/// <summary>

/// Радиус коллизии.

/// </summary>

private float radius;

public Vector2 Center { get => center; }

/// <summary>

/// Создание коллайдера.

/// </summary>

/// <param name="center">Центр коллизии</param>

/// <param name="radius">Радиус коллизии</param>

public Collider(Vector2 center, float radius)

{

this.center = center;

this.radius = radius;

}

/// <summary>

/// Проверка пересечения между двумя коллайдерами.

/// </summary>

/// <param name="collider">Коллайдер, с которым будет осуществляться проверка.</param>

/// <returns>True, если есть пересечение, иначе false.</returns>

public bool CheckCollision(Collider collider)

{

float dx = center.X - collider.center.X;

float dy = center.Y - collider.center.Y;

double distance = Math.Sqrt(dx \* dx + dy \* dy);

return distance < radius + collider.radius;

}

}

}

**Листинг программного класса GameObject.cs:**

using System;

using OpenTK;

using System.Collections.Generic;

namespace GameEngineLibrary

{

/// <summary>

/// Класс, который описывает объект на сцене.

/// </summary>

public class GameObject : IDisposable

{

private bool disposed = false;

private Dictionary<string, IComponent> components;

/// <summary>

/// Список добавленных скриптов.

/// </summary>

private List<Script> scripts;

/// <summary>

/// Список внутренних объектов.

/// </summary>

public List<GameObject> InnerObjects { get; private set; }

/// <summary>

/// Создание нового игрового объекта.

/// </summary>

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

/// <param name="position">Положение объекта на сцене.</param>

/// <param name="rotationPoint">Точка поворота объекта.</param>

/// <param name="scale">Масштаб объекта.</param>

/// <param name="rotation">Угол поворота объекта.</param>

public GameObject(Texture2D texture, Vector2 position,

Vector2 rotationPoint, Vector2 scale, double rotation)

{

InnerObjects = new List<GameObject>();

scripts = new List<Script>();

components = new Dictionary<string, IComponent>();

Transform transform = new Transform();

transform.Position = position;

transform.RotationPoint = rotationPoint;

transform.Scale = scale;

transform.Rotation = rotation;

components["transform"] = transform;

if (texture != null)

{

components["texture"] = texture;

}

}

/// <summary>

/// Создание нового игрового объекта.

/// </summary>

public GameObject() : this(null, Vector2.Zero, Vector2.Zero, Vector2.One, 0)

{

}

/// <summary>

/// Создание нового игрового объекта с текстурой.

/// </summary>

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

public GameObject(Texture2D texture) : this()

{

components["texture"] = texture;

}

/// <summary>

/// Добавление объекта внутрь другого объекта.

/// Объекты внутри других объектов должны отрисовываться

/// относительно позиции родительского объекта.

/// </summary>

/// <param name="gameObject">Объект для вставки.</param>

public void AddInnerObject(GameObject gameObject)

{

InnerObjects.Add(gameObject);

Transform transform = gameObject.GetComponent("transform") as Transform;

if (transform != null)

{

transform.Parent = this;

}

}

/// <summary>

/// Удаление объекта из другого объекта.

/// Объекты внутри других объектов должны отрисовываться

/// относительно позиции родительского объекта.

/// </summary>

/// <param name="gameObject">Объект для удаления.</param>

public void RemoveInnerObject(GameObject gameObject)

{

InnerObjects.Remove(gameObject);

Transform transform = gameObject.GetComponent("transform") as Transform;

if (transform != null)

{

transform.Parent = null;

}

}

/// <summary>

/// Добавление скрипта для объекта.

/// Скрипт определяет основное поведение объекта.

/// </summary>

/// <param name="script">Скрипт для добавления.</param>

public void AddScript(Script script)

{

scripts.Add(script);

script.SetControlledObject(this);

script.Init();

}

/// <summary>

/// Удаление скрипта объекта.

/// Скрипт определяет основное поведение объекта.

/// </summary>

/// <param name="script">Скрипт для удаления.</param>

public void RemoveScript(Script script)

{

scripts.Remove(script);

script.SetControlledObject(null);

}

/// <summary>

/// Установить коллайдер объекту.

/// </summary>

/// <param name="collider">Новый коллайдер.</param>

public void SetCollider(Collider collider)

{

components["collider"] = collider;

}

/// <summary>

/// Привязать коллайдер к текстуре.

/// </summary>

public void UpdateColliderToTexture()

{

Transform transform = GetComponent("transform") as Transform;

Texture2D texture = GetComponent("texture") as Texture2D;

float x = transform.Position.X;

float y = transform.Position.Y;

float width = texture.Width \* transform.Scale.X;

float height = texture.Height \* transform.Scale.Y;

float radius = width / 2f;

Vector2 center = new Vector2(x + radius, y + height / 2f);

SetCollider(new Collider(center, radius));

}

/// <summary>

/// Обновить состояние объекта.

