

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования
«Гомельский государственный технический университет
имени П.О.Сухого»

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

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

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту
по дисциплине «Объектно-ориентированное программирование»

на тему: «Игровое приложение *Windows Form* «Кольцевые гонки» с
использованием графики *OpenGL*»

Исполнитель: студент группы ИТИ-21
Ковалёв И.А.

Руководитель: доцент
Курочка К.С.

Дата проверки: _____

Дата допуска к защите: _____

Дата защиты: _____

Оценка работы: _____

Подписи членов комиссии
по защите курсового проекта: _____

Гомель 2025

СОДЕРЖАНИЕ

Введение.....	5
1 Игровые приложения и средства их разработки.....	6
1.1 Особенности жанра «Гонки»	6
1.2 Игровой процесс в разработанном приложении.....	7
1.3 Графическая библиотека <i>OpenGL</i> : обзор возможностей и применение в <i>Windows Form</i> и <i>C#</i>	8
1.4 Сравнительный анализ <i>OpenGL</i> и <i>DirectX</i>	11
1.5 Шаблоны проектирования в разработке игровых механизмов.....	13
2 Программная реализация игрового приложения «Кольцевые гонки»	15
2.1 Программные средства и шаблоны проектирования «Фабричный метод» и «Декоратор»	15
2.2 Структура классов приложения «Кольцевые гонки».....	18
3 Результаты тестирования и верификации приложения «Кольцевые гонки». 27	
3.1 Принцип работы игрового приложения	27
3.2 Результаты тестирования разработанного приложения	28
3.3 Результаты верификации разработанного приложения.....	30
Заключение	34
Список используемых источников.....	35
Приложение А Листинг программы «Кольцевые гонки».....	36
Приложение Б Руководство пользователя.....	82
Приложение В Руководство программиста.....	85
Приложение Г Руководство системного программиста	87
Приложение Д Внешний вид окон интерфейса программы.....	90
Приложение Ж Результаты опытной эксплуатации.....	92
Приложение И Схема использования паттерна «фабричный метод»	93

ВВЕДЕНИЕ

Курсовой проект сосредоточен на создании игрового приложения «Кольцевые гонки» для платформы *Windows Forms* с применением графики *OpenGL*. Главная цель – разработать полноценное приложение для двух игроков, где реализована механика гонок на одном экране с появлением призов на трассе. В процессе работы решаются задачи, связанные с созданием алгоритмов для управления автомобилями и взаимодействия объектов, использованием *OpenGL* для обеспечения качественной графики, применением шаблонов проектирования для гибкости кода, а также тестированием и проверкой работоспособности приложения.

Для реализации используются актуальные технологии и инструменты. Язык *C#* в среде *Windows Forms* упрощает разработку интерфейса и логики, а библиотека *OpenGL* обеспечивает плавное и производительное отображение игрового процесса. Применение таких шаблонов проектирования, как «фабричный метод» и «декоратор», делает код модульным и легко расширяемым, что соответствует современным подходам к разработке программного обеспечения.

Разрабатываемое приложение может стать полезным примером для изучения основ разработки игр и базой для дальнейших улучшений. В рамках работы анализируются существующие подходы, разрабатывается алгоритмическая основа, реализуется программная часть и проводится тестирование, что позволяет оценить эффективность предложенных решений.

1 ИГРОВЫЕ ПРИЛОЖЕНИЯ И СРЕДСТВА ИХ РАЗРАБОТКИ

1.1 Особенности жанра «Гонки»

Разрабатываемая игра относится к жанру «Гонки», который с первых дней своего появления неизменно привлекает внимание как игроков, так и разработчиков. Первые образцы этого жанра появились в конце 70-х – начале 80-х годов, когда аркадные автоматы с симуляторами гонок занимали центральное место в развлекательных залах торговых центров. Тогда игроки впервые испытали азарт стремительного движения и риск скоростных заездов, что быстро сделало жанр популярным среди широкой аудитории.

Одной из ключевых особенностей гоночных игр всегда было стремление передать реалистичное ощущение скорости и динамики. Разработчики уделяют особое внимание физике движения транспортных средств, тщательно моделируя нюансы ускорения, торможения, управляемости на поворотах и взаимодействия с различными дорожными покрытиями. Такая проработка позволяет игроку почувствовать себя настоящим гонщиком, где даже малейшая ошибка может повлиять на результат заезда.

С развитием технологий жанр «Гонки» претерпел значительные изменения. Современные симуляторы включают не только улучшенную физическую модель, но и продвинутые системы искусственного интеллекта, позволяющие создавать конкурентную среду с динамичным поведением соперников. Кроме того, интеграция технологий виртуальной (VR) и дополненной реальности (AR) открывает новые горизонты, позволяя полностью погрузиться в атмосферу настоящего заезда и сделать игровой процесс максимально интерактивным и захватывающим.

Немаловажным аспектом является развитие многопользовательских онлайн-соревнований. Благодаря этому игроки со всего мира могут участвовать в турнирах, чемпионатах и индивидуальных заездах, что способствует становлению киберспорта в этом направлении. Регулярное обновление контента, введение новых трасс, автомобилей и игровых режимов позволяют жанру оставаться актуальным и востребованным даже спустя десятилетия с момента его появления.

Современные гоночные игры также предлагают возможность тонкой настройки автомобилей. Игроки могут экспериментировать с характеристиками, модифицировать агрегаты и оптимизировать управляемость транспортного средства, что превращает каждый заезд в уникальное стратегическое испытание. Такой подход не только повышает вовлеченность, но и стимулирует развитие личных навыков, позволяя каждому пользователю находить индивидуальный стиль вождения.

Таким образом, жанр «Гонки» продолжает эволюционировать, объединяя в себе элементы аркадного азарта и глубоких симуляций. Технологический прогресс и инновационные подходы делают его не просто развлечением, а полноценной экосистемой, в которой каждая деталь – от реалистичной физики

до детализированного аудиовизуального оформления – способствует созданию по-настоящему захватывающего опыта для игроков всех уровней.

1.2 Игровой процесс в разрабатываемом приложении

Игровой процесс в приложениях, реализующих жанр кольцевых гонок, представляет собой сложную и многогранную систему, в которой сочетаются динамика, стратегия и реакция игрока. В данной курсовой работе разрабатывается игра «Кольцевые гонки» для двух пользователей, играющих на одном экране, где каждая секунда заезда имеет решающее значение.

Разрабатываемое приложение «Кольцевые гонки» представляет собой динамичную игру для двух пользователей, играющих на одном экране, где каждый игрок управляет гоночным автомобилем на кольцевой трассе. Цель игры – первым проехать пять кругов, при этом ключевым элементом становится не только скорость, но и грамотное использование бонусов, а также тактическое планирование расхода топлива. Ниже приведён подробный анализ основных аспектов игрового процесса, реализованных в данном проекте.

В основе приложения лежит идея состязания двух гонщиков на замкнутой трассе, где каждый заезд становится испытанием не только скорости, но и внимательности игрока. Игра начинается с недостаточным запасом топлива, что вынуждает участников активно собирать бонусы, расположенные по всей трассе. Таким образом, успех определяется умением сочетать агрессивное вождение с точным расчетом расхода топлива. Ограничение движения за границы трассы, которое приводит к снижению скорости, добавляет элемент дисциплины и требует от игрока постоянного контроля за положением автомобиля. Тестовое изображение игрового поля (трассы) представлено на рисунке 1.1.

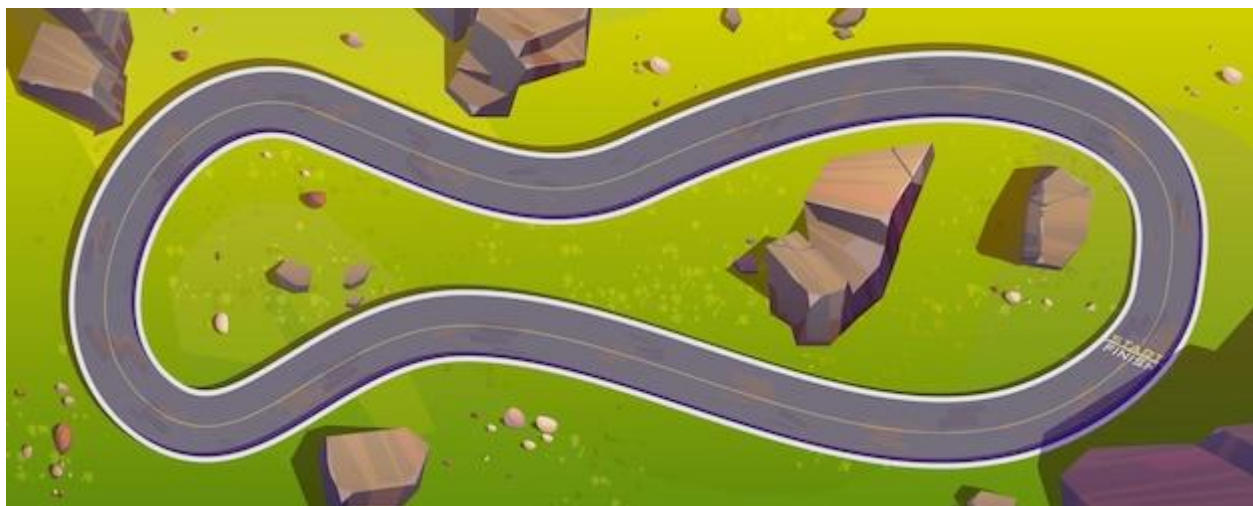


Рисунок 1.1 – Тестовое изображение игрового поля

Одним из центральных элементов игрового процесса является система расхода топлива. При запуске заезда у автомобиля имеется ограниченный запас топлива, которого недостаточно для прохождения полного круга. Это вынуждает

игроков искать и собирать бонусы, появляющиеся в случайных местах трассы. В игре реализованы следующие типы бонусов:

- топливо: пополнение запаса, необходимого для продолжения гонки. Собирая топливо, игрок получает возможность преодолеть критические участки трассы, избегая остановок и вынужденных замедлений;

- ускорение: временное увеличение максимальной скорости автомобиля, что позволяет сделать решающий обгон на прямых участках трассы. Такой бонус может кардинально изменить расстановку сил в заезде;

- замедление: временное снижение скорости соперника или корректировка собственной динамики, что помогает точнее проходить повороты или избегать столкновений.

Эта система бонусов вносит элемент стратегии: игрокам необходимо не только своевременно реагировать на появление бонусов, но и грамотно планировать момент их использования, чтобы оптимизировать расход топлива и максимально увеличить шансы на победу.

Режим игры для двух пользователей на одном экране создает условия для непосредственного противостояния, где каждый игрок видит действия соперника в режиме реального времени. Это исключает задержки, присущие онлайн-соревнованиям, и гарантирует синхронное отображение всех игровых событий. Такая организация игрового процесса способствует более тесному взаимодействию между игроками, повышая уровень адреналина и конкурентоспособности.

При этом система управления, рассчитанная на двух игроков, позволяет каждому участнику полностью контролировать своего автомобиля. Возможность мгновенного реагирования на действия соперника и оперативное использование бонусов превращают каждую гонку в напряженное и захватывающее состязание, где успех определяется не только техническими характеристиками автомобиля, но и стратегией, быстрой реакцией и тактическим мышлением.

1.3 Графическая библиотека *OpenGL*: обзор возможностей и применение в *Windows Form* и *C#*

В области создания визуальных приложений и компьютерной графики, графическая библиотека *OpenGL* занимает одну из ключевых позиций благодаря своей универсальности и способности эффективно использовать ресурсы графического оборудования. *OpenGL* представляет собой стандартизированный программный интерфейс (*API*), разработанный для взаимодействия с графическими ускорителями, поддерживающий создание как двухмерной, так и трехмерной графики [1, стр. 24]. Его кроссплатформенный характер делает его привлекательным выбором для разработчиков, стремящихся создавать приложения, способные работать на различных операционных системах и аппаратных конфигурациях.

Для проектов, подобных "Кольцевым гонкам", *OpenGL* предоставляет фундаментальные инструменты для отрисовки всех визуальных элементов: от статических изображений трассы до динамично перемещающихся объектов, таких как автомобили и призы. Возможность использования аппаратного ускорения графического процессора через *OpenGL* позволяет добиться высокой производительности рендеринга, что критически важно для обеспечения плавного отображения игрового процесса и создания захватывающего визуального ряда.

Современные версии *API*, начиная с третьего поколения, акцентируют внимание на использовании шейдерных программ – небольших исполняемых фрагментов кода, которые выполняются непосредственно на графическом процессоре. Этот переход к программируемому конвейеру обработки графики открыл широчайшие возможности для реализации сложных визуальных эффектов, детального управления процессами текстурирования, освещения и постобработки, предоставляя разработчикам беспрецедентный контроль над финальным изображением [3, стр. 74]. Схема графического конвейера *OpenGL* представлена на рисунке 1.2. Такая эволюция делает *OpenGL* не только мощным, но и гибким инструментом для реализации самых современных алгоритмов рендеринга.

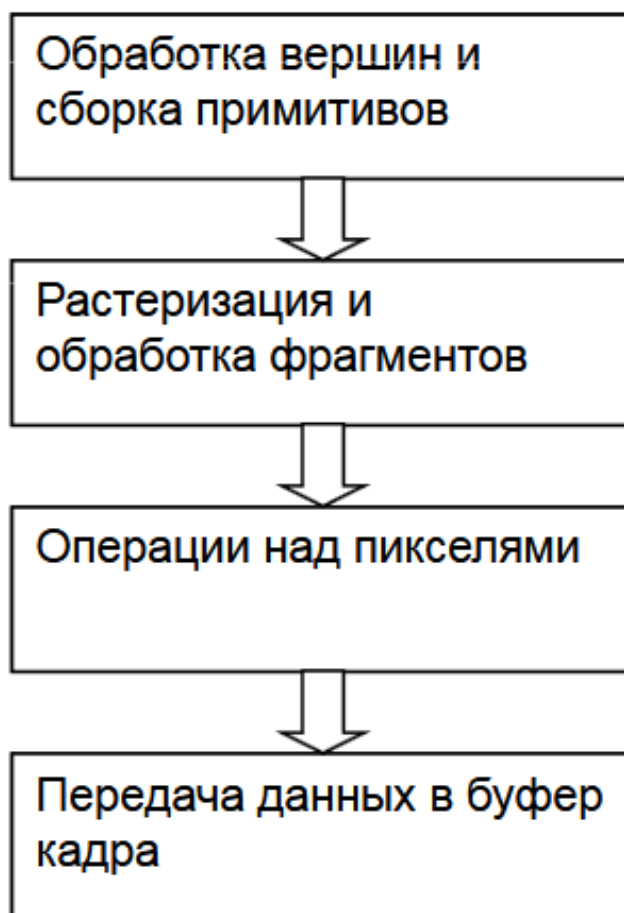


Рисунок 1.2 – Схема графического конвейера

OpenGL предоставляет ряд ключевых возможностей, которые делают его незаменимым инструментом для разработки графики [5, стр. 2]:

- аппаратное ускорение: использование *GPU* позволяет значительно ускорить рендеринг, что особенно важно для динамичных сцен и игр, где требуется высокая частота обновления кадров;

- кроссплатформенность: *OpenGL* поддерживается на различных операционных системах, включая *Windows*, *macOS* и *Linux*, что облегчает переносимость приложений;

- поддержка 2D и 3D графики: *API* позволяет работать как с двумерными изображениями (например, спрайтовой графикой), так и с трехмерными объектами, что дает разработчику широкие возможности для творчества;

- гибкость через шейдеры: программируемые шейдеры дают возможность реализовывать индивидуальные визуальные эффекты, варьируя освещение, тени, отражения и прочие параметры в реальном времени;

- оптимизация рендеринга: многочисленные функции *OpenGL* позволяют оптимизировать процесс отрисовки за счет использования буферов, инстансинга и других современных методов.

Эти возможности обеспечивают высокую производительность и визуальную привлекательность конечного продукта, что является основополагающим для реализации игровых приложений с насыщенной графикой.

В контексте реализации приложения «Кольцевые гонки», ключевым аспектом стала возможность объединения стандартного фреймворка для создания пользовательских интерфейсов *Windows* с мощными средствами аппаратного ускорения графики. Выбор связки *Windows Forms* на языке *C#* для элементов управления и *OpenGL* для визуализации позволил создать удобный для пользователя интерфейс и одновременно достичь высокой производительности при отрисовке игровых сцен, что важно для динамичных аркадных гонок, требующих плавности и отзывчивости.

Для обеспечения взаимодействия между управляемым кодом *C#* и низкоуровневым графическим *API OpenGL* применяются специализированные библиотеки-обертки. Такие решения, как *OpenTK*, *SharpGL* или *OpenGL.Net*, упрощают процесс интеграции, предоставляя набор классов и методов, соответствующих функциям *OpenGL*, доступных непосредственно из *C#* [2, стр. 52]. Эти библиотеки значительно облегчают задачи инициализации графического контекста, создания окна или элемента управления, предназначенного для рендеринга, а также управления графическими ресурсами и состоянием конвейера *OpenGL*. В проекте «Кольцевые гонки» использовался именно такой подход для бесшовного сочетания привычной среды разработки пользовательских интерфейсов с возможностями высокопроизводительной графики.

Практическая реализация рендеринга *OpenGL* в приложении на базе *Windows Forms* включает создание специального компонента визуализации, который интегрируется в структуру формы. В случае использования *OpenTK*, таким компонентом является *GLControl* – элемент управления,

предоставляющий область для отрисовки графики и механизмы для взаимодействия с контекстом *OpenGL* [4, стр. 26]. Разработчику при этом требуется учитывать специфику управления ресурсами графического процессора, корректно инициализировать графический контекст при старте приложения и синхронизировать процесс отрисовки с игровым циклом для обеспечения стабильной частоты кадров, что особенно актуально в игровых приложениях.

Сочетание *OpenGL* с *Windows Forms* и *C#* в приложении «Кольцевые гонки» представляет собой эффективное решение, обеспечивающее баланс между удобством разработки интерфейса и графической мощностью. Этот подход не только позволил успешно реализовать визуализацию игровых объектов и анимацию, но и способствовал интеграции объектно-ориентированных архитектурных решений, таких как применение шаблонов проектирования "фабричный метод" и "декоратор", для создания гибкой и расширяемой структуры кода. В итоге, выбранная интеграция технологий способствовала созданию производительного и визуально привлекательного приложения, демонстрируя потенциал современных средств разработки графических приложений на платформе *.NET*.

1.4 Сравнительный анализ *OpenGL* и *DirectX*

В контексте выбора графического *API* для разработки приложений, важное место занимает сравнение *OpenGL* и *DirectX*. Изначально *OpenGL* создавался как программный интерфейс, не привязанный к конкретной операционной системе, что обеспечивает ему возможность функционировать на широком спектре платформ, включая распространенные десктопные ОС, такие как *Windows*, *macOS* и дистрибутивы *Linux*. Это качество делает *OpenGL* привлекательным выбором для разработчиков, цель которых — создание мультиплатформенных приложений [2, стр. 29]. В противоположность этому, *DirectX* разработан корпорацией *Microsoft* и предназначен эксклюзивно для работы в среде операционных систем *Windows*, что, естественно, ограничивает его применимость в проектах, требующих совместимости с другими программными окружениями.

С точки зрения внутренней архитектуры, *OpenGL* функционирует по принципу клиент-серверной модели, где приложение выступает в роли клиента, отправляющего команды графическому драйверу (серверу), который берет на себя основную работу по их обработке и управлению графическими ресурсами [1, стр. 77]. Такая модель обеспечивает определенную гибкость *API*. *DirectX*, в частности его компоненты, такие как *Direct3D*, используют иной подход, где управление ресурсами в большей степени возлагается на само приложение. Это дает разработчикам более тонкий контроль над процессами рендеринга, но может потребовать и более детальной работы со стороны программиста.

Вопрос производительности между этими двумя *API* на протяжении долгого времени оставался предметом активных дискуссий среди разработчиков. В ранних своих версиях *DirectX* иногда подвергался критике за излишнюю

сложность и не всегда оптимальную производительность по сравнению с *OpenGL* [4, стр. 80]. Однако с эволюцией и совершенствованием технологий *Microsoft* существенно улучшила *DirectX*, и современные его версии демонстрируют производительность, вполне сопоставимую с *OpenGL*. Некоторые специалисты указывают на то, что *DirectX* может предоставлять более прямой доступ к низкоуровневым функциям графического оборудования, что в определенных, специфических сценариях может позитивно сказываться на финальной производительности.

Что касается простоты освоения и удобства использования, *OpenGL* часто воспринимается как *API* с более понятной и последовательной структурой, что может облегчать порог вхождения для новых разработчиков. *DirectX*, особенно в его исторических версиях, считался более сложным для изучения, отчасти из-за своей архитектуры и необходимости более глубокого управления ресурсами на стороне приложения. Тем не менее, с выходом новых версий и улучшений, предпринятых *Microsoft*, *DirectX* стал более доступным и интуитивно понятным для разработчиков.

В части поддержки новых функций оборудования, *OpenGL* предоставляет производителям графических карт возможность добавлять собственные расширения к *API* [4, стр. 88]. Это позволяет довольно быстро внедрять поддержку новейших возможностей железа. Однако обратной стороной такого подхода может стать фрагментация *API* и потенциальные сложности с обеспечением одинаковой функциональности и поведения на оборудовании разных производителей. *DirectX* придерживается более строгого подхода к спецификациям и реализации функций, что обеспечивает большую согласованность и предсказуемость работы на различных системах, но может приводить к некоторой задержке во внедрении поддержки самых последних инноваций в графическом оборудовании.

Хотя оба *API* изначально разрабатывались для работы с 3D-графикой, они также поддерживают 2D-графику. В случае *OpenGL* для работы с 2D-графикой используются ортографические проекции и соответствующие функции рендеринга. *DirectX* включает компонент *Direct2D*, специально предназначенный для работы с 2D-графикой, что упрощает разработку 2D-приложений на платформе *Windows*. Сравнение *OpenGL* и *DirectX* представлено в таблице 1.1.

Таблица 1.1 – Сравнение *API*

Характеристика	<i>OpenGL</i>	<i>DirectX</i>
Кроссплатформенность	<i>Windows, macOS, Linux</i>	<i>Windows, Xbox</i>
Оптимизация	Хорошая, но требует дополнительной настройки	Глубокая интеграция с ОС, высокая производительность
Аппаратное ускорение	Есть, но зависит от драйверов	Гарантированное аппаратное ускорение
Сообщество	Активное, много учебных материалов	Хорошая документация от <i>Microsoft</i>

Выбор между *OpenGL* и *DirectX* зависит от конкретных требований проекта. Если необходимо обеспечить кроссплатформенность и гибкость, *OpenGL* является предпочтительным выбором. Для проектов, ориентированных исключительно на *Windows* и требующих глубокого взаимодействия с системой, *DirectX* может предоставить более оптимальные возможности.

