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: «Приложение, реализующее игру «Кольцевые гонки» с использованием Windows Form и графики OpenGL»

Исполнитель: студент группы ИТИ-21

Медянцев С.А.

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 2020

СОДЕРЖАНИЕ

[Введение 5](#_Toc40999997)

[1 Этапы и средства разработки игровых приложений 6](#_Toc40999998)

[1.1 Этапы разработки игрового приложения 6](#_Toc40999999)

[1.2 Актуальность разработки игрового приложения 7](#_Toc41000000)

[1.3 Гоночный игровой жанр 7](#_Toc41000001)

[1.4 Особенности существующих решений 10](#_Toc41000002)

[1.5 OpenGL 12](#_Toc41000003)

[1.6 Microsoft DirectX 13](#_Toc41000004)

[1.7 Выводы аналитического обзора 14](#_Toc41000005)

[2 Архитектура и структура игрового приложения 15](#_Toc41000006)

[2.1 Архитектура приложения 15](#_Toc41000007)

[2.2 Структура приложения 17](#_Toc41000008)

[2.3 Паттерны проектирования 20](#_Toc41000009)

[3 Компоненты программного комплекса 23](#_Toc41000010)

[3.1 Окно запуска приложения 23](#_Toc41000011)

[3.2 Игровой двигатель 23](#_Toc41000012)

[3.3 Элементы управления автомобилем 24](#_Toc41000013)

[3.4 Физика движения автомобиля 25](#_Toc41000014)

[3.5 Механизм поворота автомобиля 26](#_Toc41000015)

[3.6 Коллизии 26](#_Toc41000016)

[3.7 Внутриигровой интерфейс 28](#_Toc41000017)

[3.8 Игровые механизмы 28](#_Toc41000018)

[3.9 Модульные тесты 30](#_Toc41000019)

[3.10 Результаты опытной эксплуатации игрового приложения 31](#_Toc41000020)

[Заключение 33](#_Toc41000021)

[Список использованных источников 34](#_Toc41000022)

[Приложение А Листинг программы 35](#_Toc41000023)

[Приложение Б Листинг модульных тестов 80](#_Toc41000024)

[Приложение В Схемы шасси автомобиля 82](#_Toc41000025)

[Приложение Г Интерфейс приложения 84](#_Toc41000026)

[Приложение К Руководство пользователя 86](#_Toc41000027)

[Приложение Л Руководство программиста 88](#_Toc41000028)

[Приложение М Руководство системного программиста 89](#_Toc41000029)

[Приложение Н Схема использования паттерна «Фабричный метод» 90](#_Toc41000030)

# ВВЕДЕНИЕ

Компьютерные игры являются популярным способом проведения досуга среди многих людей уже не первое десятилетие, а высокие темпы развития игровой индустрии обусловлены прежде всего наличием большого потребительского спроса на различные типы игровых приложений. Более того, постоянно растущая заинтересованность пользователей в игровых продуктах породила большое многообразие игровых жанров.

Ставший уже, по сути, классическим гоночный жанр компьютерных игр продолжает активно развиваться на протяжении почти пяти десятилетий. «Гонки» полюбились игрокам всех возрастов и плотно заняли свою нишу в развлекательном сегменте благодаря тому, что представляют собой слияние простых и понятных игровых механизмов с острыми ощущениями от игрового процесса. Каждый заезд бросает игроку некий вызов, является испытанием. Скоростные заезды требуют от игрока предельной внимательности, быстроты реакции, точности и правильности выполняемых действий.

Игры этого жанра предлагают игроку множество различных способов проведения гонок, огромное количество разнообразных транспортных средств, современную графику, а также уникальные физические модели поведения транспорта. Они особенно широко распространены среди аудитории, заинтересованной в автомобилях и автомобильных соревнованиях. Гоночные игры развиваются параллельно с другими массовыми жанрами, а потребность в них остаётся стабильной на протяжении долгого времени.

Стоит отметить также, что на рынке компьютерных игр «гонки» занимают верхние позиции по количеству проданных копий. Опираясь на наблюдения за рынком, можно сказать, что особой популярностью пользуются гоночные игры аркадной направленности, которые характеризуются не только лёгкостью в управлении и нетребовательностью к вычислительной мощности компьютера, но и простотой в освоении новым игроком. На сегодняшний день масштабные работы в этой ветви игровой индустрии ведутся самыми крупными игровыми студиями мира, а многие из представителей гоночных игр остаются популярны среди игроков спустя десятилетия.

На основе всего вышесказанного, можно сделать вывод, что разработка новой игры в гоночном жанре является актуальной и востребованной среди широкого круга игроков.

# 1 ЭТАПЫ И СРЕДСТВА РАЗРАБОТКИ ИГРОВЫХ ПРИЛОЖЕНИЙ

## 1.1 Этапы разработки игрового приложения

Разработка любого игрового приложения начинается, прежде всего, с создания концепта. На этом этапе формируется основная идея и задача игры, поверхностно прорабатывается игровой дизайн. Исходя из имеющейся задумки, происходит выбор подходящего игрового жанра, в соответствии с которым создаётся более детальное описание проекта. Также производится поиск существующих игр, схожих по жанру. Одновременно с этим определяется сеттинг игры, на котором основывается специфика будущих механик игры, её будущего внешнего вида и других особенностей. В ходе создания концепта не важна точность в деталях будущего проекта, а внесение корректировок на этапе реализации является нормальным явлением при разработке игровых приложений.

Далее осуществляется переход к созданию прототипа игрового приложения. Целью создания прототипа является формирование первого впечатления от особенностей игрового процесса, тестирование некоторых важных игровых механик или же просто оценивание качества работы реализованных алгоритмов. Обычно прототипы игровых приложений представляют собой комплекс из реализованного игрового двигателя, компонента для отрисовки упрощённой графики, базовой поддержки упрощённых механизмов управления и базовой физической модели игровых объектов. Внутриигровой интерфейс на данном этапе, как правило, отсутствует.

После создания прототипа производится поверхностное тестирование имеющегося функционала, при необходимости вносятся правки в фундаментальные компоненты приложения. Следующим шагом является создание упрощённой версии игры с полноценным функционалом, по которой имеется возможность оценить потенциал игрового процесса. Обычно на этом этапе все особенности игры представляются в черновом варианте.

Затем производится наполнение проекта графическим и звуковым контентом. Сюда относится создание эскизов игровых объектов и локаций, рисование спрайтов, проработка структуры и читабельности внутриигрового интерфейса. Также при необходимости создаются анимации. Производятся работы по балансировке игры и её оптимизации, путём корректировки алгоритмов и способов взаимодействия внутриигровых компонентов.

В момент, когда наполненность игры становится максимальной (реализованы все игровые механики, элементы графики, внутриигровой интерфейс и прочие жанровые особенности), осуществляется тестирование приложения с целью выявления проблем с внутриигровым балансом.

## 1.2 Актуальность разработки игрового приложения

Рынок цифровых развлечений безостановочно расширяется. Популярность компьютерных игр возрастает с каждым годом, также не угасает интерес пользователей и к современной аппаратной части игровых компьютеров, и к новым моделям компьютерной игровой периферии. Самые успешные игровые проекты продаются миллионными тиражами. Более того, игры стали частью повседневной жизни многих людей на одном уровне с литературой, музыкой и кино, вбирая в себя всё лучшее из каждой развлекательной области и привнося что-то новое. Из этого следует вывод, что разработка компьютерных игр на сегодняшний день является не только рентабельной, но и востребованной.

Анализируя предпочтения игроков на основе глобальной статистики, несложно заметить, что запросы варьируются на основе множества факторов: возрастной категории игрока, его пола, сферы деятельности, местонахождения и пр.

Следовательно, данное наблюдение говорит о том, что разработка игры в любом существующем жанре является оправданным шагом, как минимум, благодаря вариативности предпочтений среди огромной численности игроков.

При рассмотрении европейского контингента игроков, выявляется явное превосходство в популярности одних игровых жанров над другими. Наиболее популярными игровыми жанрами среди европейских игроков являются:

– ролевые игры;

– спортивные игры;

– гоночные игры;

– игры-песочницы с открытым миром.

В частности, игры гоночного жанра игроки выбирают по причине личной заинтересованности в различных транспортных средствах, склонности к соревновательным особенностям игрового процесса и пр. Такие игры в большинстве случаев не являются чрезмерно требовательными к производительности компьютеров игроков, а имеют достаточно невысокие системные требования.

## 1.3 Гоночный игровой жанр

Гоночный игровой жанр по своей сути напоминает жанр спортивных игр. Данные жанры имеют схожие элементы и особенности воздействия на игрока. Гонки – это игры с видом от первого или от третьего лица, в которых игроку предоставляется управление каким-либо видом транспорта: наземным, воздушным, водным. Однако зачастую большинство игр производится с упором на наземные виды транспорта: автомобили. Это происходит из-за того, что большая часть игроков имеет более высокую степень осведомлённости об автомобилях, нежели чем о других видах транспорта из реальной жизни. Подобная бытовая осведомлённость об игровом объекте на основании личного жизненного опыта повышает привлекательность автомобильных гонок среди потенциальной аудитории.

Однако лишь малая часть игр гоночного жанра предоставляют игроку только управление транспортным средством на открытой местности при отсутствии иных особых событий. Гоночные игры в своей основной массе ставят игрока в ситуацию, в которой ему необходимо участвовать в соревнованиях среди других транспортных средств. Целью таких соревнований является победа. Сам же сценарий, при котором выполняются все условия для победного исхода, изменяется в зависимости от выбранного гоночного режима. Такие соревнования происходят либо между двумя игроками на одном компьютере при использовании обособленных устройств ввода, либо между несколькими игроками в режиме игры по сети, либо против внутриигрового искусственного интеллекта.

Гоночные игры делятся на две основные категории:

– гоночные симуляторы;

– аркадные гоночные игры.

Каждая из категорий характеризуется как уровнями сложности, так и наличием специальных игровых механик.

Гоночные автосимуляторы ставят перед собой целью максимально точную передачу ощущений от управления автомобилем. Ключевой особенностью таких игр является физика поведения транспортных средств, которая детально передаёт сложность управления реальным гоночным автомобилем.

В играх данного жанра игрок управляет лицензированными автомобилями, существующими в реальности, для которых реализовывается большой и детально проработанный набор физических параметров, позволяющий ощутить разницу в управлении автомобилей разных классов и брендов. Более того, в симуляторах детализированными физическими качествами обладают ещё и различные типы дорожных покрытий.

Графическая составляющая в играх такого рода старается максимально приблизиться к фотореализму для того, чтобы дать игроку ощущение полного погружения. Также особое внимание при разработке гоночных симуляторов уделяется тонкостям настройки множества характеристик каждого автомобиля. В ходе каждого заезда игроку необходимо следить за многочисленными показателями приборов и датчиков автомобиля: износ шин автомобиля, температура масла, уровень топлива в баке и пр.

При игре в гоночные симуляторы основной акцент ставится на правильность методики прохождения поворотов и точное маневрирование, причём для осуществления точного управления зачастую рекомендуется использовать дорогостоящие специализированные устройства ввода и вывода: гоночные рули для компьютеров, педали, селекторы передач и др.

Для игроков без опыта в симуляторах предусмотрен целый ряд помощников в управлении автомобилем, которые помогают освоить управление автомобилем и понять принципы игровой физики. Самыми распространёнными помощниками являются:

– антиблокировочная система;

– помощь при заносе;

– автоматическое переключение передач.

Также немаловажным фактом является проработка повреждений транспорта, которые влияют как на его внешность, так и на его ходовые качества, что значительно усложняет процесс гонки, ограничивая количество необдуманных игроком действий.

В аркадных гонках основной упор делается на получение игроком удовольствия от игры, благодаря упрощённым схемам управления и повышенному темпу игрового процесса. Также использование упрощённого физического движка, благодаря которому снижается входной порог для игрока является важным отличием от гоночных симуляторов. Упрощения в физике движения транспорта позволяют сделать возможным ослабление некоторых пунктов правил игры.

Соревнования имеют уникальные условия, отличающиеся от привычных условий проведения реальных гоночных состязаний. Например, прохождение опасных участков трассы на высокой скорости или же использование «скольжения» автомобиля без замедления для прохождения виражей являются нереалистичными элементами с точки зрения реализма. Аркадные гонки также зачастую предусматривают систему специальных призов, которые генерируются на участке пути в ходе заезда и мгновенно либо улучшают или ухудшают характеристики автомобиля, либо активизируют особые игровые возможности, контролируемые игроком.

Система повреждений в аркадных гонках развита слабо и не влияет на ходовые характеристики транспортного средства. Как правило, все столкновения в таких играх лишь временно замедляют игрока.

Графика в гонках такого типа обычно не столько реалистичная, сколько красочная и по-разному улучшается с помощью нестандартных приёмов цветокоррекции, эффекта дисторсии камеры игрока, различных ярких спецэффектов.

Следует отметить также, что в таких играх зачастую не имеется возможности настраивать характеристики автомобилей, а в случае, когда такая возможность присутствует, для изменения доступен лишь небольшой перечень параметров автомобиля, при этом такие изменения нередко являются неосуществимыми в реальности.

Заезды в аркадных гонках обычно проходят на загородных шоссе, скоростных трассах и в городах.

По типу заезды делятся на:

– кольцевые гонки;

– гонки по прямой с контрольными точками;

– гонки на выживание;

– прыжки на автомобилях и др.

## 1.4 Особенности существующих решений

Жанр гоночных игр на первый взгляд прост и понятен, однако для проектирования собственного игрового продукта в данном жанре сперва проводится анализ существующих игровых приложений. В ходе анализа разбираются ключевые особенности игрового процесса, способы отрисовки графики, существующие варианты игрового и визуального дизайна.

Популярными примерами аркадных гоночных игр являются: серия игр «The Need for Speed», серия игр «Forza Horizon», «Burnout Paradise» и др.

В свою очередь среди самых известных гоночных автомобильных симуляторов выделяются: серия игр «Forza Motorsport», проект «Assetto Corsa», серия «Project CARS», авторский проект «GranTurismo».

Большое количество популярных симуляторов и аркадных игр выпускаются, как правило, для всех актуальных платформ: PlayStation 4, Xbox One, MS Windows. Также все они поддерживают различные устройства ввода и многопользовательский режим игры с активными системами контроля за нарушениями.

Во всех популярных аркадных гонках существует два режима игры: карьера и свободный заезд. В режиме карьера игрок должен выигрывать серии гонок, чтобы разблокировать новые транспортные средства, трассы и пр. В свободном заезде игрок может управлять любым автомобилем из обширного игрового списка на любой из возможных игровых локаций. Персонализация автомобилей во многих аркадах заключается в изменении их внешнего вида, а также в настройках производительности.

Наиболее популярные гоночные симуляторы известны благодаря своим уникальным движкам, имеющим минимально возможные задержки между пользовательским вводом и реакцией автомобиля, а также сложным физически правильным системам динамического изменения характеристик автомобилей и погодных условий. Все трассы и автомобили в данных проектах проработаны с высочайшей степенью точности и детализации, благодаря исполнению по официальным лицензированным чертежам. Следует отметить также, что режим карьеры является в таких играх симулятором реальных соревновательных событий с турнирными таблицами, строгими правилами, точным контролем за временем и пр.

Наблюдая за самыми успешными игровыми гоночными проектами можно сделать вывод о том, что они во многом определяют вектор дальнейшего развития гоночных игр, а большое количество применяемых в них новых игровых механик в последствии перенимается и используется менее крупными командами разработчиков в своих более простых гоночных игровых приложениях.

Современные крупные проекты создаются с использованием передовых технологий. Практически все из них отлично оптимизированы под многопоточность процессора, многие задействуют весь потенциал графической подсистемы компьютера.

Менее крупные проекты сосредотачиваются же в основном на части разнообразия и особенностей игрового процесса, привнося в жертву графику. Такой подход позволяет сольным разработчикам и малочисленным командам разработчиков не тратить огромное количество ресурсов на попытки достигнуть того же качества оптимизации и уровня графики, который имеется в проектах высшего класса.

При разработке небольших проектов разработчики сосредотачивают своё внимание на двух-трёх главных механиках, определённых жанром. Поэтому среди небольших игр гоночного жанра основная масса из них является аркадными играми. Такие приложения, как правило, не поддерживают многопоточность и нестандартные устройства ввода. Движок и физические модели поведения объектов в таких создаются самостоятельно на каком-либо языке программирования в целях экономии ресурсов. Графика же реализуется при помощи специальных графических библиотек для выбранного языка программирования. Так основными средствами в разработке подобных приложений являются всевозможные интегрированные среды разработки и библиотеки-обёртки для OpenGL и DirectX. Пример внешнего вида приложения, реализованного таким способом представлен на рисунке 1.1.



Рисунок 1.1 – Внешний вид аркадной гоночной игры

## 1.5 *OpenGL*

OpenGL (Open Graphics Library) – согласно [1] это спецификация, которая описывает независимый от платформы программный интерфейс для написания приложений, использующих двумерную и трёхмерную компьютерную графику. OpenGL является не зависимым от оконной системы и операционной системы и позволяет разработчикам программного обеспечения для персональных компьютеров и рабочих станций создавать высокопроизводительные приложения с привлекательной графикой для таких областей, как создание контента, виртуальная реальность, игровая индустрия и пр. OpenGL включает в себя описание более 250 функций для рисования сложных трёхмерных сцен из простых примитивов.

Архитектурно OpenGL решает такие задачи, как сокрытие сложности адаптации различных 3D-ускорителей, предоставляя разработчику единый API и сокрытие различия в возможностях аппаратных платформ, требуя реализации недостающей функциональности с помощью программной эмуляции.

Основным принципом работы OpenGL является получение наборов векторных графических примитивов в виде точек, линий и многоугольников с последующей математической обработкой полученных данных и построением растровой картинки на экране или в памяти. Векторные трансформации и растеризация выполняются графическим конвейером, представленным на рисунке 1.2. Большая часть команд OpenGL либо добавляют графические примитивы на вход в конвейер, либо конфигурируют конвейер на различное исполнение трансформаций.

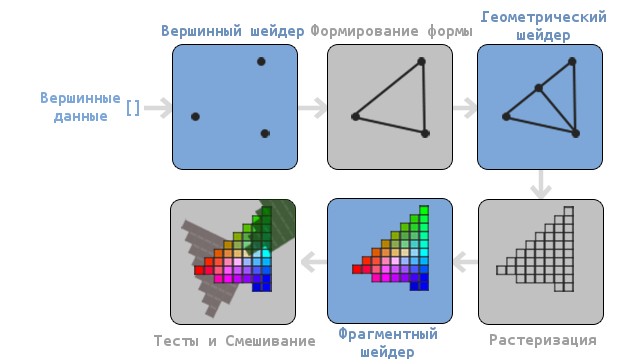


Рисунок 1.2 – Схема графического конвейера OpenGL.