/// </summary>

/// <param name="delta">Время, прошедшее между кадрами.</param>

public void Update(TimeSpan delta)

{

foreach (Script script in scripts)

{

script.Update(delta);

}

foreach (GameObject gameObject in InnerObjects)

{

gameObject.Update(delta);

}

Animation2D animation = GetComponent("texture") as Animation2D;

if (animation != null)

{

animation.Update(delta);

}

}

/// <summary>

/// Получение компонентов объекта по названию.

/// </summary>

/// <param name="key">Название компонента.</param>

/// <returns></returns>

public IComponent GetComponent(string key)

{

IComponent component;

components.TryGetValue(key, out component);

return component;

}

/// <summary>

/// Добавление компонента объекту.

/// </summary>

/// <param name="key">Название компонента.</param>

/// <param name="component">Компонент для добавления.</param>

public void AddComponent(string key, IComponent component)

{

components[key] = component;

}

/// <summary>

/// Уничтожение игрового объекта.

/// </summary>

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

/// <summary>

/// Уничтожение игрового объекта.

/// </summary>

/// <param name="disposing">Если уже уничтожен.</param>

public void Dispose(bool disposing)

{

if (!disposed)

{

if (disposing)

{

if (GetComponent("texture") is Texture2D texture)

{

texture.Dispose();

}

}

disposed = true;

}

}

/// <summary>

/// Деструктор игрового объекта.

/// </summary>

~GameObject()

{

Dispose(false);

}

}

}

**Листинг программного класса IComponent.cs:**

namespace GameEngineLibrary

{

/// <summary>

/// Маркерный интерфейс, который помечает все компоненты в GameObject.

/// </summary>

public interface IComponent

{

}

}

**Листинг программного класса RandomMeneger.cs:**

using System;

namespace GameEngineLibrary

{

/// <summary>

/// Класс, отвечающий за случайные числа в игре.

/// </summary>

public static class RandomManager

{

private static Random random = new Random();

/// <summary>

/// Возвращает неотрицательное случайное целое число.

/// </summary>

/// <returns>Неотрицательное случайное целое число</returns>

public static int Next()

{

return random.Next();

}

/// <summary>

/// Возвращает неотрицательное случайное целое число,

/// которое меньше определенного максимального значения.

/// </summary>

/// <param name="maxValue">Максимальное значение.</param>

/// <returns>Неотрицательное случайное целое число</returns>

public static int Next(int maxValue)

{

return random.Next(maxValue);

}

/// <summary>

/// Возвращает неотрицательное случайное целое число

/// в указанном диапазоне.

/// </summary>

/// <param name="minValue">Минимальное значение.</param>

/// <param name="maxValue">Максимальное значение.</param>

/// <returns>Неотрицательное случайное целое число</returns>

public static int Next(int minValue, int maxValue)

{

return random.Next(minValue, maxValue);

}

/// <summary>

/// Возвращает случайное число с плавающей точкой,

/// которое больше или равно 0,0 и меньше 1,0.

/// </summary>

/// <returns></returns>

public static double NextDouble()

{

return random.NextDouble();

}

}

}

**Листинг программного класса Renderer.cs:**

using System;

using System.Collections.Generic;

using OpenTK;

using OpenTK.Graphics.OpenGL;

namespace GameEngineLibrary

{

/// <summary>

/// Класс, отвечающий за отрисовку объектов на сцене.

/// </summary>

public class Renderer

{

/// <summary>

/// Сцена для отрисовки.

/// </summary>

private Scene scene;

/// <summary>

/// Создание нового объекта для отрисовки объектов на сцене.

/// </summary>

/// <param name="scene">Сцена, которую будет рендерить объект.</param>

public Renderer(Scene scene)

{

this.scene = scene;

}

/// <summary>

/// Отрисовать объекты на сцене.

/// </summary>

public void Render()

{

Render(scene.GetGameObjects());

}

/// <summary>

/// Отрисовать объекты из переданного массива.

/// </summary>

/// <param name="objectsToRender">Массив объектов для отрисовки.</param>

private void Render(List<GameObject> objectsToRender)

{

foreach (GameObject gameObject in objectsToRender)

{

Render(gameObject.InnerObjects);

RenderObject(gameObject);

}

}

/// <summary>

/// Отрисовать объект на сцене.

/// </summary>

/// <param name="gameObject">Объект для отрисовки.</param>

private void RenderObject(GameObject gameObject)

{

Texture2D texture = gameObject.GetComponent("texture") as Texture2D;

Transform transform = gameObject.GetComponent("transform") as Transform;

if (texture == null || transform == null)

{

return;

}

Vector2 rotationPoint = transform.RotationPoint;

Vector2 position = transform.Position;

double rotation = transform.Rotation;

Vector2[] vertices = new Vector2[4]

{

new Vector2(0, 0),

new Vector2(1, 0),

new Vector2(1, 1),

new Vector2(0, 1)

};

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

GL.Begin(PrimitiveType.Quads);

GL.Color3(texture.Color);

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

{

GL.TexCoord2(vertices[i]);

vertices[i].X \*= texture.Width;

vertices[i].Y \*= texture.Height;

vertices[i].X -= rotationPoint.X;

vertices[i].Y -= rotationPoint.Y;

vertices[i] = new Vector2(

(float)(Math.Cos(rotation) \* vertices[i].X -

Math.Sin(rotation) \* vertices[i].Y),

(float)(Math.Sin(rotation) \* vertices[i].X +

Math.Cos(rotation) \* vertices[i].Y));

vertices[i].X += rotationPoint.X;

vertices[i].Y += rotationPoint.Y;

vertices[i] \*= transform.Scale;

vertices[i] += position;

GL.Vertex2(vertices[i]);

}

GL.End();

}

/// <summary>

/// Устанавливает сцену для отрисовки.