1.5 Шаблоны проектирования в разработке игровых механизмов

Современная разработка игр предъявляет высокие требования к архитектуре программного обеспечения, требуя создания систем, способных легко адаптироваться к новым функциям и растущей сложности. Проверенные временем подходы к решению типовых задач проектирования, известные как шаблоны, играют центральную роль в структурировании кода, повышая его гибкость, понимаемость и устойчивость к возможным проблемам. В особенности это касается игровых приложений, где элементы часто зависят друг от друга, а логика должна обрабатывать множество динамических состояний. Применение шаблонов в таких проектах переходит из категории рекомендации в категорию необходимости. Задачи, такие как процедурная генерация внутриигровых объектов, динамическое изменение свойств игровых сущностей или управление сложными последовательностями событий, требуют использования методик, которые минимизируют жесткую связь между компонентами и позволяют добавлять новые возможности без переработки уже существующего кода.

Одним из основополагающих шаблонов в разработке интерактивных приложений является «фабричный метод». Этот паттерн находит свое применение там, где требуется создавать объекты, реализующие один и тот же интерфейс, но имеющие различное внутреннее устройство. В играх, где объекты, такие как бонусы или противники, появляются на игровом поле в зависимости от определенных условий или случайно, фабричный метод позволяет передать ответственность за создание экземпляров специализированным классам-генераторам (фабрикам). Такой подход не только упрощает процесс добавления новых видов объектов в игру, но и способствует централизованному контролю над их свойствами и правилами появления. Рассматривая процесс создания призов на трассе – например, бонусов топлива, ускорения или замедления – каждый тип приза может быть ассоциирован со своей собственной фабрикой. Это дает возможность гибко управлять частотой появления каждого типа бонуса, определять их начальные характеристики и даже задавать условия, при которых они могут появляться на определенных участках уровня. Более того, использование фабричного метода хорошо согласуется с принципами внедрения зависимостей, что упрощает независимое тестирование отдельных модулей игры.

Еще одним важным инструментом для создания динамических игровых систем является шаблон "декоратор". В игровых сценариях, где игровые объекты могут временно изменять свое поведение или комбинировать различные эффекты, использование традиционного наследования может быстро привести к

разрастанию иерархии классов и стать трудноуправляемым. Декоратор эффективно решает эту проблему, позволяя динамически "оборачивать" объект дополнительной функциональностью. Например, базовый объект автомобиля со своими стандартными характеристиками движения может быть дополнен декоратором, который временно модифицирует его максимальную скорость для реализации эффекта ускорения. Каждый декоратор обычно реализует тот же интерфейс, что и декорируемый объект, что делает процесс добавления или удаления эффектов прозрачным для остальной части кода, использующей объект. Это особенно ценно в системах, где состояние объектов может меняться в реальном времени.

Важно также отметить, как шаблоны проектирования могут взаимодействовать с графическими библиотеками, такими как *OpenGL*. Например, декораторы, предназначенные для изменения визуального представления объекта (цвет, эффекты свечения), могут инкапсулировать специфичные вызовы к графическому *API*, тем самым отделяя логику визуальных эффектов от деталей низкоуровневой отрисовки. Такой подход соответствует принципам проектирования, направленным на минимизацию зависимостей и повышение модульности.

В заключение, внедрение шаблонов проектирования в механику игры – это не просто формальное следование стандартам, а стратегически обоснованное решение, направленное на построение устойчивой, масштабируемой и легко поддерживаемой архитектуры. Применение паттернов позволяет разработчикам сосредоточиться на творческих задачах геймдизайна и реализации новых идей, не будучи скованными жесткостью или запутанностью кодовой базы. В условиях быстро развивающейся игровой индустрии, способность эффективно использовать и комбинировать шаблоны проектирования является одним из ключевых навыков для любого профессионального игрового программиста.

2 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ «КОЛЬЦЕВЫЕ ГОНКИ»

2.1 Программные средства и шаблоны проектирования «Фабричный метод» и «Декоратор»

Для разработки программного обеспечения используется интегрированная среда *Visual Studio 2022*, а также компоненты, обеспечивающие поддержку платформы *.NET* и позволяющие создавать как библиотеки классов, так и клиентские приложения на базе *Windows Forms*.

Для интеграции с *OpenGL* подключаются необходимые библиотеки: *OpenTK* и *OpenTK.GLControl*.

Для создания спрайтовой графики в формате *.png* используется графический редактор *Aseprite*.

С целью обеспечения расширяемости и гибкости архитектуры разрабатываемого приложения применяются шаблоны проектирования.

Фабричный метод (*Factory Method*) – это один из наиболее популярных порождающих шаблонов проектирования. Его основная задача – делегировать создание объектов подклассам, тем самым позволяя использовать в коде объекты, не зная их конкретных классов. Это особенно важно в ситуациях, когда система должна быть расширяема и модифицируема без изменения существующего кода.

Идея шаблона заключается в создании абстрактного метода (или интерфейса), который определяет общий способ создания объектов. Конкретные реализации этого метода в подклассах возвращают объекты конкретных типов. Таким образом, клиентский код работает с абстракцией, а конкретный тип объекта подсовывается фабрикой.

В разрабатываемом приложении «Кольцевые гонки» используется шаблон проектирования Фабричный метод (*Factory Method*) для создания различных типов призов (топливо, ускорение, замедление) без привязки к конкретным классам в клиентском коде. Это позволяет изолировать процесс создания объектов от их использования и облегчает добавление новых типов призов в будущем. Этот подход позволяет создавать призы динамически и гибко управлять их типами, не нарушая принцип открытости/закрытости. Например, чтобы добавить новый тип приза, достаточно реализовать новый класс-награду и соответствующую фабрику, не изменяя остальной код.

На рисунке 2.1 представлена схема реализации паттерна «фабричный метод».

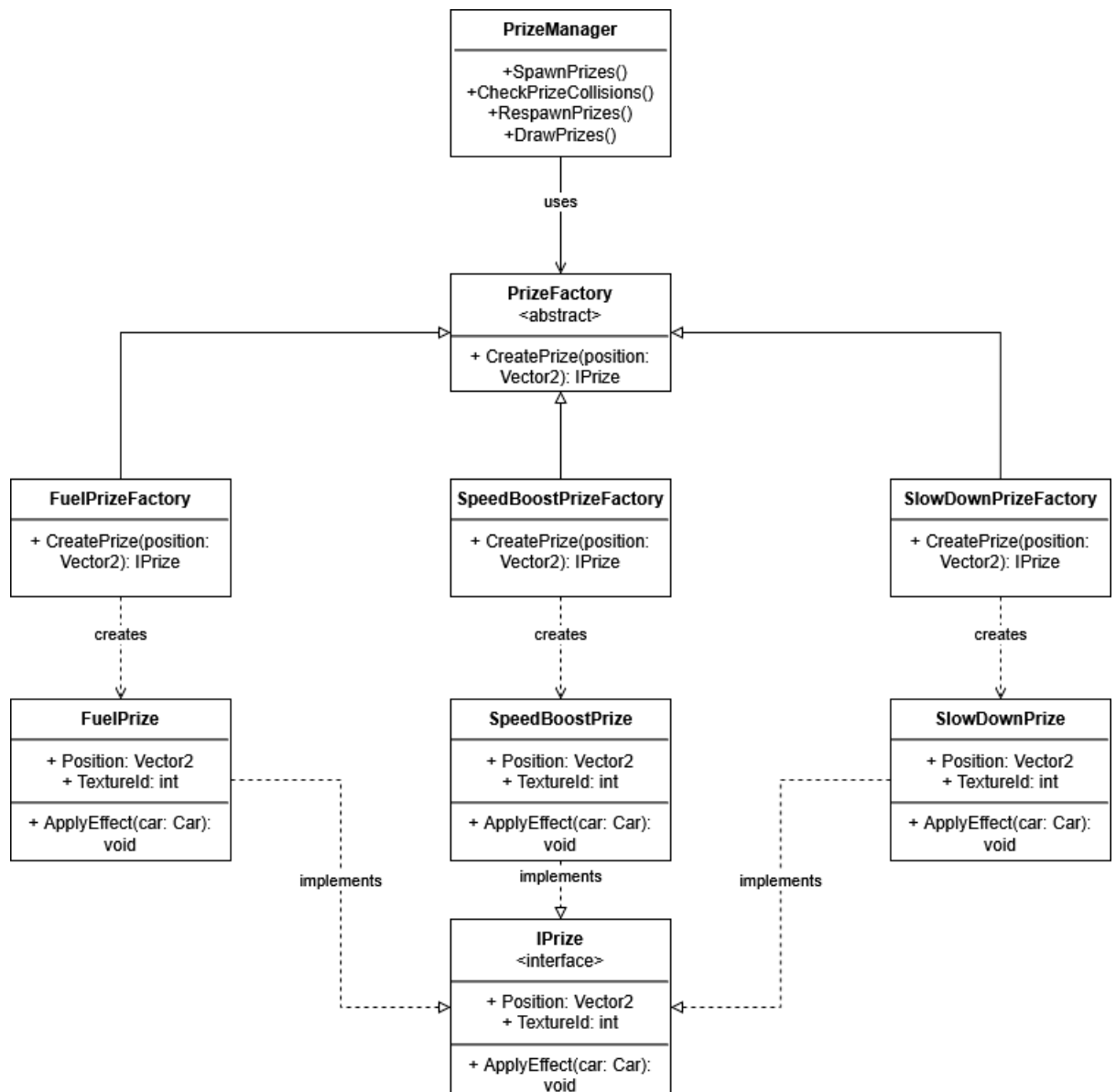


Рисунок 2.1 – Схема реализации паттерна «фабричный метод»

В свою очередь шаблон проектирования «Декоратор» относится к структурным шаблонам и предназначен для динамического добавления объектам новой функциональности без изменения их исходного кода. Это особенно полезно, когда нужно модифицировать поведение объектов, не прибегая к громоздкому наследованию и не нарушая принцип открытости/закрытости (*OCP*) из *SOLID*.

Основная идея «Декоратора» состоит в том, чтобы обернуть один объект другим, который реализует тот же интерфейс или наследует тот же базовый класс. Объект-декоратор перехватывает вызовы и, при необходимости, добавляет новое поведение – до, после или вместо вызова оригинального объекта. При этом основной объект не знает, что он обернут – он продолжает функционировать, как обычно. Таким образом, декораторы можно наслаивать друг на друга, создавая цепочку поведения, расширяющую возможности базового объекта.

В приложении «Кольцевые гонки» с помощью шаблона проектирования «Декоратор» (*Decorator*) происходит изменение максимальной скорости машины при подборе приза. Основной класс *Car* содержит ссылку на текущий активный декоратор (*_currentDecorator*), который может динамически изменять поведение машины, не изменяя её структуру.

Каждый конкретный эффект, такой как ускорение или замедление, наследует абстрактный класс *CarDecorator*, который определяет общий интерфейс для всех декораторов. При активации эффекта машина вызывает метод, который устанавливает новый декоратор, применяющий соответствующее изменение – например, изменение параметра *ForwardMaxSpeed* у конфигурации движения машины. Через заданный интервал времени декоратор автоматически завершает своё действие и возвращает параметры в исходное состояние.

Таким образом, декоратор оборачивает машину дополнительным временным поведением без изменения самого класса *Car*. Это позволяет гибко расширять поведение автомобиля (например, добавлять новые эффекты) и применять их независимо друг от друга, что является типичным применением шаблона Декоратор.

На рисунке 2.2 представлена схема реализации паттерна «декоратор».

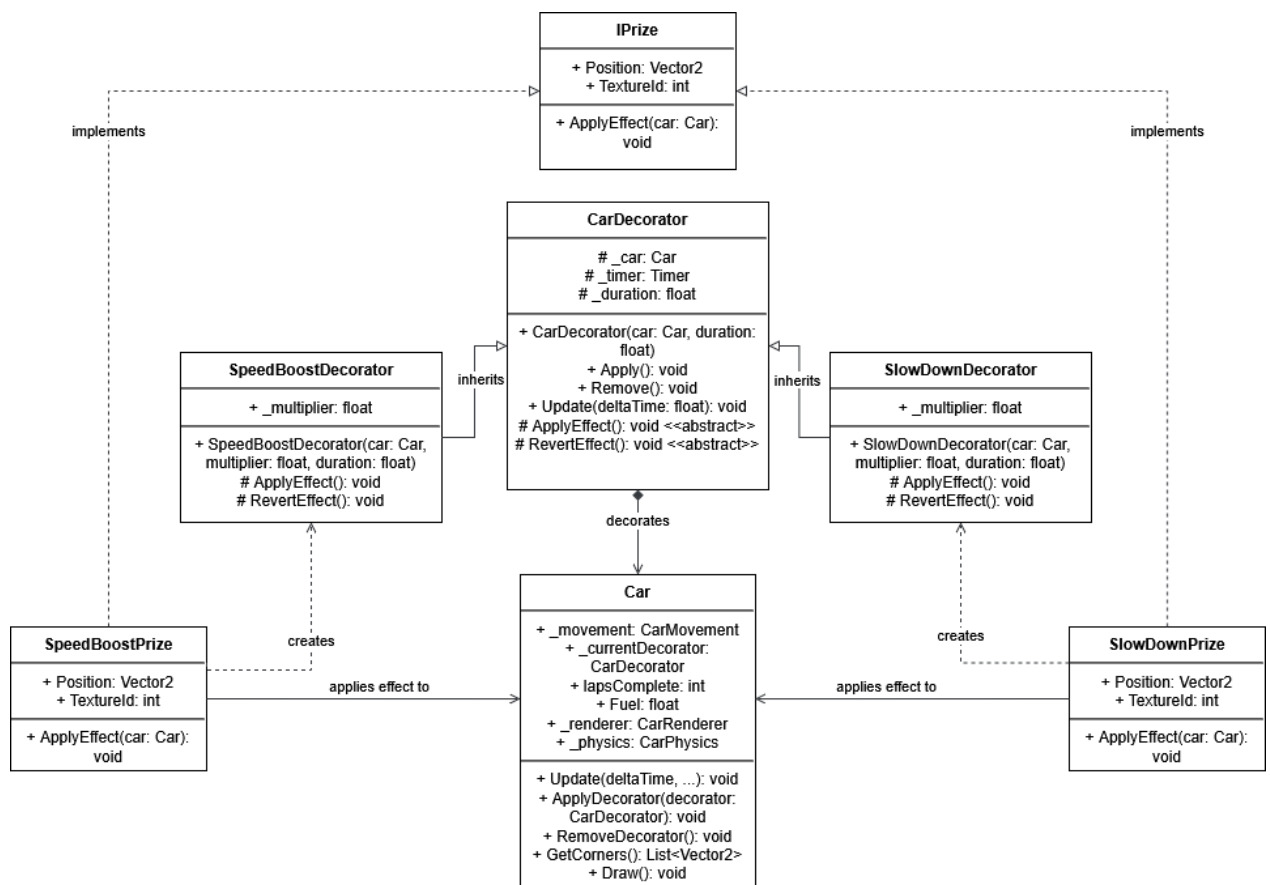


Рисунок 2.2 – Схема реализации паттерна «декоратор»

2.2 Структура классов приложения «Кольцевые гонки»

2.2.1 Проектирование структуры классов приложения "Кольцевые гонки" осуществляется с учетом принципов объектно-ориентированного подхода и направлено на создание гибкой, расширяемой и легко поддерживаемой архитектуры. Выделяются основные функциональные слои: пользовательский интерфейс (*UI*), реализуемый средствами *Windows Forms*, и ядро игровой логики, ответственное за симуляцию игрового мира, взаимодействие объектов и управление игровым процессом.

В основе иерархии классов лежит разделение на управляющие компоненты (контроллеры и менеджеры) и игровые сущности (объекты, присутствующие в игровом мире). Контроллеры отвечают за управление состоянием приложения и взаимодействие с пользовательским интерфейсом. Менеджеры координируют действия и состояния игровых объектов. Игровые сущности инкапсулируют собственное состояние и поведение.

Использование шаблонов проектирования оказывает существенное влияние на структуру. Например, паттерн «Фабричный метод» реализуется для создания различных типов бонусов через общий интерфейс, а паттерн «Декоратор» применяется для динамического изменения характеристик автомобилей под действием бонусов.

2.2.2 Архитектура проекта разделяет логику представления (*UI*), основную игровую механику и вспомогательные утилиты по разным пространствам имен (*RingRaceApp*, *RingRaceLab*, *RingRaceLab.Game*, *RingRaceLab.Menu*), обеспечивая модульность и управляемость кода.

Основным окном приложения является класс *Form1* (Приложение А, код программы *Form1.cs*), расположенный в пространстве имен *RingRaceApp*. Ключевая роль *Form1* заключается в инициализации и управлении двумя основными состояниями приложения: главным меню и непосредственно игровым процессом. Это достигается через создание экземпляров контроллеров, реализующих интерфейсы *IMenuController* и *IGameController*. Форма хранит ссылки на панели (*MenuPanel* и *GamePanel*), связанные с каждым контроллером, и добавляет их в коллекцию своих элементов управления. Переключение между меню и игрой осуществляется методами *ShowMenu()* и *ShowGame()*, которые отвечают за скрытие одной панели и отображение другой.

Когда игрок выбирает старт гонки в меню, управление переходит к *GameController*, который, в свою очередь, создает и запускает *GameManager* (Приложение А, код программы *GameManager.cs*). *GameManager*, является сердцем игрового процесса. Он отвечает за координацию всех аспектов активной гонки. В его конструктор передаются все необходимые данные для инициализации сессии: информация о выбранной трассе, стартовые позиции машин, координаты финишной линии, текстуры для автомобилей обоих игроков и размеры игрового окна. Основные обязанности *GameManager*:

- управление игровым циклом: основная логика каждого кадра выполняется в методе *GameManager.Update()*. В каждом вызове этого метода происходит расчет времени, прошедшего с предыдущего кадра (*deltaTime*).

Синхронизация действий игроков и состояния игрового мира реализуется путем последовательной обработки ввода от обоих игроков и обновления состояния их автомобилей, а также других игровых объектов (бонусов, коллизий) в рамках одного вызова *Update*, используя рассчитанный *deltaTime*. Это гарантирует, что игровые события для обоих участников обрабатываются одновременно с точки зрения игрового времени, обеспечивая единую и непротиворечивую картину игрового процесса на одном экране;

- управление сущностями: *GameManager* использует экземпляр *EntityManager* для хранения и управления списком активных игровых сущностей, таких как трасса и автомобили игроков (*Car1*, *Car2*);

- отслеживание состояния игры: он следит за состоянием машин, проверяет пересечение финишной линии для подсчета кругов и генерирует событие *OnCarFinished*, когда один из игроков завершает необходимое количество кругов;

- отрисовка сцены: метод *Draw()* отвечает за визуализацию игрового мира. Он очищает экран, затем последовательно вызывает методы *Draw()* у всех активных сущностей, а также у менеджера бонусов и отрисовщика интерфейса.

Важно отметить базовые классы и структуры проекта, а также игровую трассу и коллизии:

- *GameEntity* (Приложение А, код программы *GameEntity.cs*): это простой абстрактный класс, служащий основой для всех объектов, которые должны быть отрисованы на игровом экране;

- *EntityManager* (Приложение А, код программы *EntityManager.cs*): расположенный в пространстве имен *RingRaceLab.Game*, этот класс представляет собой простой контейнер для управления списком объектов *GameEntity*;

- *Track* (Приложение А, код программы *Track.cs*): класс, представляющий гоночную трассу, наследуется от *GameEntity*. В конструкторе он получает путь к файлу текстуры, массив стартовых позиций, координаты для определения финишной линии и размеры игрового поля. Метод *Draw()* этого класса отвечает за отрисовку текстуры трассы на весь экран;

- *CollisionMask* (Приложение А, код программы *CollisionMask.cs*): этот класс играет важную роль в определении границ трассы, по которым могут перемещаться машины. Основная функция класса – метод *CheckCollision(Car car)*, который проверяет, не вышли ли углы переданной машины за пределы допустимой зоны. Если хотя бы один угол находится на не проезжаемой поверхности, метод возвращает *true*, сигнализируя о столкновении;

- *FinishLine* (Приложение А, код программы *FinishLine.cs*): небольшой класс, представляющий собой линию старта/финиша. Он хранит координаты начальной и конечной точек линии. Метод *CheckCrossing* использует вспомогательный класс *LineIntersection* для определения, пересекла ли машина эту линию за последний кадр, и в каком направлении.

На рисунке 2.3 представлена высокоуровневая архитектура приложения, иллюстрирующая основные компоненты и их взаимосвязи, что дает общее представление о структуре программы.

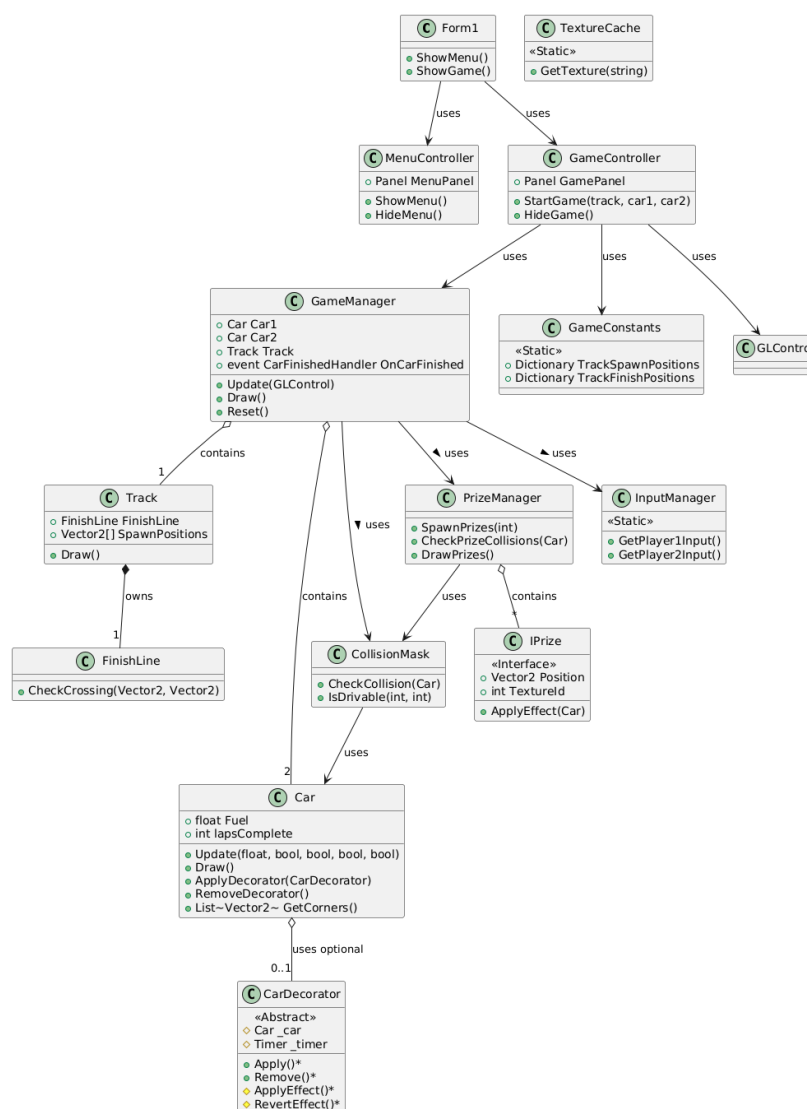


Рисунок 2.3 – Высокоуровневая архитектура приложения

2.2.3 Центральной игровой сущностью, управляемой игроком, является класс *Car* (Приложение А, код программы *Car.cs*). Как и трасса, он наследуется от базового класса *GameEntity*, что обязывает его реализовывать метод *Draw()* для отрисовки. Архитектурно класс *Car* построен по принципу композиции: он не содержит всю логику внутри себя, а делегирует специфические задачи отдельным компонентам: *_movement* (движение), *_renderer* (отрисовка) и *_physics* (физические расчеты для коллизий).