## 1.6 *Microsoft* *DirectX*

Microsoft DirectX – согласно [2] это набор программных интерфейсов, разработанных для решения задач, связанных с программированием под Microsoft Windows. Технология DirectX позволяет программистам создавать в OC Windows приложения со встроенным доступом к аппаратным средствам без необходимости знать специфику аппаратной части компьютера. Фактически DirectX выполняет роль промежуточного звена между программой и драйвером, преобразуя обобщенные команды в команды, специфические для того или иного устройства. В интерфейсе DirectX реализованы принципы модели COM (Component Object Model), что позволяет оформлять необходимые функции в виде компонентов или объектов. В составе набора API DirectX присутствуют такие элементы, как DirectDraw, DirectSound, Directlnput, DirectPlay и Direct3D.

Direct3D – это подсистема, используемая для создания трёхмерных графики. Она состоит из программных интерфейсов низкого и высокого уровней, которые обеспечивают несколько базовых возможностей для создания изображения и осуществляют комплекс операций по его образованию. Direct3D реализует быстрый программный рендеринг с применением полного конвейера рендеринга 3D-графики. Для корректного воспроизведения система файлов Direct3D хранит информацию, необходимую для трехмерного рендеринга, включая текстуры, порядок расположения объектов, маршрут анимации и другие детали. Механизм рендеринга Direct3D реализуется с помощью трех матриц: трансформации, освещенности и растеризации. На рисунке 1.3 представлена схема трёхэтапной конвейерной архитектуры Direct3D.

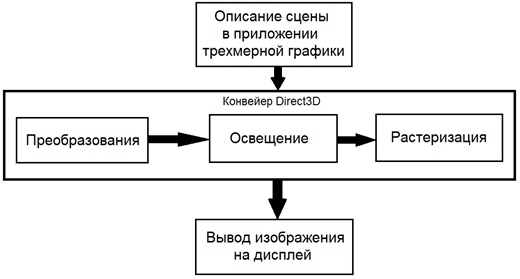


Рисунок 1.3 – Схема архитектуры Direct3D

## 1.7 Выводы аналитического обзора

Исходя из описания этапов разработки игрового приложения, анализа существующих решений и некоторых средств реализации формируется образ технического задания, подлежащего последующей реализации.

В ходе анализа существующих решений и их особенностей выясняется, что данный сегмент игровой индустрии пользуется популярностью среди широких масс людей. Гоночные симуляторы имеют более низкую популярность за счёт своей сложности и необходимости наличия дополнительной периферии для осуществления игрового процесса, то выбор жанра для разработки производится в сторону аркадного гоночного жанра. Однако для придания проекту более уникальной окраски производится смешение двух игровых жанров, что в результате помогает добавить нехарактерные для конкретного жанра игровые механики. Кроме того, аркадные игры не только охватывают большую аудиторию, но и являются более простыми в реализации.

Следовательно, для наиболее успешной реализации игры в гоночном жанре необходимо следовать специфике аркадных гоночных игр. Также с целью сохранения баланса между проектировочными ожиданиями и реальным игровым приложением, которое разрабатывается в рамках определённых ресурсных ограничений, рациональным подходом является использование 2D-графики и упрощённой физической модели, обеспечивающей простое и понятное управление автомобилем.

Также производится выбор способа его разработки и реализации. На основе сравнения графических средств OpenGL и DirectX между собой следует вывод, что оба средства обеспечивают практический одинаковый уровень функциональности. OpenGL является кроссплатформенным и показывает отличные результаты в быстродействии при работе с отрисовкой двумерных объектов, в то время как у DirectX имеются программные интерфейсы для работы со звуком.

Для данного проекта в качестве языка программирования выбран объектно-ориентированный язык программирования C# и интегрированная среда разработки Visual Studio 2019. А в качестве графического средства для разработки игрового приложения выбрана библиотека OpenTK как реализация спецификации OpenGL для языка C#.

# 2 АРХИТЕКТУРА И СТРУКТУРА ИГРОВОГО ПРИЛОЖЕНИЯ

## 2.1 Архитектура приложения

При разработке архитектуры игрового приложения, как правило, руководствуются несколькими основными соображениями об оптимальной работе приложения, а также вводятся некоторые допущения, связанные с особенностями конкретного проекта. Отсутствие единой оптимальной схемы работы игровых приложений, не отменяет необходимости придерживаться некоторых универсальных критериев для создания игровых программных продуктов при разработке.

Прежде всего, необходимо, чтобы игровое приложение было в достаточной степени производительным. Программа должна эффективно использовать ресурсы компьютера и быстро выполнять имеющиеся в ней функции.

Также необходимо, чтобы приложение могло поддерживать добавление нового функционала. Подобное сохранение гибкости приложений, в особенности игровых, является весьма разумным требованием, так как на практике очень часто возникают ситуации, в которых необходимо организовать добавление нового функционала, либо же расширить имеющиеся наборы объектов. Зачастую полезно продумывать возможные изменения приложения в будущем. Например, в данном игровом проекте есть шанс возникновения необходимости изменения:

– дизайна внутриигрового интерфейса;

– формы игровой сцены;

– дизайна игровых объектов.

Разрабатываемая система должна поддерживать возможность расширения без изменения кода существующий реализации. Для этой цели сперва создаётся самый базовый функционал приложения, который затем будет усложняться за счёт внедрения дополнительных программных реализаций.

Принимая во внимание вышеописанные рекомендации по проектированию архитектуры игрового приложения, необходимо также позаботиться и о снижении сложности реализации. Снизить сложность создания приложения позволяет использование метода декомпозиции.

Явление декомпозиции представляет собой способ разделения сложной системы на несколько частей. При декомпозиции происходит разделение приложения на отдельные модули, каждый из которых обладает своим перечнем выполняемых функций. Каждый модуль проектируется как независимый блок всей системы, а затем используется в качестве готового фрагмента программы при написании других модулей.

При успешном проектировании архитектуры приложение должно представлять из себя набор блоков, функционал которых накладывается друг на друга, образуя цельную систему. Классы в таких блоках обычно выделяются на основе разделения больших модулей приложения на более мелкие подмодули.

Деление комплексной системы осуществляется также на основе требуемого от приложения функционала. Поэтому выделение модуля подразумевает выделение его функций и данных, с которыми ему придётся работать. Однако следует отметить, что для игровых приложений зачастую характерна высокая связность подмодулей внутри каждого модуля, что несколько затрудняет расширение игрового приложения нестандартным функционалом.

С целью снижения сложности реализации данный игровой проект имеет модульную архитектуру. Схема взаимодействия модулей представлена на рисунке 2.1.



Рисунок 2.1 – Схема взаимодействия модулей игрового приложения

Архитектура приложения включает в себя четыре основных модуля:

– окно запуска приложения;

– игровой двигатель;

– компоненты;

– объекты.

Окно запуска, помимо взаимодействия с пользователем, предназначается для осуществления запуска игрового двигателя и первоначальную инициализацию игрового приложения различными объектами и необходимыми для их работы компонентами.

Игровой двигатель является главным модулем игрового приложения. В нём выполняется обработка основных данных некоторым базовым программным кодом, который является пригодным к расширению и повторному использованию. Игровой двигатель выполняет роль центрального контроллера, который осуществляет работу всех дополнительных модулей приложения.

Компоненты в игровом приложении позволяют одному игровому объекту поддерживать работу сразу с несколькими разными наборами функций, не связывая эти наборы между собой, согласно [3, с. 213]. Каждый компонент можно использовать в различных частях приложения, однако в данном проекте они применяются внутри базовых классов, которые предоставляют реализацию минимально необходимого функционала каких-либо игровых механик, либо определяют базовые модели игровых объектов. В данном проекте выделяется 3 вида компонентов:

– компонент отрисовки графики;

– компонент обсчёта физической модели твёрдого тела;

– компонент спрайтовой анимации.

Объект в данном проекте представляется в качестве фундаментальной структуры, от которой наследуются производные структуры. В последствии при разработке объекты предназначаются для использования в качестве агрегаторов для других объектов. Для каждого такого объекта создаётся экземпляр, происходит его инициализация, а затем корректным образом передаётся управление этим экземпляром игровому двигателю.

Каждый объект создаёт свои собственные компоненты при инициализации в зависимости от необходимости того или иного функционала. Такой подход обеспечивает гарантированность создания необходимых для работы компонентов несмотря на то, что объект становится более сложным для конфигурирования.

При таком жёстком связывании каждого компонента со своим объектом-владельцем взаимодействие между различными компонентами происходит либо через изменение состояния объекта-владельца, либо через обращения друг к другу напрямую.

Основные игровые механики также строятся на основе создания специальных объектов, которые наследуют основную реализацию от единого базового объекта и определения дополнительного необходимого функционала. При создании игровых механик зачастую используются уже существующие шаблоны проектирования, которые помогают наиболее эффективно решать задачи подобного рода.

## 2.2 Структура приложения

Реализация представленной архитектуры осуществляется путём создания абстрактных и конкретных классов и определения в них необходимых полей, свойств и методов, предоставляющих требуемый функционал. Каждый такой класс определяется в соответствии с возлагаемой на него задачей.

Графический интерфейс для запуска игрового окна создаётся при помощи визуального и программного средств разработки графического интерфейса Windows Forms.

При запуске игрового окна, отрисовка графической части которого будет происходить средствами OpenGL, в памяти устройства будут создаваться необходимые для работы приложения игровые объекты, а также подгружаемые ресурсы (элементы спрайтовой графики).

После успешного запуска данного окна происходит старт игрового процесса, при котором пользователю передается право взаимодействия с игровыми объектами посредством ввода команд с клавиатуры. Обновление состояний объектов и обработка ввода пользователей происходят в классе EngineCore игрового двигателя. Данный класс наследуется от класса GameWindow из библиотеки OpenTK, который имеет в себе базовую реализацию игрового цикла.

Базовый функционал для модулей компонентов и объектов, который отвечает основным требованиям для перехода к дальнейшей реализации полного функционала игрового приложения определяется в классах Component и GameObject.

Класс Component является абстрактным базовым классом, от которого наследуются основные компоненты приложения, отвечающие за отрисовку объектов, обновление физических параметров и анимацию спрайтовой графики.

Описание класса Component:

* поле attachedTo показывает, какому объекту типа GameObject принадлежит компонент;
* свойство IsEnabled показывает, активен ли компонент;
* метод PerformComponent позволяет задействовать функционал компонента.

Схема иерархии наследования объектов от класса Component представлена на рисунке 2.2.

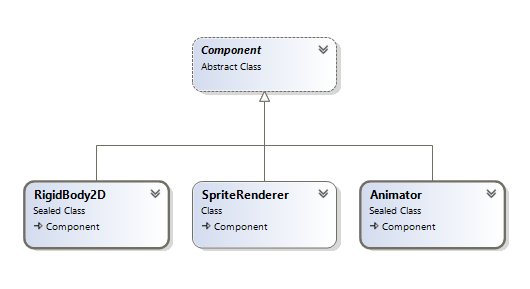


Рисунок 2.2 – Схема иерархии наследования от класса Component

Класс GameObject является абстрактным базовым классом для наследуемых игровых объектов, который определяет основной функционал каждого объекта.

Описание класса GameObject:

* поле components хранит список объектов типа Component, принадлежащих каждому экземпляру объекту-наследнику;
* свойство Position хранит текущую позицию игрового объекта в типе Vector2;
* метод Update() обновляет параметры игрового объекта за промежуток времени между отрисовкой каждого кадра;
* метод FixedUpdate() обновляет параметры игрового объекта за фиксированный промежуток времени;

Схема иерархии наследования объектов от класса GameObject представлена на рисунке 2.3.

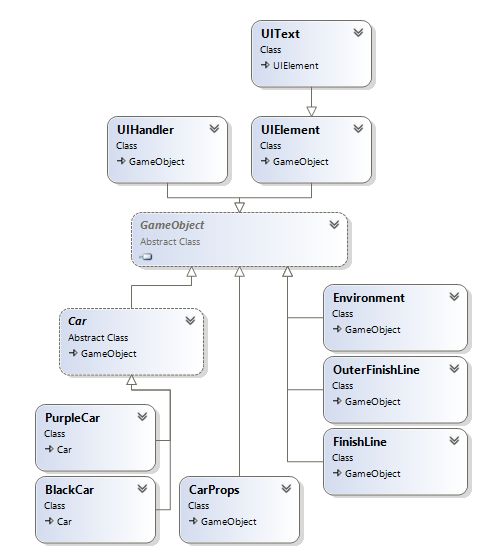


Рисунок 2.3 – Схема иерархии наследования от класса GameObject

От основных объектов проекта далее наследуются другие игровые объекты, реализующие необходимый полный функционал приложения. В частности, объекты Car и Environment содержат в себе необходимую логику для наличия контролируемых игроками гоночных автомобилей и трассы соответственно.

Класс Car является экземплярным классом-наследником класса GameObject. Данный класс представляет собой основной класс игрока и содержит определения для осуществления работы основных и дополнительных механик игры: осуществление обработки пользовательского ввода, подсчёт количества пройденных кругов, расход топлива, разрешение коллизий и пр.

Описание класса Car:

* поле properties содержит ссылку на класс, содержащий характеристики автомобиля;
* поле fuelLevel хранит текущий уровень топлива в автомобиле;
* поле lapsPassed определяет количество пройденных автомобилем кругов;
* метод FixedUpdate() вызывается каждый раз при обновлении состояния игровых объектов и производит обновление состояния автомобиля;
* метод GetUserInput() принимает и обрабатывает пользовательский ввод для управления автомобилем;

Графический компонент, представленный классом SpriteRenderer, отвечает за полный цикл отрисовки спрайтовой графики. Отрисовка графики происходит посредством оперирования классом Sprite, который определяет графический объект спрайта. Каждый спрайт задаётся шириной и высотой, масштабом, номером слоя, на котором он размещается относительно других спрайтов, а также спрайт имеет свою собственную позицию, значение смещения по позиции, угол поворота и точку поворота. Основная же графическая информация спрайта хранится в поле класса Texture2D, в котором определяется идентификатор сформированной текстуры данного спрайта, директорию хранения спрайта, а также высоту и ширину текстуры. Коллекция спрайтов, которая определяется для хранения нескольких спрайтов одного объекта, представляется классом SpriteBatch.

Компонент анимации, представленный классом Animator, построен на основных принципах работы класса SpriteRenderer. Он также имеет коллекцию спрайтов типа SpriteBatch для отрисовки, однако основной его функцией является периодическая смена текущего спрайта на следующий спрайт в коллекции.

Физический компонент представляется классом RigidBody2D. Данный класс содержит в себе необходимые поля для хранения как текущего значения параметра скорости твёрдого тела, так и констант: массы твёрдого тела, коэффициента сопротивления, коэффициента торможения, силы тяги. Также класс RigidBody2D содержит единственный метод, предназначенный для обновления текущего значения скорости твёрдого тела в зависимости от внешних условий.

## 2.3 Паттерны проектирования

Огромное множество современных приложений, в том числе и игровых, зачастую строится на основе схожих основных алгоритмов работы. Как итог, для реализации подобных задач существуют готовые оптимальные в той или иной ситуации решения, называемые шаблонами или паттернами проектирования. Паттерны проектирования являются многократно применяемым решением проблемы в рамках определённого контекста. В данном случае некоторые из них наиболее эффективно подходят для решения задач, связанных с реализацией игровых механик.

Паттерн «Декоратор» в данном приложении используется для реализации возможности динамического изменения характеристик гоночного автомобиля игрока. Для реализации данного паттерна согласно [4] сперва создаётся абстрактный класс-декоратор, который в качестве базового класса имеет тот же класс, что и декорируемые им объекты.

В данном проекте реализация декоратора определяется в классе CarPropsDecorator, который является наследником класса CarProps (Racing.Objects), содержащего определение характеристик гоночного автомобиля. Данный класс хранит в себе ссылку на декорируемый объект типа CarProps, а также встроенный таймер времени жизни декорации и метод для снятия декорации RemoveDecorator().

Для расширения класса декоратора дополнительной функциональностью, обусловленной особенностями игрового процесса: улучшение и ухудшение характеристик автомобиля – созданы конкретные классы BoostProps и SlowdownProps, являющиеся наследниками класса CarPropsDecorator.

Схема реализации паттерна «Декоратор» представлена на рисунке 2.4.

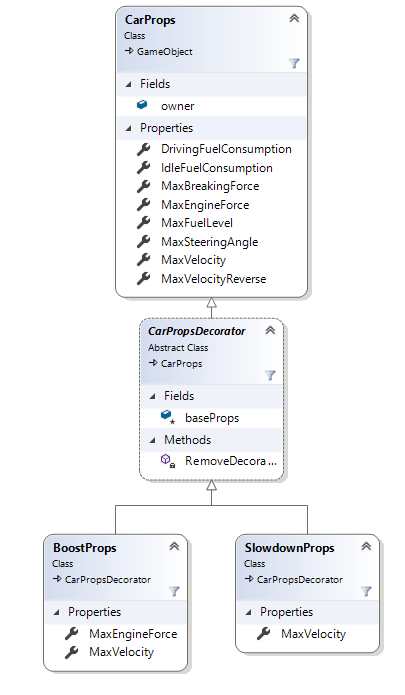


Рисунок 2.4 – Схема реализации паттерна «Декоратор»

Паттерн «Фабричный метод» используется для многократного создания призов на игровом поле.

Для реализации паттерна «Фабричного метод» согласно [5] сперва производится описание абстрактного класса приза Prize (Racing.Prizes), классы-наследники которого будут представлять необходимую реализацию класса приза. Для текущего игрового приложения создаётся 3 класса-наследника класса Prize: BoostPrize, FuelPrize и SlowdownPrize. Каждый из перечисленных конкретных классов представляет реализацию призов ускорения, топлива и замедления, экземпляры которых необходимо неоднократно создавать в процессе работы приложения.

В данном проекте паттерн «Фабричный метод» применяется в своём упрощённом виде. Для реализации шаблона проектирования данного типа сначала создаётся интерфейс IPrizeFactory, который является необходимым для создания объектов призов. Данный интерфейс содержит в себе определение фабричного метода GeneratePrize(), возвращающего объект типа Prize.