/// </summary>

/// <param name="scene">Новая сцена для отрисовки.</param>

public void SetSceneToRender(Scene scene)

{

this.scene = scene;

}

}

}

**Листинг программного класса Scene.cs:**

using System;

using System.Collections.Generic;

using System.Windows;

namespace GameEngineLibrary

{

/// <summary>

/// Интерфейс, описывающий пользовательскую сцену.

/// </summary>

public abstract class Scene : IDisposable

{

/// <summary>

/// Окно, в котором отрисовывается сцена.

/// </summary>

public Window GameWindow { get; }

/// <summary>

/// Список созданных текстур.

/// </summary>

private List<Texture2D> textures;

/// <summary>

/// Список объектов на сцене.

/// </summary>

private List<GameObject> objects;

/// <summary>

/// Временный массив с объектами, который нужен

/// для возможности добавления объектов на сцену

/// во время выполнения скриптов.

/// </summary>

private List<GameObject> objectsToAdd;

/// <summary>

/// Временный массив с объектами, который нужен

/// для возможности удаления объектов со сцены

/// во время выполнения скриптов.

/// </summary>

private List<GameObject> objectsToRemove;

/// <summary>

/// Создание сцены.

/// </summary>

/// <param name="window">Окно, в котором отрисовывается сцена.</param>

public Scene(Window window)

{

GameWindow = window;

textures = new List<Texture2D>();

objects = new List<GameObject>();

objectsToRemove = new List<GameObject>();

objectsToAdd = new List<GameObject>();

}

/// <summary>

/// Метод, который вызывается 1 раз для инициализации сцены.

/// </summary>

public abstract void Init();

/// <summary>

/// Метод, который вызывается перед отрисовкой кадра.

/// В этом методе параметры сцены должны обновляться.

/// </summary>

/// <param name="delta">Время, прошедшее между кадрами.</param>

public void Update(TimeSpan delta)

{

foreach (GameObject gameObject in objects)

{

gameObject.Update(delta);

}

UpdateObjectsArray();

}

/// <summary>

/// Метод, позволяющий обновлять количество объектов для обработки динамически.

/// </summary>

private void UpdateObjectsArray()

{

if (objectsToRemove.Count != 0)

{

foreach (GameObject gameObject in objectsToRemove)

{

objects.Remove(gameObject);

}

objectsToRemove.Clear();

}

if (objectsToAdd.Count != 0)

{

objects.AddRange(objectsToAdd);

objectsToAdd.Clear();

}

}

/// <summary>

/// Получение списка с объектами на сцене.

/// </summary>

/// <returns>Список объектов на сцене.</returns>

public List<GameObject> GetGameObjects()

{

return objects;

}

/// <summary>

/// Метод добавления объекта на сцену.

/// </summary>

/// <param name="gameObject">Объект для добавления.</param>

public void AddGameObject(GameObject gameObject)

{

objectsToAdd.Add(gameObject);

}

/// <summary>

/// Метод удаления объекта со сцены.

/// </summary>

/// <param name="gameObject">Объект для удаления.</param>

public void RemoveGameObject(GameObject gameObject)

{

objectsToRemove.Add(gameObject);

}

/// <summary>

/// Метод добавления текстур на сцену.

/// </summary>

/// <param name="texture">Текстура для добавления.</param>

public void AddTexture(Texture2D texture)

{

textures.Add(texture);

}

/// <summary>

/// Уничтожение сцены.

/// </summary>

public void Dispose()

{

foreach (Texture2D texture in textures)

{

texture.Dispose();

}

foreach (GameObject gameObject in objects)

{

gameObject.Dispose();

}

}

}

}

**Листинг программного класса Script.cs:**

using System;

namespace GameEngineLibrary

{

/// <summary>

/// Интерфейс, описывающий скрипт, который определяет поведение игрового объекта.

/// </summary>

public abstract class Script

{

/// <summary>

/// Объект, которым управляет скрипт.

/// </summary>

protected GameObject controlledObject;

/// <summary>

/// Метод, инициализирующий данные скрипта.

/// </summary>

public virtual void Init()

{

}

/// <summary>

/// Метод, который содержит основную логику программы.

/// Данный метод вызывается в каждом кадре игры.

/// </summary>

/// <param name="delta">Время, прошедшее между кадрами.</param>

public abstract void Update(TimeSpan delta);

/// <summary>

/// Устанавливает объект, который будет контролировать скрипт.