При создании объекта *Car* (внутри *GameManager*) в его конструктор передаются начальная позиция, путь к файлу текстуры и объект *CarConfig*. На основе этих данных создаются внутренние компоненты *CarMovement*, *CarRenderer* и *CarPhysics*. Помимо компонентов, класс *Car* хранит собственное состояние, важное для игры:

- *lapsComplete*: счетчик пройденных кругов;
- *Fuel*: уровень топлива. Топливо расходуется во время движения;

- *_currentDecorator*: ссылка на текущий активный бонусный эффект (декоратор), если он есть.

Класс *Car* содержит метод *Update*, который вызывается каждый кадр из *GameManager*. Он принимает время кадра (*deltaTime*) и булевы флаги, соответствующие нажатию клавиш управления (вперед, назад, влево, вправо). Внутри метода происходит следующее:

- расход топлива: рассчитывается количество потраченного топлива за кадр. Расход зависит от текущей скорости машины (*_movement.CurrentSpeed*), времени кадра и коэффициента расхода;

- делегирование движения: если у машины еще есть топливо, вызывается метод *Update* компонента *_movement*, которому передаются *deltaTime* и флаги пользовательского ввода. Именно *CarMovement* обчисляет изменение скорости, угла поворота и позиции;

- обновление декоратора: если к машине применен временный эффект, вызывается метод *Update* объекта *_currentDecorator*.

Класс *Car* также предоставляет методы *ApplyDecorator* и *RemoveDecorator* для управления временными эффектами. Отрисовка машины полностью делегируется компоненту *_renderer*, а расчет координат углов – компоненту *_physics*.

Класс *CarConfig* (Приложение А, код программы *CarConfig.cs*) представляет собой простой контейнер данных, который хранит набор параметров, определяющих характеристики автомобиля (ускорение, максимальная скорость, размер и т.д.).

Использование *CarConfig* позволяет легко настраивать поведение машин и потенциально создавать разные типы автомобилей с уникальными характеристиками, передавая разные конфигурации в конструктор *Car*.

Класс *CarMovement* (Приложение А, код программы *CarMovement.cs*) инкапсулирует всю сложную логику, связанную с перемещением и ориентацией машины в пространстве. Класс также содержит ссылку на объект *CarConfig* для доступа к параметрам физики.

Основная работа происходит в методе *Update*. Скорость изменяется в зависимости от нажатых клавиш «вперед» или «назад». Применяются разные коэффициенты ускорения из *CarConfig* в зависимости от того, набирает ли машина скорость в текущем направлении или тормозит/начинает движение в противоположном. Скорость ограничивается максимальными значениями из конфигурации. Машина может поворачивать, только если она движется (*Math.Abs(CurrentSpeed) > 0.1f*). Скорость поворота не постоянна, она зависит от текущей скорости машины относительно максимальной – чем выше скорость, тем медленнее поворот. Угол изменяется на основе нажатых клавиш «влево» или «вправо» и времени кадра. Метод *UpdatePosition* отвечает за обновления позиции машины.

Компонент *CarRenderer* (Приложение А, код программы *CarRenderer.cs*) отвечает исключительно за отрисовку спрайта машины в правильном месте и под нужным углом.

Класс *CarPhysics* (Приложение А, код программы *CarPhysics.cs*) предоставляет информацию о физических границах машины, необходимую для системы определения столкновений.

Топливо является важным ресурсом в игре. Как упоминалось, оно расходуется в *Car.Update* пропорционально скорости. Если топливо заканчивается (*Fuel* ≤ 0), машина больше не может двигаться, так как вызов *_movement.Update()* блокируется.

Подсчет кругов реализован в *GameManager*. После каждого обновления позиции машины, менеджер проверяет, пересекла ли машина финишную линию за этот кадр. В соответствии с результатом обновляется счетчик *Car.lapsComplete*. Когда счетчик достигает установленного значения (5 кругов), *GameManager* генерирует событие *OnCarFinished*, что приводит к завершению гонки.

Диаграмма, фокусирующаяся на классе *Car* и его компонентах (связях) представлена на рисунке 2.4.

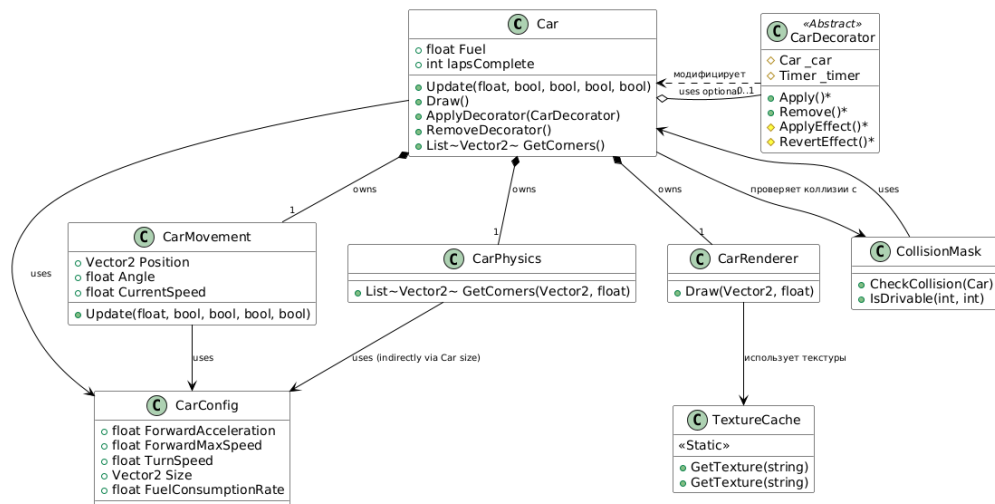


Рисунок 2.4 – Диаграмма класса *Car*, его компонентов и связей

2.2.4 Для добавления динамики и элемента случайности в гоночный процесс, на трассе размещаются коллекционные предметы – бонусы (призы). Эти бонусы могут давать игроку временное преимущество (например, ускорение или пополнение топлива) или накладывать негативный эффект. Управление всем жизненным циклом этих бонусов возложено на класс *PrizeManager* (Приложение А, код программы *PrizeManager.cs*), расположенный в пространстве имен *RingRaceLab.Game*.

PrizeManager является ключевым компонентом системы бонусов. Он создается в *GameManager* и получает ссылки на фабрики для создания различных типов бонусов, а также на систему коллизий (*CollisionMask*) для определения корректных мест размещения. *PrizeManager* отвечает за следующие аспекты:

- размещение: метод *SpawnPrizes* используется для первоначального и последующего размещения бонусов на карте;

- проверка подбора: метод *CheckPrizeCollisions* вызывается для каждой машины в каждом кадре игры. Он проверяет расстояние от машины до каждого активного бонуса. Если машина подъезжает достаточно близко к бонусу, считается, что он подобран;

- поддержание количества: *PrizeManager* использует внутренний таймер (*_prizeRespawnTimer*) для периодического вызова метода *RespawnPrizes*. Этот метод проверяет, не стало ли количество активных бонусов на трассе меньше установленного минимума (*MIN_PRIZES*). Если бонусов мало, он вызывает *SpawnPrizes*;

- отрисовка: метод *DrawPrizes* отвечает за визуализацию всех активных бонусов. Он перебирает список *_activePrizes* и отрисовывает каждый бонус на его позиции, используя соответствующую текстуру.

Для создания экземпляров бонусов используется паттерн «Фабричный метод». Это позволяет *PrizeManager* не зависеть от конкретных классов бонусов, а работать с ними через общий интерфейс и абстрактную фабрику. Интерфейс *IPrize* (Приложение А, код программы *IPrize.cs*) определяет общий контракт для всех бонусов: он требует наличия свойств *Position* (позиция на карте) и *TextureId* (идентификатор текстуры для отрисовки), а также метода *ApplyEffect(Car car)*, который инкапсулирует логику эффекта, применяемого к машине при подборе бонуса. Абстрактный класс *PrizeFactory* (Приложение А, код программы *PrizeFactory.cs*) объявляет абстрактный метод *CreatePrize(Vector2 position)*. Для каждого типа бонуса (*FuelPrize*, *SpeedBoostPrize*, *SlowDownPrize*) существует своя конкретная реализация фабрики (*FuelPrizeFactory*, *SpeedBoostPrizeFactory*, *SlowDownPrizeFactory*), которая наследуется от *PrizeFactory* и реализует метод *CreatePrize*, возвращая экземпляр соответствующего конкретного бонуса.

В проекте реализованы три типа бонусов, каждый со своим эффектом:

- *FuelPrize* (Приложение А, код программы *FuelPrize.cs*): при подборе немедленно пополняет запас топлива подобравшей машины на 25 единиц, но не выше максимального значения 100;

- *SpeedBoostPrize* (Приложение А, код программы *SpeedBoostPrize.cs*): при подборе применяет к машине временный эффект ускорения. Это достигается путем создания и применения объекта *SpeedBoostDecorator*;

- *SlowDownPrize* (Приложение А, код программы *SlowDownPrize.cs*): при подборе применяет к машине временный эффект замедления, создавая и применяя *SlowDownDecorator*.

Для реализации временных эффектов, таких как ускорение и замедление, используется паттерн «Декоратор». Этот паттерн позволяет динамически добавлять или изменять функциональность объекта (*Car*), «оборачивая» его в один или несколько объектов-декораторов.

CarDecorator (Приложение А, код программы *CarDecorator.cs*) это базовый абстрактный класс для всех временных эффектов. В конструкторе он принимает объект *Car*, к которому будет применяться эффект, и длительность (*duration*) эффекта в секундах. Метод *Apply()* запускает таймер, записывает время начала действия эффекта и вызывает абстрактный метод *ApplyEffect()*, где дочерний класс должен реализовать логику модификации машины. Метод

Remove() останавливает таймер и вызывает абстрактный метод *RevertEffect()*, где дочерний класс должен отменить внесенные изменения, возвращая машину к исходному состоянию. Когда таймер завершает работу, автоматически вызывается метод *OnTimerEnd*, который просто сообщает объекту *Car* о необходимости снять текущий декоратор (*RemoveDecorator()*).

SpeedBoostDecorator и *SlowDownDecorator* (конкретные декораторы) это классы (Приложение А, код программы *SpeedBoostDecorator.cs* и *SlowDownDecorator.cs*), которые наследуются от *CarDecorator*. Они реализуют методы *ApplyEffect* и *RevertEffect*.

Класс *Car* управляет применением декораторов. Метод *ApplyDecorator* сначала удаляет предыдущий декоратор (если он был), а затем сохраняет ссылку на новый декоратор и вызывает его метод *Apply*. Метод *RemoveDecorator* вызывает *Remove* у текущего декоратора и обнуляет ссылку на него.

Структура классов, реализующих систему бонусов и применение шаблонов проектирования «Фабричный метод» и «Декоратор», представлена на рисунке 2.4.

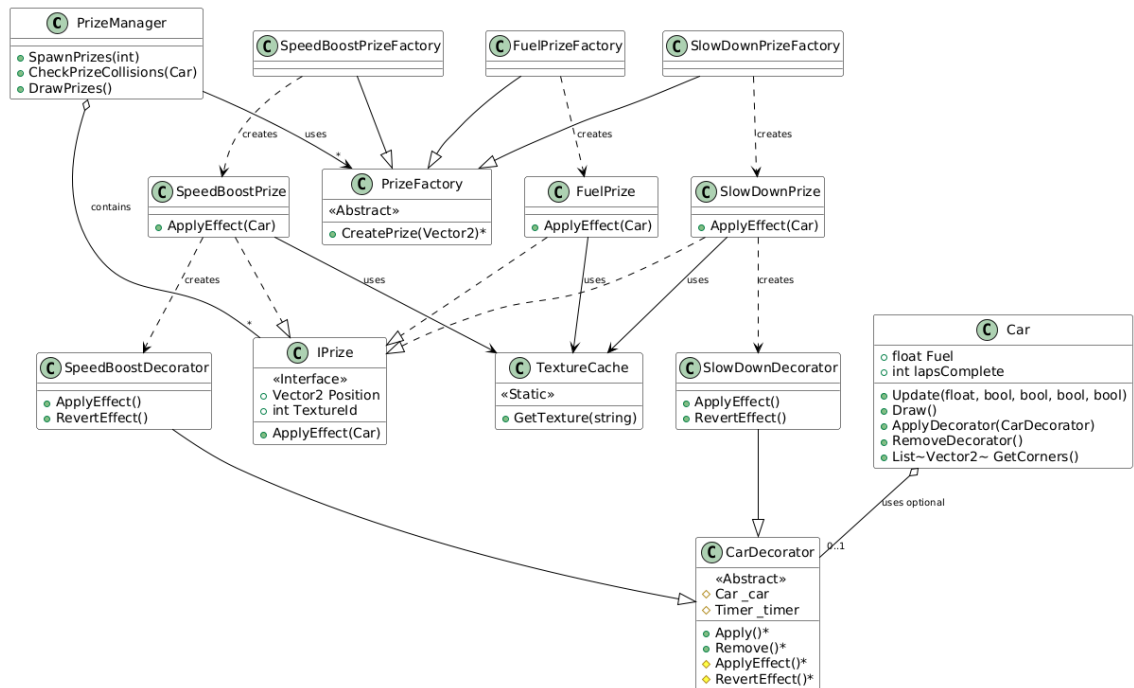


Рисунок 2.4 – Структура классов, реализующих систему бонусов

Такая система с использованием фабрик для создания бонусов и декораторов для временных эффектов обеспечивает гибкость и расширяемость.

2.2.5 Архитектура разрабатываемого приложения четко разделяет два основных состояния: главное меню и непосредственно игровой процесс. Это разделение реализуется с помощью интерфейсов *IMenuController* и *IGameController* (Приложение А, код программы *IMenuController.cs* и *IGameController.cs*). Главная форма приложения взаимодействует с логикой меню и игры через эти интерфейсы, что уменьшает связанность компонентов. Каждый контроллер управляет своей панелью – *MenuPanel* для меню и

GamePanel для игры. *Form1* отвечает за переключение видимости между этими панелями при смене состояний.

За управление логикой и построением пользовательского интерфейса главного меню отвечают классы *MenuController* и *MenuBuilder* (Приложение А, код программы *MenuController.cs* и *MenuBuilder.cs*).

MenuController реализует интерфейс *IMenuController*, управляет видимостью *MenuPanel* и предоставляет *Form1* доступ к выбору, сделанному пользователем.

MenuBuilder – это класс, который конструирует все визуальные элементы меню внутри *MenuPanel*.

Когда пользователь запускает игру из меню, управление переходит к *GameController*, реализующему *IGameController*. Он отвечает за управление игровым состоянием и соответствующей панелью *GamePanel*.

GameController (Приложение А, код программы *GameController.cs*) создает *GamePanel*, которая изначально скрыта. Главным элементом этой панели является *GLControl* (из библиотеки *OpenTK*) – это элемент управления *Windows Forms*, который предоставляет поверхность для рендеринга с использованием *OpenGL*.

Когда гонка завершена, *GameController* уведомляет об этом объект *GameBuilder* (Приложение А, код программы *GameBuilder.cs*). *GameBuilder* отвечает за построение небольшого интерфейса, отображаемого поверх игрового экрана после финиша.

Структура классов, отвечающих за пользовательский интерфейс и управление переключением между экранами приложения, показана на рисунке 2.5.

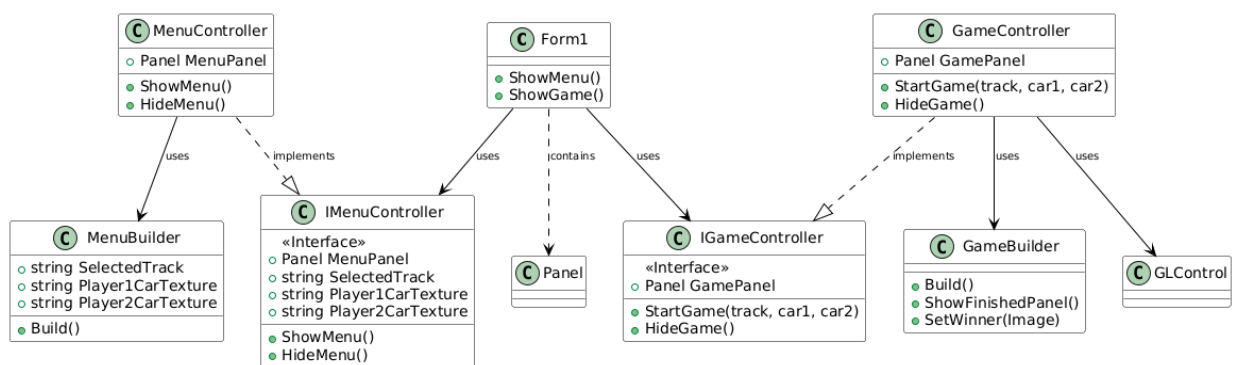


Рисунок 2.5 – Пользовательский интерфейс и управление переключением между экранами приложения

Во время гонки игрокам необходимо видеть актуальную информацию о состоянии их машины. За это отвечает *HUDRenderer* (Приложение А, код программы *HUDRenderer.cs*).

Для получения информации о нажатых клавишах используется класс *InputManager* (Приложение А, код программы *InputManager.cs*) из

RingRaceLab.Game. Он инкапсулирует логику опроса клавиатуры. *GameManager* использует экземпляр *InputManager* в своем методе *Update*.

2.2.6 Центральным элементом для рендеринга в игровом режиме является *GLControl*, расположенный на *GamePanel* в *GameController*. Для проецирования 2D-координат на экран используется ортографическая проекция (*GL.Ortho*), настраиваемая в методе *SetupViewport*..

Загрузка текстур выполняется с помощью статического класса *TextureLoader* (Приложение А, код программы *TextureLoader.cs*). Он использует *System.Drawing.Bitmap* для чтения данных из файлов изображений, создает текстурные объекты *OpenGL* и загружает в них пиксельные данные. Для оптимизации и избежания повторной загрузки одних и тех же текстур из файла используется *TextureCache* (Приложение А, код программы *TextureCache.cs*).

Проект включает несколько полезных вспомогательных статических классов:

- *GameConstants* (Приложение А, код программы *GameConstants.cs*): содержит константные данные, специфичные для игровых трасс;
- *LineIntersection* (Приложение А, код программы *LineIntersection.cs*): предоставляет статический метод *CheckLineCrossing* для определения факта пересечения двух отрезков на плоскости.

Структура классов, отвечающих за графический вывод, обработку коллизий и набор общих вспомогательных утилит, показана на рисунке 2.6.

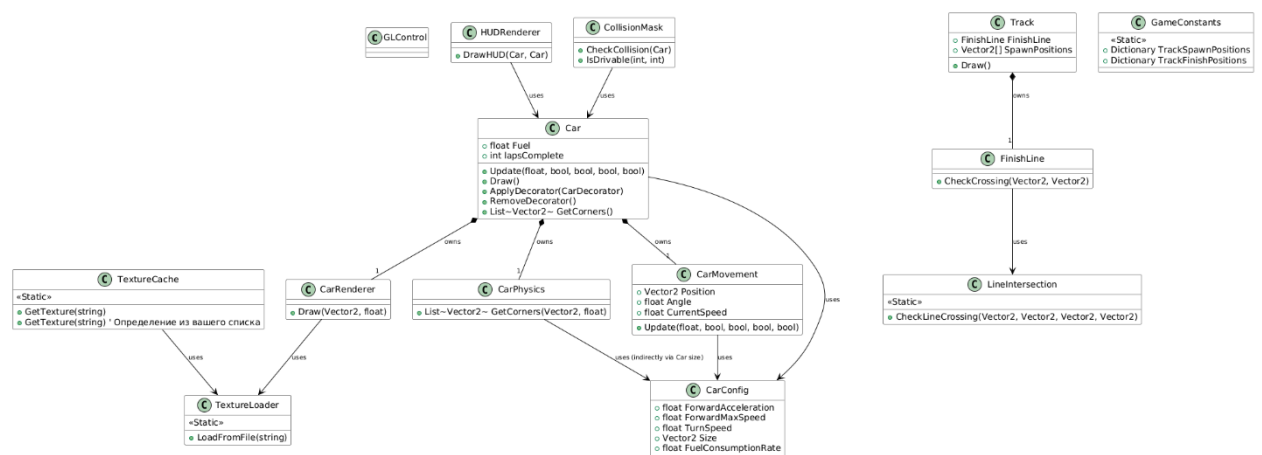


Рисунок 2.6 – Структура классов, отвечающих за графический вывод, обработку коллизий и набор общих вспомогательных утилит

Основной поток данных во время игрового процесса выглядит следующим образом: *InputManager* считывает ввод с клавиатуры, затем *GameManager* получает этот ввод, обновляет состояние машин (вызывая *Car.Update*, который использует *CarMovement*), проверяет столкновения машин с бонусами (*PrizeManager*) и границами трассы (*CollisionMask*), проверяет пересечение финишной линии (*FinishLine*), обновляет данные для HUD, затем *GameManager* инициирует отрисовку, вызывая методы *Draw* у трассы, машин (*CarRenderer*), бонусов (*PrizeManager*) и HUD (*HUDRenderer*).

3 ТЕСТИРОВАНИЯ И ВЕРИФИКАЦИЯ ПРИЛОЖЕНИЯ «КОЛЬЦЕВЫЕ ГОНКИ»

3.1 Принцип работы игрового приложения

Приложение «Кольцевые гонки» представляет собой динамичное двухмерное гоночное состязание для двух игроков, управление в котором осуществляется с одной клавиатуры. Основная цель каждого участника – стать первым, кто успешно преодолет пять полных кругов по выбранной трассе, умело управляя своим автомобилем, рационально расходуя топливо и используя попадающиеся на пути бонусы.

Запуск приложения открывает главное меню, где игрокам предоставляется возможность выбрать игровую трассу из доступных вариантов и внешний вид (текстуру) для каждого из двух гоночных автомобилей. После того как выбор сделан и подтвержден нажатием кнопки старта, приложение переходит в игровое состояние.