Реализация интерфейса IPrizeFactory происходит в конкретном классе PrizeGenerator (Racing.Prizes), который определяет свою реализацию метода GeneratePrize(). Метод GeneratePrize() класса PrizeGenerator возвращает конкретный определённый тип приза на основании использования алгоритма генерации случайных чисел.

Схема реализации паттерна «Фабричный метод» приведена в приложениии Н.

# 3 КОМПОНЕНТЫ ПРОГРАММНОГО КОМПЛЕКСА

## 3.1 Окно запуска приложения

Запуск игрового приложения осуществляется при помощи предшествующего ему окна запуска, которое реализуется при помощи технологии Windows Forms Application.

Библиотека OpenTK поставляется со своей собственной системой окон для использования функций OpenGL, однако не имеет встроенных функций вывода текста в этих окнах согласно [6].

Класс GameWindow, описывающий основное игровое окно располагается в пространстве имён OpenTK данной библиотеки, которая подключается к проекту файлом OpenTK.dll внутри Visual Studio 2019 при помощи официального менеджера пакетов NuGet для .NET.

## 3.2 Игровой двигатель

В текущем приложении реализация игрового двигателя представлена классом EngineCore, размещённом в пространстве имён RGEngine. Класс EngineCore является базовым окном, т.е. наследником класса GameWindow (OpenTK), который содержит в себе базовую реализацию необходимых методов для создания и отрисовки игрового окна OpenGL, а также поддержки пользовательского ввода и загрузки дополнительных ресурсов.

Класс EngineCore содержит в себе статический список игровых объектов типа GameObject, которых хранит в себе все текущие объекты на сцене. Статическим полем так же задана величина времени фиксированного обновления состояний игровых объектов сцены deltaTimeFixedUpdate типа double.

По нажатию на кнопку в диалоговом окне запуска приложения происходит создание экземпляра класса EngineCore, добавление в него игровых объектов, настройка игрового окна и вызов метода Run() класса GameWindow (OpenTK), который запускает основной игровой цикл согласно [6].

Метод OnLoad() в данном классе вызывается один раз при входе в метод Run() и производит основную первичную настройку игрового окна: очищает пространство для отрисовки кадра и устанавливает игровую камеру при помощи вызова метода Camera.SetView (RGEngine).

Далее вызывается метод OnUpdateFrame(), который производит подготовку игровых объектов сцены к отрисовке путём обновления их состояний и состояний их компонентов, а также обновляет состояние объекта, который отвечает за хранение значений пользовательского ввода в данный момент времени. Внутри OnUpdateFrame() происходит вызов методов PerformFixedUpdate() и PerformUpdate() для каждого объекта GameObject сцены.

После подготовки к отрисовке кадра происходит непосредственно создание одного нового кадра в методе OnRenderFrame(). В данном методе происходит вызов статического метода RenderEntireFrame() класса-отрисовщика SpriteRenderer (RGEngine.Graphics). Метод RenderEntireFrame() производит размещение спрайтов игровых объектов по необходимым позициям на экране, обеспечивает их поворот относительно центра экрана, а также обеспечивает корректность масштаба отрисовки спрайтовой графики. После создания кадра в методе OnRenderFrame() происходит вызов метода SwapBuffers() класса GameWindow (OpenTK), который производит смену «заднего» буфера на «передний», то есть выводит новый кадр из памяти непосредственно на экран.

В процессе работы программы данные методы вызываются в представленном порядке циклически до момента закрытия приложения, согласно [7, с. 339]. Схема данного цикла представлена на рисунке 3.1.

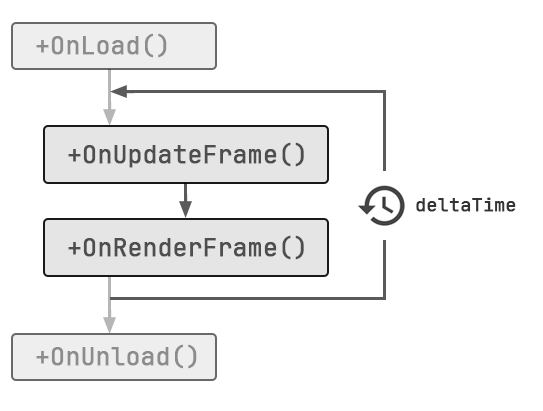


Рисунок 3.1 – Схема работы игрового цикла

## 3.3 Элементы управления автомобилем

Каждый из игроков имеет возможность управлять своим гоночным автомобилем с клавиатуры. Для каждого игрока предусмотрен отдельный набор клавиш, который описывается в окне запуска игры. На одного игрока приходится по 6 клавиш: 4 клавиши для непосредственно управления передвижением автомобиля в пространстве и 2 клавиши для переключения режимов коробки передач гоночного автомобиля.

Элементы управления автомобилем:

– «Газ» обеспечивает приложение постоянной силы тяги к автомобилю;

– «Тормоз» обеспечивает приложение к автомобилю постоянной силы сопротивления движению;

– «Поворот влево» обеспечивает выворот ведущих колёс на определённый угол влево относительно передней части автомобиля;

– «Поворот вправо» обеспечивает выворот ведущих колёс на определённый угол вправо относительно передней части автомобиля.

Переключение режимов коробки передач автомобиля:

– в режиме «Drive» крутящий момент двигателя автомобиля передаётся на ведущие колёса в прямом направлении;

– в режиме «Reverse» крутящий момент двигателя автомобиля передаётся на ведущие колёса в обратном направлении.

Для реализации управления игровыми объектами c клавиатуры были использованы классы KeyboardState и Keyboard, описанные в пространстве имен OpenTK.Input библиотеки OpenTK. Данные о состоянии клавиатуры, хранящиеся в свойстве CurrentKeyboardState класса InputController (RGEngine.Input), обновляются в методе Update() того же класса с каждой следующей итерацией основного игрового цикла.

## 3.4Физика движения автомобиля

Так как главным игровым объектом в данном проекте является гоночный автомобиль, то простое свободное перемещение его на плоскости с линейной скоростью является недостаточным. При разработке модели движения автомобиля по поверхности трассы одной из основных задач являлось описание корректного физического поведения автомобиля. В частности, была необходимость в написании кода для физически правильного разгона и торможения автомобиля.

За физические расчёты в данном проекте отвечает класс RigidBody2D, который является наследником класса Component. Этот класс описывает поведение физического твёрдого тела, имеющего конкретную массу, скорость и коэффициент сцепления с другими поверхностями.

В методе PerformComponent() производится оперирование исходными данными и входным параметром (количеством пройденного времени) с целью обновления значения скорости твёрдого тела в данный момент времени.

Для в основе алгоритма для метода PerformComponent() лежат второй и третий законы Ньютона. Приложенная к автомобилю сила складывается из силы тяги двигателя автомобиля, силы торможения автомобиля в данный момент времени и силы сопротивления движению. Получение значения величины ускорения автомобиля производится путём деления приложенной силы на массу автомобиля. Обновление значения скорости автомобиля производится путём сложения предыдущего значения скорости с величиной изменения скорости за данный промежуток времени.

## 3.5 Механизм поворота автомобиля

Для реализации поворота автомобиля недостаточно простого вращения спрайта автомобиля относительно его центра. Угол поворота спрайта автомобиля присваивается спрайту перед отрисовкой на экран в методе Update() класса SpriteRenderer (RGEngine.Graphics) на основе положения физической модели шасси автомобиля в пространстве сцены. Схема модель шасси автомобиля в данном проекте имеет вид, представленный на рисунке В.1.

Согласно данной модели, автомобиль имеет 2 оси, на каждую из которых приходится по одному колесу. Заднее колесо (backWheel) имеет фиксацию относительно задней оси автомобиля, переднее (frontWheel) – имеет возможность поворота на определённый максимальный угол steeringAngle. Центр шасси автомобиля равноудалён от переднего и заднего колёс и располагается на оси, проходящей через оба колеса. Центр автомобиля хранится в свойстве Position класса-родителя GameObject.

Расстояние между колёсами автомобиля называется колёсной базой и задается полем wheelBase в абстрактном классе Car, описывающем автомобиль. При движении автомобиля каждое колесо должно перемещаться в пространстве на некоторое расстояние в направлении, указанном нормалью к оси вращения колеса. Причём имеется необходимость в сохранении значения величины колёсной базы при движении автомобиля. Изменение описанных параметров отображено при помощи схемы, представленной на рисунке В.2.

Направление движения заднего колеса совпадает с направлением, в которое обращена передняя часть автомобиля и определена углом carDirectionAngle. Переднее же колесо не всегда направлено в ту же сторону, поэтому в расчётах положения автомобиля на сцене необходимо учитывать собственный угол поворота переднего колеса.

Новое положение центра шасси автомобиля рассчитывается как среднее арифметическое координат положения переднего и заднего колёс. Новый угол поворота (направления движения) автомобиля, который задаётся полем carDirectionAngle в классе Car, рассчитывается как арктангенс от величин смещения переднего и заднего колёс относительно их предыдущих позиций в соответствии с рисунком В.3.

## 3.6Коллизии

Для того чтобы внедрить в игровое приложение какие-либо ограничения по перемещению игровых объектов на сцене, связанные с особенностями рельефа или расположением непроходимых объектов на практике используются коллайдеры. Коллайдер является объектом, который наделяет твёрдое физическое тело особыми свойствами: реагирование на столкновения твёрдого тела с другими телами или на иные внешние силы.

В данном проекте применяется подход, при котором производится описание полигонального коллайдера, способного выступать в роли статического или динамического типа коллайдеров. Это достигается за счёт того, что на основной класс коллайдера PolyCollider (RGEngine.Physics) приходится реализация методов создания и обновления ограничивающего многоугольника, имеющего свою собственную позицию в пространстве, и методов обнаружения возможных коллизий. Остальная часть работы по разрешению возможных коллизий выносится во внешние классы, что позволяет не только достаточно гибко настраивать реакцию игровых объектов на столкновения, но и повторно использовать код, описывающий данный коллайдер.

Класс PolyCollider содержит в себе массив вершин ограничивающего многоугольника, который в данном проекте представлен прямоугольником, ссылку на GameObject, к которому данный коллайдер прикреплён, а также размер ограничивающего прямоугольника, внешний вид которого (выделены чёрным цветом поверх спрайтов объектов) представлен на рисунке 3.2. В основном методе класса PolyCollider DetectCollision(), определяющем коллизии, происходит обновление координат вершин ограничивающего прямоугольника относительно положения его центра, далее осуществляется поворот ограничивающего прямоугольника относительно центра системы экранных координат. Затем в цикле согласно [8, с. 370] происходит поиск оси разделения между двумя потенциально соприкасающимися коллайдерами путём перебора всех рёбер двух ограничивающих прямоугольников с целью нахождения максимальной и минимальной отрицательных нормалей по каждой опорной точке.



Рисунок 3.2 – Внешний вид ограничивающих прямоугольников

При фактическом пересечении двух коллайдеров детектируется коллизия, вызывающая метод OnColliderTriggered(). Данный метод поднимает событие ColliderTriggered, передавая в качестве аргументов CollisionEventArgs два пересекающихся объекта типа GameObject. Впоследствии при необходимости происходит обработка возникшей коллизии с дальнейшим её разрешением.

Для объекта автомобиля в данном проекте разрешение коллизий с другими твёрдыми телами сцены происходит в методе FixedUpdate() класса Car (Racing.Objects). Разрешение осуществляется путём возврата автомобиля на предыдущую позицию, которая являлась текущей в момент до обнаружения столкновения. Также необходимо отметить, что при столкновении скорость автомобиля уменьшается и меняет своё направление, благодаря чему удаётся создать информативное для пользователя визуальное отображение соударения с непроходимым объектом.

При столкновении автомобиля с другим автомобилем на сцене разрешения коллизий не происходит. Данное упрощение связано с особенностями игрового процесса.

## 3.7 Внутриигровой интерфейс

Ввиду отсутствия готовых реализаций для отображения текстовой информации внутри библиотеки OpenTK согласно [6] вывод информации о состоянии автомобилей каждого из игроков выполняется путём отрисовки вручную созданных спрайтов текстовых глифов.

Базовым классом для отображения интерфейса является UIElement, наследуемый от GameObject. Данный класс содержит в себе компонент SpriteRenderer, который обеспечивает отрисовку элемента. Текстовый интерфейс реализуется в классе UIText, который является классом-наследником UIElement и хранит в себе статический список текстовых текстур, загружаемых в память при вызове статического конструктора.

Основной метод ParseStringToSprites() сперва производит сопоставление входящего значения строки с имеющимися текстурами текстовых глифов, а затем создает и возвращает готовый список текстовых спрайтов, размещённых в одну строку, обращаясь к списку хранимых текстур и к позиции элемента интерфейса на экране.

С целью более удобного расположения каждого тестового элемента был выбран моноширинный шрифт, наиболее подходящий по стилистике к игре.

Размещением текстовых элементов интерфейса занимается класс UIHandler, который в своём конструкторе сперва задаёт положение каждого из элементов на экране, а затем производит обновление отображаемой информации в методе FixedUpdate().

## 3.8Игровые механизмы

Механизм работы призов в данном игровом приложении позволяет динамически изменять характеристики гоночных автомобилей игроков.

Генерация призов на трассе осуществляется случайным образом. С момента запуска игры происходит запуск внутриигрового таймера GameTimer (RGEngine.Support) внутри класса-контроллера PrizeGenerator (Racing.Prizes), который через фиксированный промежуток времени вызывает событие Elapsed, метод-обработчик AddPrizeOnScene() которого производит добавление нового приза на сцену посредством метода GeneratePrize().

Фабричный метод GeneratePrize() класса PrizeGenerator возвращает приз одного из трёх типов на основе получения случайного натурального числа из метода Next(). Однако данному фабричному методу предшествует вызов метода GetPrizePosition(), который возвращает случайным образом сгенерированную позицию в пределах трассы для размещения нового приза. Алгоритм выбора случайной позиции основывается также на основе получения случайных натуральных чисел, однако перед генерацией непосредственно самих значений координат происходит выбор случайной области (0-3) для размещения приза, по позиции граничных точек которой с учётом необходимых отступов производится случайное размещение приза.

Активация приза осуществляется посредством детектирования коллизии между автомобилем и объектом приза. При возникновении такой коллизии происходит обработка события ColliderTriggered методом Prize\_ColliderTriggered() класса Car. Далее приз удаляется со сцены вызовом метода Remove() абстрактного класса Prize, и происходит вызов метода ApplyPrize(), реализация которого находится в каждом экземплярном классе-наследнике абстрактного класса Prize. Метод ApplyPrize() обеспечивает наложение декоратора на свойства CarProps автомобиля либо изменение некоторых параметров автомобиля напрямую.

Декорирование характеристик автомобиля ввиду особенностей игрового процесса производится на непродолжительный промежуток времени, по истечению которого декорация снимается. По этой причине после наложения декоратора на свойства автомобиля происходит запуск внутриигрового таймера GameTimer (RGEngine.Support) внутри класса CarPropsDecorator (Racing.Objects), который обновляется посредством вызова метода Update().По истечению времени осуществляется вызов события Elapsed, которое сигнализирует о необходимости снятия декорации с объекта, хранящего характеристики автомобиля. Метод-обработчик события Elapsed RemoveDecorator() класса CarPropsDecorator производит возвращение характеристик автомобиля к базовым значениям.

Следует также отметить, что в виду особенностей игрового процесса при подборе нескольких призов, их эффект не суммируется, а действующим призом считается последний подобранный.

Подсчёт количества пройденных игроком кругов осуществляется путём установления отметки true в ячейку массива пройденных кругов laps типа bool. Данный массив хранится в классе Car (Racing.Objects) и хранит информацию о пройденных кругов индивидуально для каждого игрока. Отметка о каждом пройденном круге ставится после того, как автомобиль игрока полностью проезжает коллайдер финишной черты и касается о выходной финишный коллайдер, в ходе чего вызывается событие ColliderTriggered, метод-обработчик FinishLine\_ColliderTriggered() которого проверяет правильность положения автомобиля относительно финишной черты и производит проверку условия выполнения цели игры по проезду 5 кругов.

В момент создания объекта автомобиля в его конструкторе полю fuelLevel класса Car присваивается значение свойства MaxFuelLevel класса характеристик автомобиля CarProps (Racing.Objects). Свойство MaxFuelLevel описывает максимальную вместимость топливного бака автомобиля. Расход топлива автомобиля варьируется в зависимости от скорости его движения, а его потребление происходит при каждом вызове метода ApplyFuelConsumption() внутри FixedUpdate(). В данном методе сперва производится проверка наличия топлива в баке, затем происходит непосредственно изменение значения поля fuelLevel.

В случае, если уровень топлива оказывается меньше, чем 0,001 условных единиц, то происходит вызов события EndedRace с передачей в качестве параметра GameEventArgs ссылки на объект текущего автомобиля с предварительным изменением текущего id автомобиля на строковый id автомобиля-победителя.

## 3.9 Модульные тесты

Модульное тестирование приложения производится по компонентам программного обеспечения. На данном этапе проверяется правильность работы реализаций необходимых для работы приложения алгоритмов.

Для тестирования создаётся дополнительный проект UnitTesting, в котором содержатся основные тестовые методы. Данные методы размещаются в классах:

– UnitTestClassInstantiating;

– UnitTestDecorator.

Ниже приводится описание данных методов:

– тестовый метод TestInstantiatingCarProps() производит тестирование правильности инициализации полей экземпляра класса CarProps;

– тестовый метод Decorate\_CarProps\_BoostProps() производит тестирование правильности наложения декоратора BoostCarProps на экземпляр класса CarProps;

– тестовый метод Decorate\_CarProps\_SlowdownProps() производит тестирование правильности наложения декоратора SlowdownProps на экземпляр класса CarProps.

Успешное выполнение описанных модульных тестов, результаты которого представлены на рисунке 3.3, свидетельствует о корректности работы тестируемых методов. Далее проводится основное тестирование путём опытного использования приложения, которое позволяет выявлять различного рода ошибки компоновки методов и недостатки балансировки свойств игровых объектов. Листинг модульных тестов представлен в приложении Б.

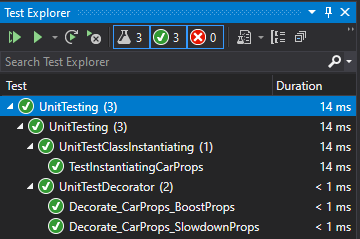


Рисунок 3.3 – Результат успешного выполнения модульных тестов

## 3.10 Результаты опытной эксплуатации игрового приложения

При запуске приложения отображается загрузочное окно Windows Forms, внешний вид которого изображён на рисунке Г.1.