/// </summary>

/// <param name="controlledObject">Объект, контролируемый скриптом.</param>

public void SetControlledObject(GameObject controlledObject)

{

this.controlledObject = controlledObject;

}

}

}

**Листинг программного класса Texture2D.cs:**

using OpenTK.Graphics.OpenGL;

using System;

using System.Drawing;

using System.Drawing.Imaging;

namespace GameEngineLibrary

{

/// <summary>

/// Класс, который описывает двухмерную текстуру.

/// </summary>

public class Texture2D : IComponent, IDisposable

{

/// <summary>

/// Переменная, отвечающая за состояние объекта.

/// </summary>

protected bool disposed = false;

/// <summary>

/// Идентификатор текстуры.

/// </summary>

public virtual int ID { get; private set; }

/// <summary>

/// Ширина текстуры.

/// </summary>

public int Width { get; private set; }

/// <summary>

/// Высота текстуры.

/// </summary>

public int Height { get; private set; }

/// <summary>

/// Цвет, который будет наложен на текстуру.

/// </summary>

public Color Color { get; set; }

/// <summary>

/// Создание новой текстуры.

/// </summary>

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

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

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

public Texture2D(int id, int width, int height)

{

ID = id;

Width = width;

Height = height;

Color = Color.Transparent;

}

/// <summary>

/// Загрузка текстуры из файла.

/// </summary>

/// <param name="path">Путь к файлу.</param>

/// <returns>Загруженная текстура.</returns>

public static Texture2D LoadTexture(string path)

{

int id = GL.GenTexture();

GL.BindTexture(TextureTarget.Texture2D, id);

Bitmap bitmap = new Bitmap(path);

BitmapData data = bitmap.LockBits(

new Rectangle(0, 0, bitmap.Width, bitmap.Height),

ImageLockMode.ReadOnly,

System.Drawing.Imaging.PixelFormat.Format32bppArgb);

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.Nearest);

GL.TexParameter(TextureTarget.Texture2D,

TextureParameterName.TextureMagFilter,

(int)TextureMagFilter.Nearest);

return new Texture2D(id, bitmap.Width, bitmap.Height);

}

/// <summary>

/// Сравнение двух объектов.

/// </summary>

/// <param name="obj">Объект для сравнения.</param>

/// <returns>True, если объекты равны и false в противном случае.</returns>

public override bool Equals(object obj)

{

return obj is Texture2D texture &&

ID == texture.ID &&

Width == texture.Width &&

Height == texture.Height;

}

/// <summary>

/// Получение hash-кода объекта.

/// </summary>

/// <returns>Целое число.</returns>

public override int GetHashCode()

{

var hashCode = 1463928665;

hashCode = hashCode \* -1521134295 + ID.GetHashCode();

hashCode = hashCode \* -1521134295 + Width.GetHashCode();

hashCode = hashCode \* -1521134295 + Height.GetHashCode();

return hashCode;

}

/// <summary>

/// Уничтожение текстуры.

/// </summary>

public virtual void Dispose()

{

if (disposed)

return;

GL.DeleteTexture(ID);

disposed = true;

}

}

}

**Листинг программного класса Transform.cs:**

using OpenTK;

namespace GameEngineLibrary

{

/// <summary>

/// Объект, отвечающий за расположение объекта на экране.

/// </summary>

public class Transform : IComponent

{

/// <summary>

/// Объект-родитель для объекта, которому принадлежит данный экземпляр Transform.

/// </summary>

public GameObject Parent { get; set; }

/// <summary>

/// Точка, вокруг которой будет поворачиваться объект.

/// </summary>

public Vector2 RotationPoint { get; set; }

/// <summary>

/// Локальные координаты объекта относительно родительского объекта.

/// </summary>

private Vector2 localPosition;

/// <summary>

/// Позиция объекта.

/// </summary>

public Vector2 Position

{

get

{

if (Parent == null)

{

return localPosition;

}

Transform transform = Parent.GetComponent("transform") as Transform;

if (transform != null)

{

return transform.Position + localPosition;

}

return localPosition;

}

set

{

localPosition = value;

}

}

/// <summary>

/// Масштабирование объект.

/// </summary>

public Vector2 Scale { get; set; }

/// <summary>

/// Угол поворота объекта.

/// </summary>

public double Rotation { get; set; }

}

}

**Листинг программного класса BallPhusicsComponent.cs:**

using GameEngineLibrary;

using OpenTK;

using System;

using System.Collections.Generic;

namespace GameLibrary.Components

{

public class BallPhysicsComponent : IComponent

{

private Transform transform;

private List<Vector2> impulses = new List<Vector2>();

private Vector2 gravity;

private Vector2 direction;

private float ground;

public BallPhysicsComponent(GameObject gameObject, float gravity = 100, float ground = 300)

{

transform = gameObject.GetComponent("transform") as Transform;

this.gravity = Vector2.UnitY \* gravity;

this.direction = Vector2.Zero;

this.ground = ground;

}

public void UpdateObjectPosition(float delta)

{

Vector2 acceleration = gravity \* delta;