В момент начала гонки на экране появляется выбранная трасса с расставленными на ней призами и двумя автомобилями игроков, расположенными на стартовых позициях. Игровой процесс управляется компонентом *GameManager*, который координирует действия всех игровых объектов. Каждое обновление игрового кадра начинается с обработки ввода игрока. *InputManager* считывает нажатия клавиш для каждого из двух игроков – стандартно это клавиши *WASD* для первого игрока и стрелки для второго. Полученные данные о вводе передаются соответствующим автомобилям для определения их дальнейших действий. Полученные данные о вводе от обоих игроков обрабатываются в рамках одного игрового цикла, где происходит последовательное обновление позиций, скоростей и состояний каждого автомобиля, а также других динамических элементов игры (например, проверка подбора призов). Таким образом достигается синхронизация игрового процесса для обоих участников в реальном времени на одном устройстве.

Движение каждого автомобиля рассчитывается в его компоненте *CarMovement*. На основе текущей скорости, угла поворота и нажатых клавиш управления (вперед, назад, влево, вправо) происходит изменение позиции и ориентации машины. Скорость автомобиля регулируется ускорением и замедлением, а также ограничена максимальными значениями. Поворот зависит от текущей скорости – чем выше скорость, тем менее маневренным становится автомобиль. Важным аспектом является расход топлива: при движении уровень топлива в машине постепенно уменьшается. Если топливо заканчивается, автомобиль теряет способность к передвижению.

В процессе движения автомобили могут сталкиваться с границами трассы. Система коллизий, реализованная классом *CollisionMask* на основе специальной карты коллизий, определяет, находится ли какая-либо из угловых точек автомобиля на непроезжей части. При обнаружении столкновения позиция автомобиля откатывается к предыдущему состоянию, а его скорость значительно снижается, имитируя отскок.

На трассе случайным образом появляются бонусы, управляемые *PrizeManager*. Существует три типа бонусов: топливо, ускорение и замедление. При наезде на бонус (проверка осуществляется по расстоянию между центром машины и центром бонуса), бонус исчезает, а его эффект применяется к подобравшему автомобилю. Бонус топлива моментально пополняет запас горючего. Бонусы ускорения и замедления применяют временные эффекты, которые модифицируют характеристики автомобиля, такие как максимальная скорость, на определенный промежуток времени. Эти временные изменения реализуются с использованием паттерна "Декоратор", где на автомобиль "навешивается" дополнительный объект, изменяющий его поведение на время действия бонуса. По истечении времени действия декоратор автоматически снимается, и характеристики автомобиля возвращаются к исходным.

Игровой процесс продолжается, пока один из игроков не преодолеет необходимое количество кругов. Пересечение финишной линии отслеживается классом *FinishLine*, который определяет факт пересечения и направление движения. При каждом пересечении финишной линии в правильном направлении счетчик кругов соответствующего автомобиля увеличивается. Как только счетчик достигает пяти, *GameManager* фиксирует финиш этого игрока.

По завершении гонки одним из игроков, игровой процесс останавливается, и на экране отображается панель с информацией о победителе. С этой панели игрок может вернуться в главное меню приложения, нажав соответствующую кнопку.

Во время гонки на экране постоянно отображается *HUD* (*Heads-Up Display*), предоставляющий актуальную информацию о состоянии автомобилей, в частности, об уровне топлива и активности временных бонусов (ускорения или замедления). Отрисовка всей игровой сцены – трассы, автомобилей, бонусов и *HUD* – происходит в каждом кадре с использованием библиотеки *OpenGL* через компонент *GLControl*. Для обеспечения плавности изображения применяются техники двойной буферизации и ортографическая проекция.

Принцип работы игрового приложения "Кольцевые гонки" основан на циклическом обновлении состояния игры в ответ на ввод игрока и взаимодействие объектов в игровом мире, начиная с выбора параметров гонки в меню, продолжая динамичным заездом с учетом физики движения, расхода топлива и использования бонусов, и завершая отображением результатов после финиша.

3.2 Результаты тестирования разработанного приложения

Для обеспечения надежности и корректной работы разработанного игрового приложения «Кольцевые гонки» было проведено модульное тестирование ключевых компонентов игровой логики. Тестирование осуществлялось в рамках отдельного проекта *RingRaceTestProject*, использующего стандартные средства модульного тестирования, предоставляемые таким пространством имен как, *Microsoft.VisualStudio.TestTools.UnitTesting*. Такой подход позволил проверить

функциональность отдельных классов и методов в изоляции, подтверждая их соответствие предъявляемым требованиям перед интеграцией в общую систему приложения.

В процессе тестирования использовался набор тестовых классов, каждый из которых был нацелен на верификацию определенной части функциональности разработанных библиотек (*RingRaceLab*). Были разработаны следующие основные тестовые классы:

- *CarMovementTests*: данный класс предназначен для тестирования класса *CarMovement*, отвечающего за симуляцию движения и поворотов гоночных автомобилей. Тесты включают проверку корректности изменения скорости автомобиля при ускорении вперед и назад, а также при применении замедления в отсутствие воздействия на педаль газа. Особое внимание уделено проверке того, что скорость не превышает установленных максимальных значений. Также верифицируется правильность изменения угла поворота автомобиля в зависимости от направления движения и входных данных, имитирующих повороты налево и направо. Для подтверждения ожидаемых результатов используются утверждения *Assert.IsTrue* для проверки булевых условий (например, скорость увеличилась или уменьшилась) и *Assert.AreEqual* (или сравнение с допуском для чисел с плавающей запятой, как показано в коде) для сравнения численных значений скорости и угла;

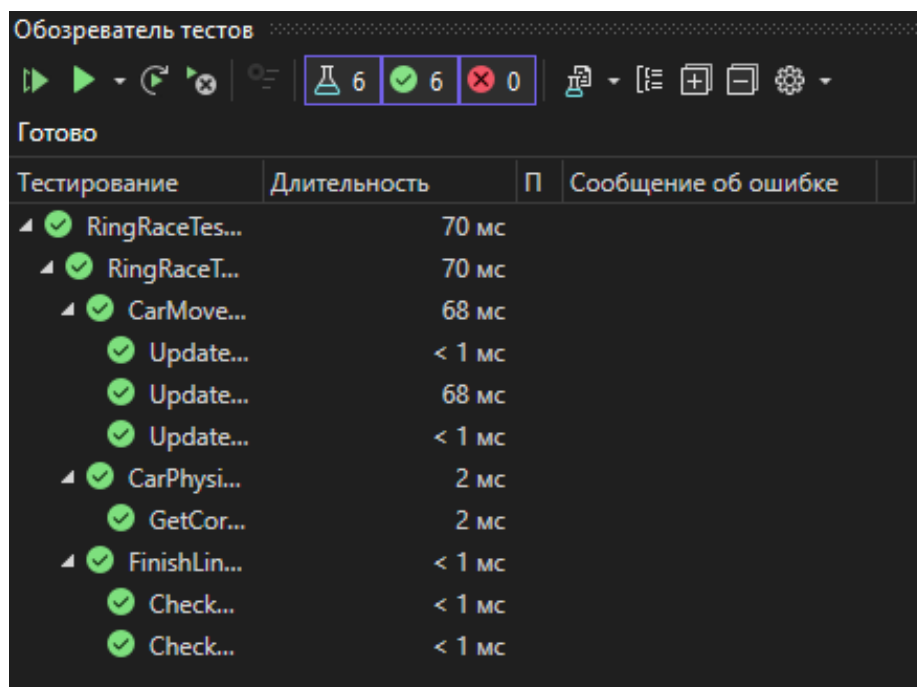
- *CarPhysicsTests*: этот тестовый класс фокусируется на верификации функциональности класса *CarPhysics*, который занимается расчетом физических параметров, критически важных для системы коллизий, в частности, определяет координаты углов прямоугольной модели автомобиля в мировых координатах. Тесты проверяют, что метод *GetCorners* возвращает правильное количество углов (четыре) и что рассчитанные координаты углов соответствуют ожидаемым значениям для различных положений и углов ориентации автомобиля. Это включает проверку расчетов с учетом поворота автомобиля на различные углы. Для сравнения координат углов, являющихся векторами, используются методы, учитывающие допустимую погрешность при работе с числами с плавающей запятой, такие как сравнение расстояния между ожидаемой и фактической точкой с малым порогом, а также *Assert.AreEqual* для количества углов;

- *FinishLineTests*: класс *FinishLineTests* предназначен для тестирования класса *FinishLine* и его способности корректно определять факт пересечения автомобилем линии старта/финиша. Тесты верифицируют логику метода *CheckCrossing*, проверяя как сценарии, в которых автомобиль действительно пересекает заданную линию между двумя последовательными позициями (ожидается результат, отличный от нуля, указывающий направление пересечения), так и сценарии, когда пересечения не происходит (ожидается результат, равный нулю). Используются утверждения *Assert.IsTrue* и *Assert.IsFalse* для подтверждения факта обнаружения или отсутствия пересечения линии, а также проверка конкретных возвращаемых значений метода *CheckCrossing* (1, -1, 0).

Каждый тестовый метод в указанных классах следует стандартной структуре *Arrange-Act-Assert*. На этапе *Arrange* подготавливаются необходимые

объекты и данные для теста. На этапе *Act* выполняется тестируемая операция или метод. На этапе *Assert* проверяется, соответствует ли результат выполнения ожидаемому поведению с помощью различных методов класса *Assert*.

На рисунке 3.1 представлены результаты тестирования приложения «Кольцевые гонки».



Тестирование	Длительность	П	Сообщение об ошибке
RingRaceTes...	70 мс	✓	
RingRaceT...	70 мс	✓	
CarMove...	68 мс	✓	
Update...	< 1 мс	✓	
Update...	68 мс	✓	
Update...	< 1 мс	✓	
CarPhysi...	2 мс	✓	
GetCor...	2 мс	✓	
FinishLin...	< 1 мс	✓	
Check...	< 1 мс	✓	
Check...	< 1 мс	✓	

Рисунок 3.1 – Результаты тестирования приложения «Кольцевые гонки»

Проведение модульного тестирования позволило подтвердить, что разработанные классы, составляющие основу игровой механики и взаимодействия объектов в приложении «Кольцевые гонки», функционируют корректно согласно заложенной логике. Тестирование охватило критически важные аспекты, такие как физика движения, определение положения объектов и детектирование ключевых игровых событий (пересечение финишной линии), что является фундаментом для стабильной и предсказуемой работы всего приложения.

3.3 Результаты верификации разработанного приложения

При запуске игрового приложения «Кольцевые гонки» пользователь видит главное меню, представляющее собой начальный интерфейс. На этом экране можно выбрать одну из доступных трасс и определить внешний вид гоночных автомобилей для обоих игроков. После подтверждения выбора и нажатия кнопки «Старт» происходит переход в игровое состояние.

Управление каждым автомобилем осуществляется с клавиатуры. Первый игрок использует клавиши *W*, *A*, *S*, *D* для движения и поворотов, а второй игрок – соответствующие клавиши со стрелками. Нажатие клавиш «вперед» (*W* или стрелка вверх) и «назад» (*S* или стрелка вниз) регулирует скорость автомобиля,

а «влево» (*A* или стрелка влево) и «вправо» (*D* или стрелка вправо) – направление движения.

Игровое поле формируется в соответствии с выбранной трассой. Автомобили игроков появляются на заранее определенных стартовых позициях на трассе. Изображение одной из трассы представлено на рисунке 3.2.

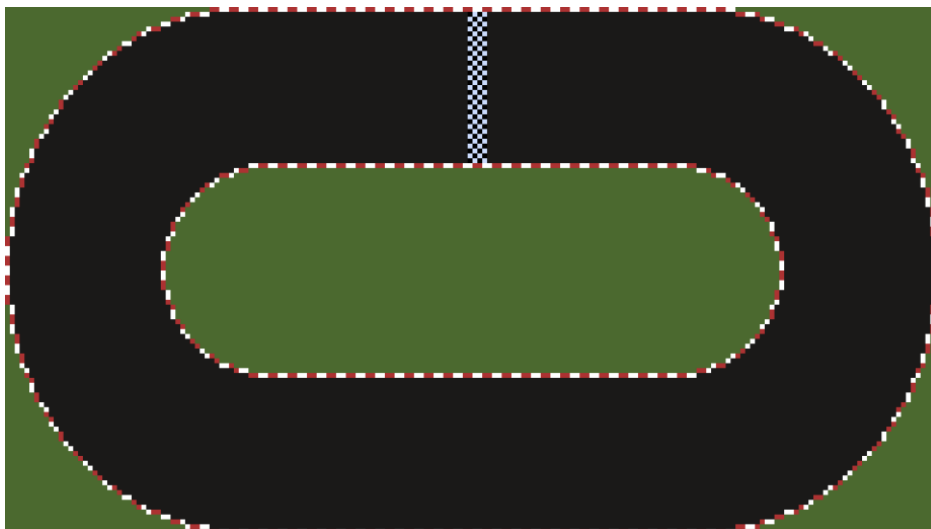


Рисунок 3.2 – Игровая трасса

На трассе также случайным образом распределяются призы, призванные разнообразить игровой процесс (рисунок 3.3).



Рисунок 3.3 – Призы, появляющиеся на трассе

Одной из ключевых механик игры является расходование топлива при движении. Начальный запас топлива ограничен, и для продолжения гонки необходимо собирать соответствующие бонусы на трассе. Помимо топлива, на трассе появляются бонусы ускорения и замедления. Подбор этих бонусов временно изменяет характеристики автомобиля, влияя на его максимальную скорость.

Взаимодействие с границами трассы обрабатывается при помощи карты коллизий. При наезде на препятствие движение автомобиля блокируется, и происходит небольшой отскок.

Цель игры для каждого игрока – первым проехать пять полных кругов. Прохождение круга фиксируется при пересечении финишной линии в правильном направлении.

В процессе игры на экране отображается информация о состоянии каждого автомобиля, включая уровень топлива и индикацию активных временных эффектов от бонусов (рисунок 3.4).

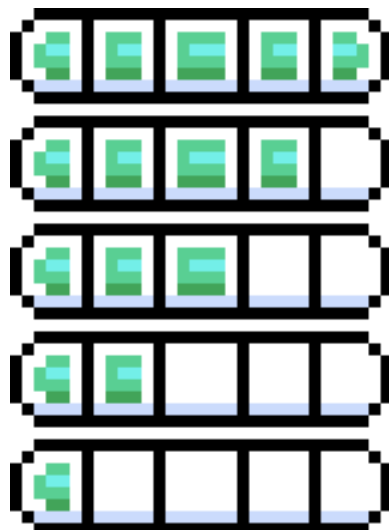


Рисунок 3.5 – Индикаторы бонусов

По завершении гонки одним из игроков на экране появляется уведомление о победителе (рисунок 3.5). На нем указано какой из игроков победил в состязании. Также присутствует кнопка, отвечающая за выход в главное меню.



Рисунок 3.6 – Уведомление о победителе

После окончания игры игроки могут вернуться в главное меню для начала нового заезда. Приложение обеспечивает возможность повторного запуска игры и выбора других параметров (трассы, автомобилей).

Верификация показала, что приложение корректно обрабатывает ввод игрока, управляет движением и взаимодействием автомобилей с игровым миром (трасса, призы), отслеживает состояние топлива и прогресс прохождения трассы, а также корректно определяет победителя и отображает соответствующую информацию. Функциональность бонусов и их временное влияние на характеристики автомобилей также соответствуют заявленной механике.

ЗАКЛЮЧЕНИЕ

Результатом выполнения проекта является игровое приложения «Кольцевые гонки», представляющее собой двухмерную аркадную гонку для двух пользователей, соревнующихся на одном экране.

В процессе разработки приложения проведен анализ особенностей жанра гонок и современных средств создания игровых приложений, включая графическую библиотеку *OpenGL* и возможности платформы *Windows Forms* на языке *C#*. Структура программного обеспечения определена путем декомпозиции общей задачи на более мелкие, управляемые компоненты и модули, что позволило эффективно организовать процесс реализации. Значительным этапом стала разработка ядра игровой логики, ответственного за управление игровым процессом, взаимодействие объектов и поддержание состояния игры, а также реализация специфических игровых механик, таких как движение автомобилей, система коллизий, учет топлива и подсчет кругов.

Для обеспечения гибкости, расширяемости и модульности кода при разработке широко применены современные шаблоны проектирования, такие как «Фабричный метод» для создания разнообразных игровых объектов (например, призов) и «Декоратор» для динамического добавления временных эффектов к игровым сущностям (например, ускорение или замедление автомобилей). Такой подход позволил создать архитектуру, легко адаптируемую к возможным будущим изменениям и дополнениям функционала.

Проведенная верификация приложения подтвердила корректность работы основных игровых механик и взаимодействия компонентов. Тестирование функциональности каждого элемента, от обработки ввода игрока до логики финиша, позволило выявить и устранить потенциальные ошибки, минимизируя вероятность некорректного поведения приложения в процессе эксплуатации.

Созданное приложение обладает интуитивно понятным интерфейсом, который начинается с простого меню выбора трассы и автомобилей, переходя к непосредственному игровому процессу с наглядным отображением необходимой информации (*HUD*). Игра не требует от пользователя специальных навыков, что делает ее доступной для широкой аудитории.

Курсовой проект проверен в системе «Антиплагиат». Оценка оригинальности составляет 96 процентов. Авторские права на программную часть проекта принадлежат автору курсовой работы.

Список использованных источников

1. Краснов, А. В. *OpenGL: Руководство разработчика*. – М.: Диалектика, 2019. – 320 с.
2. Петров, Б. С. Программирование графики в C# с использованием *OpenGL*. – СПб.: Питер, 2020. – 256 с.
3. Сидоров, В. Н. Графические библиотеки и их применение в *.NET*. – Новосибирск: НГТУ, 2018. – 198 с.
4. Иванова, Г. Л. Трёхмерная графика в *Windows Forms*. – Казань: Казанский университет, 2021. – 274 с.
5. *OpenTK Documentation* [Электронный ресурс] // Официальный сайт *OpenTK*. URL: <https://opentk.net/learn/documentation> (дата обращения: 14.03.2025).

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг программы «Кольцевые гонки»

Код программы *Form1.cs*:

```
using System.Windows.Forms;
using RingRaceLab; // Предполагается, что это пространство имен содержит IMenuController, IGameController,
GameController, MenuController

namespace RingRaceApp
{
    /// <summary>
    /// Главная форма приложения, управляющая переключением между меню и игровым процессом.
    /// </summary>
    public partial class Form1 : Form
    {
        /// <summary>
        /// Контроллер меню для управления отображением и логикой меню.
        /// </summary>
        private readonly IMenuController _menuController;

        /// <summary>
        /// Контроллер игры для управления игровым процессом.
        /// </summary>
        private readonly IGameController _gameController;

        /// <summary>
        /// Инициализирует новый экземпляр класса <see cref="Form1"/>.
        /// Настраивает форму на полноэкранный режим без рамок,
        /// создает контроллеры меню и игры, добавляет их панели в форму
        /// и отображает начальное меню.
        /// </summary>
        public Form1()
        {
            InitializeComponent();

            // Настройки окна: полноэкранный режим без рамок
            FormBorderStyle = FormBorderStyle.None;
            WindowState = FormWindowState.Maximized;
            // Установка фиксированных размеров (может потребоваться доработка для разных разрешений)
            Width = 1920;
            Height = 1080;
            Text = "2D Car Game with Two Cars";
            KeyPreview = true; // Позволяет форме получать события клавиатуры до дочерних элементов
            DoubleBuffered = true; // Включает двойную буферизацию для уменьшения мерцания

            // Создаём контроллеры, передавая методы для переключения экранов и размеры формы
            _gameController = new GameController(ShowMenu, Width, Height);
            _menuController = new MenuController(_gameController, ShowGame, Width, Height);

            // Добавляем панели контроллеров в коллекцию элементов управления формы
            Controls.Add(_menuController.MenuPanel);
            Controls.Add(_gameController.GamePanel);

            // Запускаем приложение с отображения меню
            ShowMenu();
        }
    }
}
```

```

/// <summary>
/// Отображает меню, скрывая игровой экран.
/// </summary>
private void ShowMenu()
{
    _gameController.HideGame(); // Скрываем панель игры
    _menuController.ShowMenu(); // Отображаем панель меню
}

/// <summary>
/// Отображает игровой экран, скрывая меню, и запускает игру
/// с выбранными настройками (трек и текстуры машин).
/// </summary>
private void ShowGame()
{
    _menuController.HideMenu(); // Скрываем панель меню
    // Запускаем игру, передавая выбранный трек и текстуры машин из контроллера меню
    _gameController.StartGame(
        _menuController.SelectedTrack,
        _menuController.Player1CarTexture,
        _menuController.Player2CarTexture
    );
}

/// <summary>
/// Переопределяет CreateParams для включения расширенного стиля окна,
/// что помогает устранить мерцание при перерисовке.
/// </summary>
protected override CreateParams CreateParams
{
    get
    {
        CreateParams cp = base.CreateParams;
        // 0x02000000 (WS_EX_COMPOSITED) включает двойную буферизацию на уровне окна
        cp.ExStyle |= 0x02000000;
        return cp;
    }
}
}
}

```

Код программы *EntityManager.cs*:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RingRaceLab.Game
{
    /// <summary>
    /// Управляет коллекцией игровых сущностей (<see cref="GameEntity"/>) в игре.
    /// Предоставляет методы для добавления, удаления и получения сущностей.
    /// </summary>
    public class EntityManager
    {
        /// <summary>
        /// Коллекция всех игровых сущностей, управляемых менеджером.
        /// </summary>
        public readonly List<GameEntity> _entities = new List<GameEntity>();
    }
}

```

```

    /// <summary>
    /// Добавляет указанную игровую сущность в коллекцию.
    /// </summary>
    /// <param name="entity">Сущность для добавления.</param>
    public void AddEntity(GameEntity entity)
    {
        _entities.Add(entity);
    }

    /// <summary>
    /// Удаляет указанную игровую сущность из коллекции.
    /// </summary>
    /// <param name="entity">Сущность для удаления.</param>
    public void RemoveEntity(GameEntity entity)
    {
        _entities.Remove(entity);
    }

    /// <summary>
    /// Возвращает перечислимую коллекцию всех игровых сущностей, управляемых менеджером.
    /// </summary>
    /// <returns>Коллекция сущностей.</returns>
    public IEnumerable<GameEntity> GetEntities()
    {
        return _entities;
    }
}

```

Код программы *GameEntity.cs*:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RingRaceLab
{
    public abstract class GameEntity
    {
        public abstract void Draw();
    }
}

```

Код программы *GameManager.cs*:

```

using OpenTK;
using System;
using System.Collections.Generic;
using OpenTK.Graphics;
using OpenTK.Graphics.OpenGL;
using System.Diagnostics;
using System.Windows.Forms;
using System.Linq;
using System.Timers;
using System.Threading;
using System.Drawing;
using RingRaceLab.Game;