Сразу же после ознакомления пользователя с тематикой игры по внешнему виду обложки и названию пользователь может начать игру по нажатию кнопки «Start». Далее открывается диалоговое окно с инструкцией к приложению, в котором отображается цель игры, раскладка клавиш управления и краткое описание игровых призов. Данное диалоговое окно изображено на рисунке Г.2.

По нажатию на кнопку «Go» диалогового окна запускается непосредственно само игровое окно приложения, которое имеет вид, представленный на рисунке Г.3.

Игроки имеют возможность управлять своим гоночным автомобилем. Для первого игрока предоставляется автомобиль чёрного цвета, для второго – сиреневого. Каждому игроку в начале гонки выдаётся по 35 единиц топлива, которого не хватит для того, чтобы завершить все 5 кругов заезда. При каждом проезде круга, пересекая финишную черту, игроку прибавляется пройденный круг.

Автомобиль игрока не может выехать за пределы гоночной трассы, и при попытке соударения с границей трассы произведёт отскок, величина которого зависит от скорости движения автомобиля в данный момент времени.

В ходе игры на гоночной трассе появляются призы через каждые 5 секунд. Максимальное количество призов, одновременно находящихся на трассе, равняется 5. Призы создаются случайным образом в случайной позиции в пределах гоночной трассы. Внешний вид призов представлен на рисунке 3.4.

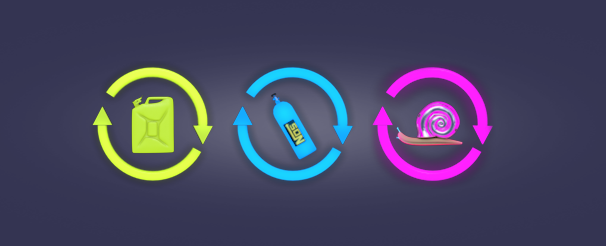


Рисунок 3.4 – Внешний вид призов

Также данные призы имеют заметную характерную анимацию для более просто обнаружения пользователем приза.

Приз «Топливо» при подборе игроком добавляет ему 7 единиц топлива.

Приз «Ускорение» при подборе игроком ускоряет его автомобиль, увеличивая максимальную скорость движения автомобиля в полтора раза и увеличивая мощность двигателя автомобиля в два раза, что характерным образом сказывается на величине ускорения автомобиля игрока.

Приз «Замедление» при подборе игроком замедляет его автомобиль, уменьшая максимальную скорость его движения в полтора раза, однако изменяя при этом показатели мощности двигателя автомобиля.

В случае, если запас топлива у одного из игроков иссякает раньше, чем у другого, то победа присуждается игроку, у которого осталось больше топлива. При победе одного из игроков на экране отображается победное изображение автомобиля-победителя с подписью. Внешний вид победного изображения представлен на рисунке Г.4.

# ЗАКЛЮЧЕНИЕ

Результатом законченного курсового проекта является аркадная гоночная 2D-игра с видом сверху в жанре «Кольцевые гонки» для операционной системы Microsoft Windows. Игровое приложение реализовано при помощи .NET технологии Windows Forms Application и графической библиотеки-обёртки OpenTK для C#.

После анализа поставленной задачи и формирования технического задания была спроектирована архитектура игрового приложения, которая вбирает в себя логику все аспекты логики работы обобщённых алгоритмов отрисовки графики, обработки пользовательского ввода, обсчёта физических и геометрических моделей объектов. Было создано множество схем работы приложения при различных сценариях поведения объектов. Были построены иерархии различных классов, используемых в приложении.

Также были реализованы такие паттерны проектирования, как «Фабричный метод» и «Декоратор» с целью упрощения внедрения основных игровых механизмов. Благодаря такому подходу данное приложение имеет возможность расширения своего функционала. В частности, для добавления в игру новых призов будут создаваться соответствующие классы призов без необходимости изменения структуры базовых классов приложения.

В процессе создания курсового проекта были реализованы компоненты, отвечающие за обработку игровой физики, а также за отрисовку графики на экран; были проработаны классы, осуществляющие обработку пользовательского ввода посредством использования методов из библиотеки OpenTK; разработаны классы со вспомогательным функционалом (собственный класс игрового таймера, класс игровой камеры, класс для загрузки текстур из файлов, классы для отображения цифровой информации на экране и др.).

Для оптимизации игрового процесса была произведена многоэтапная балансировка характеристик описательных полей классов объектов сцены путём тщательного тестирования приложения.

Эксклюзивно создавался уникальный графический материал, используемый в приложении: текстовые элементы, элементы интерфейса, спрайты автомобилей, спрайты игровых призов, спрайты анимации призов, гоночная трасса, заставка экрана окончания игры, изображение загрузочного окна, изображение инструкции для диалогового окна.

Выполненная курсовая работа, проверенная в системе «Антиплагиат», имеет оригинальность, соответствующую значению 97,21%, и является уникальной. Цитирования обозначены ссылками на публикации, указанные в «Списке использованных источников».

# Список использованных источников

1. OpenGL Wiki // Khronos [Electronic resource]. – Режим доступа: https://www.khronos.org/opengl/wiki/FAQ#What\_is\_OpenGL.3F. – Дата доступа: 09.05.2020.
2. Getting started with DirectX Graphics // Microsoft [Electronic resource]. – 2018. – Режим доступа: https://docs.microsoft.com/en-us/windows/win32/getting-started-with-directx-graphics. – Дата доступа: 09.05.2020.
3. Nystrom, R. Game Programming Patterns. – first edition. – Genever Benning, 2014. – 354 p.
4. Паттерн проектирования «Декоратор» // Metanit.com [Электронный ресурс]. – 2015. – Режим доступа: https://metanit.com/sharp/patterns/4.1.php. – Дата доступа: 09.05.2020.
5. Паттерн проектирования «Фабричный метод» // Metanit.com [Электронный ресурс]. – 2015. – Режим доступа: https://metanit.com/sharp/patterns/2.1.php. – Дата доступа: 09.05.2020.
6. Perks, D. LearnOpenTK // OpenTK [Electronic resource]. – Режим доступа: https://opentk.net/learn/index.html. – Дата доступа: 09.05.2020.
7. Gregory, J. Game Engine Architecture. – third edition. – New York : CRC Press, 2018. – 1240 p.
8. Sung, K., Pavleas, J., Arnez, F., Pace, J. Build your own 2D Game Engine and Create Great Web Games. – first edition. – Apress, 2015. – 468 p.

# ПРИЛОЖЕНИЕ А

(обязательное)

**Листинг программы**

**Листинги проекта RGEngine:**

**Листинг класса Component.cs:**

namespace RGEngine.BaseClasses

{

/// <summary>

/// Base class for everything attached to GameObjects.

/// </summary>

public abstract class Component

{

/// <summary>

/// The game object this component is attached to.

/// </summary>

internal readonly GameObject attachedTo;

/// <summary>

/// Switches ON and OFF the current component.

/// </summary>

public bool IsEnabled { get; set; }

internal Component(GameObject gameObject)

{

this.attachedTo = gameObject;

IsEnabled = true;

}

/// <summary>

/// Method that allows to use the Component potential.

/// </summary>

/// <param name="deltaTime"></param>

internal abstract void PerformComponent(double deltaTime);

internal virtual void InitializeComponent() { }

}

/// <summary>

/// Defines an object update type.

/// </summary>

internal interface IFixedUpdatable { }

}

**Листинг класса GameObject.cs:**

namespace RGEngine.BaseClasses

{

/// <summary>

/// Base class for all entities in RGEngine scenes.

/// </summary>

public abstract class GameObject

{

/// <summary>

/// Stores a list of the components for Update().

/// </summary>

internal readonly List<Component> components = new List<Component>();

/// <summary>

/// Keeps current position of GameObject.

/// </summary>

public Vector2 Position { get; set; }

/// <summary>

/// Stores a GameObjectCollider.

/// </summary>

public PolyCollider collider;

private float rotation;

/// <summary>

/// Keeps current rotation of GameObject.

/// </summary>

public float Rotation

{

get => this.rotation;

set

{

this.rotation = value;

OnRotationChanged(new EventArgs());

}

}

/// <summary>

/// Event that raised when rotation occured.

/// </summary>

public event EventHandler RotationChanged;

/// <summary>

/// Method that invokes a RotationChanged ivent.

/// </summary>

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

protected virtual void OnRotationChanged(EventArgs e)

{

var handler = RotationChanged;

handler?.Invoke(this, e);

}

/// <summary>

/// Adds a Component attached to a GameObject.

/// </summary>

/// <param name="componentName"></param>

/// <returns></returns>

public Component AddComponent(string componentName)

{

Component newComponent = null;

switch(componentName)

{

case "SpriteRenderer":

newComponent = new SpriteRenderer(this);

break;

case "RigidBody2D":

newComponent = new RigidBody2D(this);

break;

case "Animator":

newComponent = new Animator(this);

break;

}

components.Add(newComponent);

return newComponent;

}

/// <summary>

/// Returns a Component attached to a GameObject.

/// </summary>

/// <typeparam name="T"></typeparam>

/// <returns></returns>

public T GetComponent<T>() where T : Component

{

foreach (var component in components)

{

var @return = component as T;

if (@return != null && @return.GetType() == typeof(T))

return @return;

}

return null;

}

/// <summary>

/// Runs once per frame. Independent from Game Physics.

/// Uses a variable time-step.

/// </summary>

public virtual void Update(double deltaTime) { }

/// <summary>

/// Updates everything that is need to be applied to a Rigidbody.

/// Uses a fixed time-step.

/// </summary>

public virtual void FixedUpdate(double fixedDeltaTime) { }

/// <summary>

/// Updates a game object and it's components

/// over a variable (render) delta time interval.

/// </summary>

/// <param name="deltaTime"></param>

internal void PerformUpdate(double deltaTime)

{

Update(deltaTime);

foreach (var component in components)

{

if (!(component is IFixedUpdatable))

component.PerformComponent(deltaTime);

}

}

/// <summary>

/// Updates a game object and it's components

/// over a fixed delta time interval.

/// </summary>

/// <param name="fixedDeltaTime"></param>

internal void PerformFixedUpdate(double fixedDeltaTime)

{

FixedUpdate(fixedDeltaTime);

foreach (var component in components)

{

if (component is IFixedUpdatable)

component.PerformComponent(fixedDeltaTime);

}

}

}

}

**Листинг класса Animator.cs:**

namespace RGEngine.Graphics

{

/// <summary>

/// Defines an animation component.

/// </summary>

public sealed class Animator : Component

{

/// <summary>

/// Default ctor.

/// </summary>

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

public Animator(GameObject gameObject)

: base(gameObject)

{

AnimationSprites = new SpriteBatch();

AnimationSprites.BatchUpdated += AnimationSprites\_OnBatchUpdated;

}

private void AnimationSprites\_OnBatchUpdated(object sender, EventArgs e)

{

if (AnimationSprites.Quantity > 0)

{

ienumerator = AnimationSprites.GetEnumerator();

ienumerator.MoveNext();

}

}

/// <summary>

/// Stores all sprites for animation.

/// </summary>

public SpriteBatch AnimationSprites { get; set; }

/// <summary>

/// Defines Frames Per Second rate for animation.

/// </summary>

public int FPS { get; set; }

private double frameLiveTime;

private IEnumerator<Sprite> ienumerator;

internal override void PerformComponent(double deltaTime)

{

var delay = 1f / FPS;

if (ienumerator != null)

{

base.attachedTo.GetComponent<SpriteRenderer>().RenderQueue.RemoveSprite(ienumerator.Current.Texture.Path);

frameLiveTime += deltaTime;

if (frameLiveTime >= delay)

{

frameLiveTime = 0;

if (!ienumerator.MoveNext())

{

ienumerator.Reset();

ienumerator.MoveNext();

}

}

base.attachedTo.GetComponent<SpriteRenderer>().RenderQueue.AddSprite(ienumerator.Current);

}

}

}

}

**Листинг класса Sprite.cs:**

namespace RGEngine.Graphics

{

/// <summary>

/// Represents a Sprite object for use in 2D.

/// </summary>

public class Sprite : IComparable<Sprite>

{

/// <summary>

/// Stores a loaded 2D texture for sprite existence.

/// </summary>

public Texture2D Texture { get; set; }

/// <summary>

/// Sprite width.

/// </summary>

public int Width { get; }

/// <summary>

/// Sprite height.

/// </summary>

public int Height { get; }

/// <summary>

/// Sptire order in scene layers.

/// </summary>

public int ZIndex { get; set; }

/// <summary>

/// Scale of a sprite.

/// </summary>

public Vector2 Scale { get; }

/// <summary>

/// Position of a sprite.

/// </summary>

public Vector2 Position { get; set; }

/// <summary>

/// Offset over position of a sprite.

/// </summary>

public Vector2 Offset { get; set; }

/// <summary>

/// Sprite rotation.

/// </summary>

public float Rotation { get; set; }

/// <summary>

/// Sprite's point of rotation.

/// </summary>

public Vector2 PointOfRotation { get; set; }

/// <summary>

/// Default ctor.

/// </summary>

/// <param name="texture"></param>

public Sprite(Texture2D texture)

{

this.Texture = texture;

this.Width = texture.Width;

this.Height = texture.Height;

this.Scale = new Vector2(0.3f, 0.3f);

this.Offset = Vector2.Zero;

this.ZIndex = 0;

}

/// <summary>

/// Ctor that tunes the sprite in more details.

/// </summary>

/// <param name="texture"></param>

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

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

/// <param name="offset"></param>

/// <param name="zIndex"></param>

public Sprite(Texture2D texture, Vector2 position, Vector2 scale, Vector2 offset, int zIndex)

{

this.Texture = texture;

this.Width = texture.Width;

this.Height = texture.Height;

this.Position = position;

this.Scale = scale;

this.Offset = offset;

this.ZIndex = zIndex;

}

/// <summary>

/// Compares Sprites by Z-index.

/// </summary>

/// <param name="otherSprite"></param>

/// <returns></returns>

public int CompareTo(Sprite otherSprite)

{

if (this.ZIndex > otherSprite.ZIndex)

return 1;

else if (this.ZIndex < otherSprite.ZIndex)

return -1;

return 0;

}

}

}

**Листинг класса SpriteBatch.cs:**

namespace RGEngine.Graphics

{

/// <summary>

/// Defines a class that stores a batch of sprites.

/// </summary>

public class SpriteBatch : IEnumerable<Sprite>

{

/// <summary>

/// Stores all sprites for some gameObject sorted in Z-index.

/// </summary>

private List<Sprite> sprites = new List<Sprite>();

/// <summary>

/// Indicates amount of sprites stored in sprite batch.

/// </summary>

public int Quantity => sprites.Count;

/// <summary>

/// Basic indexer.

/// </summary>

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

/// <returns></returns>

public Sprite this[int index]

{

get => sprites[index];

set => sprites[index] = value;

}

/// <summary>

/// Event that raised when current batch was updated.

/// </summary>

public event EventHandler BatchUpdated;

/// <summary>

/// Adds a sprite to a sprite batch.

/// </summary>

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

public void AddSprite(Sprite sprite)

{

sprites.Add(sprite);

sprites.Sort();

BatchUpdated?.Invoke(this, new EventArgs());

}

/// <summary>

/// Removes a sprite from a sprite batch.

/// </summary>

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

public void RemoveSprite(string name)

{

BatchUpdated?.Invoke(this, new EventArgs());

foreach (var sprite in sprites.ToList<Sprite>())

{

if (sprite.Texture.Path == name)

{

sprites.Remove(sprite);

}

}

}

/// <summary>

/// Returns created from textures SpriteBatch object.

/// </summary>

/// <param name="textures"></param>

/// <returns></returns>

public static SpriteBatch CreateSpriteBatch(params Texture2D[] textures)

{

if (textures.Length == 0)

return null;

var batch = new SpriteBatch();

for (int i = 0; i < textures.Length; i++)

{

batch.AddSprite(new Sprite(textures[i]));

}

return batch;

}

/// <summary>

/// Returns created from sprites SpriteBatch object.

/// </summary>

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

/// <returns></returns>

public static SpriteBatch CreateSpriteBatch(params Sprite[] sprites)

{

if (sprites.Length == 0)

return null;

var batch = new SpriteBatch();

for (int i = 0; i < sprites.Length; i++)

{

batch.AddSprite(sprites[i]);

}

return batch;

}

/// <summary>

/// IEnumerator implementation.

/// </summary>

/// <returns></returns>

public IEnumerator<Sprite> GetEnumerator()

{

return ((IEnumerable<Sprite>)sprites).GetEnumerator();

}

IEnumerator IEnumerable.GetEnumerator()

{

return ((IEnumerable<Sprite>)sprites).GetEnumerator();

}

}

}

**Листинг класса SpriteRenderer.cs:**

namespace RGEngine.Graphics

{

/// <summary>

/// Renders a Sprite for 2D graphics.

/// </summary>

public class SpriteRenderer : Component

{

/// <summary>

/// Enables blending textures with alpha channel.

/// </summary>

static SpriteRenderer()

{

GL.Enable(EnableCap.Texture2D);

GL.Enable(EnableCap.Blend);

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

}

/// <summary>

/// Default ctor.

/// </summary>

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

public SpriteRenderer(GameObject gameObject) :

base(gameObject)

{

RenderQueue = new SpriteBatch();

}

/// <summary>

/// Holds the queue of Sprite objects to render.

/// </summary>

public SpriteBatch RenderQueue { get; set; }

// Renders a single sprite object.

internal static void RenderSprite(Sprite sprite)

{

// Define and init the 4 vertices of a rect sprite.

var vertices = new Vector2[4]

{

new Vector2(0f, 0f),

new Vector2(1f, 0f),

new Vector2(1f, 1f),

new Vector2(0f, 1f)

};

GL.MatrixMode(MatrixMode.Modelview);

GL.PushMatrix();

var X = sprite.Position.X + sprite.PointOfRotation.X + sprite.Offset.X \* sprite.Scale.X;

var Y = sprite.Position.Y + sprite.PointOfRotation.Y + sprite.Offset.Y \* sprite.Scale.Y;

GL.Translate(X, Y, 0f);

GL.Rotate(sprite.Rotation, 0.0f, 0f, 1f);

GL.Translate(-X, -Y, 0f);

GL.BindTexture(TextureTarget.Texture2D, sprite.Texture.Id);

GL.Begin(PrimitiveType.Quads);

// Process each texture vertex.

for (int i = 0; i < vertices.Length; i++)

{

// Set current texture coords.