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

{

acceleration += impulses[i] \* delta;

}

impulses.Clear();

direction += acceleration;

if (Math.Abs(direction.X) < 0.01)

{

direction.X = 0;

}

transform.Position += direction;

Vector2 position = transform.Position;

if (position.Y > ground)

{

direction.Y = ground;

transform.Position = new Vector2(position.X, ground);

}

}

public void ResetDirection()

{

direction = Vector2.Zero;

}

public void AddImpulse(Vector2 impulse)

{

impulses.Add(impulse);

}

}

}

**Листинг программного класса FastPlayerPropertiesDecorator.cs:**

namespace GameLibrary.Components

{

public class FastPlayerPropertiesDecorator : PlayerPropertiesDecorator

{

public override float Speed => base.Speed \* 2.1f;

public FastPlayerPropertiesDecorator(PlayerPropertiesComponent playerProperties)

: base (playerProperties)

{

}

}

}

**Листинг программного класса** **PlayerPropertiesComponent.cs:**

using GameEngineLibrary;

namespace GameLibrary.Components

{

public class PlayerPropertiesComponent : IComponent

{

public virtual float Speed { get => 300; }

public virtual float Power { get => 100; }

}

}

**Листинг программного класса** **PlayerPropertiesDecorator.cs:**

namespace GameLibrary.Components

{

public class PlayerPropertiesDecorator : PlayerPropertiesComponent

{

private PlayerPropertiesComponent playerProperties;

public override float Speed => playerProperties.Speed;

public override float Power => playerProperties.Power;

public PlayerPropertiesDecorator(PlayerPropertiesComponent playerProperties)

{

this.playerProperties = playerProperties;

}

}

}

**Листинг программного класса** **PowerfulPlayerPropertiesDecorator.cs:**

namespace GameLibrary.Components

{

public class PowerfulPlayerPropertiesDecorator : PlayerPropertiesDecorator

{

public override float Power => base.Power \* 2.1f;

public PowerfulPlayerPropertiesDecorator(PlayerPropertiesComponent playerProperties)

: base (playerProperties)

{

}

}

}

**Листинг программного класса BattleScene.cs:**

using System.Windows;

using System.Windows.Controls;

using GameEngineLibrary;

using GameLibrary.Components;

using GameLibrary.Scripts;

using OpenTK;

using OpenTK.Input;

namespace GameLibrary

{

/// <summary>

/// Сцена сражения в игре.

/// </summary>

public class BattleScene : Scene

{

private const string BACKGROUND\_TEXTURE\_PATH = @"../../../GameLibrary/Resources/beach.jpg";

private const string PLAYER\_TEXTURE\_PATH = @"../../../GameLibrary/Resources/player.png";

private const string BALL\_TEXTURE\_PATH = @"../../../GameLibrary/Resources/ball.png";

private const string SPEED\_BONUS\_PATH = @"../../../GameLibrary/Resources/1.png";

private const string POWER\_BONUS\_PATH = @"../../../GameLibrary/Resources/speed.png";

private TextBlock firstPlayerScore;

private TextBlock secondPlayerScore;

/// <summary>

/// Создание сцены.

/// </summary>

/// <param name="window">Окно, в котором будет отрисовываться сцена.</param>

public BattleScene(Window window, TextBlock firstPlayerScore, TextBlock secondPlayerScore)

: base (window)

{

this.firstPlayerScore = firstPlayerScore;

this.secondPlayerScore = secondPlayerScore;

}

/// <summary>

/// Инициализация сцены.

/// </summary>

public override void Init()

{

AddGameObject(CreateBackground());

Texture2D speedBonusTexture = Texture2D.LoadTexture(SPEED\_BONUS\_PATH);

AddTexture(speedBonusTexture);

Texture2D powerBonusTexture = Texture2D.LoadTexture(POWER\_BONUS\_PATH);

AddTexture(powerBonusTexture);

Texture2D player1Texture = Texture2D.LoadTexture(PLAYER\_TEXTURE\_PATH);

AddTexture(player1Texture);

GameObject player1 = new GameObject(player1Texture, new Vector2(-400, 300), new Vector2(0,0), new Vector2(1,1), 0);

player1.AddComponent("playerProperties", new PlayerPropertiesComponent());

player1.AddScript(new PlayerKeyboardControlScript(this, Key.A, Key.D, -800, -128));

AddGameObject(player1);

Texture2D player2Texture = Texture2D.LoadTexture(PLAYER\_TEXTURE\_PATH);

AddTexture(player2Texture);

GameObject player2 = new GameObject(player1Texture, new Vector2(400, 300), new Vector2(0, 0), new Vector2(1, 1), 0);

player2.AddComponent("playerProperties", new PlayerPropertiesComponent());

player2.AddScript(new PlayerKeyboardControlScript(this, Key.Left, Key.Right, 0, 672));

AddGameObject(player2);

Texture2D ballTexture = Texture2D.LoadTexture(BALL\_TEXTURE\_PATH);

AddTexture(ballTexture);