```

```

namespace RingRaceLab
{
    /// <summary>
    /// Управляет основным игровым циклом, сущностями, столкновениями, призами и отображением HUD.
    /// </summary>
    public class GameManager
    {
        /// <summary>
        /// Менеджер сущностей для хранения и управления всеми игровыми объектами.
        /// </summary>
        private readonly EntityManager _entityManager = new EntityManager();

        /// <summary>
        /// Менеджер призов для создания, управления и обработки сбора призов.
        /// </summary>
        private readonly PrizeManager _prizeManager;

        /// <summary>
        /// Рендерер HUD для отображения информации о состоянии игроков.
        /// </summary>
        private readonly HUDRenderer _hudRenderer;

        /// <summary>
        /// Система обнаружения столкновений, основанная на маске коллизий.
        /// </summary>
        private readonly CollisionMask _collisionSystem;

        /// <summary>
        /// Менеджер ввода для обработки действий игроков.
        /// </summary>
        private readonly InputManager _inputManager = new InputManager();

        /// <summary>
        /// Секундомер для измерения времени между кадрами (для дельта времени).
        /// </summary>
        private readonly Stopwatch _stopwatch = new Stopwatch();

        /// <summary>
        /// Делегат для события завершения гонки автомобилем.
        /// </summary>
        /// <param name="finishedCar">Автомобиль, завершивший гонку.</param>
        public delegate void CarFinishedHandler(Car finishedCar);

        /// <summary>
        /// Событие, возникающее при завершении гонки одним из автомобилей.
        /// </summary>
        public event CarFinishedHandler OnCarFinished;

        /// <summary>
        /// Игровой трек.
        /// </summary>
        public Track Track { get; private set; }

        /// <summary>
        /// Первый автомобиль игрока.
        /// </summary>
        public Car Car1 { get; private set; }

        /// <summary>
        /// Второй автомобиль игрока.
        /// </summary>
        public Car Car2 { get; private set; }
    }
}

```

```

/// <summary>
/// Текущий уровень индикатора топлива для первого игрока.
/// </summary>
private int _currentFuelLevel1 = 5;

/// <summary>
/// Текущий уровень индикатора ускорения для первого игрока.
/// </summary>
private int _currentSpeedLevel1 = 0;

/// <summary>
/// Текущий уровень индикатора замедления для первого игрока.
/// </summary>
private int _currentSlowLevel1 = 0;

/// <summary>
/// Текущий уровень индикатора топлива для второго игрока.
/// </summary>
private int _currentFuelLevel2 = 5;

/// <summary>
/// Текущий уровень индикатора ускорения для второго игрока.
/// </summary>
private int _currentSpeedLevel2 = 0;

/// <summary>
/// Текущий уровень индикатора замедления для второго игрока.
/// </summary>
private int _currentSlowLevel2 = 0;

/// <summary>
/// Словарь для хранения загруженных текстур по имени.
/// </summary>
private Dictionary<string, int> _textures = new Dictionary<string, int>();

/// <summary>
/// Инициализирует новый экземпляр класса <see cref="GameManager"/>.
/// Создает и настраивает все игровые компоненты: трек, автомобили,
/// систему столкновений, менеджеры сущностей и призов, а также HUD.
/// </summary>
/// <param name="trackTexture">Путь к текстуре трека.</param>
/// <param name="collisionMap">Путь к изображению маски коллизий.</param>
/// <param name="spawnPositions">Массив стартовых позиций для автомобилей.</param>
/// <param name="finishPosition">Позиция финишной линии (начало и конец отрезка).</param>
/// <param name="player1CarTexture">Путь к текстуре автомобиля первого игрока.</param>
/// <param name="player2CarTexture">Путь к текстуре автомобиля второго игрока.</param>
/// <param name="Width">Ширина игрового окна.</param>
/// <param name="Height">Высота игрового окна.</param>
/// <exception cref="ArgumentException">Выбрасывается, если задано менее двух стартовых
позиций.</exception>
public GameManager(string trackTexture, string collisionMap, Vector2[] spawnPositions, Vector2[]
finishPosition, string player1CarTexture, string player2CarTexture, int Width, int Height)
{
    _stopwatch.Start();
    if (spawnPositions == null || spawnPositions.Length < 2)
        throw new ArgumentException("Необходимо задать хотя бы две стартовые позиции.",
nameof(spawnPositions));
    Track = new Track(trackTexture, spawnPositions, finishPosition[0], finishPosition[1], Width, Height);
    CarConfig config1 = new CarConfig
    {
        Size = new Vector2(Width / 60, (int)(Height / 67.5)),
    };
    CarConfig config2 = new CarConfig

```

```

    {
        Size = new Vector2(Width / 60, (int)(Height / 67.5)),
    };
    Car1 = new Car(spawnPositions[0], player1CarTexture, config1);
    Car2 = new Car(spawnPositions[1], player2CarTexture, config2);
    _collisionSystem = new CollisionMask(collisionMap);
    _entityManager.AddEntity(Track);
    _entityManager.AddEntity(Car1);
    _entityManager.AddEntity(Car2);

    PrizeFactory[] prizeFactories = {
        new FuelPrizeFactory(),
        new SpeedBoostPrizeFactory(),
        new SlowDownPrizeFactory()
    };
    _prizeManager = new PrizeManager(prizeFactories, _collisionSystem, Width, Height);
    _prizeManager.SpawnPrizes(10);

    LoadTextures();
    _hudRenderer = new HUDRenderer(_textures, Width, Height);
}

/// <summary>
/// Загружает текстуры, используемые в игре, в словарь.
/// </summary>
private void LoadTextures()
{
    // Индикаторы
    _textures["fuel0"] = TextureLoader.LoadFromFile("sprites/fuel_indicator_0.png");
    _textures["fuel1"] = TextureLoader.LoadFromFile("sprites/fuel_indicator_1.png");
    _textures["fuel2"] = TextureLoader.LoadFromFile("sprites/fuel_indicator_2.png");
    _textures["fuel3"] = TextureLoader.LoadFromFile("sprites/fuel_indicator_3.png");
    _textures["fuel4"] = TextureLoader.LoadFromFile("sprites/fuel_indicator_4.png");
    _textures["fuel5"] = TextureLoader.LoadFromFile("sprites/fuel_indicator_5.png");

    _textures["speed0"] = TextureLoader.LoadFromFile("sprites/speed_indicator_0.png");
    _textures["speed1"] = TextureLoader.LoadFromFile("sprites/speed_indicator_1.png");
    _textures["speed2"] = TextureLoader.LoadFromFile("sprites/speed_indicator_2.png");
    _textures["speed3"] = TextureLoader.LoadFromFile("sprites/speed_indicator_3.png");
    _textures["speed4"] = TextureLoader.LoadFromFile("sprites/speed_indicator_4.png");
    _textures["speed5"] = TextureLoader.LoadFromFile("sprites/speed_indicator_5.png");

    _textures["slow0"] = TextureLoader.LoadFromFile("sprites/slow_indicator_0.png");
    _textures["slow1"] = TextureLoader.LoadFromFile("sprites/slow_indicator_1.png");
    _textures["slow2"] = TextureLoader.LoadFromFile("sprites/slow_indicator_2.png");
    _textures["slow3"] = TextureLoader.LoadFromFile("sprites/slow_indicator_3.png");
    _textures["slow4"] = TextureLoader.LoadFromFile("sprites/slow_indicator_4.png");
    _textures["slow5"] = TextureLoader.LoadFromFile("sprites/slow_indicator_5.png");
}

/// <summary>
/// Обновляет уровни индикаторов для HUD на основе текущего состояния автомобиля
/// (уровня топлива и активных эффектов от призов).
/// </summary>
/// <param name="car">Автомобиль, для которого обновляются индикаторы.</param>
/// <param name="fuel">Ссылка на переменную уровня индикатора топлива.</param>
/// <param name="speed">Ссылка на переменную уровня индикатора скорости.</param>
/// <param name="slow">Ссылка на переменную уровня индикатора замедления.</param>
private void UpdateIndicators(Car car, ref int fuel, ref int speed, ref int slow)
{
    if (car.Fuel > 80) fuel = 5;
    else if (car.Fuel > 60) fuel = 4;
}

```

```

else if (car.Fuel > 40) fuel = 3;
else if (car.Fuel > 20) fuel = 2;
else if (car.Fuel > 0) fuel = 1;
else fuel = 0;
if (car._currentDecorator != null)
{
    TimeSpan elapsed = DateTime.Now - car._currentDecorator.timerStartTime;
    int remaining = (int)((car._currentDecorator._timer.Interval - elapsed.TotalMilliseconds) / 1000) + 1;

    if (car._currentDecorator is SpeedBoostDecorator)
    {
        if (remaining == 5)
            speed = 5;
        else if (remaining == 4)
            speed = 4;
        else if (remaining == 3)
            speed = 3;
        else if (remaining == 2)
            speed = 2;
        else if (remaining == 1)
            speed = 1;
        else
            speed = 0;
        slow = 0;
    }
    else
    {
        if (remaining == 5)
            slow = 5;
        else if (remaining == 4)
            slow = 4;
        else if (remaining == 3)
            slow = 3;
        else if (remaining == 2)
            slow = 2;
        else if (remaining == 1)
            slow = 1;
        else
            slow = 0;
        speed = 0;
    }
}
else
{
    speed = 0;
    slow = 0;
}
}

/// <summary>
/// Обновляет состояние игрового мира за один кадр.
/// Обработывает ввод, обновляет положение автомобилей, проверяет коллизии
/// и обновляет состояние индикаторов.
/// </summary>
/// <param name="glControl">Элемент управления GLControl для запроса перерисовки.</param>
public void Update(GLControl glControl)
{
    float deltaTime = (float)_stopwatch.Elapsed.TotalSeconds;
    _stopwatch.Restart();

    var input1 = _inputManager.GetPlayer1Input();
    var input2 = _inputManager.GetPlayer2Input();

```



```

UpdateCar(Car1, deltaTime, input1);
UpdateCar(Car2, deltaTime, input2);

_prizeManager.CheckPrizeCollisions(Car1);
_prizeManager.CheckPrizeCollisions(Car2);

UpdateIndicators(Car1, ref _currentFuelLevel1, ref _currentSpeedLevel1, ref _currentSlowLevel1);
UpdateIndicators(Car2, ref _currentFuelLevel2, ref _currentSpeedLevel2, ref _currentSlowLevel2);
glControl.Invalidate();
}

/// <summary>
/// Обновляет состояние одного автомобиля (движение, вращение, коллизии, финиш).
/// </summary>
/// <param name="car">Автомобиль для обновления.</param>
/// <param name="deltaTime">Время, прошедшее с предыдущего кадра.</param>
/// <param name="input">Ввод игрока для данного автомобиля.</param>
private void UpdateCar(Car car, float deltaTime, (bool forward, bool backward, bool left, bool right) input)
{
    Vector2 oldPos = car._movement.Position;
    float oldAngle = car._movement.Angle;

    car.Update(deltaTime, input.forward, input.backward, input.left, input.right);

    int hasCrossed = Track.FinishLine.CheckCrossing(oldPos, car._movement.Position);
    // Проверка пересечения финиша
    if (hasCrossed == 1)
    {
        car.lapsComplete++;
        // Триггерим событие финиша
        if (car.lapsComplete == 5)
        {
            OnCarFinished?.Invoke(car);
        }
    }
    else if (hasCrossed == -1)
    {
        car.lapsComplete--;
    }

    if (_collisionSystem.CheckCollision(car))
    {
        car._movement.Position = oldPos;
        car._movement.Angle = oldAngle;
        car._movement.CurrentSpeed *= -0.25f; // Логика отскока
    }
}

/// <summary>
/// Выполняет отрисовку всех игровых сущностей и HUD.
/// </summary>
public void Draw()
{
    GL.Clear(ClearBufferMask.ColorBufferBit);
    foreach (var entity in _entityManager._entities)
    {
        entity.Draw();
    }

    _prizeManager.DrawPrizes();

    _hudRenderer.DrawHUD(Car1, Car2);
}

```

```

    }
    /// <summary>
    /// Сбрасывает состояние игры, переинициализируя автомобили и секундомер.
    /// </summary>
    /// <param name="spawnPositions">Стартовые позиции для новых автомобилей.</param>
    public void Reset(Vector2[] spawnPositions)
    {
        // Переинициализируем машины
        Car1 = new Car(spawnPositions[0], "sprites/car2.png", new CarConfig());
        Car2 = new Car(spawnPositions[1], "sprites/car1.png", new CarConfig());

        // Сбросим внутреннее состояние
        _stopwatch.Restart();
    }
}

```

Код программы *HUDRenderer.cs*:

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using OpenTK.Graphics.OpenGL;

namespace RingRaceLab.Game
{
    /// <summary>
    /// Класс для отрисовки пользовательского интерфейса (HUD) в игре,
    /// включая индикаторы топлива и эффектов призов для автомобилей.
    /// </summary>
    public class HUDRenderer
    {
        /// <summary>
        /// Словарь, содержащий загруженные текстуры для индикаторов по их ключам.
        /// </summary>
        private readonly Dictionary<string, int> _textures;

        /// <summary>
        /// Ширина области отрисовки HUD.
        /// </summary>
        private readonly int Width;

        /// <summary>
        /// Высота области отрисовки HUD.
        /// </summary>
        private readonly int Height;

        /// <summary>
        /// Инициализирует новый экземпляр класса <see cref="HUDRenderer"/>.
        /// </summary>
        /// <param name="textures">Словарь текстур, используемых для отрисовки индикаторов.</param>
        /// <param name="Width">Ширина области отрисовки.</param>
        /// <param name="Height">Высота области отрисовки.</param>
        public HUDRenderer(Dictionary<string, int> textures, int Width, int Height)
        {
            _textures = textures;
            this.Width = Width;
            this.Height = Height;
        }
    }
}

```

```

    }

    /// <summary>
    /// Отрисовывает элементы HUD для двух автомобилей, включая индикаторы топлива, скорости и
    замедления.
    /// </summary>
    /// <param name="car1">Первый автомобиль, для которого отрисовывается HUD.</param>
    /// <param name="car2">Второй автомобиль, для которого отрисовывается HUD.</param>
    public void DrawHUD(Car car1, Car car2)
    {
        // Получение уровней индикаторов для Car1
        int fuelLevel1 = GetFuelLevel(car1.Fuel);
        int speedLevel1 = GetDecoratorLevel(car1, true); // true для SpeedBoostDecorator
        int slowLevel1 = GetDecoratorLevel(car1, false); // false для SlowDownDecorator

        // Отрисовка индикаторов для Car1 (в левой части экрана)
        DrawIndicator(new Rectangle(Width / 384, Height / 216, Width / 12, Height / 27), $"fuel{fuelLevel1}");
        DrawIndicator(new Rectangle(Width / 384, (int)(Height / 21.6), Width / 12, Height / 27),
            $"speed{speedLevel1}");
        DrawIndicator(new Rectangle(Width / 384, (int)(Height / 11.36), Width / 12, Height / 27),
            $"slow{slowLevel1}");

        // Получение уровней индикаторов для Car2
        int fuelLevel2 = GetFuelLevel(car2.Fuel);
        int speedLevel2 = GetDecoratorLevel(car2, true);
        int slowLevel2 = GetDecoratorLevel(car2, false);

        // Отрисовка индикаторов для Car2 (в правой части экрана)
        DrawIndicator(new Rectangle(Width - (int)(Width / 11.63), Height / 216, Width / 12, Height / 27),
            $"fuel{fuelLevel2}");
        DrawIndicator(new Rectangle(Width - (int)(Width / 11.63), (int)(Height / 21.6), Width / 12, Height / 27),
            $"speed{speedLevel2}");
        DrawIndicator(new Rectangle(Width - (int)(Width / 11.63), (int)(Height / 11.36), Width / 12, Height / 27),
            $"slow{slowLevel2}");
    }

    /// <summary>
    /// Определяет уровень индикатора топлива на основе текущего количества топлива.
    /// </summary>
    /// <param name="fuel">Текущее количество топлива.</param>
    /// <returns>Уровень индикатора от 0 до 5.</returns>
    private int GetFuelLevel(float fuel)
    {
        if (fuel > 80) return 5;
        if (fuel > 60) return 4;
        if (fuel > 40) return 3;
        if (fuel > 20) return 2;
        if (fuel > 0) return 1;
        return 0;
    }

    /// <summary>
    /// Определяет уровень индикатора эффекта приза (скорость или замедление)
    /// на основе оставшегося времени действия эффекта.
    /// </summary>
    /// <param name="car">Автомобиль, для которого проверяется эффект.</param>
    /// <param name="isSpeed">Если true, проверяется эффект ускорения; если false, проверяется эффект
    замедления.</param>
    /// <returns>Уровень индикатора от 0 до 5, основанный на оставшихся секундах.</returns>
    private int GetDecoratorLevel(Car car, bool isSpeed)
    {
        if (car._currentDecorator == null) return 0;
        TimeSpan elapsed = DateTime.Now - car._currentDecorator.timerStartTime;

```

```

        int remaining = (int)((car._currentDecorator._timer.Interval - elapsed.TotalMilliseconds) / 1000) + 1; //
        Оставшиеся секунды

        if ((isSpeed && car._currentDecorator is SpeedBoostDecorator) ||
            (!isSpeed && car._currentDecorator is SlowDownDecorator)) // Предполагается наличие
        SlowDownDecorator
        {
            // Уровни индикаторов соответствуют оставшимся секундам до 5
            if (remaining >= 5) return 5;
            if (remaining == 4) return 4;
            if (remaining == 3) return 3;
            if (remaining == 2) return 2;
            if (remaining == 1) return 1;
            return 0; // Эффект закончился
        }
        return 0; // Другой тип декоратора или нет нужного эффекта
    }

    /// <summary>
    /// Отрисовывает один индикатор HUD в заданном прямоугольнике с использованием указанной текстуры.
    /// </summary>
    /// <param name="rect">Прямоугольник, в котором будет отрисован индикатор.</param>
    /// <param name="textureKey">Ключ текстуры индикатора в словаре текстур.</param>
    private void DrawIndicator(Rectangle rect, string textureKey)
    {
        // Получаем ID текстуры по ключу
        if (!_textures.TryGetValue(textureKey, out int textureId)) return; // Выходим, если текстура не найдена

        GL.Enable(EnableCap.Texture2D); // Включаем использование текстур
        GL.BindTexture(TextureTarget.Texture2D, textureId); // Привязываем нужную текстуру

        // Начинаем отрисовку квадрата (прямоугольника)
        GL.Begin(PrimitiveType.Quads);
        // Задаем текстурные координаты и координаты вершин для каждого угла прямоугольника
        GL.TexCoord2(0, 0); GL.Vertex2(rect.Left, rect.Top); // Левый верхний угол текстуры к левому
        // верхнему углу прямоугольника
        GL.TexCoord2(1, 0); GL.Vertex2(rect.Right, rect.Top); // Правый верхний угол текстуры к правому
        // верхнему углу прямоугольника
        GL.TexCoord2(1, 1); GL.Vertex2(rect.Right, rect.Bottom); // Правый нижний угол текстуры к правому
        // нижнему углу прямоугольника
        GL.TexCoord2(0, 1); GL.Vertex2(rect.Left, rect.Bottom); // Левый нижний угол текстуры к левому
        // нижнему углу прямоугольника
        GL.End(); // Заканчиваем отрисовку

        GL.Disable(EnableCap.Texture2D); // Выключаем использование текстур
    }
}

```

Код программы *InputManager.cs*:

```

using OpenTK.Input;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RingRaceLab.Game
{
    /// <summary>

```

```

/// Управляет получением пользовательского ввода с клавиатуры для игроков.
/// </summary>
public class InputManager
{
    /// <summary>
    /// Получает состояние ввода с клавиатуры для первого игрока (клавиши W, S, A, D).
    /// </summary>
    /// <returns>Кортеж булевых значений, указывающих, нажаты ли клавиши Вперед, Назад, Влево, Вправо
    соответственно.</returns>
    public (bool forward, bool backward, bool left, bool right) GetPlayer1Input() => (
        Keyboard.GetState().IsKeyDown(Key.W),
        Keyboard.GetState().IsKeyDown(Key.S),
        Keyboard.GetState().IsKeyDown(Key.A),
        Keyboard.GetState().IsKeyDown(Key.D)
    );

    /// <summary>
    /// Получает состояние ввода с клавиатуры для второго игрока (клавиши стрелок).
    /// </summary>
    /// <returns>Кортеж булевых значений, указывающих, нажаты ли клавиши Вверх (Вперед), Вниз (Назад),
    Влево, Вправо соответственно.</returns>
    public (bool forward, bool backward, bool left, bool right) GetPlayer2Input() => (
        Keyboard.GetState().IsKeyDown(Key.Up),
        Keyboard.GetState().IsKeyDown(Key.Down),
        Keyboard.GetState().IsKeyDown(Key.Left),
        Keyboard.GetState().IsKeyDown(Key.Right)
    );
}
}

```

Код программы *PrizeManager.cs*:

```

using OpenTK;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using OpenTK.Graphics.OpenGL;

namespace RingRaceLab.Game
{
    /// <summary>
    /// Управляет призами на трассе.
    /// </summary>
    public class PrizeManager
    {
        private readonly List<IPrize> _activePrizes = new List<IPrize>();
        private readonly PrizeFactory[] _prizeFactories;
        private readonly CollisionMask _collisionSystem;

        /// <summary>
        /// Таймер возрождения призов.
        /// </summary>
        private readonly System.Timers.Timer _prizeRespawnTimer;

        private const int MIN_PRIZES = 5;

        /// <summary>
        /// Максимальное количество призов.
        /// </summary>
        public const int MAX_PRIZES = 10;
    }
}

```

```

private const int RESPAWN_INTERVAL = 3000;
private readonly int Width;
private readonly int Height;

/// <summary>
/// Инициализирует PrizeManager.
/// </summary>
/// <param name="prizeFactories">Фабрики призов.</param>
/// <param name="collisionSystem">Система коллизий.</param>
/// <param name="Width">Ширина.</param>
/// <param name="Height">Высота.</param>
public PrizeManager(PrizeFactory[] prizeFactories, CollisionMask collisionSystem, int Width, int Height)
{
    _prizeFactories = prizeFactories;
    _collisionSystem = collisionSystem;
    this.Width = Width;
    this.Height = Height;
    _prizeRespawnTimer = new System.Timers.Timer(RESPAWN_INTERVAL);
    _prizeRespawnTimer.Elapsed += (s, e) => RespawnPrizes();
    _prizeRespawnTimer.AutoReset = true;
    _prizeRespawnTimer.Start();
}

/// <summary>
/// Создает и размещает призы.
/// </summary>
/// <param name="count">Количество.</param>
public void SpawnPrizes(int count)
{
    Random rand = new Random();
    int maxAttempts = 100;
    int spawned = 0;
    while (spawned < count && maxAttempts-- > 0)
    {
        Vector2 position = new Vector2(rand.Next((int)(Width / 38.4), _collisionSystem.Width - (int)(Width / 38.4)),
rand.Next((int)(Width / 38.4), _collisionSystem.Height - (int)(Width / 38.4)));
        if (IsValidPosition(position))
        {
            var factory = _prizeFactories[rand.Next(_prizeFactories.Length)];
            _activePrizes.Add(factory.CreatePrize(position));
            spawned++;
        }
    }
}

private bool IsValidPosition(Vector2 position)
{
    if (!_collisionSystem.IsDrivable((int)position.X, (int)position.Y)) return false;
    lock (_activePrizes)
    {
        foreach (var prize in _activePrizes)
        {
            if (Vector2.Distance(position, prize.Position) < (int)(Width / 38.4)) return false;
        }
    }

    return true;
}

/// <summary>
/// Проверяет столкновения автомобиля с призами.
/// </summary>

```