GL.TexCoord2(vertices[i]);

// 'Normalize' position.

vertices[i].X = sprite.Width \* (vertices[i].X - 0.5f);

vertices[i].Y = sprite.Height \* (vertices[i].Y - 0.5f);

vertices[i] \*= sprite.Scale;

// Translation of vertex.

vertices[i] += sprite.Position + sprite.Offset \* sprite.Scale;

GL.Vertex2(vertices[i]);

}

GL.End();

GL.PopMatrix();

}

/// <summary>

/// Renders a list of all the sprites from the scene.

/// </summary>

/// <param name="batch"></param>

internal static void RenderSpritesQueue(SpriteBatch batch)

{

for (int i = 0; i < batch.Quantity; i++)

{

SpriteRenderer.RenderSprite(batch[i]);

}

}

/// <summary>

/// Renders all objects from the scene.

/// </summary>

/// <param name="gameObjects"></param>

internal static void RenderEntireFrame(List<GameObject> gameObjects)

{

GL.Clear(ClearBufferMask.ColorBufferBit);

foreach (var gameObject in gameObjects)

{

if (!(gameObject is INonRenderable))

SpriteRenderer.RenderSpritesQueue(gameObject.GetComponent<SpriteRenderer>().RenderQueue);

}

}

/// <summary>

/// Updates sprite position and rotation state.

/// </summary>

/// <param name="deltaTime"></param>

internal override void PerformComponent(double deltaTime)

{

foreach (var sprite in RenderQueue)

{

sprite.Position = base.attachedTo.Position + sprite.Offset;

sprite.Rotation = base.attachedTo.Rotation;

}

}

}

/// <summary>

/// Interface that disables rendering for an object.

/// </summary>

public interface INonRenderable { }

}

**Листинг класса Texture2D.cs:**

namespace RGEngine.Graphics

{

/// <summary>

/// Describes a 2D texture.

/// </summary>

public class Texture2D

{

/// <summary>

/// Specifies a texture identifier.

/// </summary>

public int Id { get; }

/// <summary>

/// Texture's width.

/// </summary>

public int Width { get; }

/// <summary>

/// Texture's height.

/// </summary>

public int Height { get; }

/// <summary>

/// A path to a texture.

/// </summary>

public string Path { get; }

/// <summary>

/// Initializes a new instance of a Texture2D.

/// </summary>

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

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

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

/// <param name="path"></param>

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

{

this.Id = id;

this.Width = width;

this.Height = height;

this.Path = path;

}

}

}

**Листинг класса InputController.cs:**

namespace RGEngine.Input

{

/// <summary>

/// Describes a controller for user input.

/// </summary>

public static class InputController

{

/// <summary>

/// Stores current state of Keyboard object.

/// </summary>

public static KeyboardState CurrentKeyboardState { get; private set; }

/// <summary>

/// Updates CurrentKeyboardState value.

/// </summary>

public static void Update()

{

CurrentKeyboardState = Keyboard.GetState();

}

}

}

**Листинг класса PolyCollider.cs:**

namespace RGEngine.Physics

{

/// <summary>

/// Describes a polygon collider for an object.

/// </summary>

public class PolyCollider

{

/// <summary>

/// Keeps the list of all objects that use collider.

/// </summary>

public static List<GameObject> allCollidersAttached = new List<GameObject>();

private Vector2[] vertices = new Vector2[4];

private GameObject attachedTo;

private Vector2 size;

/// <summary>

/// Basic ctor that instantiates SIZED collider to an object.

/// </summary>

/// <param name="object"></param>

/// <param name="size"></param>

public PolyCollider(GameObject @object, Vector2 size)

{

if (@object is ICollidable)

{

allCollidersAttached.Add(@object);

this.attachedTo = @object;

this.size = size;

Update(attachedTo.Position);

}

}

/// <summary>

/// Event that raised when collision occured.

/// </summary>

public event EventHandler<CollisionEventArgs> ColliderTriggered;

/// <summary>

/// Method that invokes a ColliderTriggered ivent.

/// </summary>

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

protected virtual void OnColliderTriggered(CollisionEventArgs e)

{

var handler = ColliderTriggered;

handler?.Invoke(this, e);

}

private void Update(Vector2 position)

{

var dx = size.X / 2f;

var dy = size.Y / 2f;

vertices[0] = new Vector2(position.X - dx, position.Y - dy);

vertices[1] = new Vector2(position.X - dx, position.Y + dy);

vertices[2] = new Vector2(position.X + dx, position.Y + dy);

vertices[3] = new Vector2(position.X + dx, position.Y - dy);

}

private Vector2 GetNormal(Vector2[] vertices, int startPoint)

{

int endPoint;

if (startPoint + 1 >= vertices.Length)

endPoint = 0;

else

endPoint = startPoint + 1;

var vertexA = vertices[startPoint];

var vertexB = vertices[endPoint];

var edge = new Vector2(vertexB.X - vertexA.X, vertexB.Y - vertexA.Y);

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

}

private Vector2 GetProjection(Vector2 normal)

{

var result = new Vector2();

bool isNull = true;

foreach (var vertex in this.vertices)

{

var projection = normal.X \* vertex.X + normal.Y \* vertex.Y;

if (isNull)

{

result = new Vector2(projection, projection);

isNull = false;

}

if (projection > result.X)

result.X = projection;

if (projection < result.Y)

result.Y = projection;

}

return result;

}

/// <summary>

/// Method that checks a collision between this and other collider.

/// </summary>

/// <param name="other"></param>

/// <returns></returns>

public bool DetectCollision(GameObject other)

{

PolyCollider otherCollider = other.collider;

Update(this.attachedTo.Position);

otherCollider.Update(otherCollider.attachedTo.Position);

Rotate(attachedTo.Rotation);

int allCount = this.vertices.Length + otherCollider.vertices.Length;

var allVertices = new Vector2[allCount];

this.vertices.CopyTo(allVertices, 0);

otherCollider.vertices.CopyTo(allVertices, this.vertices.Length);

Vector2 normal;

bool isCollide = false;

for (int i = 0; i < allCount && !isCollide; i++)

{

normal = GetNormal(allVertices, i);

var currentProjection = GetProjection(normal);

var otherProjection = otherCollider.GetProjection(normal);

if (currentProjection.X < otherProjection.Y ||

otherProjection.X < currentProjection.Y)

return false;

}

OnColliderTriggered(new CollisionEventArgs(this.attachedTo, other));

return true;

}

/// <summary>

/// Rotates collider by some amount of degrees.

/// </summary>

/// <param name="angleInDegrees"></param>

public void Rotate(float angleInDegrees)

{

var angle = MathHelper.DegreesToRadians(angleInDegrees);

Update(Vector2.Zero);

for (int i = 0; i < vertices.Length; i++)

{

var tmp = vertices[i];

tmp.X = (float)(vertices[i].X \* Math.Cos(angle) - vertices[i].Y \* Math.Sin(angle));

tmp.Y = (float)(vertices[i].X \* Math.Sin(angle) + vertices[i].Y \* Math.Cos(angle));

vertices[i] = tmp + this.attachedTo.Position;

}

}

/// <summary>

/// Draws a collider to the scene.

/// </summary>

public void Draw()

{

GL.Disable(EnableCap.Blend);

GL.MatrixMode(MatrixMode.Modelview);

GL.PushMatrix();

GL.Begin(PrimitiveType.LineLoop);

for (int i = 0; i < vertices.Length; i++)

{

GL.Vertex2(vertices[i]);

}

GL.End();

GL.PopMatrix();

GL.Enable(EnableCap.Blend);

}

}

/// <summary>

/// Interface for an object that supports collisions.

/// </summary>

public interface ICollidable { }

/// <summary>

/// Interface for an object that does not need collision resolution.

/// </summary>

public interface INonResolveable { }

/// <summary>

/// Event args for collision events.

/// </summary>

public class CollisionEventArgs : EventArgs

{

/// <summary>

/// First collided object.

/// </summary>

public readonly GameObject one;

/// <summary>

/// Second collided object.

/// </summary>

public readonly GameObject another;

/// <summary>

/// Basic ctor that init objects that collided.

/// </summary>

/// <param name="one"></param>

/// <param name="another"></param>

public CollisionEventArgs(GameObject one, GameObject another)

{

this.one = one;

this.another = another;

}

}

}

**Листинг класса RigidBody2D.cs:**

namespace RGEngine.Physics

{

/// <summary>

/// A physics component for 2D sprites.

/// </summary>

public sealed class RigidBody2D : Component, IFixedUpdatable

{

/// <summary>

/// Default ctor.

/// </summary>

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

public RigidBody2D(GameObject gameObject)

: base(gameObject)

{

}

/// <summary>

/// Responsible for object velocity.

/// </summary>

public float velocity;

/// <summary>

/// Defines object mass.

/// </summary>

public float mass;

/// <summary>

/// Defines a friction coefficient.

/// </summary>

public float frictionConst;

/// <summary>

/// Defines engine force.

/// </summary>

public float engineForce;

/// <summary>

/// Defines breaking force.

/// </summary>

public float breakingForce;

internal override void PerformComponent(double deltaTime)

{

var frictionForce = frictionConst \* velocity;

var attachedForce = engineForce - breakingForce - frictionForce;

float acceleration = attachedForce / mass;

velocity += acceleration \* (float)deltaTime;

}

}

}

**Листинг класса ContentLoader.cs:**

namespace RGEngine.Support

{

/// <summary>

/// Defines a class that supports loading additional content to the game.

/// </summary>

public class ContentLoader

{

/// <summary>

/// Loads a texture from an image.

/// </summary>

/// <param name="texturePath"></param>

/// <returns></returns>

public static Texture2D LoadTexture(string texturePath)

{

// Check if a file exists.

if (!File.Exists(texturePath))

{

throw new FileNotFoundException("That is unable to load a texture file.", texturePath);

}

// Genereate a new texture type and

// bind it to ID.

int textureId = GL.GenTexture();

GL.BindTexture(TextureTarget.Texture2D, textureId);

// Load a texture to a bitmap.

Bitmap bitmap = new Bitmap(texturePath);

// Convert texture bitmap to a bits data.

System.Drawing.Imaging.BitmapData data = bitmap.LockBits(new Rectangle(0, 0, bitmap.Width, bitmap.Height),

System.Drawing.Imaging.ImageLockMode.ReadOnly,

System.Drawing.Imaging.PixelFormat.Format32bppArgb);

// Specify a texture 2D image.

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

PixelFormat.Bgra, PixelType.UnsignedByte, data.Scan0);

bitmap.UnlockBits(data);

// TextureWrapS applies wrapping mode for S-axis.

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapS,

(int)TextureWrapMode.Clamp);

// TextureWrapS applies wrapping mode for T-axis.

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapT,

(int)TextureWrapMode.Clamp);

// Minimizing filtering setup.

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter,

(int)TextureMinFilter.Linear);

// Magnifying filtering setup.

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter,

(int)TextureMagFilter.Linear);

return new Texture2D(textureId, bitmap.Width, bitmap.Height, texturePath);

}

}

}

**Листинг класса GameTimer.cs:**

namespace RGEngine.Support

{

/// <summary>

///

/// </summary>

public class GameTimer

{

/// <summary>

/// Basic ctor that init timer with 0 interval.

/// </summary>

public GameTimer()

{

this.Interval = 0;

}

/// <summary>

/// Basic ctor that init timer with param interval value.

/// </summary>

/// <param name="interval"></param>

public GameTimer(double interval)

{

this.Interval = interval;

}

private double elapsedTime;

/// <summary>

/// Keeps time-interval value.

/// </summary>

public double Interval { get; set; }

/// <summary>

/// Keeps the number of invokations of a timer Elapsed event.

/// </summary>

public int Invokations { get; private set; }

/// <summary>

/// Event that raised when timer interval elapsed.

/// </summary>

public event EventHandler Elapsed;

/// <summary>

/// Updates the timer with deltaTime value.

/// </summary>

/// <param name="deltaTime"></param>

public void Update(double deltaTime)

{

elapsedTime += deltaTime;

if (Math.Abs(elapsedTime - Interval) <= 0.01f)

{

Elapsed?.Invoke(this, new EventArgs());

Invokations += 1;

elapsedTime = 0;

}

}

}

}

**Листинг класса Camera.cs:**

namespace RGEngine

{

/// <summary>

/// A Camera is a device through which the player views the world.

/// </summary>

internal class Camera

{

/// <summary>

/// Keeps current position of Camera.

/// </summary>

public Vector2 position;

/// <summary>

/// Keeps current zoom value of Camera.

/// </summary>

public double zoom;

/// <summary>

/// Keeps current rotation of Camera.

/// </summary>

public double rotation;

public Camera(Vector2 camPosition, double camZoomValue, double camRotationValue)

{

this.position = camPosition;

this.zoom = camZoomValue;

this.rotation = camRotationValue;

}

public static void SetView(int gameWidth, int gameHeight)

{

// If enabled and no fragment shader is active,

// two-dimensional texturing is performed.

GL.Viewport(0, 0, gameWidth, gameHeight);

// Setup the OpenGL workflow to set the view.

GL.MatrixMode(MatrixMode.Projection);

GL.LoadIdentity();

// Put the View in the center of the window.

// Left, Rigth, Bottom, Top.

GL.Ortho(-gameWidth / 2f, gameWidth / 2f, gameHeight / 2f, -gameHeight / 2f, 0f, 1f);

}

}

}

**Листинг класса EngineCore.cs:**

namespace RGEngine

{

/// <summary>

/// Main core class of the engine that controlls all parts

/// like Graphics, Physics, Input etc.

/// </summary>

public class EngineCore : GameWindow

{

/// <summary>

/// Stores all current objects in the game.

/// </summary>

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

private double totalTimeElapsed;

/// <summary>

/// Defines a time interval for FixedUpdate().

/// </summary>

public static double deltaTimeFixedUpdate;

private bool EnableColliderDrawing { get; }

/// <summary>

/// Default ctor for EngineCore class.

/// </summary>

public EngineCore()

{

deltaTimeFixedUpdate = 0.001f;

}

/// <summary>

/// Ctor that gives ability to enable collider drawing on the scene.

/// </summary>

/// <param name="enableCollidersDrawing"></param>

public EngineCore(bool enableCollidersDrawing)

: this()

{

this.EnableColliderDrawing = enableCollidersDrawing;

}

/// <summary>

/// Inits the game on loading.

/// </summary>

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

protected override void OnLoad(EventArgs e)

{

GL.ClearColor(Color.White);

Camera.SetView(Width, Height);

base.OnLoad(e);

}

/// <summary>

/// Defines frames rendering.

/// </summary>

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

protected override void OnRenderFrame(FrameEventArgs e)

{

SpriteRenderer.RenderEntireFrame(gameObjects.ToList<GameObject>());

if (EnableColliderDrawing)

foreach (var @object in gameObjects.ToList<GameObject>())

@object.collider?.Draw();

SwapBuffers();

base.OnRenderFrame(e);

}

/// <summary>

/// Do all the preparations before frame rendering and updates the game state.

/// </summary>

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

protected override void OnUpdateFrame(FrameEventArgs e)

{

InputController.Update();

totalTimeElapsed += e.Time;

// Prevents redundant Updating of FixedUpdate() method.

// \*REASON: Rendering is slower than physics fixed updates.\*

//

// Cheks if the time elapsed is bigger than deltaTimeFixedUpdate

// than choose elapsed time from the last frame render for a FixedUpdate.

if (totalTimeElapsed >= deltaTimeFixedUpdate)

{

foreach (var gameObject in gameObjects.ToList<GameObject>())

{

gameObject.PerformFixedUpdate(totalTimeElapsed);

}

}

totalTimeElapsed = 0f;

foreach (var gameObject in gameObjects.ToList<GameObject>())

{

gameObject.PerformUpdate(e.Time);

}

base.OnUpdateFrame(e);

}

/// <summary>

/// Defines actions that are made on GameWindow Resized event.

/// </summary>

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

protected override void OnResize(EventArgs e)

{

Camera.SetView(Width, Height);

base.OnResize(e);

}

/// <summary>

/// Do actions on unloading the game.

/// </summary>

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

protected override void OnUnload(EventArgs e)

{

gameObjects.RemoveRange(0, gameObjects.Count);

base.OnUnload(e);

}

/// <summary>

/// Adds a GameObject to a static gameObjects list.

/// </summary>

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

public static void AddGameObject(GameObject gameObject)

{

gameObjects.Add(gameObject);

}

/// <summary>

/// Removes Adds a GameObject from a static gameObjects list.

/// </summary>

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

public static void RemoveGameObject(GameObject gameObject)

{

gameObjects.Remove(gameObject);

}

}

}

**Листинги проекта Racing:**

**Листинг класса Car.cs:**

namespace Racing.Objects

{

/// <summary>

/// Defines an abstract car object.

/// </summary>

public abstract class Car : GameObject, ICollidable

{

/// <summary>

/// Ctor that set a basic car object settings properly.

/// </summary>

public Car()

{

spriteRenderer = (SpriteRenderer)AddComponent("SpriteRenderer");

rigidBody2D = (RigidBody2D)AddComponent("RigidBody2D");

// Set defaults for a default car.

properties = new CarProps(this);

this.fuelLevel = properties.MaxFuelLevel;

// Set default wheelBase value and start car position.

wheelBase = 80f;

SetStartCarPosition(new Vector2(150f, 150f));

// Set default RigidBody parameters for a basic Car object.

rigidBody2D.mass = 1200f;

rigidBody2D.frictionConst = 750f;

base.collider = new PolyCollider(this, new Vector2(110f, 45f));

base.collider.ColliderTriggered += FinishLine\_ColliderTriggered;

base.collider.ColliderTriggered += Prize\_ColliderTriggered;

// Difine finished laps array.

this.laps = new bool[5 + 1];

}

#region Fields and Props

/// <summary>

/// SpriteRenderer component.

/// </summary>

protected SpriteRenderer spriteRenderer;

/// <summary>

/// RigidBody2D component.

/// </summary>

protected RigidBody2D rigidBody2D;

/// <summary>

/// Constraint on min car velocity.

/// </summary>

protected const float velocityConstraint = 6f;

/// <summary>

/// Keeps all car properties.

/// </summary>

public CarProps properties;

/// <summary>

/// Defines a distance between front and rear wheel.

/// </summary>

protected float wheelBase;

/// <summary>

/// Responsible for car front wheels steering angle.

/// </summary>

protected float steeringAngle;

/// <summary>

/// Defines current car direction on scene (in radians).