GameObject ball = new GameObject(ballTexture, new Vector2(-400, -400), new Vector2(0, 0), new Vector2(1, 1), 0);

BallPhysicsComponent physics = new BallPhysicsComponent(ball);

ball.AddComponent("physics", physics);

ball.AddScript(new PhysicScript(physics));

ball.AddScript(new BallScript(player1, player2, firstPlayerScore, secondPlayerScore));

AddGameObject(ball);

GameObject bonusSpawner = new GameObject();

BonusFactory bonusFactory = new BonusFactory(this, player1, player2, speedBonusTexture, powerBonusTexture);

bonusSpawner.AddScript(new BonusSpawnerScript(this, bonusFactory));

AddGameObject(bonusSpawner);

}

private GameObject CreateBackground()

{

Texture2D backgroundTex = Texture2D.LoadTexture(BACKGROUND\_TEXTURE\_PATH);

AddTexture(backgroundTex);

return new GameObject(backgroundTex,

new Vector2((float)-GameWindow.Width, (float)-GameWindow.Height),

Vector2.Zero, new Vector2(1.3f, 1.3f), 0);

}

}

}

**Листинг программного класса** **BallScript.cs:**

using GameEngineLibrary;

using GameLibrary.Components;

using OpenTK;

using System;

using System.Windows.Controls;

namespace GameLibrary.Scripts

{

public class BallScript : Script

{

private GameObject[] players;

private BallPhysicsComponent physics;

private TextBlock firstPlayerScore;

private TextBlock secondPlayerScore;

private float ground;

private float left;

private float right;

private int firstScore;

private int secondScore;

public BallScript(GameObject firstPlayer, GameObject secondPlayer, TextBlock firstPlayerScore, TextBlock secondPlayerScore, float ground = 350, float left = -800, float right = 800)

{

players = new GameObject[] { firstPlayer, secondPlayer };

this.firstPlayerScore = firstPlayerScore;

this.secondPlayerScore = secondPlayerScore;

this.ground = ground;

this.left = left;

this.right = right;

firstScore = 0;

secondScore = 0;

}

public override void Init()

{

physics = controlledObject.GetComponent("physics") as BallPhysicsComponent;

}

public override void Update(TimeSpan delta)

{

Collider ballCollider = controlledObject.GetComponent("collider") as Collider;

if (ballCollider.Center.X < left)

{

IncreseSecondScore();

return;

}

if (ballCollider.Center.X > right)

{

IncreseFirstScore();

return;

}

if (ballCollider.Center.Y > ground)

{

if (ballCollider.Center.X > 0)

{

IncreseFirstScore();

}

else

{

IncreseSecondScore();

}

return;

}

foreach (GameObject player in players)

{

Collider playerCollider = player.GetComponent("collider") as Collider;

if (playerCollider.CheckCollision(ballCollider))

{

Vector2 direction = playerCollider.Center - ballCollider.Center;

PlayerPropertiesComponent playerProperties = player.GetComponent("playerProperties") as PlayerPropertiesComponent;

physics.AddImpulse(Vector2.NormalizeFast(direction) \* -playerProperties.Power \* (float)delta.TotalMilliseconds);

}

}

}

private void IncreseFirstScore()

{

firstScore++;

firstPlayerScore.Text = firstScore.ToString();

Transform transform = controlledObject.GetComponent("transform") as Transform;

Transform playerTransform = players[1].GetComponent("transform") as Transform;

transform.Position = new Vector2(playerTransform.Position.X, -400);

controlledObject.UpdateColliderToTexture();

physics.ResetDirection();

}

private void IncreseSecondScore()

{

secondScore++;

secondPlayerScore.Text = secondScore.ToString();

Transform transform = controlledObject.GetComponent("transform") as Transform;

Transform playerTransform = players[0].GetComponent("transform") as Transform;

transform.Position = new Vector2(playerTransform.Position.X, -400);

controlledObject.UpdateColliderToTexture();

physics.ResetDirection();

}

}

}

**Листинг программного класса** **PhysicScript.cs:**

using GameEngineLibrary;

using GameLibrary.Components;

using System;

namespace GameLibrary.Scripts

{

/// <summary>

/// Класс, описывающий физику объекта,

/// на который действуют различные силы.

/// </summary>

public class PhysicScript : Script

{

private BallPhysicsComponent \_physics;

public PhysicScript(BallPhysicsComponent physics)

{

\_physics = physics;

}

/// <summary>

/// Обновление состояния скрипта.

/// </summary>

/// <param name="delta">Время, прошедшее с предыдущего кадра.</param>

public override void Update(TimeSpan delta)

{

\_physics.UpdateObjectPosition((float) delta.TotalSeconds);

controlledObject.UpdateColliderToTexture();

}

}

}

**Листинг программного класса** **PlayerKeyboardControlScript.cs:**

using OpenTK.Input;

using GameEngineLibrary;

using OpenTK;

using System;

using GameLibrary.Components;

namespace GameLibrary.Scripts

{

/// <summary>

/// Скрипт, отвечающий за передвижение персонажа при помощи клавиатуры.