```

/// <param name="car">Автомобиль.</param>
public void CheckPrizeCollisions(Car car)
{
    lock (_activePrizes)
    {
        for (int i = _activePrizes.Count - 1; i >= 0; i--)
        {
            if (Vector2.Distance(car._movement.Position, _activePrizes[i].Position) < (int)(Width / 38.4))
            {
                _activePrizes[i].ApplyEffect(car);
                _activePrizes.RemoveAt(i);
            }
        }
    }
}

private void RespawnPrizes()
{
    lock (_activePrizes)
    {
        if (_activePrizes.Count < MIN_PRIZES)
        {
            int needed = MAX_PRIZES - _activePrizes.Count;
            SpawnPrizes(needed);
        }
    }
}

/// <summary>
/// Отрисовывает призы.
/// </summary>
public void DrawPrizes()
{
    lock (_activePrizes)
    {
        foreach (var prize in _activePrizes)
        {
            GL.PushMatrix();
            GL.Enable(EnableCap.Texture2D);
            GL.BindTexture(TextureTarget.Texture2D, prize.TextureId);
            GL.Begin(PrimitiveType.Quads);
            GL.TexCoord2(0, 0); GL.Vertex2(prize.Position.X - Width / 240, prize.Position.Y - (int)(Height / 67.5));
            GL.TexCoord2(1, 0); GL.Vertex2(prize.Position.X + Width / 240, prize.Position.Y - (int)(Height / 67.5));
            GL.TexCoord2(1, 1); GL.Vertex2(prize.Position.X + Width / 240, prize.Position.Y + (int)(Height / 67.5));
            GL.TexCoord2(0, 1); GL.Vertex2(prize.Position.X - Width / 240, prize.Position.Y + (int)(Height / 67.5));
            GL.End();
            GL.PopMatrix();
        }
    }
}

/// <summary>
/// Сбрасывает состояние.
/// </summary>
public void Reset()
{
    _prizeRespawnTimer.Stop();
    lock (_activePrizes)
    {
        _activePrizes.Clear();
    }
}
}

```

Код программы *TextureCache.cs*:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RingRaceLab.Game
{
    public static class TextureCache
    {
        private static Dictionary<string, int> _textures = new Dictionary<string, int>();

        public static int GetTexture(string path)
        {
            if (!_textures.ContainsKey(path))
            {
                _textures[path] = TextureLoader.LoadFromFile(path);
            }
            return _textures[path];
        }
    }
}
```

Код программы *TextureLoader.cs*:

```
using System;
using System.Collections.Generic;
using System.Drawing.Imaging;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using OpenTK;
using OpenTK.Graphics.OpenGL;

namespace RingRaceLab
{
    /// <summary>
    /// Загрузчик текстур для OpenGL.
    /// </summary>
    public static class TextureLoader
    {
        /// <summary>
        /// Загружает текстуру из файла.
        /// </summary>
        /// <param name="path">Путь к файлу текстуры.</param>
        /// <returns>Идентификатор текстуры OpenGL.</returns>
        /// <exception cref="FileNotFoundException">Файл текстуры не найден.</exception>
        public static int LoadFromFile(string path)
        {
            if (!File.Exists(path))
                throw new FileNotFoundException("Texture not found", path);
            int id;
            using (var bitmap = new Bitmap(path))
            {
                GL.GenTextures(1, out id);
                GL.BindTexture(TextureTarget.Texture2D, id);
            }
        }
    }
}
```



```

        BitmapData data = bitmap.LockBits(
            new Rectangle(0, 0, bitmap.Width, bitmap.Height),
            ImageLockMode.ReadOnly,
            System.Drawing.Imaging.PixelFormat.Format32bppArgb);

        GL TexImage2D(TextureTarget.Texture2D, 0,
            PixelInternalFormat.Rgba, data.Width, data.Height, 0,
            OpenTK.Graphics.OpenGL.PixelFormat.Bgra,
            PixelType.UnsignedByte, data.Scan0);

        bitmap.UnlockBits(data);

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

```

Код программы *GameBuilder.cs*:

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Security.Cryptography.X509Certificates;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace RingRaceLab.Menu
{
    /// <summary>
    /// Строитель игрового интерфейса, в частности панели завершения игры.
    /// </summary>
    public class GameBuilder
    {
        /// <summary>
        /// Панель игры.
        /// </summary>
        private readonly Panel GamePanel;

        /// <summary>
        /// Флаг состояния.
        /// </summary>
        public bool flag = false;

        private readonly Action _exitToMenu;
        private PictureBox player;
        private FlowLayoutPanel GameFinishedPanel;
        private readonly int Width;
        private readonly int Height;

        /// <summary>
        /// Инициализирует GameBuilder.
        /// </summary>
        /// <param name="GamePanel">Панель игры.</param>
        /// <param name="exitToMenu">Действие для выхода в меню.</param>
    }
}

```

```

/// <param name="Width">Ширина.</param>
/// <param name="Height">Высота.</param>
public GameBuilder(Panel GamePanel, Action exitToMenu, int Width, int Height)
{
    this.GamePanel = GamePanel;
    _exitToMenu = exitToMenu;
    this.Width = Width;
    this.Height = Height;
}

/// <summary>
/// Строит интерфейс завершения игры.
/// </summary>
public void Build()
{
    GameFinishedPanel = new FlowLayoutPanel
    {
        FlowDirection = FlowDirection.TopDown,
        Visible = false,
        Width = (int)(this.Width / 3.84),
        Height = (int)(this.Height / 2.16),
        Location = new Point(Width / 2 - (int)(Width / 7.68), Height / 2 - (int)(Height / 4.32)),
        BackgroundImage = Image.FromFile("sprites/EndGamePanel.png"),
    };
    player = new PictureBox
    {
        Height = (int)(Height / 10.8),
        Width = (int)(Width / 3.88),
        BackColor = Color.Transparent,
        SizeMode = PictureBoxSizeMode.CenterImage,
    };
    PictureBox win = new PictureBox
    {
        Width = (int)(Width / 3.88),
        Height = (int)(Height / 10.8),
        BackColor = Color.Transparent,
        SizeMode = PictureBoxSizeMode.CenterImage,
        Image = Image.FromFile("sprites/win.png")
    };
    Button ExitToMenuButton = new Button
    {
        Width = (int)(Width / 6.87),
        Height = (int)(Height / 8.3),
        FlatStyle = FlatStyle.Flat,
        Margin = new Padding((int)(Width / 17.45), Height / 18, 0, 0),
        BackgroundImageLayout = ImageLayout.Stretch,
        BackgroundImage = Image.FromFile("sprites/ExitToMenuButton.png"),
        BackColor = Color.Transparent
    };
    ExitToMenuButton.Click += ExitToMenuButton_Click;
    ExitToMenuButton.FlatAppearance.BorderSize = 0;
    ExitToMenuButton.FlatAppearance.MouseDownBackColor = Color.Transparent;
    ExitToMenuButton.FlatAppearance.MouseOverBackColor = Color.Transparent;
    GameFinishedPanel.Controls.Add(player);
    GameFinishedPanel.Controls.Add(win);
    GameFinishedPanel.Controls.Add(ExitToMenuButton);
    GamePanel.Controls.Add(GameFinishedPanel);
}

private void ExitToMenuButton_Click(object sender, EventArgs e)
{
    _exitToMenu();
}

```

```

        flag = false;
        GameFinishedPanel.Visible = false;
    }

    /// <summary>
    /// Устанавливает изображение победителя.
    /// </summary>
    /// <param name="image">Изображение победителя.</param>
    public void SetWinner(Image image)
    {
        player.Image = image;
    }

    /// <summary>
    /// Показывает панель завершения игры.
    /// </summary>
    public void ShowFinishedPanel()
    {
        GameFinishedPanel.Visible = true;
    }
}
}

```

Код программы *GameConstants.cs*:

```

using OpenTK;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RingRaceLab
{
    public static class GameConstants
    {
        public static Dictionary<string, Vector2[]> TrackSpawnPositions = new Dictionary<string, Vector2[]>
        {
            { "sprites/road1.png", new[] { new Vector2(919, 82), new Vector2(919, 246) } },
            { "sprites/road2.png", new[] { new Vector2(895, 170), new Vector2(895, 70) } },
            { "sprites/road3.png", new[] { new Vector2(895, 280), new Vector2(895, 230) } }
        };

        public static Dictionary<string, Vector2[]> TrackFinishPositions = new Dictionary<string, Vector2[]>
        {
            { "sprites/road1.png", new[] { new Vector2(950, 10), new Vector2(950, 319) } },
            { "sprites/road2.png", new[] { new Vector2(928, 10), new Vector2(928, 269) } },
            { "sprites/road3.png", new[] { new Vector2(928, 10), new Vector2(928, 269) } }
        };
    }
}

```

Код программы *GameController.cs*:

```

using OpenTK.Graphics;
using OpenTK;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;

```

```

using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using OpenTK.Graphics.OpenGL;
using RingRaceLab.Menu;

namespace RingRaceLab
{
    /// <summary>
    /// Контроллер игры.
    /// </summary>
    public class GameController : IGameController
    {
        /// <summary>
        /// Панель игры.
        /// </summary>
        public Panel GamePanel { get; }

        /// <summary>
        /// Действие выхода в меню.
        /// </summary>
        public readonly Action _exitToMenu;

        /// <summary>
        /// Менеджер игры.
        /// </summary>
        private GameManager _gameManager;

        /// <summary>
        /// Элемент управления OpenGL.
        /// </summary>
        private GLControl _glControl;

        /// <summary>
        /// PictureBox игрока.
        /// </summary>
        public PictureBox player;

        /// <summary>
        /// Строитель UI игры.
        /// </summary>
        private readonly GameBuilder gameBuilder;

        /// <summary>
        /// Инициализирует контроллер.
        /// </summary>
        /// <param name="exitToMenu">Действие выхода в меню.</param>
        /// <param name="Width">Ширина.</param>
        /// <param name="Height">Высота.</param>
        public GameController(Action exitToMenu, int Width, int Height)
        {
            GamePanel = new Panel
            {
                Dock = DockStyle.Fill,
                BackColor = Color.Black,
                Visible = false
            };

            _exitToMenu = exitToMenu;
            gameBuilder = new GameBuilder(GamePanel, _exitToMenu, Width, Height);
            gameBuilder.Build();
            SetupGL();
        }
    }
}

```

```

private void SetupGL()
{
    _glControl = new GLControl(new GraphicsMode(32, 0, 0, 4))
    {
        Dock = DockStyle.Fill
    };
    _glControl.Load += (s, e) =>
    {
        GL.ClearColor(Color4.CornflowerBlue);
        GL.Enable(EnableCap.Blend);
        GL.BlendFunc(BlendingFactor.SrcAlpha, BlendingFactor.OneMinusSrcAlpha);
        SetupViewport();
    };

    _glControl.Paint += (s, e) =>
    {
        if (!gameBuilder.flag)
        {
            _gameManager?.Update(_glControl);
            _gameManager?.Draw();
        }

        _glControl.SwapBuffers();
    };

    _glControl.Resize += (s, e) => SetupViewport();

    _glControl.KeyDown += OnKeyDown;
    GamePanel.Controls.Add(_glControl);
}

private void SetupViewport()
{
    GL.Viewport(0, 0, _glControl.ClientSize.Width, _glControl.ClientSize.Height);
    GL.MatrixMode(MatrixMode.Projection);
    GL.LoadIdentity();
    GL.Ortho(0, _glControl.ClientSize.Width, _glControl.ClientSize.Height, 0, -1, 1);
    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity();
}

/// <summary>
/// Запускает игру.
/// </summary>
/// <param name="track">Имя трека.</param>
/// <param name="player1Car">Текстура машины игрока 1.</param>
/// <param name="player2Car">Текстура машины игрока 2.</param>
public void StartGame(string track, string player1Car, string player2Car)
{
    GamePanel.Show();
    _glControl.Focus();

    string collisionMap = track.Replace(".png", "_map.png");
    Vector2[] spawnPositions = GameConstants.TrackSpawnPositions[track];
    Vector2[] finishPositions = GameConstants.TrackFinishPositions[track];
    _gameManager = new GameManager(track, collisionMap, spawnPositions, finishPositions, player1Car,
player2Car, _glControl.Width, _glControl.Height);
    _gameManager.OnCarFinished += OnCarFinished;
    _glControl.Invalidate();
}

```

```

private void OnKeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Escape)
    {
        _gameManager.OnCarFinished -= OnCarFinished;
        _exitToMenu();
    }
}

/// <summary>
/// Скрывает игру.
/// </summary>
public void HideGame() => GamePanel.Hide();

private void OnCarFinished(Car car)
{
    gameBuilder.flag = true;
    gameBuilder.SetWinner(Image.FromFile(car._renderer._texturePath.Contains("blue") ?
"sprites/player1_win.png" : "sprites/player2_win.png"));
    gameBuilder.ShowFinishedPanel();
}
}
}

```

Код программы *IGameController.cs*:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace RingRaceLab
{
    public interface IGameController
    {
        Panel GamePanel { get; }
        void StartGame(string track, string player1Car, string player2Car);
        void HideGame();
    }
}

```

Код программы *IMenuController.cs*:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace RingRaceLab
{
    public interface IMenuController
    {
        Panel MenuPanel { get; }
        void ShowMenu();
        void HideMenu();
    }
}

```

```

        string SelectedTrack { get; }
        string Player1CarTexture { get; }
        string Player2CarTexture { get; }
    }
}

```

Код программы *MenuBuilder.cs*:

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace RingRaceLab
{
    public class MenuBuilder
    {
        private readonly int Width;
        private readonly int Height;
        public string SelectedTrack => trackList[trackIndex];
        public string Player1CarTexture => player1Cars[player1Index].Replace("_menu", "");
        public string Player2CarTexture => player2Cars[player2Index].Replace("_menu", "");

        private readonly Panel _menuPanel;
        private readonly Action _onStartGame;

        private PictureBox trackPreview;
        private List<string> trackList = new List<string> { "sprites/road1.png", "sprites/road2.png", "sprites/road3.png" };
        private int trackIndex = 0;

        private List<string> player1Cars = new List<string> { "sprites/car1_blue_menu.png",
"sprites/car2_blue_menu.png" };
        private List<string> player2Cars = new List<string> { "sprites/car1_red_menu.png", "sprites/car2_red_menu.png"
};
        private int player1Index = 0;
        private int player2Index = 0;

        public MenuBuilder(Panel panel, Action onStartGame, int Width, int Height)
        {
            this.Width = Width;
            this.Height = Height;
            _menuPanel = panel;
            _onStartGame = onStartGame;
            Build();
        }

        private void StyleButton(Button btn)
        {
            btn.FlatStyle = FlatStyle.Flat;
            btn.FlatAppearance.BorderSize = 0;
            btn.BackColor = Color.Transparent;
            btn.FlatAppearance.MouseOverBackColor = Color.Transparent;
            btn.FlatAppearance.MouseDownBackColor = Color.Transparent;
        }

        public void Build()
        {
            // фон и двойная буферизация
            _menuPanel.BackgroundImage = Image.FromFile("sprites/background_menu.png");

```

```

SetDoubleBuffered(_menuPanel);

var outer = new TableLayoutPanel
{
    Dock = DockStyle.Fill,
    ColumnCount = 3,
    RowCount = 1,
    BackColor = Color.Transparent
};
outer.ColumnStyles.Add(new ColumnStyle(SizeType.Percent, 33f));
outer.ColumnStyles.Add(new ColumnStyle(SizeType.Percent, 33f));
outer.ColumnStyles.Add(new ColumnStyle(SizeType.Percent, 33f));

_menuPanel.Controls.Add(outer);
outer.Controls.Add(BuildPlayerPanel(true), 0, 0);
outer.Controls.Add(BuildCenterPanel(), 1, 0);
outer.Controls.Add(BuildPlayerPanel(false), 2, 0);
// Добавляем кнопку выхода
var exitButton = new Button
{
    Size = new Size(Width/13, (int)(Height/7.5)),
    BackgroundImage = Image.FromFile("sprites/close.png"),
    BackgroundImageLayout = ImageLayout.Stretch,
    Anchor = AnchorStyles.Right | AnchorStyles.Top,
    FlatStyle = FlatStyle.Flat
};
StyleButton(exitButton);
exitButton.Click += (s, e) => Application.Exit();

// Добавляем кнопку на панель и выводим поверх остальных элементов
_menuPanel.Controls.Add(exitButton);
exitButton.BringToFront();
}

private TableLayoutPanel BuildPlayerPanel(bool isPlayer1)
{
    var tl = new TableLayoutPanel
    {
        Dock = DockStyle.Fill,
        RowCount = 5,
        ColumnCount = 1,
    };
    tl.RowStyles.Add(new RowStyle(SizeType.Percent, 10f)); // верхний отступ
    tl.RowStyles.Add(new RowStyle(SizeType.AutoSize)); // текст
    tl.RowStyles.Add(new RowStyle(SizeType.AutoSize)); // превью машины
    tl.RowStyles.Add(new RowStyle(SizeType.AutoSize)); // кнопки
    tl.RowStyles.Add(new RowStyle(SizeType.Percent, 17f)); // нижний отступ

    // --- 1) текстовый заголовок
    var textBox = new PictureBox
    {
        Image = Image.FromFile(isPlayer1 ? "sprites/player1_menu.png" : "sprites/player2_menu.png"),
        SizeMode = PictureBoxSizeMode.StretchImage,
        Size = new Size(Width/3, (int)(Height/10.8)),
        Dock = DockStyle.Top,
        Anchor = AnchorStyles.Right
    };
    tl.Controls.Add(textBox, 0, 1);

    // --- 2) превью машины
    var carPreview = new PictureBox
    {
        SizeMode = PictureBoxSizeMode.CenterImage,

```



```

        Size = new Size(Width/12, (int)(Height/3.4)),
        Dock = DockStyle.Top,
        Margin = new Padding(0, Height/108, 0, Height/108)
    };
    void UpdateCarPreview()
    {
        var list = isPlayer1 ? player1Cars : player2Cars;
        var idx = isPlayer1 ? player1Index : player2Index;
        var img = Image.FromFile(list[idx]);
        img.RotateFlip(RotateFlipType.Rotate90FlipNone);
        carPreview.Image = img;
    }
    UpdateCarPreview();
    tl.Controls.Add(carPreview, 0, 2);

    // --- 3) кнопки переключения
    var btnL = new Button { Size = new Size(Width / 13, (int)(Height / 7.5)), BackgroundImageLayout =
ImageLayout.Stretch };
    var btnR = new Button { Size = new Size(Width / 13, (int)(Height / 7.5)), BackgroundImageLayout =
ImageLayout.Stretch };
    btnL.BackgroundImage = Image.FromFile("sprites/button_left.png");
    btnR.BackgroundImage = Image.FromFile("sprites/button_right.png");
    StyleButton(btnL);
    StyleButton(btnR);

    btnL.Click += (s, e) =>
    {
        if (isPlayer1)
            player1Index = (player1Index - 1 + player1Cars.Count) % player1Cars.Count;
        else
            player2Index = (player2Index - 1 + player2Cars.Count) % player2Cars.Count;
        UpdateCarPreview();
    };
    btnR.Click += (s, e) =>
    {
        if (isPlayer1)
            player1Index = (player1Index + 1) % player1Cars.Count;
        else
            player2Index = (player2Index + 1) % player2Cars.Count;
        UpdateCarPreview();
    };

    var flow = new FlowLayoutPanel
    {
        FlowDirection = FlowDirection.LeftToRight,
        Dock = DockStyle.Bottom,
        AutoSize = true,
        Anchor = AnchorStyles.Bottom,
        Padding = new Padding(0, 0, 0, Height / 54)
    };
    flow.Controls.Add(btnL);
    flow.Controls.Add(btnR);
    tl.Controls.Add(flow, 0, 3);

    return tl;
}

private TableLayoutPanel BuildCenterPanel()
{
    var tl = new TableLayoutPanel
    {
        Dock = DockStyle.Fill,
        RowCount = 4,

```

```

        ColumnCount = 1,
    };
    tl.RowStyles.Add(new RowStyle(SizeType.Percent, 100f)); // пустое пространство
    tl.RowStyles.Add(new RowStyle(SizeType.AutoSize)); // превью трассы
    tl.RowStyles.Add(new RowStyle(SizeType.AutoSize)); // кнопки L/R
    tl.RowStyles.Add(new RowStyle(SizeType.AutoSize)); // кнопка Start

    // --- trackPreview
    trackPreview = new PictureBox
    {
        Image = Image.FromFile(trackList[trackIndex]),
        SizeMode = PictureBoxSizeMode.StretchImage,
        Size = new Size(Width/3, (int)(Height/3.2)),
        Dock = DockStyle.Top,
        Margin = new Padding(0, Height/54, 0, Height/54)
    };
    tl.Controls.Add(trackPreview, 0, 1);

    // --- кнопки L/R
    var btnLeft = new Button { Size = new Size(Width / 13, (int)(Height / 7.5)), BackgroundImageLayout =
    ImageLayout.Stretch };
    var btnRight = new Button { Size = new Size(Width / 13, (int)(Height / 7.5)), BackgroundImageLayout =
    ImageLayout.Stretch };
    btnLeft.BackgroundImage = Image.FromFile("sprites/button_left.png");
    btnRight.BackgroundImage = Image.FromFile("sprites/button_right.png");
    StyleButton(btnLeft);
    StyleButton(btnRight);

    btnLeft.Click += (s, e) =>
    {
        trackIndex = (trackIndex - 1 + trackList.Count) % trackList.Count;
        trackPreview.Image = Image.FromFile(trackList[trackIndex]);
    };
    btnRight.Click += (s, e) =>
    {
        trackIndex = (trackIndex + 1) % trackList.Count;
        trackPreview.Image = Image.FromFile(trackList[trackIndex]);
    };

    var flowLR = new FlowLayoutPanel
    {
        FlowDirection = FlowDirection.LeftToRight,
        AutoSize = true,
        Dock = DockStyle.Top,
        Anchor = AnchorStyles.Top,
        Margin = new Padding(0, 0, 0, Height / 54)
    };
    flowLR.Controls.Add(btnLeft);
    flowLR.Controls.Add(btnRight);
    tl.Controls.Add(flowLR, 0, 2);

    // --- btnStart
    var btnStart = new Button
    {
        Size = new Size(Width/3, (int)(Height/4.15)),
        BackgroundImage = Image.FromFile("sprites/button_up.png"),
        BackgroundImageLayout = ImageLayout.Stretch,
        Dock = DockStyle.Top,
        Margin = new Padding(0, 0, 0, Height / 54)
    };
    StyleButton(btnStart);
    btnStart.Click += (s, e) => _onStartGame();
    tl.Controls.Add(btnStart, 0, 3);

```

```

        return tl;
    }

    // Отсутствие мерцания
    public static void SetDoubleBuffered(Control c)
    {
        if (SystemInformation.TerminalServerSession) return;
        var prop = typeof(Control).GetProperty("DoubleBuffered",
            System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Instance);
        prop.SetValue(c, true, null);
    }
}