/// </summary>

protected float carDirectionAngle;

/// <summary>

/// Keeps a driving mode.

/// Drive or Reverse.

/// </summary>

protected int drivingMode;

/// <summary>

/// Defines a position of a front wheel.

/// </summary>

protected Vector2 frontWheel;

/// <summary>

/// Defines a position of a back wheel.

/// </summary>

protected Vector2 backWheel;

private float fuelLevel;

/// <summary>

/// Responsible for getting and setting current fuel level.

/// </summary>

public float FuelLevel

{

get => this.fuelLevel;

set

{

if (this.fuelLevel + value >= properties.MaxFuelLevel)

this.fuelLevel = properties.MaxFuelLevel;

else

this.fuelLevel = value;

}

}

/// <summary>

/// String ID name of a car.

/// </summary>

public string id;

private bool[] laps;

private int beingLocatedOnFinishLine;

private int \_lapsPassed;

/// <summary>

/// Responsible for counting laps that were passed.

/// </summary>

public int LapsPassed

{

get

{

var result = 0;

for (int i = 0; i < this.laps.Length; i++)

{

if (this.laps[i])

result++;

}

return result;

}

}

#endregion

/// <summary>

/// Event that raised when End of race occured.

/// </summary>

public event EventHandler<GameEventArgs> EndedRace;

/// <summary>

/// Invokes EndedRace event.

/// </summary>

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

protected virtual void OnEndedRace(GameEventArgs e)

{

var handler = EndedRace;

handler?.Invoke(this, e);

}

/// <summary>

/// Override of FixedUpdate method.

/// </summary>

/// <param name="fixedDeltaTime"></param>

public override void FixedUpdate(double fixedDeltaTime)

{

var steer = MathHelper.DegreesToRadians(steeringAngle);

Vector2 deltaBackWheel = Vector2.Zero;

Vector2 deltaFrontWheel = Vector2.Zero;

deltaBackWheel.X = rigidBody2D.velocity \* (float)fixedDeltaTime \*

(float)Math.Cos(carDirectionAngle);

deltaBackWheel.Y = rigidBody2D.velocity \* (float)fixedDeltaTime \*

(float)Math.Sin(carDirectionAngle);

// Calculations that push the front wheel forward and

// conserve the wheelBase distance.

var distance = rigidBody2D.velocity \* (float)fixedDeltaTime;

var B = (wheelBase - distance) \* Math.Cos(steer);

var C = distance \* (2 \* wheelBase - distance);

var calc = Math.Sqrt(B \* B + C) - B;

deltaFrontWheel.X = (float)calc \*

(float)Math.Cos(carDirectionAngle + steer);

deltaFrontWheel.Y = (float)calc \*

(float)Math.Sin(carDirectionAngle + steer);

backWheel += deltaBackWheel;

frontWheel += deltaFrontWheel;

base.Position = (frontWheel + backWheel) / 2;

// Detect and resolve collisions.

foreach (var @object in EngineCore.gameObjects.ToList<GameObject>())

{

// Car DOES NOT collide with other car.

if (@object is ICollidable && !(@object is Car))

{

if (!ReferenceEquals(this, @object))

{

if (collider.DetectCollision(@object))

{

if (!(@object is INonResolveable))

{

backWheel -= 1.5f \* deltaBackWheel;

frontWheel -= 1.5f \* deltaFrontWheel;

rigidBody2D.velocity /= -2.75f;

}

}

}

}

}

base.Position = (frontWheel + backWheel) / 2;

carDirectionAngle = (float)Math.Atan2(frontWheel.Y - backWheel.Y,

frontWheel.X - backWheel.X);

base.Rotation = MathHelper.RadiansToDegrees(carDirectionAngle);

ApplyFuelConsumprion(fixedDeltaTime);

}

#region ColliderTriggered handlers

/// <summary>

/// Handles the ColliderTriggered event when car collides a finish line.

/// </summary>

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

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

private void FinishLine\_ColliderTriggered(object sender, CollisionEventArgs e)

{

if (ReferenceEquals(this, e.one))

{

// Checks for finishline crossing.

if (e.another is FinishLine)

{

if ((this.Rotation < 90f &&

this.Rotation > -90f &&

rigidBody2D.velocity > 0) ||

(this.Rotation > 90f &&

this.Rotation < -90f &&

rigidBody2D.velocity < 0))

{

if (!this.laps[\_lapsPassed])

{

this.laps[\_lapsPassed] = true;

}

this.beingLocatedOnFinishLine++;

}

if (LapsPassed == 5 + 1)

OnEndedRace(new GameEventArgs(this));

}

else

{

if (e.another is OuterFinishLine)

{

if (this.beingLocatedOnFinishLine > 0)

{

this.\_lapsPassed = LapsPassed;

this.beingLocatedOnFinishLine = 0;

}

}

}

}

}

/// <summary>

/// Handles the ColliderTriggered event when car collides a prize.

/// </summary>

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

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

private void Prize\_ColliderTriggered(object sender, CollisionEventArgs e)

{

foreach (var @object in EngineCore.gameObjects.ToList<GameObject>())

{

if (@object is Prize)

{

if (ReferenceEquals(@object, e.another))

{

((Prize)@object).PickUp(this);

}

}

}

}

#endregion

/// <summary>

/// Applies a fuel consuption throught time.

/// </summary>

/// <param name="fixedDeltaTime"></param>

/// <returns></returns>

public void ApplyFuelConsumprion(double fixedDeltaTime)

{

// End of race condition.

if (this.fuelLevel <= 0.001)

{

// Choose another car as the winner.

if (this.id == "Black")

this.id = "Purple";

else if (this.id == "Purple")

this.id = "Black";

OnEndedRace(new GameEventArgs(this));

return;

}

var consumptionTime = 5f;

if (rigidBody2D.velocity == 0)

this.fuelLevel -= (properties.IdleFuelConsumption \* (float)fixedDeltaTime)

/ consumptionTime;

else

this.fuelLevel -= (properties.DrivingFuelConsumption \* (float)fixedDeltaTime)

/ consumptionTime;

}

#region User Input Methods

/// <summary>

/// Gets user input for the main car control.

/// </summary>

/// <param name="gas"></param>

/// <param name="brake"></param>

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

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

protected virtual void GetUserInput(Key gas, Key brake, Key left, Key right)

{

// Check for max and min velocity-values.

if (rigidBody2D.velocity >= properties.MaxVelocity)

rigidBody2D.velocity = properties.MaxVelocity;

if (rigidBody2D.velocity <= properties.MaxVelocityReverse)

rigidBody2D.velocity = properties.MaxVelocityReverse;

// Fixing a STOP-point.

if (rigidBody2D.velocity <= velocityConstraint &&

rigidBody2D.velocity >= -velocityConstraint)

rigidBody2D.velocity = 0;

if (InputController.CurrentKeyboardState.IsKeyDown(gas))

{

rigidBody2D.engineForce = drivingMode \* properties.MaxEngineForce;

}

else if (InputController.CurrentKeyboardState.IsKeyDown(brake))

{

if (rigidBody2D.velocity == 0)

rigidBody2D.breakingForce = 0;

else if (rigidBody2D.velocity > 0)

rigidBody2D.breakingForce = properties.MaxBreakingForce;

else

rigidBody2D.breakingForce = -properties.MaxBreakingForce;

}

else

{

rigidBody2D.breakingForce = 0;

rigidBody2D.engineForce = 0;

}

if (InputController.CurrentKeyboardState.IsKeyDown(left))

{

this.steeringAngle = -properties.MaxSteeringAngle;

}

else if (InputController.CurrentKeyboardState.IsKeyDown(right))

{

this.steeringAngle = properties.MaxSteeringAngle;

}

else

{

this.steeringAngle = 0f;

}

}

/// <summary>

/// Gets user input to update a gearbox state.

/// </summary>

/// <param name="reverse"></param>

/// <param name="drive"></param>

protected virtual void UpdateGearboxState(Key reverse, Key drive)

{

if (InputController.CurrentKeyboardState.IsKeyDown(drive))

{

drivingMode = (int)DrivingModes.Drive;

}

else if (InputController.CurrentKeyboardState.IsKeyDown(reverse))

{

drivingMode = (int)DrivingModes.Reverse;

}

}

#endregion

/// <summary>

/// Sets a start car position.

/// </summary>

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

protected virtual void SetStartCarPosition(Vector2 startPosition)

{

base.Position = startPosition;

this.frontWheel = base.Position +

this.wheelBase / 2 \* new Vector2((float)Math.Cos(carDirectionAngle), (float)Math.Sin(carDirectionAngle));

this.backWheel = base.Position -

this.wheelBase / 2 \* new Vector2((float)Math.Cos(carDirectionAngle), (float)Math.Sin(carDirectionAngle));

}

}

/// <summary>

/// Defines main driving modes.

/// </summary>

public enum DrivingModes

{

/// <summary>

/// Returns -1 cause all forces are in 'negative' direction.

/// </summary>

Reverse = -1,

/// <summary>

/// Returns 0 cause all forces are in 'neutral' direction.

/// </summary>

Neutral = 0,

/// <summary>

/// Returns 1 cause all forces are in 'positive' direction.

/// </summary>

Drive = 1

}

}

**Листинг класса BlackCar.cs:**

namespace Racing.Objects

{

/// <summary>

/// Defines a player on a black car.

/// </summary>

public class BlackCar : Car

{

/// <summary>

/// Default ctor.

/// </summary>

public BlackCar()

{

base.id = "Black";

SetStartCarPosition(new Vector2(85f, 325f));

var vehicleTexture = ContentLoader.LoadTexture(@"Contents\Cars\black.png");

var vehicleSprite = new Sprite(vehicleTexture, Position, new Vector2(0.4f, 0.4f),

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

spriteRenderer.RenderQueue = SpriteBatch.CreateSpriteBatch(vehicleSprite);

}

/// <summary>

/// Override of FixedUpdate with new user input.

/// </summary>

/// <param name="fixedDeltaTime"></param>

public override void FixedUpdate(double fixedDeltaTime)

{

UpdateGearboxState(Key.Q, Key.E);

GetUserInput(Key.W, Key.S, Key.A, Key.D);

base.FixedUpdate(fixedDeltaTime);

}

}

}

**Листинг класса PurpleCar.cs:**

namespace Racing.Objects

{

/// <summary>

/// Defines a player on a purple car.

/// </summary>

public class PurpleCar : Car

{

/// <summary>

/// Default ctor.

/// </summary>

public PurpleCar()

{

base.id = "Purple";

SetStartCarPosition(new Vector2(85f, 435f));

var vehicleTexture = ContentLoader.LoadTexture(@"Contents\Cars\purple.png");

var vehicleSprite = new Sprite(vehicleTexture, Position, new Vector2(0.4f, 0.4f),

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

spriteRenderer.RenderQueue = SpriteBatch.CreateSpriteBatch(vehicleSprite);

}

/// <summary>

/// Override of FixedUpdate with new user input.

/// </summary>

/// <param name="fixedDeltaTime"></param>

public override void FixedUpdate(double fixedDeltaTime)

{

UpdateGearboxState(Key.Keypad7, Key.Keypad9);

GetUserInput(Key.Keypad8, Key.Keypad5, Key.Keypad4, Key.Keypad6);

base.FixedUpdate(fixedDeltaTime);

}

}

}

**Листинг класса Environment.cs:**

namespace Racing.Objects

{

/// <summary>

/// Defines an environment class that holds a background

/// and collidable bounds.

/// </summary>

public class Environment : GameObject, ICollidable

{

/// <summary>

/// Default ctor.

/// </summary>

/// <param name="backgroundPath"></param>

public Environment(string backgroundPath)

{

spriteRenderer = (SpriteRenderer)AddComponent("SpriteRenderer");

base.Position = new Vector2(0f, 0f);

var bgTexture = ContentLoader.LoadTexture(backgroundPath);

var bgSprite = new Sprite(bgTexture, Position, new Vector2(1f, 1f),

new Vector2(0f, 0f), -1);

spriteRenderer.RenderQueue = SpriteBatch.CreateSpriteBatch(bgSprite);

// Middle collider.

base.collider = new PolyCollider(this, new Vector2(1310f, 480f));

// Add screen bounds.

EngineCore.AddGameObject(new Bound(new Vector2(-945f, 0f), new Vector2(20f, 1080f)));

EngineCore.AddGameObject(new Bound(new Vector2(945f, 0f), new Vector2(20f, 1080f)));

EngineCore.AddGameObject(new Bound(new Vector2(0f, 525f), new Vector2(1920f, 20f)));

EngineCore.AddGameObject(new Bound(new Vector2(0f, -525f), new Vector2(1920f, 20f)));

}

private class Bound : GameObject, ICollidable, INonRenderable

{

public Bound(Vector2 position, Vector2 size)

{

base.Position = position;

base.collider = new PolyCollider(this, size);

}

}

private SpriteRenderer spriteRenderer;

}

/// <summary>

/// Defines game event args.

/// </summary>

public class GameEventArgs : EventArgs

{

/// <summary>

/// GameObject member.

/// </summary>

public readonly GameObject @object;

/// <summary>

/// Default ctor.

/// </summary>

/// <param name="object"></param>

public GameEventArgs(GameObject @object)

{

this.@object = @object;

}

}

}

**Листинг класса FinishLine.cs:**

namespace Racing.Objects

{

/// <summary>

/// Defines a finish line object.

/// </summary>

public class FinishLine : GameObject, INonRenderable, ICollidable, INonResolveable

{

/// <summary>

/// Default ctor with position and collider settings.

/// </summary>

public FinishLine()

{

//base.Position = new Vector2(10f, 380f);

base.Position = new Vector2(150f, 380f);

base.collider = new PolyCollider(this, new Vector2(15f, 250f));

}

}

}

**Листинг класса OuterFinishLine.cs:**

namespace Racing.Objects

{

/// <summary>

/// Defines a finish line outer collider.

/// </summary>

public class OuterFinishLine : GameObject, INonRenderable, ICollidable, INonResolveable

{

/// <summary>

/// Default ctor with position and collider settings.

/// </summary>

public OuterFinishLine()

{

//base.Position = new Vector2(150f, 380f);

base.Position = new Vector2(290f, 380f);

base.collider = new PolyCollider(this, new Vector2(15f, 250f));

}

}

}

**Листинг класса CarProps.cs:**

namespace Racing.Objects

{

/// <summary>

/// Defines a main set of car properties.

/// </summary>

public class CarProps : GameObject, INonResolveable, INonRenderable

{

/// <summary>

/// Object-owner.

/// </summary>

public Car owner;

/// <summary>

/// Init with default values.

/// </summary>

/// <param name="owner"></param>

public CarProps(Car owner)

{

this.owner = owner;

MaxEngineForce = 430000f;

MaxVelocity = 480f;

MaxVelocityReverse = -250f;

MaxSteeringAngle = 25f;

MaxBreakingForce = 500000f;

MaxFuelLevel = 35f;

IdleFuelConsumption = 1f;

DrivingFuelConsumption = 3f;

}

/// <summary>

/// Basic ctor.

/// </summary>

public CarProps()

{

}

/// <summary>

/// Defines max engine force.

/// </summary>

public virtual float MaxEngineForce { get; set; }

/// <summary>

/// Defines max velocity.

/// </summary>

public virtual float MaxVelocity { get; set; }

/// <summary>

/// Defines max velocity in reverse.

/// </summary>

public float MaxVelocityReverse { get; set; }

/// <summary>

/// Defines max steering angle.

/// </summary>

public float MaxSteeringAngle { get; set; }

/// <summary>

/// Defines max breaking force.

/// </summary>

public float MaxBreakingForce { get; set; }

/// <summary>

/// Defines max fuel level.

/// </summary>

public float MaxFuelLevel { get; set; }

/// <summary>

/// Defines idle fuel consumption.

/// </summary>

public float IdleFuelConsumption { get; set; }

/// <summary>

/// Defines driving fuel consumption.

/// </summary>

public float DrivingFuelConsumption { get; set; }

}

}

**Листинг класса CarPropsDecorator.cs:**

namespace Racing.Objects

{

/// <summary>

/// Desines a decorator for a car props.

/// </summary>

public abstract class CarPropsDecorator : CarProps

{

/// <summary>

/// Keeps base props object.

/// </summary>

protected CarProps baseProps;

/// <summary>

/// Sets prize life timer.

/// </summary>

protected GameTimer lifeTimer;

/// <summary>

/// Sets decorator life time period in a timer.

/// </summary>

/// <param name="baseProps"></param>

public CarPropsDecorator(CarProps baseProps)

: base(baseProps.owner)

{

this.baseProps = baseProps;

base.owner = baseProps.owner;

this.lifeTimer = new GameTimer(3f);

this.lifeTimer.Elapsed += (sender, e) => RemoveDecorator();

EngineCore.AddGameObject(this);

}

/// <summary>

/// Testing ctor.

/// </summary>

public CarPropsDecorator(CarProps baseProps, Car owner)

{

this.baseProps = baseProps;

base.owner = owner;

}

/// <summary>

/// Overrides FixedUpdate with inner timer update.

/// </summary>

/// <param name="fixedDeltaTime"></param>

public override void FixedUpdate(double fixedDeltaTime)

{

lifeTimer.Update(fixedDeltaTime);

base.FixedUpdate(fixedDeltaTime);

}

private void RemoveDecorator()

{

var list = EngineCore.gameObjects.ToList<GameObject>();

foreach (var @object in list)

{

if ((@object is Car) && ReferenceEquals(@object, base.owner))

{

var car = (Car)@object;

car.properties = this.baseProps;

}

}

EngineCore.RemoveGameObject(this);

}

}

}

**Листинг класса BoostProps.cs:**

namespace Racing.Objects

{

/// <summary>

/// Defines props with applied fuel prize.

/// </summary>

public class BoostProps : CarPropsDecorator

{

/// <summary>

/// Default ctor.

/// </summary>

/// <param name="props"></param>

public BoostProps(CarProps props)

: base(props)

{

}

/// <summary>

/// Testing ctor.

/// </summary>

/// <param name="props"></param>

/// <param name="owner"></param>

public BoostProps(CarProps props, Car owner)

: base(props, owner)

{

}

/// <summary>

/// Overrides MaxVelocity prop.

/// </summary>

public override float MaxVelocity

{

get => baseProps.MaxVelocity \* 1.5f;

set => base.MaxVelocity = value;

}

/// <summary>

/// Overrides MaxEngineForce prop.