/// </summary>

public class PlayerKeyboardControlScript : Script

{

private Key left;

private Key right;

private float leftBoard;

private float rightBoard;

Scene scene;

/// <summary>

/// Создание контроллера для персонажа.

/// </summary>

/// <param name="scene">Сцена, в которой происходит перемещение объекта.</param>

/// <param name="left">Левая клавиша</param>

/// <param name="right">Правая клавиша</param>

public PlayerKeyboardControlScript(Scene scene, Key left, Key right, float leftBoard, float rightBoard)

{

this.scene = scene;

this.left = left;

this.right = right;

this.leftBoard = leftBoard;

this.rightBoard = rightBoard;

}

/// <summary>

/// Обновление состояния скрипта.

/// </summary>

/// <param name="delta">Время, прошедшее с предыдущего кадра.</param>

public override void Update(TimeSpan delta)

{

PlayerPropertiesComponent playerProperties = controlledObject.GetComponent("playerProperties") as PlayerPropertiesComponent;

Vector2 speed = new Vector2(playerProperties.Speed, 0);

KeyboardState keyboard = Keyboard.GetState();

Transform transform = controlledObject.GetComponent("transform") as Transform;

Vector2 translate = Vector2.Zero;

if (keyboard[left])

{

translate -= speed \* (float)delta.TotalSeconds;

}

if (keyboard[right])

{

translate += speed \* (float)delta.TotalSeconds;

}

transform.Position += translate;

if (transform.Position.X < leftBoard || transform.Position.X > rightBoard)

{

transform.Position -= translate;

}

controlledObject.UpdateColliderToTexture();

}

}

}

**Листинг программного класса** **BonusFactory.cs:**

using GameEngineLibrary;

using System;

namespace GameLibrary.Scripts

{

public class BonusFactory

{

private Texture2D speedBonusTexture;

private Texture2D powerBonusTexture;

private Scene scene;

private GameObject firstPlayer;

private GameObject secondPlayer;

public BonusFactory(Scene scene, GameObject firstPlayer, GameObject secondPlayer, Texture2D speedBonusTexture, Texture2D powerBonusTexture)

{

this.speedBonusTexture = speedBonusTexture;

this.powerBonusTexture = powerBonusTexture;

this.scene = scene;

this.firstPlayer = firstPlayer;

this.secondPlayer = secondPlayer;

}

public GameObject CreateRandomBonus()

{

Random random = new Random();

GameObject bonus = null;

switch (random.Next(2))

{

case 0:

bonus = new GameObject(speedBonusTexture);

bonus.AddScript(new SpeedBonusScript(scene, firstPlayer, secondPlayer));

break;

case 1:

bonus = new GameObject(powerBonusTexture);

bonus.AddScript(new PowerBonusScript(scene, firstPlayer, secondPlayer));

break;

}

return bonus;

}

}

}

**Листинг программного класса** **BonusSpawnerScript.cs:**

using GameEngineLibrary;

using OpenTK;

using System;

namespace GameLibrary.Scripts

{

public class BonusSpawnerScript : Script

{

private Random random = new Random();

private BonusFactory factory;

private Scene scene;

private float ground;

public BonusSpawnerScript(Scene scene, BonusFactory factory, float ground = 300)

{

this.scene = scene;

this.factory = factory;

this.ground = ground;

}

public override void Update(TimeSpan delta)

{

if (random.Next(1000) == 0)

{

int width = (int)scene.GameWindow.Width;

GameObject bonus = factory.CreateRandomBonus();

Transform transform = bonus.GetComponent("transform") as Transform;

transform.Position = new Vector2(random.Next(width \* 2) - width, ground);

bonus.UpdateColliderToTexture();

scene.AddGameObject(bonus);

}

}

}

}

**Листинг программного класса** **PickableBonusScript.cs:**

using GameEngineLibrary;

using System;

namespace GameLibrary.Scripts

{

public abstract class PickableBonusScript : Script

{

private Scene scene;

private GameObject[] players;

public PickableBonusScript(Scene scene, GameObject firstPlayer, GameObject secondPlayer)

{

this.scene = scene;

players = new GameObject[] { firstPlayer, secondPlayer };

}

public override void Init()

{

controlledObject.UpdateColliderToTexture();

}

public override void Update(TimeSpan delta)

{

Collider bonusCollider = controlledObject.GetComponent("collider") as Collider;

foreach (GameObject player in players)

{

Collider playerCollider = player.GetComponent("collider") as Collider;

if (playerCollider.CheckCollision(bonusCollider))

{

Picked(player);

scene.RemoveGameObject(controlledObject);

return;

}

}

}

protected abstract void Picked(GameObject player);

}

}

**Листинг программного класса** **PowerBonusScript.cs:**

using GameEngineLibrary;

using GameLibrary.Components;

namespace GameLibrary.Scripts

{

public class PowerBonusScript : PickableBonusScript

{

public PowerBonusScript(Scene scene, GameObject firstPlayer, GameObject secondPlayer) : base(scene, firstPlayer, secondPlayer)