```

Код программы *MenuController.cs*:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace RingRaceLab
{
    public class MenuController : IMenuController
    {
        public Panel MenuPanel { get; }
        private readonly Action _onStartGame;
        private readonly MenuBuilder _menuBuilder;
        public string SelectedTrack => _menuBuilder.SelectedTrack;
        public string Player1CarTexture => _menuBuilder.Player1CarTexture;
        public string Player2CarTexture => _menuBuilder.Player2CarTexture;

        public MenuController(IGameController gameController, Action onStartGame, int Width, int Height)
        {
            _onStartGame = onStartGame;
            MenuPanel = new Panel { Dock = DockStyle.Fill };
            _menuBuilder = new MenuBuilder(MenuPanel, _onStartGame, Width, Height);
        }

        public void ShowMenu() => MenuPanel.Show();
        public void HideMenu() => MenuPanel.Hide();
    }
}

```

Код программы *Car.cs*:

```

using OpenTK;
using System;
using System.Collections.Generic;
using System.Linq;

namespace RingRaceLab
{
    /// <summary>
    /// Представляет игровой автомобиль.
    /// </summary>
    public class Car : GameEntity // Убедитесь, что GameEntity доступен

```

```

{
    /// <summary>
    /// Управление движением автомобиля.
    /// </summary>
    internal CarMovement _movement;

    /// <summary>
    /// Рендерер автомобиля.
    /// </summary>
    internal CarRenderer _renderer;

    /// <summary>
    /// Физика автомобиля.
    /// </summary>
    internal CarPhysics _physics;

    /// <summary>
    /// Количество завершенных кругов.
    /// </summary>
    public int lapsComplete = -1;

    /// <summary>
    /// Текущий уровень топлива.
    /// </summary>
    public float Fuel { get; set; } = 100;

    /// <summary>
    /// Активный декоратор (эффект приза).
    /// </summary>
    public CarDecorator _currentDecorator;

    /// <summary>
    /// Инициализирует новый автомобиль.
    /// </summary>
    /// <param name="startPosition">Стартовая позиция.</param>
    /// <param name="texturePath">Путь к текстуре.</param>
    /// <param name="config">Конфигурация автомобиля.</param>
    public Car(Vector2 startPosition, string texturePath, CarConfig config)
    {
        _movement = new CarMovement(startPosition, config);
        _renderer = new CarRenderer(texturePath, config.Size);
        _physics = new CarPhysics(config.Size);
    }

    /// <summary>
    /// Обновляет состояние автомобиля (движение, топливо, декоратор).
    /// </summary>
    /// <param name="deltaTime">Время с последнего обновления.</param>
    /// <param name="moveForward">Двигаться вперед?</param>
    /// <param name="moveBackward">Двигаться назад?</param>
    /// <param name="turnLeft">Повернуть влево?</param>
    /// <param name="turnRight">Повернуть вправо?</param>
    public void Update(float deltaTime, bool moveForward, bool moveBackward, bool turnLeft, bool turnRight)
    {
        Fuel -= Math.Abs(_movement.CurrentSpeed) * deltaTime * _movement._config.FuelConsumptionRate;
        Fuel = Math.Max(0, Fuel);

        if (Fuel > 0)
        {
            _movement.Update(deltaTime, moveForward, moveBackward, turnLeft, turnRight);
        }

        // Обновление декоратора

```

```

        _currentDecorator?.Update(deltaTime);
    }

    /// <summary>
    /// Применяет декоратор к автомобилю.
    /// </summary>
    /// <param name="newDecorator">Декоратор для применения.</param>
    public void ApplyDecorator(CarDecorator newDecorator)
    {
        _currentDecorator?.Remove();
        _currentDecorator = newDecorator;
        _currentDecorator.Apply();
    }

    /// <summary>
    /// Удаляет текущий декоратор с автомобиля.
    /// </summary>
    public void RemoveDecorator()
    {
        _currentDecorator?.Remove();
        _currentDecorator = null;
    }

    /// <summary>
    /// Отрисовывает автомобиль.
    /// </summary>
    public override void Draw()
    {
        _renderer.Draw(_movement.Position, _movement.Angle);
    }

    /// <summary>
    /// Получает координаты углов автомобиля.
    /// </summary>
    /// <returns>Список координат углов.</returns>
    public List<Vector2> GetCorners()
    {
        return _physics.GetCorners(_movement.Position, _movement.Angle);
    }
}
}

```

Код программы *CarConfig.cs*:

```

using OpenTK;
namespace RingRaceLab
{
    /// <summary>
    /// Конфигурация параметров автомобиля.
    /// </summary>
    public class CarConfig
    {
        public float ForwardAcceleration { get; set; } = 150f;
        public float ForwardMaxSpeed { get; set; } = 400f;
        public float ReverseAcceleration { get; set; } = 120f;
        public float ReverseMaxSpeed { get; set; } = 200f;
        public float Deceleration { get; set; } = 75f;
        public float TurnSpeed { get; set; } = 180f;
        public Vector2 Size { get; set; } = new Vector2(32f, 16f);
        public float FuelConsumptionRate { get; set; } = 0.01f;
    }
}

```

Код программы *CarDecorator.cs*:

```
using System;
using System.Timers;

namespace RingRaceLab
{
    /// <summary>
    /// Абстрактный базовый класс для декораторов автомобиля.
    /// </summary>
    public abstract class CarDecorator
    {
        /// <summary>
        /// Декорируемый автомобиль.
        /// </summary>
        protected readonly Car _car;

        /// <summary>
        /// Таймер длительности эффекта.
        /// </summary>
        public readonly Timer _timer;

        /// <summary>
        /// Длительность эффекта в секундах.
        /// </summary>
        protected readonly float _duration;

        /// <summary>
        /// Время начала действия таймера.
        /// </summary>
        public DateTime timerStartTime;

        /// <summary>
        /// Инициализирует новый экземпляр <see cref="CarDecorator"/>.
        /// </summary>
        /// <param name="car">Автомобиль для декорирования.</param>
        /// <param name="duration">Длительность эффекта в секундах.</param>
        public CarDecorator(Car car, float duration)
        {
            _car = car;
            _duration = duration;
            _timer = new Timer(duration * 1000);
            _timer.Elapsed += OnTimerEnd;
        }

        /// <summary>
        /// Применяет эффект декоратора и запускает таймер.
        /// </summary>
        public virtual void Apply()
        {
            _timer.Start();
            timerStartTime = DateTime.Now;
            ApplyEffect(); // Абстрактный метод для специфичного эффекта
        }

        /// <summary>
        /// Останавливает таймер и отменяет эффект декоратора.
        /// </summary>
        public virtual void Remove()
        {
            _timer.Stop();
            RevertEffect(); // Абстрактный метод для отмены специфичного эффекта
        }
    }
}
```

```

    }

    /// <summary>
    /// Обновляет состояние декоратора (если необходимо).
    /// </summary>
    /// <param name="deltaTime">Время, прошедшее с последнего обновления.</param>
    public void Update(float deltaTime)
    {
        // Логика обновления (если требуется в наследниках)
    }

    /// <summary>
    /// Применяет специфичный эффект декоратора к автомобилю. Реализуется в наследниках.
    /// </summary>
    protected abstract void ApplyEffect();

    /// <summary>
    /// Отменяет специфичный эффект декоратора. Реализуется в наследниках.
    /// </summary>
    protected abstract void RevertEffect();

    /// <summary>
    /// Обработчик события окончания таймера. Удаляет декоратор из автомобиля.
    /// </summary>
    private void OnTimerEnd(object sender, ElapsedEventArgs e)
    {
        _car.RemoveDecorator(); // Удаление декоратора через метод автомобиля
    }
}
}

```

Код программы *CarMovement.cs*:

```

using OpenTK;
using System;

namespace RingRaceLab
{
    /// <summary>
    /// Управляет движением и ориентацией автомобиля.
    /// </summary>
    public class CarMovement
    {
        /// <summary>
        /// Текущая позиция автомобиля.
        /// </summary>
        public Vector2 Position { get; set; }

        /// <summary>
        /// Текущий угол ориентации автомобиля (в градусах).
        /// </summary>
        public float Angle { get; set; }

        /// <summary>
        /// Текущая скорость автомобиля.
        /// </summary>
        public float CurrentSpeed { get; set; }

        /// <summary>
        /// Конфигурация параметров движения автомобиля.
        /// </summary>
        public CarConfig _config; // Убрал internal, так как используется из Car
    }
}

```

```

/// <summary>
/// Инициализирует управление движением автомобиля.
/// </summary>
/// <param name="startPosition">Начальная позиция.</param>
/// <param name="config">Конфигурация движения.</param>
public CarMovement(Vector2 startPosition, CarConfig config)
{
    Position = startPosition;
    _config = config;
}

/// <summary>
/// Обновляет позицию, угол и скорость автомобиля.
/// </summary>
/// <param name="deltaTime">Время, прошедшее с последнего обновления.</param>
/// <param name="moveForward">Флаг движения вперед.</param>
/// <param name="moveBackward">Флаг движения назад.</param>
/// <param name="turnLeft">Флаг поворота влево.</param>
/// <param name="turnRight">Флаг поворота вправо.</param>
public void Update(float deltaTime, bool moveForward, bool moveBackward, bool turnLeft, bool turnRight)
{
    // Переносим всю логику движения из старого класса Car
    if (moveForward && !moveBackward)
    {
        {
            if (CurrentSpeed > 0)
            {
                CurrentSpeed += _config.ForwardAcceleration * deltaTime;
                CurrentSpeed = Math.Min(CurrentSpeed, _config.ForwardMaxSpeed);
            }
            else
            {
                CurrentSpeed += _config.ForwardAcceleration * deltaTime * 2;
                CurrentSpeed = Math.Min(CurrentSpeed, _config.ForwardMaxSpeed);
            }
        }
    }
    else if (moveBackward && !moveForward)
    {
        {
            if (CurrentSpeed < 0)
            {
                CurrentSpeed -= _config.ReverseAcceleration * deltaTime;
                CurrentSpeed = Math.Max(CurrentSpeed, -_config.ReverseMaxSpeed);
            }
            else
            {
                CurrentSpeed -= _config.ReverseAcceleration * deltaTime * 3;
                CurrentSpeed = Math.Max(CurrentSpeed, -_config.ReverseMaxSpeed);
            }
        }
    }
    else
    {
        {
            ApplyDeceleration(deltaTime);
        }
    }

    UpdateRotation(deltaTime, turnLeft, turnRight);
    UpdatePosition(deltaTime);
}

private void ApplyDeceleration(float deltaTime)
{
    {
        if (CurrentSpeed > 0)
        {

```



```

        CurrentSpeed = Math.Max(0, CurrentSpeed - _config.Deceleration * deltaTime);
    }
    else if (CurrentSpeed < 0)
    {
        CurrentSpeed = Math.Min(0, CurrentSpeed + _config.Deceleration * deltaTime);
    }
}

private void UpdateRotation(float deltaTime, bool turnLeft, bool turnRight)
{
    if (Math.Abs(CurrentSpeed) > 0.1f)
    {
        float speedFactor = Math.Max(
            (_config.ForwardMaxSpeed - Math.Abs(CurrentSpeed)) / _config.ForwardMaxSpeed,
            0.8f
        );

        float effectiveTurnSpeed = _config.TurnSpeed * speedFactor;

        if (turnLeft) Angle -= effectiveTurnSpeed * deltaTime;
        if (turnRight) Angle += effectiveTurnSpeed * deltaTime;
    }
}

private void UpdatePosition(float deltaTime)
{
    float rad = MathHelper.DegreesToRadians(Angle);
    Vector2 direction = new Vector2((float)Math.Cos(rad), (float)Math.Sin(rad));
    Position += direction * CurrentSpeed * deltaTime;
}
}
}

```

Код программы *CarPhysics.cs*:

```

using OpenTK;
using System;
using System.Collections.Generic;

namespace RingRaceLab
{
    /// <summary>
    /// Управляет физическими расчетами для автомобиля.
    /// </summary>
    public class CarPhysics
    {
        /// <summary>
        /// Размер автомобиля.
        /// </summary>
        private readonly Vector2 _size;

        /// <summary>
        /// Инициализирует физику автомобиля.
        /// </summary>
        /// <param name="size">Размер автомобиля.</param>
        public CarPhysics(Vector2 size)
        {
            _size = size;
        }

        /// <summary>
        /// Вычисляет мировые координаты углов автомобиля.
    }
}

```

```

/// </summary>
/// <param name="position">Позиция автомобиля.</param>
/// <param name="angle">Угол поворота автомобиля в градусах.</param>
/// <returns>Список мировых координат углов автомобиля.</returns>
public List<Vector2> GetCorners(Vector2 position, float angle)
{
    float rad = MathHelper.DegreesToRadians(angle);
    float cos = (float)Math.Cos(rad);
    float sin = (float)Math.Sin(rad);

    Vector2[] localCorners = {
        new Vector2(-_size.X, -_size.Y),
        new Vector2(_size.X, -_size.Y),
        new Vector2(_size.X, _size.Y),
        new Vector2(-_size.X, _size.Y)
    };

    List<Vector2> globalCorners = new List<Vector2>();
    foreach (var local in localCorners)
    {
        // Поворот и смещение
        float rotatedX = local.X * cos - local.Y * sin;
        float rotatedY = local.X * sin + local.Y * cos;
        globalCorners.Add(new Vector2(
            position.X + rotatedX,
            position.Y + rotatedY
        ));
    }

    return globalCorners;
}
}
}

```

Код программы *CarRender.cs*:

```

using OpenTK;
using OpenTK.Graphics.OpenGL;

namespace RingRaceLab
{
    public class CarRenderer
    {
        private readonly int _textureId;
        private readonly Vector2 _size;
        public readonly string _texturePath;

        public CarRenderer(string texturePath, Vector2 size)
        {
            _size = size;
            _textureId = TextureLoader.LoadFromFile(texturePath);
            _texturePath = texturePath;
        }

        public void Draw(Vector2 position, float angle)
        {
            GL.PushMatrix();
            GL.Translate(position.X, position.Y, 0);
            GL.Rotate(angle, 0, 0, 1);

            GL.Enable(EnableCap.Texture2D);
            GL.BindTexture(TextureTarget.Texture2D, _textureId);
        }
    }
}

```

```

        GL.Begin(PrimitiveType.Quads);
        GL.TexCoord2(0, 0); GL.Vertex2(-_size.X, -_size.Y);
        GL.TexCoord2(1, 0); GL.Vertex2(_size.X, -_size.Y);
        GL.TexCoord2(1, 1); GL.Vertex2(_size.X, _size.Y);
        GL.TexCoord2(0, 1); GL.Vertex2(-_size.X, _size.Y);
        GL.End();

        GL.Disable(EnableCap.Texture2D);
        GL.PopMatrix();
    }
}
}

```

Код программы *FuelPrize.cs*:

```

using OpenTK;
using System.Windows.Forms;
using System;
using RingRaceLab.Game;

namespace RingRaceLab
{
    /// <summary>
    /// Приз, добавляющий топливо автомобилю.
    /// </summary>
    public class FuelPrize : IP Prize
    {
        /// <summary>
        /// Позиция приза.
        /// </summary>
        public Vector2 Position { get; set; }

        /// <summary>
        /// ID текстуры приза.
        /// </summary>
        public int TextureId { get; }

        /// <summary>
        /// Инициализирует приз топлива.
        /// </summary>
        /// <param name="position">Позиция приза.</param>
        public FuelPrize(Vector2 position)
        {
            Position = position;
            try
            {
                TextureId = TextureCache.GetTexture("sprites/fuel_prize.png");
            }
            catch (Exception ex)
            {
                MessageBox.Show($"Ошибка загрузки текстуры: {ex.Message}");
                TextureId = -1;
            }
        }

        /// <summary>
        /// Применяет эффект приза (добавляет топливо) к машине.
        /// </summary>
        /// <param name="car">Машина, к которой применяется эффект.</param>
        public void ApplyEffect(Car car)
        {

```

```

        car.Fuel = Math.Min(car.Fuel + 25, 100);
    }
}

```

Код программы *IPrize.cs*:

```

using OpenTK;

namespace RingRaceLab
{
    public interface IPrize
    {
        Vector2 Position { get; set; }
        int TextureId { get; }
        void ApplyEffect(Car car);
    }
}

```

Код программы *PrizeFactory.cs*:

```

using OpenTK;

namespace RingRaceLab
{
    /// <summary>
    /// Абстрактная фабрика для создания различных типов призов.
    /// </summary>
    public abstract class PrizeFactory
    {
        /// <summary>
        /// Создает новый экземпляр приза.
        /// </summary>
        /// <param name="position">Позиция приза.</param>
        /// <returns>Созданный приз.</returns>
        public abstract IPrize CreatePrize(Vector2 position);
    }

    /// <summary>
    /// Фабрика для создания призов топлива.
    /// </summary>
    public class FuelPrizeFactory : PrizeFactory
    {
        /// <summary>
        /// Создает приз топлива.
        /// </summary>
        /// <param name="position">Позиция приза.</param>
        /// <returns>Приз топлива.</returns>
        public override IPrize CreatePrize(Vector2 position) => new FuelPrize(position);
    }

    /// <summary>
    /// Фабрика для создания призов ускорения.
    /// </summary>
    public class SpeedBoostPrizeFactory : PrizeFactory
    {
        /// <summary>
        /// Создает приз ускорения.
        /// </summary>
        /// <param name="position">Позиция приза.</param>

```

```

    /// <returns>Приз ускорения.</returns>
    public override IPrize CreatePrize(Vector2 position) => new SpeedBoostPrize(position);
}

/// <summary>
/// Фабрика для создания призов замедления.
/// </summary>
public class SlowDownPrizeFactory : PrizeFactory
{
    /// <summary>
    /// Создает приз замедления.
    /// </summary>
    /// <param name="position">Позиция приза.</param>
    /// <returns>Приз замедления.</returns>
    public override IPrize CreatePrize(Vector2 position) => new SlowDownPrize(position);
}
}

```

Код программы *SlowDownDecorator.cs*:

```

using System.Timers;

namespace RingRaceLab
{
    /// <summary>
    /// Декоратор, применяющий эффект замедления к машине.
    /// </summary>
    public class SlowDownDecorator : CarDecorator
    {
        /// <summary>
        /// Множитель, применяемый к максимальной скорости.
        /// </summary>
        private readonly float _multiplier;

        /// <summary>
        /// Исходная максимальная скорость машины до применения эффекта.
        /// </summary>
        private float _originalSpeed;

        /// <summary>
        /// Инициализирует новый экземпляр декоратора замедления.
        /// </summary>
        /// <param name="car">Машина для декорирования.</param>
        /// <param name="multiplier">Множитель для замедления.</param>
        /// <param name="duration">Длительность эффекта в миллисекундах.</param>
        public SlowDownDecorator(Car car, float multiplier, float duration)
            : base(car, duration)
        {
            _multiplier = multiplier;
        }

        /// <summary>
        /// Применяет эффект замедления, уменьшая максимальную скорость машины.
        /// </summary>
        protected override void ApplyEffect()
        {
            _originalSpeed = _car._movement._config.ForwardMaxSpeed;
            _car._movement._config.ForwardMaxSpeed *= _multiplier;
        }

        /// <summary>
        /// Отменяет эффект замедления, восстанавливая исходную максимальную скорость.
    }
}

```

```

    /// </summary>
    protected override void RevertEffect()
    {
        _car._movement._config.ForwardMaxSpeed = _originalSpeed;
    }
}

```

Код программы *SlowDownPrize.cs*:

```

using OpenTK;
using System.Windows.Forms;
using System;
using RingRaceLab.Game;

namespace RingRaceLab
{
    /// <summary>
    /// Приз, применяющий эффект замедления к автомобилю.
    /// </summary>
    public class SlowDownPrize : IPrize
    {
        /// <summary>
        /// Позиция приза.
        /// </summary>
        public Vector2 Position { get; set; }

        /// <summary>
        /// ID текстуры приза.
        /// </summary>
        public int TextureId { get; }

        /// <summary>
        /// Инициализирует приз замедления.
        /// </summary>
        /// <param name="position">Позиция приза.</param>
        public SlowDownPrize(Vector2 position)
        {
            Position = position;
            try
            {
                TextureId = TextureCache.GetTexture("sprites/slow_prize.png");
            }
            catch (Exception ex)
            {
                MessageBox.Show($"Ошибка загрузки текстуры: {ex.Message}");
                TextureId = -1; // Используйте значение по умолчанию
            }
        }

        /// <summary>
        /// Применяет эффект замедления к машине.
        /// </summary>
        /// <param name="car">Машина, к которой применяется эффект.</param>
        public void ApplyEffect(Car car)
        {
            car.ApplyDecorator(new SlowDownDecorator(car, 0.5f, 5f));
        }
    }
}

```

Код программы *SpeedBoostDecorator.cs*:

```
using System.Timers;

namespace RingRaceLab
{
    /// <summary>
    /// Декоратор, применяющий эффект ускорения к машине.
    /// </summary>
    public class SpeedBoostDecorator : CarDecorator
    {
        /// <summary>
        /// Множитель, применяемый к максимальной скорости.
        /// </summary>
        private readonly float _multiplier;

        /// <summary>
        /// Исходная максимальная скорость машины до применения эффекта.
        /// </summary>
        private float _originalSpeed;

        /// <summary>
        /// Инициализирует новый экземпляр декоратора ускорения.
        /// </summary>
        /// <param name="car">Машина для декорирования.</param>
        /// <param name="multiplier">Множитель для ускорения.</param>
        /// <param name="duration">Длительность эффекта в миллисекундах.</param>
        public SpeedBoostDecorator(Car car, float multiplier, float duration)
            : base(car, duration)
        {
            _multiplier = multiplier;
        }

        /// <summary>
        /// Применяет эффект ускорения, увеличивая максимальную скорость машины.
        /// </summary>
        protected override void ApplyEffect()
        {
            _originalSpeed = _car._movement._config.ForwardMaxSpeed;
            _car._movement._config.ForwardMaxSpeed *= _multiplier;
        }

        /// <summary>
        /// Отменяет эффект ускорения, восстанавливая исходную максимальную скорость.
        /// </summary>
        protected override void RevertEffect()
        {
            _car._movement._config.ForwardMaxSpeed = _originalSpeed;
        }
    }
}
```

Код программы *SpeedBoostPrize.cs*:

```
using OpenTK;
using System.Windows.Forms;
using System;
using RingRaceLab.Game;

namespace RingRaceLab
{
    /// <summary>
```

```

/// Приз, применяющий эффект ускорения к автомобилю.
/// </summary>
public class SpeedBoostPrize : IPrize
{
    /// <summary>
    /// Позиция приза.
    /// </summary>
    public Vector2 Position { get; set; }

    /// <summary>
    /// ID текстуры приза.
    /// </summary>
    public int TextureId { get; }