/// </summary>

public override float MaxEngineForce

{

get => baseProps.MaxEngineForce \* 2f;

set => base.MaxEngineForce = value;

}

}

}

**Листинг класса SlowdownProps.cs:**

namespace Racing.Objects

{

/// <summary>

/// Defines props with applied slowdown prize.

/// </summary>

public class SlowdownProps : CarPropsDecorator

{

/// <summary>

/// Default ctor.

/// </summary>

/// <param name="props"></param>

public SlowdownProps(CarProps props)

: base(props)

{

}

/// <summary>

/// Testing ctor.

/// </summary>

/// <param name="props"></param>

/// <param name="owner"></param>

public SlowdownProps(CarProps props, Car owner)

: base(props, owner)

{

}

/// <summary>

/// Overrides MaxVelocity prop.

/// </summary>

public override float MaxVelocity

{

get => baseProps.MaxVelocity / 1.5f;

set => base.MaxVelocity = value;

}

}

}

**Листинг класса UIHandler.cs:**

namespace Racing.Objects.UI

{

/// <summary>

/// Defines a control object for the UI.

/// </summary>

public class UIHandler : GameObject, INonRenderable, INonResolveable

{

/// <summary>

/// Default ctor.

/// </summary>

public UIHandler()

{

SubscribeOnPlayers();

// Add info ui elements.

EngineCore.AddGameObject(new UIText("lapsBlack", new Vector2(-355, -208), new Vector2(1.05f, 1.05f)));

EngineCore.AddGameObject(new UIText("lapsPurple", new Vector2(505, -208), new Vector2(1.05f, 1.05f)));

EngineCore.AddGameObject(new UIText("fuelBlack", new Vector2(-388, -147), new Vector2(1.05f, 1.05f)));

EngineCore.AddGameObject(new UIText("fuelPurple", new Vector2(474, -147), new Vector2(1.05f, 1.05f)));

}

private void SubscribeOnPlayers()

{

var list = EngineCore.gameObjects.ToList<GameObject>();

foreach (var @object in list)

{

if (@object is Car)

{

((Car)@object).EndedRace += (sender, e) => DisplayWinner(e.@object);

}

}

}

private void DisplayWinner(GameObject winner)

{

var blackTex = @"Contents\UI\Winners\blackWinsText.png";

var purpleTex = @"Contents\UI\Winners\purpleWinsText.png";

var car = winner as Car;

if (car != null)

{

switch (car.id)

{

case "Black":

EngineCore.AddGameObject(new UIElement(blackTex, new Vector2(0f, 0f)));

EndGame();

break;

case "Purple":

EngineCore.AddGameObject(new UIElement(purpleTex, new Vector2(0f, 0f)));

EndGame();

break;

default:

throw new Exception("There is nothing to display.");

}

}

}

private void EndGame()

{

foreach (var @object in EngineCore.gameObjects.ToList<GameObject>())

{

if (@object is Car ||

@object is Prize ||

@object is PrizeGenerator)

{

EngineCore.RemoveGameObject(@object);

}

}

}

int lastLapsBlack;

int lastLapsPurple;

int lastFuelBlack;

int lastFuelPurple;

/// <summary>

/// Overrides FixedUpdate with UI update.

/// </summary>

/// <param name="fixedDeltaTime"></param>

public override void FixedUpdate(double fixedDeltaTime)

{

// Check cars' states.

foreach (var @object in EngineCore.gameObjects)

{

if (@object is Car)

{

if (((Car)@object).id == "Black")

{

// Check for changes.

if (Math.Abs(lastLapsBlack - ((Car)@object).LapsPassed) > 0)

{

this.lastLapsBlack = ((Car)@object).LapsPassed;

DisplayTextUIElement("lapsBlack", lastLapsBlack.ToString());

}

if (Math.Abs(lastFuelBlack - (int)((Car)@object).FuelLevel) > 0)

{

this.lastFuelBlack = (int)((Car)@object).FuelLevel;

DisplayTextUIElement("fuelBlack", ((int)lastFuelBlack).ToString());

}

}

else if (((Car)@object).id == "Purple")

{

// Check for changes.

if (Math.Abs(lastLapsPurple - ((Car)@object).LapsPassed) > 0)

{

this.lastLapsPurple = ((Car)@object).LapsPassed;

DisplayTextUIElement("lapsPurple", lastLapsPurple.ToString());

}

if (Math.Abs(lastFuelPurple - (int)((Car)@object).FuelLevel) > 0)

{

this.lastFuelPurple = (int)((Car)@object).FuelLevel;

DisplayTextUIElement("fuelPurple", ((int)lastFuelPurple).ToString());

}

}

}

}

base.FixedUpdate(fixedDeltaTime);

}

private void DisplayTextUIElement(string elementName, string text)

{

foreach (var element in EngineCore.gameObjects)

{

if (element is UIText)

{

if (((UIText)element).name == elementName)

{

((UIText)element).spriteRenderer.RenderQueue =

UIText.ParseStringToSprites(text, base.Position, new Vector2(1.05f, 1.05f));

}

}

}

}

}

}

**Листинг класса UIElement.cs:**

namespace Racing.Objects.UI

{

/// <summary>

/// Defines a simple UI graphical element.

/// </summary>

public class UIElement : GameObject, IUIElement

{

/// <summary>

/// Holds SpriteRenderer component.

/// </summary>

public SpriteRenderer spriteRenderer;

/// <summary>

/// Default ctor with setting a position.

/// </summary>

/// <param name="uiTexturePath"></param>

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

public UIElement(string uiTexturePath, Vector2 position)

{

spriteRenderer = (SpriteRenderer)AddComponent("SpriteRenderer");

base.Position = position;

var uiTex = ContentLoader.LoadTexture(uiTexturePath);

var uiSprite = new Sprite(uiTex, Position, new Vector2(1f, 1f),

new Vector2(0f, 0f), 10);

spriteRenderer.RenderQueue = SpriteBatch.CreateSpriteBatch(uiSprite);

}

/// <summary>

/// Ctor for rendering SpriteBatch-ready UI elements.

/// </summary>

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

public UIElement(Vector2 position)

{

//spriteRenderer = AddComponent<SpriteRenderer>();

spriteRenderer = (SpriteRenderer)AddComponent("SpriteRenderer");

base.Position = position;

}

}

/// <summary>

/// Interface for UI element object.

/// </summary>

public interface IUIElement { }

}

**Листинг класса UIText.cs:**

namespace Racing.Objects.UI

{

/// <summary>

/// Defines a simple UI text element.

/// </summary>

public class UIText : UIElement

{

private static Texture2D[] digitTextures;

private static readonly string digits = "0123456789";

/// <summary>

/// Keeps the name of the Text element.

/// </summary>

public readonly string name;

static UIText()

{

digitTextures = LoadSymbolsTexArray(@"Contents\UI\Digits\0.png",

@"Contents\UI\Digits\1.png",

@"Contents\UI\Digits\2.png",

@"Contents\UI\Digits\3.png",

@"Contents\UI\Digits\4.png",

@"Contents\UI\Digits\5.png",

@"Contents\UI\Digits\6.png",

@"Contents\UI\Digits\7.png",

@"Contents\UI\Digits\8.png",

@"Contents\UI\Digits\9.png");

}

/// <summary>

/// Ctor that defines basic text element.

/// </summary>

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

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

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

public UIText(string name, Vector2 position, Vector2 scale)

: base(position)

{

this.name = name;

base.spriteRenderer.RenderQueue = ParseStringToSprites("0", position, scale);

}

private static Texture2D[] LoadSymbolsTexArray(params string[] texPath)

{

var textures = new Texture2D[texPath.Length];

for (int i = 0; i < texPath.Length; i++)

{

textures[i] = ContentLoader.LoadTexture(texPath[i]);

}

return textures;

}

/// <summary>

/// Parses string to a set of sprites from pre-defined textures.

/// </summary>

/// <param name="data"></param>

/// <param name="textPosition"></param>

/// <param name="symbolScale"></param>

/// <returns></returns>

public static SpriteBatch ParseStringToSprites(string data, Vector2 textPosition, Vector2 symbolScale)

{

if (data.Equals(string.Empty))

throw new ArgumentException("There is no parse data into the input string.");

var sprites = new List<Sprite>();

var startPos = textPosition;

// Create an offset with the space size of 5.

var spaceWidth = -7;

var offset = digitTextures[0].Width / 2 + spaceWidth;

foreach (var @char in data)

{

Sprite symbolSprite;

for (int i = 0; i < digits.Length; i++)

{

if (@char == digits[i])

{

symbolSprite = new Sprite(digitTextures[i], startPos, symbolScale,

textPosition, 10);

sprites.Add(symbolSprite);

// Make an offset by X-axis for the next symbol if there is one.

textPosition += new Vector2(offset, 0);

}

}

}

var textOutputSprites = SpriteBatch.CreateSpriteBatch(sprites.ToArray());

return textOutputSprites;

}

}

}

**Листинг класса Prize.cs:**

namespace Racing.Prizes

{

/// <summary>

/// Defines an abstract class for a prize.

/// </summary>

public abstract class Prize : GameObject

{

/// <summary>

/// Default ctor with setting collider and animation.

/// </summary>

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

/// <param name="texturesPath"></param>

public Prize(Vector2 position, params string[] texturesPath)

{

spriteRenderer = (SpriteRenderer)AddComponent("SpriteRenderer");

animator = (Animator)AddComponent("Animator");

// Set default position.

base.Position = position;

for (int i = 0; i < texturesPath.Length; i++)

{

var tex = ContentLoader.LoadTexture(texturesPath[i]);

var sprite = new Sprite(tex, position, new Vector2(0.55f, 0.55f), new Vector2(0f, 0f), 3);

animator.AnimationSprites.AddSprite(sprite);

}

animator.FPS = 7;

base.collider = new PolyCollider(this, new Vector2(40f, 40f));

}

/// <summary>

/// SpriteRenderer component.

/// </summary>

protected SpriteRenderer spriteRenderer;

/// <summary>

/// RigidBody2D component.

/// </summary>

protected RigidBody2D rigidBody;

/// <summary>

/// Animator component.

/// </summary>

protected Animator animator;

internal void PickUp(Car car)

{

ApplyPrize(car);

RemovePrize();

}

/// <summary>

/// Removes a prize from the scene.

/// </summary>

protected void RemovePrize()

{

PolyCollider.allCollidersAttached.Remove(this);

var list = EngineCore.gameObjects.ToList<GameObject>();

foreach (var @object in list)

{

if (ReferenceEquals(@object, this))

EngineCore.RemoveGameObject(this);

}

}

/// <summary>

/// Applies a prize to a Car.

/// </summary>

/// <param name="car"></param>

protected abstract void ApplyPrize(Car car);

}

/// <summary>

/// Defines all possible in-game prizes.

/// </summary>

public enum Prizes

{

/// <summary>

/// Adds fuel.

/// </summary>

Fuel = 1,

/// <summary>

/// Speeds up a car.

/// </summary>

Boost = 2,

/// <summary>

/// Slows a car.

/// </summary>

Slowdown = 3

}

}

**Листинг интерфейса IPrizeFactory.cs:**

namespace Racing.Prizes

{

/// <summary>

/// Interface for a prize factory.

/// </summary>

interface IPrizeFactory

{

Prize Prize { get; set; }

Prize GeneratePrize(Random random, Vector2 position);

}

}

**Листинг класса PrizeGenerator.cs:**

namespace Racing.Prizes

{

/// <summary>

/// Defines a class responsible for a prize generation.

/// </summary>

public class PrizeGenerator : GameObject

{

/// <summary>

/// Sets the geteration interval and generation area bounds.

/// </summary>

public PrizeGenerator()

{

spawnTimer = new GameTimer(5f);

spawnTimer.Elapsed += (sender, e) => AddPrizeOnScene();

rect1min = new Vector2(-870, 450);

rect1max = new Vector2(870, -450);

rect2min = new Vector2(-730, 310);

rect2max = new Vector2(730, -310);

}

internal IPrizeFactory IPrizeFactory

{

get => default;

set

{

}

}

private GameTimer spawnTimer;

// Rect1 is outer bound.

// Rect2 is inner bound.

private Vector2 rect1min;

private Vector2 rect1max;

private Vector2 rect2min;

private Vector2 rect2max;

private void AddPrizeOnScene()

{

var rand = new Random();

var prize = GeneratePrize(rand, GetPrizePosition(rand));

if (prize != null)

{

EngineCore.AddGameObject(prize);

}

}

/// <summary>

/// Main Fabric Method.

/// </summary>

public Prize GeneratePrize(Random random, Vector2 position)

{

// Limit the number of prizes located

// on scene at the same time.

var prizesOnScene = 0;

foreach (var @object in EngineCore.gameObjects)

{

if (@object is Prize)

prizesOnScene++;

}

if (prizesOnScene > 4)

return null;

var prizesCount = 3;

var randPrize = random.Next(1, prizesCount + 1);

switch (randPrize)

{

case (int)Prizes.Fuel:

return new FuelPrize(position);

case (int)Prizes.Boost:

return new BoostPrize(position);

case (int)Prizes.Slowdown:

return new SlowdownPrize(position);

default:

throw new Exception("There is no such a prize number.");

}

}

/// <summary>

/// Overrides FixedUpdate with updating inner timer.

/// </summary>

/// <param name="fixedDeltaTime"></param>

public override void FixedUpdate(double fixedDeltaTime)

{

spawnTimer?.Update(fixedDeltaTime);

base.FixedUpdate(fixedDeltaTime);

}

/// <summary>

/// Generates random prize spawn position.

/// </summary>

/// <returns></returns>

private Vector2 GetPrizePosition(Random random)

{

int randX;

int randY;

var randRegion = random.Next(4);

// INVERSED Y-axis.

switch (randRegion)

{

case 0:

randX = random.Next((int)rect1min.X, (int)rect1max.X);

randY = random.Next((int)rect1max.Y, (int)rect2max.Y);

break;

case 1:

randX = random.Next((int)rect1min.X, (int)rect2min.X);

randY = random.Next((int)rect2max.Y, (int)rect2min.Y);

break;

case 2:

randX = random.Next((int)rect2max.X, (int)rect1max.X);

randY = random.Next((int)rect2max.Y, (int)rect2min.Y);

break;

case 3:

randX = random.Next((int)rect1min.X, (int)rect1max.X);

randY = random.Next((int)rect2min.Y, (int)rect1min.Y);

break;

default:

throw new Exception("Random out of range.");

}

return new Vector2(randX, randY);

}

}

}

**Листинг класса BoostPrize.cs:**

namespace Racing.Prizes

{

/// <summary>

/// Defines a prize that speeds up a car.

/// </summary>

public class BoostPrize : Prize

{

/// <summary>

/// Loading textures.

/// </summary>

public BoostPrize(Vector2 position)

: base(position,

@"Contents\Animations\Prizes\Boost\nos1.png",

@"Contents\Animations\Prizes\Boost\nos2.png",

@"Contents\Animations\Prizes\Boost\nos3.png",

@"Contents\Animations\Prizes\Boost\nos4.png")

{

}

/// <summary>

/// Applies a specified decorator to a CarProps.

/// </summary>

/// <param name="car"></param>

protected override void ApplyPrize(Car car)

{

// Remove current decorator if there is one.

car.properties = new CarProps(car);

car.properties = new BoostProps(car.properties);

}

}

}

**Листинг класса FuelPrize.cs:**

namespace Racing.Prizes

{

/// <summary>

/// Defines a prize that refuels a car.

/// </summary>

public class FuelPrize : Prize

{

/// <summary>

/// Loading textures.

/// </summary>

public FuelPrize(Vector2 position)

: base(position,

@"Contents\Animations\Prizes\Fuel\fuel1.png",

@"Contents\Animations\Prizes\Fuel\fuel2.png",

@"Contents\Animations\Prizes\Fuel\fuel3.png",

@"Contents\Animations\Prizes\Fuel\fuel4.png")

{

}

/// <summary>

/// Adds more fuel to a Car object.

/// </summary>

/// <param name="car"></param>

protected override void ApplyPrize(Car car)

{

car.FuelLevel += 7f;

}

}

}

**Листинг класса SlowdownPrize.cs:**

namespace Racing.Prizes

{

/// <summary>

/// Defines an object that slows down a car.

/// </summary>

public class SlowdownPrize : Prize

{

/// <summary>

/// Loading textures.

/// </summary>

public SlowdownPrize(Vector2 position)

: base(position,

@"Contents\Animations\Prizes\Slowdown\slow1.png",

@"Contents\Animations\Prizes\Slowdown\slow2.png",

@"Contents\Animations\Prizes\Slowdown\slow3.png",

@"Contents\Animations\Prizes\Slowdown\slow4.png")

{

}

/// <summary>

/// Applies a specified decorator to a CarProps.