{

}

protected override void Picked(GameObject player)

{

PlayerPropertiesComponent playerProperties = player.GetComponent("playerProperties") as PlayerPropertiesComponent;

player.AddComponent("playerProperties", new PowerfulPlayerPropertiesDecorator(playerProperties));

}

}

}

**Листинг программного класса** **SpeedBonusScript.cs:**

using GameEngineLibrary;

using GameLibrary.Components;

namespace GameLibrary.Scripts

{

public class PowerBonusScript : PickableBonusScript

{

public PowerBonusScript(Scene scene, GameObject firstPlayer, GameObject secondPlayer) : base(scene, firstPlayer, secondPlayer)

{

}

protected override void Picked(GameObject player)

{

PlayerPropertiesComponent playerProperties = player.GetComponent("playerProperties") as PlayerPropertiesComponent;

player.AddComponent("playerProperties", new PowerfulPlayerPropertiesDecorator(playerProperties));

}

}

**ПРИЛОЖЕНИЕ Б**

(обязательное)

**ПРИЛОЖЕНИЕ В**

(обязательное)

**РУКОВОДСТВО СИСТЕМНОГО ПРОГРАММИСТА**

1. **Общие сведения о программе.**

Приложение «Пляжный волейбол» представляет собой платформу для игры на двоих на одном устройстве. В игре существует лишь один режим: игра на двоих. Главной целью приложения является развлечение и тренировка реакции.

Программа показала свою максимальную производительность при взаимодействии с:

– Операционной системой Windows 10;

* Компьютерные манипуляторы: клавиатура, мышь;
* Устройство вывода: компьютерный монитор.

1. **Структура программы.**

Структура приложения предполагает разделение на две части: графику и логику.

Логическая часть программы отвечает за описание объектов приложение, их свойства и взаимодействие.

Графическая часть программы отвечает за отрисовку определённых в логической части объектов, пользовательский интерфейс и графику приложе­ния в целом.

1. **Настройка программы.**

Для запуска не нужна настройка, нужно всего лишь запустить приложе­ние.

1. **Проверка программы.**

Отсутствие аварийного окончания работы программы во время игры или возникновения исключений является признаком корректной работы игрового приложения.

1. **Дополнительные возможности.**

Отсутствуют.

1. **Сообщение системному программисту.**

При возникновении непредвиденных ошибок или остановке работы при­ложения рекомендуется провести перезапуск всего приложения.

**ПРИЛОЖЕНИЕ Г**

(обязательное)

**РУКОВОДСТВО ПРОГРАММИСТА**

1. **Назначения и условия применения программы.**

Приложение «Пляжный волейбол» представляет собой платформу для игры на двоих на одном устройстве. В игре существует лишь один режим: игра на двоих. Главной целью приложения является развлечение и тренировка реакции.

Программа показала свою максимальную производительность при взаи­модействии с:

* Операционной системой Windows 10;
* Компьютерные манипуляторы: клавиатура, мышь;
* Устройство вывода: компьютерный монитор.

1. **Характеристика программы.**

Сразу же после запуска программы загружается главное меню. В глав­ном меню необходимо нажать соответствующую клавишу, сразу после этого начинается игровой процесс, который длиться до тех пор, пока игра не станет слишком сложной или игрокам не надоест.

1. **Обращение к программе.**

Для старта работы приложения необходимо запустить файл.

1. **Входные и выходные данные.**

В качестве входных данных для приложение используется язык про­граммирования C#, графика OpenGL, WPF, а также шаблоны проектирования «Фабричный метод» и «Декоратор».

1. **Сообщение программисту.**

При возникновении непредвиденных ошибок или остановке работы при­ложения рекомендуется провести перезапуск всего приложения.

**ПРИЛОЖЕНИЕ Д**

(обязательное)

**РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ**

1. **Введение.**

Игровое приложение «Волейбол» представляет из себя платформу для игры на двоих на одном устройстве. Основное применение – получение до­суга. Доступен лишь один режим – игра на двоих.

Приложение имеет два этапа работы: главное меню и непосредственный игровой процесс.

1. **Назначение и условия применения.**

Игровое приложение «Пляжный волейбол» представляет из себя плат­форму для игры на двоих на одном устройстве.

Программа показала свою максимальную производительность при взаи­модействии с:

* Операционной системой Windows 10;
* Компьютерные манипуляторы: клавиатура, мышь;

– Устройство вывода: компьютерный монитор.

1. **Подготовка к работе.**

Для старта работы приложения необходимо запустить файл.

1. **Описание операций.**

Основные манипуляции со стороны игрока:

* Старт игры;
* Перемещение игрового аватара по игровому полю влево и вправо;
* Окончание игры.

1. **Непредвиденные ситуации.**

Программа грамотно верифицирована, но в ситуации возникновения ошибки рекомендуется полный перезапуск игрового приложения.

1. **Рекомендации.**

Для полного понимания игровых механик рекомендуется для начала за­пустить тестовую игру. Затем, после полного освоения, возможен переход к полноценной игре.