/// </summary>

/// <param name="car"></param>

protected override void ApplyPrize(Car car)

{

// Remove current decorator if there is one.

car.properties = new CarProps(car);

car.properties = new SlowdownProps(car.properties);

}

}

}

**Листинги проекта GameLauncher:**

**Листинг класса Instruction.Designer.cs:**

namespace GameLauncher

{

partial class Instruction

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

System.ComponentModel.ComponentResourceManager resources = new System.ComponentModel.ComponentResourceManager(typeof(Instruction));

this.GO = new System.Windows.Forms.PictureBox();

((System.ComponentModel.ISupportInitialize)(this.GO)).BeginInit();

this.SuspendLayout();

//

// GO

//

this.GO.BackColor = System.Drawing.Color.Transparent;

this.GO.BackgroundImage = global::GameLauncher.Properties.Resources.go\_w;

this.GO.ErrorImage = null;

this.GO.InitialImage = null;

this.GO.Location = new System.Drawing.Point(1104, 599);

this.GO.Name = "GO";

this.GO.Size = new System.Drawing.Size(100, 50);

this.GO.TabIndex = 0;

this.GO.TabStop = false;

this.GO.Click += new System.EventHandler(this.GO\_Click);

this.GO.MouseEnter += new System.EventHandler(this.GO\_OnMouseEnter);

this.GO.MouseLeave += new System.EventHandler(this.GO\_OnMouseLeave);

//

// Instruction

//

this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);

this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;

this.BackgroundImage = global::GameLauncher.Properties.Resources.Instruction;

this.BackgroundImageLayout = System.Windows.Forms.ImageLayout.Stretch;

this.ClientSize = new System.Drawing.Size(1264, 681);

this.Controls.Add(this.GO);

this.Icon = ((System.Drawing.Icon)(resources.GetObject("$this.Icon")));

this.MaximizeBox = false;

this.MinimizeBox = false;

this.Name = "Instruction";

this.StartPosition = System.Windows.Forms.FormStartPosition.CenterParent;

this.Text = "Instruction";

this.Load += new System.EventHandler(this.Instruction\_Load);

((System.ComponentModel.ISupportInitialize)(this.GO)).EndInit();

this.ResumeLayout(false);

}

#endregion

private System.Windows.Forms.PictureBox GO;

}

}

**Листинг класса Instruction.cs:**

namespace GameLauncher

{

public partial class Instruction : Form

{

public Instruction()

{

InitializeComponent();

}

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

{

this.FormBorderStyle = FormBorderStyle.FixedSingle;

}

private void GO\_OnMouseEnter(object sender, EventArgs e)

{

GO.BackgroundImage = new Bitmap(@"Resources\go\_y.png");

}

private void GO\_OnMouseLeave(object sender, EventArgs e)

{

GO.BackgroundImage = new Bitmap(@"Resources\go\_w.png");

}

private void GO\_Click(object sender, EventArgs e)

{

this.Hide();

using (var racingGame = new EngineCore(false))

{

EngineCore.AddGameObject(new Racing.Objects.Environment(@"Contents\Environment\bg\_ui\_v2.png"));

EngineCore.AddGameObject(new FinishLine());

EngineCore.AddGameObject(new OuterFinishLine());

EngineCore.AddGameObject(new PurpleCar());

EngineCore.AddGameObject(new BlackCar());

EngineCore.AddGameObject(new PrizeGenerator());

EngineCore.AddGameObject(new UIHandler());

racingGame.Title = "Retro Race";

racingGame.Icon = new Icon(@"Resources\icon32.ico");

racingGame.WindowBorder = OpenTK.WindowBorder.Fixed;

racingGame.WindowState = OpenTK.WindowState.Maximized;

racingGame.Run();

}

Application.Exit();

}

}

}

**Листинг класса LaunchWindow.Designer.cs:**

namespace GameLauncher

{

partial class LaunchWindow

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null;

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

System.ComponentModel.ComponentResourceManager resources = new System.ComponentModel.ComponentResourceManager(typeof(LaunchWindow));

this.start = new System.Windows.Forms.PictureBox();

((System.ComponentModel.ISupportInitialize)(this.start)).BeginInit();

this.SuspendLayout();

//

// start

//

this.start.BackColor = System.Drawing.Color.Transparent;

this.start.BackgroundImage = global::GameLauncher.Properties.Resources.start\_w;

this.start.BackgroundImageLayout = System.Windows.Forms.ImageLayout.Center;

this.start.Location = new System.Drawing.Point(1052, 546);

this.start.Name = "start";

this.start.Size = new System.Drawing.Size(191, 60);

this.start.TabIndex = 2;

this.start.TabStop = false;

this.start.Click += new System.EventHandler(this.start\_Click);

this.start.MouseEnter += new System.EventHandler(this.start\_OnMouseEnter);

this.start.MouseLeave += new System.EventHandler(this.start\_OnMouseLeave);

//

// LaunchWindow

//

this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);

this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;

this.BackColor = System.Drawing.Color.FromArgb(((int)(((byte)(27)))), ((int)(((byte)(27)))), ((int)(((byte)(27)))));

this.BackgroundImage = global::GameLauncher.Properties.Resources.launcherBG\_hd\_logo;

this.ClientSize = new System.Drawing.Size(1264, 681);

this.Controls.Add(this.start);

this.DoubleBuffered = true;

this.Icon = ((System.Drawing.Icon)(resources.GetObject("$this.Icon")));

this.MaximizeBox = false;

this.MinimizeBox = false;

this.Name = "LaunchWindow";

this.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen;

this.Text = "Racing Game Launcher";

this.Load += new System.EventHandler(this.LaunchWindow\_Load);

((System.ComponentModel.ISupportInitialize)(this.start)).EndInit();

this.ResumeLayout(false);

}

#endregion

private System.Windows.Forms.PictureBox start;

}

}

**Листинг класса LaunchWindow.cs:**

namespace GameLauncher

{

public partial class LaunchWindow : Form

{

public LaunchWindow()

{

InitializeComponent();

}

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

{

}

private void start\_OnMouseEnter(object sender, EventArgs e)

{

start.BackgroundImage = new Bitmap(@"Resources\start\_y.png");

}

private void start\_OnMouseLeave(object sender, EventArgs e)

{

start.BackgroundImage = new Bitmap(@"Resources\start\_w.png");

}

private void start\_Click(object sender, EventArgs e)

{

this.Hide();

var popupInstruction = new Instruction();

popupInstruction.ShowDialog();

}

}

}

# ПРИЛОЖЕНИЕ Б

(обязательное)

**Листинг модульных тестов**

**Листинги тестового проекта UnitTesting:**

**Листинг тестового класса UnitTestClassInstantiating.cs:**

namespace UnitTesting

{

[TestClass]

public class UnitTestClassInstantiating

{

[TestMethod]

public void TestInstantiatingCarProps()

{

var carProps = new CarProps(null);

var result = false;

if (carProps.MaxEngineForce == 430000f

&& carProps.MaxVelocity == 480f

&& carProps.MaxVelocityReverse == -250f

&& carProps.MaxSteeringAngle == 25f

&& carProps.MaxBreakingForce == 500000f

&& carProps.MaxFuelLevel == 35f

&& carProps.IdleFuelConsumption == 1f

&& carProps.DrivingFuelConsumption == 3f)

{

result = true;

}

Assert.IsTrue(result);

}

}

}

**Листинг тестового класса UnitTestDecorator.cs:**

namespace UnitTesting

{

[TestClass]

public class UnitTestDecorator

{

[TestMethod]

public void Decorate\_CarProps\_BoostProps()

{

CarProps testProps = new CarProps();

testProps.MaxVelocity = 400f;

testProps.MaxEngineForce = 250f;

bool result = false;

testProps = new BoostProps(testProps, null);

if (testProps.MaxVelocity == 600f

&& testProps.MaxEngineForce == 500f)

result = true;

Assert.IsTrue(result);

}

[TestMethod]

public void Decorate\_CarProps\_SlowdownProps()

{

CarProps testProps = new CarProps();

testProps.MaxVelocity = 375f;

bool result = false;

testProps = new SlowdownProps(testProps, null);

if (testProps.MaxVelocity == 250f)

result = true;

Assert.IsTrue(result);

}

}

}

# ПРИЛОЖЕНИЕ В

(справочное)

**Схемы шасси автомобиля**

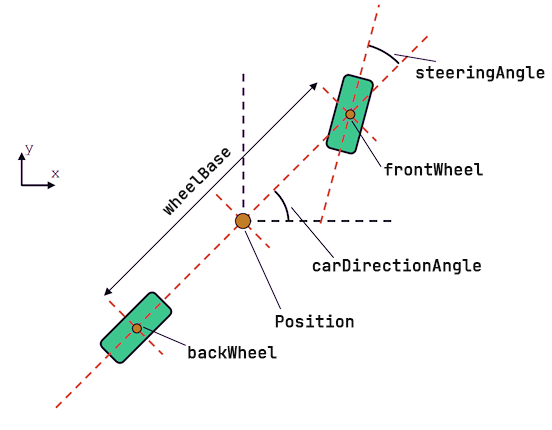


Рисунок В.1 – Схема шасси автомобиля

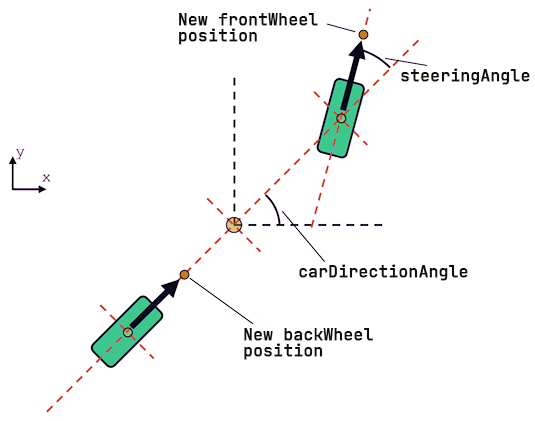


Рисунок В.2 – Схема перемещения колёс на плоскости

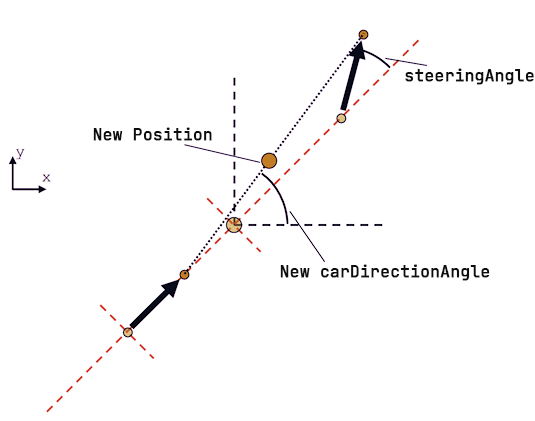


Рисунок В.3 – Схема нахождения новых значений параметров автомобиля

# ПРИЛОЖЕНИЕ Г

(справочное)

**Интерфейс приложения**



Рисунок Г.1 – Загрузочное окно приложения



Рисунок Г.2 – Диалоговое окно приложения



Рисунок Г.3 – Игровое окно приложения

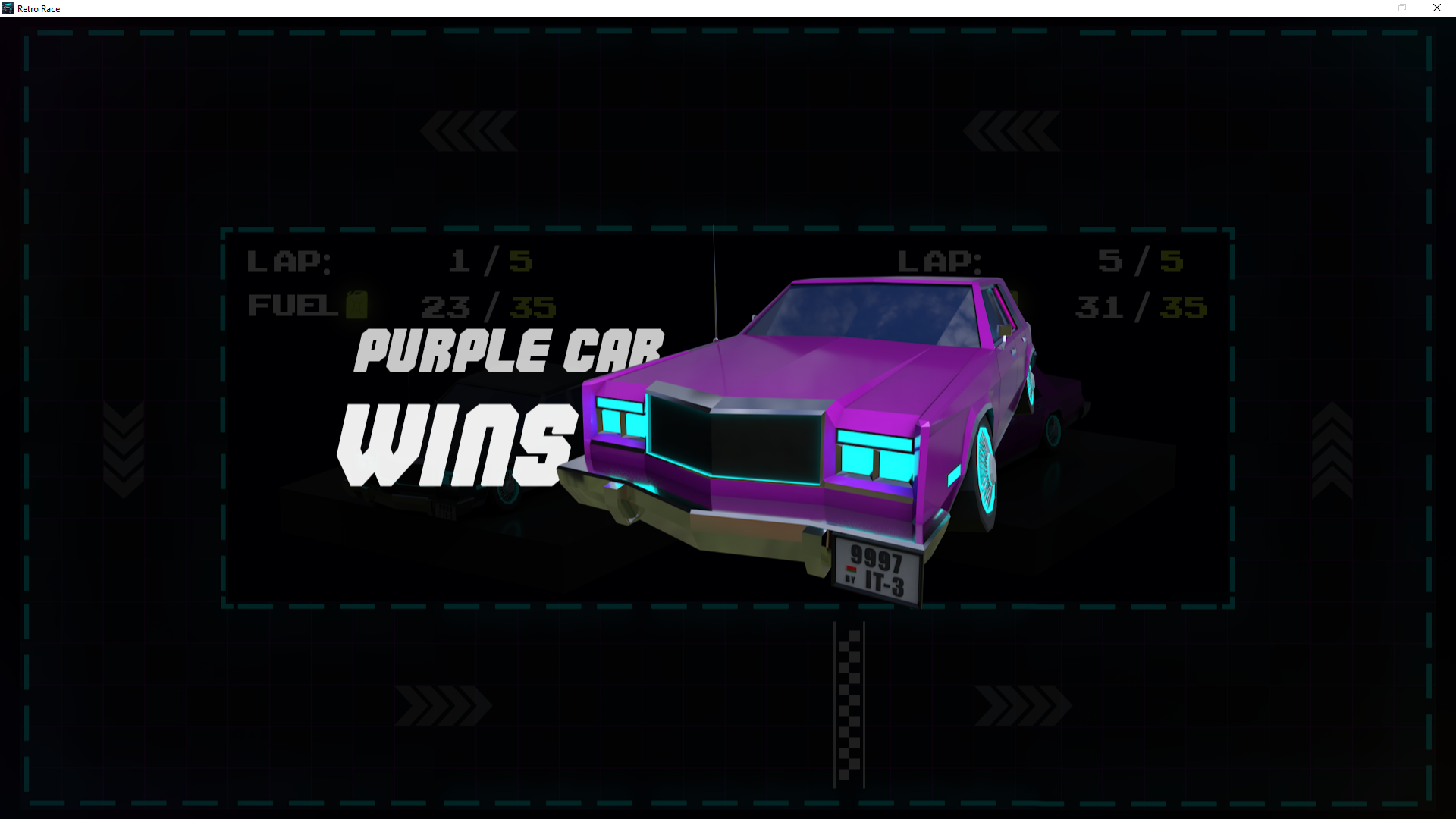


Рисунок Г.4 – Конец игры

# ПРИЛОЖЕНИЕ К

(обязательное)

**Руководство пользователя**

1. Введение.

Руководство пользователя предоставляет пользователю возможность получения базовых навыков эксплуатации игрового приложения. Указания данного руководства выполняются при:

– опытной эксплуатации приложения;

– тестировании приложения.

Данное программное приложение предназначено для организации игрового процесса пользователями. Предполагается наличие двух пользователей в качестве соперников. Управление игровыми объектами осуществляется посредством пользовательского взаимодействия с клавиатурой устройства.

Для использования программного приложения пользователь должен быть ознакомлен с:

– настоящим руководством пользователя;

– базовыми принципами движения автомобиля.

2. Назначение и условия применения.

Данное игровое приложение разработано в развлекательных целях и предназначено для игры в аркадные гонки между двумя игроками на одном компьютере. Игра требует концентрации внимания, а также базового понимания принципов движения автомобиля с целью формирования различных методов прохождения соревновательного заезда.

Для работы приложения необходимо выполнение следующих требований:

– компьютер на базе операционной системы Microsoft Windows;

– наличие клавиатуры с дополнительным блоком цифровых клавиш;

– наличие экрана с разрешением вывода 1920 на 1080 пикселей и выше;

– минимальное количество оперативной памяти в компьютере – 512 МБ.

3. Подготовка к работе.

Для запуска данного приложения необходимо перенести на компьютер папку с готовым решением RacingGame2D. Далее необходимо проследовать по пути GameLauncher\bin\Debug. Находясь в директории Debug, необходимо запустить приложение GameLauncher.exe, после чего произойдёт запуск приложения.

4. Описание операций.

Пользователям доступны следующие операции:

– находясь в окне запуска, ознакомление с правилами игры по нажатию кнопки «Start»;

– находясь в диалоговом окне, запуск игрового окна по нажатию кнопки «Go»;

– управление гоночным автомобилем при помощи клавиш: «W», «A», «S», «D» для управления чёрным автомобилем и «num8», «num4», «num5», «num6» для управления сиреневым автомобилем;

– переключение режимов движения при помощи клавиш «Q», «E» для чёрного автомобиля и «num7», «num9» для сиреневого автомобиля.

5. Аварийные ситуации.

Для того, чтобы избежать возникновения ошибок при использовании данного игрового приложения, необходимо соблюдать предписанный в главе 3 данного руководства порядок действий и условий использования.

В случаях непредвиденной остановки работы приложения следует завершить текущий процесс работы с данным приложением и произвести перезапуск приложения.

В случае появления различного рода системных ошибок, относящихся к работе данного приложения, необходимо убедиться в целостности данных, хранимых в директории приложения.

6. Рекомендации по освоению.

Данное приложение имеет достаточно простой и интуитивно понятный интерфейс, который не потребует от пользователя дополнительных знаний. Кроме того, в приложении имеется встроенное краткое руководство по эксплуатации с наглядными иллюстрациями, что позволяет ускорить и облегчить освоение программного продукта пользователем.

# ПРИЛОЖЕНИЕ Л

(обязательное)

**Руководство программиста**

1. Назначение и условия применения программы.

Данное программное приложение предназначено для организации игрового процесса пользователями. Предполагается наличие двух пользователей в качестве соперников. Управление игровыми объектами осуществляется посредством пользовательского взаимодействия с клавиатурой устройства.

Для корректной работы приложения необходимо осуществлять соблюдение следующих требований:

– компьютер на базе операционной системы Microsoft Windows;

– наличие клавиатуры с дополнительным блоком цифровых клавиш;

– наличие экрана с разрешением вывода 1920 на 1080 пикселей и выше;

– минимальное количество оперативной памяти в компьютере – 512 МБ.

1. Характеристики программы.

Данное приложение работает в автоматическом режиме. При запуске приложения работа с ним происходит в открывающихся окнах. Начальное окно является окном запуска приложения. Диалоговое окно инструктирует пользователя, предоставляет правила игры. Игровое окно предназначено непосредственно для осуществления игрового процесса.

1. Обращение к программе.

Запуск приложения производится путём открытия файла GameLauncher.exe в директории RacingGame2D\GameLauncher\bin\Debug.

1. Входные и выходные данные.

В качестве входных данных используется пользовательский ввод с клавиатуры. Выходными данными является отклик игрового приложения путём отображения изменений игрового процесса на экране.

1. Сообщения.

В процессе работы приложения не предусматривается вывода каких-либо специальных системных сообщений.

# ПРИЛОЖЕНИЕ М

(обязательное)

**Руководство системного программиста**

1. Общие сведения о программе.

Данное программное приложение предназначено для организации игрового процесса пользователями. Предполагается наличие двух пользователей в качестве соперников. Управление игровыми объектами осуществляется посредством пользовательского взаимодействия с клавиатурой устройства.

Для корректной работы приложения необходимо осуществлять соблюдение следующих требований:

– компьютер на базе операционной системы Microsoft Windows;

– наличие клавиатуры с дополнительным блоком цифровых клавиш;

– наличие экрана с разрешением вывода 1920 на 1080 пикселей и выше;

– минимальное количество оперативной памяти в компьютере – 512 МБ.

1. Структура программы.

Игровое приложение состоит из следующих логических модулей:

– графический пользовательский интерфейс Windows Forms;

– игровой двигатель;

– игровые объекты;

– компонент графической отрисовки;

– компонент физической модели твёрдого тела;

– компонент спрайтовой анимации.

1. Проверка программы.

Факт отсутствия системных ошибок свидетельствует о корректной работе приложения.

# ПРИЛОЖЕНИЕ Н

(обязательное)

**Схема использования паттерна «Фабричный метод»**