    /// <summary>
    /// Инициализирует приз ускорения.
    /// </summary>
    /// <param name="position">Позиция приза.</param>
    public SpeedBoostPrize(Vector2 position)
    {
        Position = position;
        try
        {
            TextureId = TextureCache.GetTexture("sprites/speed_prize.png");
        }
        catch (Exception ex)
        {
            MessageBox.Show($"Ошибка загрузки текстуры: {ex.Message}");
            TextureId = -1; // Используйте значение по умолчанию
        }
    }

    /// <summary>
    /// Применяет эффект ускорения к машине.
    /// </summary>
    /// <param name="car">Машина, к которой применяется эффект.</param>
    public void ApplyEffect(Car car)
    {
        car.ApplyDecorator(new SpeedBoostDecorator(car, 1.5f, 5f));
    }
}

```

Код программы *CollisionMask.cs*:

```

using OpenTK;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RingRaceLab
{
    /// <summary>
    /// Представляет маску коллизий, основанную на изображении.
    /// </summary>
    public class CollisionMask
    {
        /// <summary>
        /// Изображение коллизионной маски.

```



```

/// </summary>
private readonly Bitmap _collisionBitmap;

/// <summary>
/// Ширина маски.
/// </summary>
public int Width;

/// <summary>
/// Высота маски.
/// </summary>
public int Height;

/// <summary>
/// Массив булевых значений, указывающий, проходим ли каждый пиксель.
/// </summary>
private bool[,] _drivablePixels;

/// <summary>
/// Инициализирует новую маску коллизий из файла изображения.
/// </summary>
/// <param name="path">Путь к файлу изображения коллизионной маски.</param>
/// <exception cref="Exception">Выбрасывается, если файл не найден.</exception>
public CollisionMask(string path)
{
    if (!System.IO.File.Exists(path))
        throw new Exception("Файл коллизионной карты не найден: " + path);
    _collisionBitmap = new Bitmap(path);
    Width = _collisionBitmap.Width;
    Height = _collisionBitmap.Height;
    _drivablePixels = new bool[Width, Height];
    for (int x = 0; x < Width; x++)
    {
        for (int y = 0; y < Height; y++)
        {
            Color pixel = _collisionBitmap.GetPixel(x, y);
            // Считаем пиксель проходимым, если он темный (например, R, G, B < 100)
            _drivablePixels[x, y] = pixel.R < 100 && pixel.G < 100 && pixel.B < 100;
        }
    }
}

/// <summary>
/// Проверяет, сталкивается ли автомобиль с непроходимой областью маски.
/// </summary>
/// <param name="car">Автомобиль для проверки.</param>
/// <returns>True, если обнаружена коллизия, иначе false.</returns>
public bool CheckCollision(Car car)
{
    List<Vector2> corners = car.GetCorners();
    foreach (var corner in corners)
    {
        int x = (int)corner.X;
        int y = (int)corner.Y;
        if (!IsDrivable(x, y))
            return true;
    }
    return false;
}

/// <summary>
/// Проверяет, является ли пиксель с заданными координатами проходимым.
/// </summary>

```

```

    /// <param name="x">Координата X пикселя.</param>
    /// <param name="y">Координата Y пикселя.</param>
    /// <returns>True, если пиксель проходим и находится в пределах маски, иначе false.</returns>
    public virtual bool IsDrivable(int x, int y)
    {
        if (x < 0 || y < 0 || x >= Width || y >= Height)
            return false;
        return _drivablePixels[x, y];
    }
}

```

Код программы *FinishLine.cs*:

```

using OpenTK;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace RingRaceLab
{
    /// <summary>
    /// Представляет финишную линию на трассе.
    /// </summary>
    public class FinishLine
    {
        private Vector2 _startPoint;
        private Vector2 _endPoint;

        /// <summary>
        /// Отслеживает пересечение линии.
        /// </summary>
        private bool _hasCrossed; // Для отслеживания пересечения

        /// <summary>
        /// Инициализирует финишную линию.
        /// </summary>
        /// <param name="start">Начальная точка линии.</param>
        /// <param name="end">Конечная точка линии.</param>
        public FinishLine(Vector2 start, Vector2 end)
        {
            _startPoint = start;
            _endPoint = end;
        }

        /// <summary>
        /// Проверяет, пересек ли объект финишную линию.
        /// </summary>
        /// <param name="previousPosition">Предыдущая позиция объекта.</param>
        /// <param name="currentPosition">Текущая позиция объекта.</param>
        /// <returns>1, если пересечено в прямом направлении; -1, если в обратном; 0, если не пересечено.</returns>
        public int CheckCrossing(Vector2 previousPosition, Vector2 currentPosition)
        {
            // Проверяем пересечение линии с помощью алгоритма пересечения отрезков
            if (LineIntersection.CheckLineCrossing( // Предполагается, что LineIntersection доступен
                previousPosition,
                currentPosition,
                _startPoint,
                _endPoint
            )
        }
    }
}

```

```

        ))
    {
        // Проверяем направление пересечения (например, по оси X)
        if (previousPosition.X < currentPosition.X) // Пример проверки направления
        {
            return 1; // Пересечено в прямом направлении
        }
        else
        {
            return -1; // Пересечено в обратном направлении
        }
    }
    return 0; // Не пересечено
}
}
}

```

Код программы *LineIntersection.cs*:

```

using OpenTK;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RingRaceLab
{
    public static class LineIntersection
    {
        public static bool CheckLineCrossing(Vector2 a1, Vector2 a2, Vector2 b1, Vector2 b2)
        {
            float denominator = (a1.X - a2.X) * (b1.Y - b2.Y) - (a1.Y - a2.Y) * (b1.X - b2.X);
            if (denominator == 0) return false;

            float t = ((a1.X - b1.X) * (b1.Y - b2.Y) - (a1.Y - b1.Y) * (b1.X - b2.X)) / denominator;
            float u = -((a1.X - a2.X) * (a1.Y - b1.Y) - (a1.Y - a2.Y) * (a1.X - b1.X)) / denominator;

            return t >= 0 && t <= 1 && u >= 0 && u <= 1;
        }
    }
}

```

Код программы *Track.cs*:

```

using OpenTK;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using OpenTK.Graphics.OpenGL;
using RingRaceLab;

/// <summary>
/// Представляет игровой трек.
/// </summary>

```

```

public class Track : GameEntity // Убедитесь, что RingRaceLab.GameEntity доступен
{
    /// <summary>
    /// Финишная линия трека.
    /// </summary>
    public FinishLine FinishLine { get; private set; }

    /// <summary>
    /// ID текстуры трека.
    /// </summary>
    private int textureId;

    /// <summary>
    /// Стартовые позиции на треке.
    /// </summary>
    public Vector2[] SpawnPositions { get; private set; }

    /// <summary>
    /// Ширина трека.
    /// </summary>
    private readonly int Width;

    /// <summary>
    /// Высота трека.
    /// </summary>
    private readonly int Height;

    /// <summary>
    /// Инициализирует новый трек.
    /// </summary>
    /// <param name="texturePath">Путь к текстуре трека.</param>
    /// <param name="spawnPositions">Стартовые позиции.</param>
    /// <param name="finishStart">Начало финишной линии.</param>
    /// <param name="finishEnd">Конец финишной линии.</param>
    /// <param name="Width">Ширина трека.</param>
    /// <param name="Height">Высота трека.</param>
    /// <exception cref="ArgumentException">Выбрасывается, если стартовые позиции не заданы.</exception>
    public Track(string texturePath, Vector2[] spawnPositions, Vector2 finishStart, Vector2 finishEnd, int Width, int
Height)
    {
        if (spawnPositions == null || spawnPositions.Length == 0)
            throw new ArgumentException("Необходимо задать хотя бы одну стартовую позицию.",
nameof(spawnPositions));

        SpawnPositions = spawnPositions;
        this.Width = Width;
        this.Height = Height;
        // Предполагается, что TextureLoader доступен
        textureId = TextureLoader.LoadFromFile(texturePath);
        // Предполагается, что FinishLine доступен
        FinishLine = new FinishLine(finishStart, finishEnd);
    }

    /// <summary>
    /// Отрисовывает трек.
    /// </summary>
    public override void Draw()
    {
        GL.Enable(EnableCap.Texture2D);
        GL.BindTexture(TextureTarget.Texture2D, textureId);

        GL.Begin(PrimitiveType.Quads);
        // Отрисовка текстурированного квадрата, растянутого на весь трек

```

```

        GL.TexCoord2(0, 0); GL.Vertex2(0, 0);
        GL.TexCoord2(1, 0); GL.Vertex2(Width, 0);
        GL.TexCoord2(1, 1); GL.Vertex2(Width, Height);
        GL.TexCoord2(0, 1); GL.Vertex2(0, Height);
        GL.End();

        GL.Disable(EnableCap.Texture2D);
    }
}

```

Код программы *CarMovementTests.cs*:

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenTK;
using RingRaceLab;

namespace RingRaceTestProject
{
    [TestClass]
    public class CarMovementTests
    {
        private CarConfig _config;
        private CarMovement _movement;

        [TestInitialize]
        public void Setup()
        {
            _config = new CarConfig();
            _movement = new CarMovement(Vector2.Zero, _config);
        }

        [TestMethod]
        public void Update_ForwardAcceleration_IncreasesSpeed()
        {
            // Arrange
            float deltaTime = 0.1f;
            float initialSpeed = _movement.CurrentSpeed;

            // Act
            _movement.Update(deltaTime, true, false, false, false);

            // Assert
            Assert.IsTrue(_movement.CurrentSpeed > initialSpeed, "Скорость должна увеличиться при ускорении вперед.");
            Assert.IsTrue(_movement.CurrentSpeed <= _config.ForwardMaxSpeed, "Скорость не должна превышать максимальную.");
        }

        [TestMethod]
        public void Update_Deceleration_ReducesSpeed()
        {
            // Arrange
            _movement.CurrentSpeed = 100f;
            float deltaTime = 0.1f;

            // Act
            _movement.Update(deltaTime, false, false, false, false);

            // Assert
            Assert.IsTrue(_movement.CurrentSpeed < 100f, "Скорость должна уменьшиться при торможении.");
            Assert.IsTrue(_movement.CurrentSpeed >= 0, "Скорость не должна стать отрицательной.");
        }
    }
}

```

```

[TestMethod]
public void Update_TurnLeft_ChangesAngle()
{
    // Arrange
    _movement.CurrentSpeed = 100f;
    float deltaTime = 0.1f;
    float initialAngle = _movement.Angle;

    // Act
    _movement.Update(deltaTime, false, false, true, false);

    // Assert
    Assert.IsTrue(_movement.Angle < initialAngle, "Угол должен уменьшиться при повороте налево.");
}
}
}

```

Код программы *CarPhysicsTests.cs*:

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenTK;
using RingRaceLab;
using System.Collections.Generic;

namespace RingRaceTestProject
{
    [TestClass]
    public class CarPhysicsTests
    {
        [TestMethod]
        public void GetCorners_ReturnsCorrectCorners()
        {
            // Arrange
            Vector2 size = new Vector2(32f, 16f);
            CarPhysics physics = new CarPhysics(size);
            Vector2 position = Vector2.Zero;
            float angle = 90f; // Поворот на 90 градусов

            // Act
            List<Vector2> corners = physics.GetCorners(position, angle);

            // Assert
            Assert.AreEqual(4, corners.Count, "Должно быть 4 угла.");
            // Проверяем примерные координаты с учетом поворота на 90 градусов
            Assert.IsTrue(Vector2.Distance(corners[0], new Vector2(16f, -32f)) < 0.1f, "Неверные координаты угла 0.");
            Assert.IsTrue(Vector2.Distance(corners[1], new Vector2(16f, 32f)) < 0.1f, "Неверные координаты угла 1.");
        }
    }
}

```

Код программы *FinishLineTests.cs*:

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenTK;
using RingRaceLab;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

```

```

namespace RingRaceTestProject
{
    [TestClass]
    public class FinishLineTests
    {
        [TestMethod]
        public void CheckCrossing_CrossesLine_ReturnsTrue()
        {
            // Arrange
            FinishLine finish = new FinishLine(new Vector2(0, 0), new Vector2(0, 100));
            Vector2 previousPos = new Vector2(-10, 50);
            Vector2 currentPos = new Vector2(10, 50);
            bool isCrossed = false;
            // Act
            int crossed = finish.CheckCrossing(previousPos, currentPos);
            if (crossed == 1 || crossed == -1) isCrossed = true;
            // Assert
            Assert.IsTrue(isCrossed, "Пересечение финишной линии должно быть обнаружено.");
        }

        [TestMethod]
        public void CheckCrossing_NoCrossing_ReturnsFalse()
        {
            // Arrange
            FinishLine finish = new FinishLine(new Vector2(0, 0), new Vector2(0, 100));
            Vector2 previousPos = new Vector2(10, 150);
            Vector2 currentPos = new Vector2(20, 150);
            bool isCrossed = true;
            // Act
            int crossed = finish.CheckCrossing(previousPos, currentPos);
            if (crossed == 0) isCrossed = false;

            // Assert
            Assert.IsFalse(isCrossed, "Пересечение не должно быть обнаружено вне линии.");
        }
    }
}

```

ПРИЛОЖЕНИЕ Б

(обязательное)

Руководство пользователя

1. Введение.

Настоящее руководство пользователя содержит информацию, необходимую для успешной эксплуатации игрового приложения «Кольцевые гонки». Приложение разработано для работы на операционных системах семейства *Windows* и представляет собой двухмерную аркадную гонку для двух игроков, соревнующихся на одном экране. Игра обладает простым и интуитивно понятным интерфейсом, рассчитанным на быстрое освоение пользователями.

Основные функциональные возможности приложения включают:

- выбор одной из нескольких доступных гоночных трасс;
- выбор внешнего вида (текстуры) для каждого из двух управляемых автомобилей;
- симуляцию движения автомобилей с учетом физических параметров;
- механику расхода и пополнения топлива во время гонки;
- систему бонусных призов на трассе, влияющих на характеристики автомобилей;
- обнаружение столкновений с границами трассы;
- учет пройденных кругов и определение победителя гонки;
- отображение актуальной игровой информации (*HUD*).

Для комфортного использования программного приложения пользователю рекомендуется ознакомиться с данным руководством, а также общими правилами работы с персональным компьютером и клавиатурой.

2. Назначение и условия применения.

Игровое приложение "Кольцевые гонки" предназначено для одновременной игры двух пользователей на одном персональном компьютере. Целью игры является развлечение и организация совместного досуга. Динамичный игровой процесс способствует развитию внимания, скорости реакции, а также тактического мышления в условиях ограниченных ресурсов (топливо) и постоянно меняющейся обстановки на трассе (бонусы, положение соперника).

Для корректной работы приложения необходимо соблюдение следующих минимальных технических требований к аппаратному и программному обеспечению:

- персональный компьютер под управлением операционной системы *Windows 7* или более новой версии;
- центральный процессор с тактовой частотой не ниже 2 ГГц;
- объем оперативной памяти не менее 2 ГБ;
- наличие графического адаптера с поддержкой *OpenGL 2.0* или выше и актуальными драйверами;

- наличие клавиатуры и цветного монитора с разрешением экрана не менее 1024x768 пикселей (рекомендуется *Full HD* 1920x1080 для наилучшего отображения);

- около 50 МБ свободного места на жестком диске для установки приложения и хранения ресурсов.

3. Подготовка к работе.

Для запуска игрового приложения достаточно открыть исполняемый файл *RingRaceApp.exe*. Убедитесь, что на вашем компьютере установлены последние версии драйверов для вашей видеокарты, поскольку приложение активно использует возможности аппаратного ускорения графики через библиотеку *OpenGL*.

При успешном запуске на экране появится главное меню приложения. Если приложение запускается без каких-либо сообщений об ошибках и отображает элементы интерфейса, значит оно готово к работе.

4. Описание операций.

При первом запуске приложения открывается главное меню. В центральной части меню отображается предварительный просмотр выбранной трассы, а по бокам – области выбора автомобилей для первого и второго игроков.

В главном меню:

- выбор трассы: используйте кнопки со стрелками рядом с изображением трассы для переключения между доступными вариантами трасс;

- выбор автомобилей: в зонах игроков используйте кнопки со стрелками рядом с изображением автомобиля для выбора желаемой текстуры для машины первого (слева) и второго (справа) игрока;

- начать гонку: после выбора трассы и автомобилей нажмите центральную кнопку для перехода к игровому процессу;

- выход из приложения: для завершения работы приложения нажмите кнопку закрытия окна в правом верхнем углу или клавишу *ESC* во время игры.

Во время гонки управление автомобилями осуществляется следующим образом:

а) игрок 1 (синий автомобиль):

- 1) вперед: Клавиша *W*;
- 2) назад: Клавиша *S*;
- 3) поворот налево: Клавиша *A*;
- 4) поворот направо: Клавиша *D*;

б) игрок 2 (красный автомобиль):

- 1) вперед: Клавиша Стрелка вверх;
- 2) назад: Клавиша Стрелка вниз;
- 3) поворот налево: Клавиша Стрелка влево;
- 4) поворот направо: Клавиша Стрелка вправо.

Основной игровой процесс сводится к следующему:

- гонка: двигайтесь по трассе, стараясь не выезжать за ее пределы, чтобы избежать замедления и потери контроля.

- топливо: следите за уровнем топлива. Он отображается на *HUD* для каждого игрока. При нулевом уровне топлива автомобиль не может двигаться;

– призы: на трассе появляются бонусные призы. При наезде на приз он подбирается. Существуют призы, пополняющие топливо, а также призы, временно увеличивающие (ускорение) или уменьшающие (замедление) максимальную скорость подобравшего автомобиля. Индикаторы активных временных эффектов также отображаются на *HUD*;

– круги: цель игры – первым завершить 5 полных кругов. Счетчик кругов не отображается на *HUD*, но финиш фиксируется приложением;

– финиш: когда один из игроков завершает 5 кругов, гонка заканчивается, и на экране отображается сообщение о победителе. С экрана финиша можно вернуться в главное меню.

5. Аварийные ситуации

Для минимизации возникновения аварийных ситуаций убедитесь, что ваш компьютер соответствует минимальным системным требованиям, установлены актуальные драйверы видеокарты, и вы запускаете приложение корректным образом.

В случае непредвиденного зависания или некорректной работы приложения, рекомендуется принудительно завершить процесс через Диспетчер задач (обычно вызывается комбинацией клавиш *Ctrl + Shift + Esc* или *Ctrl + Alt + Delete*) и попробовать запустить приложение снова.

6. Рекомендации по освоению

Перед началом первой гонки полезно потренироваться в управлении автомобилем, ознакомившись с его поведением на разных скоростях и углах поворота. Понимание влияния призов и тактическое их использование может стать решающим фактором для победы, поэтому старайтесь запоминать их расположение и эффект.

ПРИЛОЖЕНИЕ В

(обязательное)

Руководство программиста

1. Назначение и условия применения программы.

Данный программный комплекс является исходным кодом игры «Кольцевые гонки», предназначенным для изучения и модификации разработчиками. Его назначение – демонстрация принципов создания 2D аркадных игр на C# с использованием аппаратного ускорения графики через *OpenTK* (интерфейс к *OpenGL*). Программа симулирует гонки, обрабатывает ввод и отображает игровую информацию. Для работы с проектом необходима интегрированная среда разработки, поддерживающая .NET (например, *Visual Studio*), и актуальные драйверы видеокарты с поддержкой *OpenGL*.

2. Характеристики программы.

Программа реализована на C# с использованием объектно-ориентированного подхода. Архитектура разделяет логику UI (*Windows Forms*) и игрового ядра, применяя шаблоны вроде «Фабричного метода» для призов и «Декоратора» для эффектов на автомобилях. Графическая подсистема основана на *OpenTK/OpenGL* для аппаратного рендеринга 2D-графики. Проект организован как решение *Visual Studio* с разделением на функциональные проекты, зависимости управляются *NuGet*. Режимы работы – разработка в IDE и исполнение игры. Явных механизмов самовосстановления нет, ошибки обрабатываются стандартными средствами .NET.

3. Обращение к программе.

«Обращение к программе» в контексте разработки означает работу с ее исходным кодом в IDE. Для этого откройте файл решения (.sln) в *Visual Studio*. Может потребоваться восстановление пакетов *NuGet* для загрузки зависимостей вроде *OpenTK*. Взаимодействие включает просмотр/изменение кода (.cs), настройку проекта, сборку (*Build Solution*) и запуск приложения для отладки (*Start Debugging*) или тестирования (*Start Without Debugging*). Исполняемый файл (.exe) создается в выходной директории проекта после успешной сборки.

4. Входные и выходные данные.

Основными входными данными в процессе выполнения игры являются пользовательский ввод (состояние клавиатуры, обрабатываемое для управления автомобилями), файлы ресурсов (текстуры .png для объектов и карты коллизий) и конфигурационные данные (параметры трасс и автомобилей из статических классов или объектов конфигурации). Выходные данные включают визуальное представление игрового мира, отрисовываемое через *GLControl* с использованием *OpenGL*, обновленное состояние игровых объектов (позиции, скорости и др. параметры) после обработки логики, а также внутренние события и текстовые сообщения для пользователя (например, о победителе).

5. Сообщения.

В ходе выполнения программы пользователю выводятся информационные сообщения. Во время игры для каждого игрока отображается текущее

количество собранных бонусов. По окончании гонки программа выводит сообщение, указывающее на победившего игрока. Эти сообщения служат для информирования пользователя о ходе и результате игры и не требуют от него каких-либо немедленных действий. Сообщения об ошибках для программиста могут выводиться средствами отладки при возникновении исключений.

ПРИЛОЖЕНИЕ Г

(обязательное)

Руководство системного программиста

1. Общие сведения о программе.

Игровое приложение «Кольцевые гонки» представляет собой автономное исполняемое приложение, разработанное для функционирования в пользовательской среде операционной системы *Windows*. Основное назначение данного программного средства – предоставление платформы для проведения локальных гоночных соревнований, рассчитанных на участие двух пользователей. Приложение реализует функции графического отображения игрового мира, обработки пользовательского ввода и применения игровой логики для симуляции гонки. Программа разработана с использованием технологий *Microsoft .NET* и взаимодействует с графической библиотекой *OpenTK*, которая служит управляемой оберткой для низкоуровневого графического *API OpenGL*. Требования к аппаратному обеспечению программы можно охарактеризовать как умеренные, что делает возможным ее запуск на большинстве современных персональных компьютеров, оснащенных стандартной графической подсистемой. Для корректного функционирования исполняемого модуля необходима операционная система семейства *Windows*, совместимая с требуемой версией *NET Framework* (поддержка начиная с *Windows 7*). Критически важным техническим условием для успешной инициализации и работы графической подсистемы является наличие в системе графического адаптера, поддерживающего *OpenGL* (рекомендуется версия 2.0 или выше), а также наличие корректно установленных и актуальных драйверов от производителя данного адаптера.

2. Структура программы.

Дистрибутив игрового приложения «Кольцевые гонки» организован как набор файлов, требуемых для его автономного запуска и исполнения. Ключевым элементом структуры является исполняемый файл приложения, имеющий расширение *.exe*. Он служит точкой входа для запуска программы. Программа активно использует набор динамически подключаемых библиотек, представленных файлами с расширением *.dll*. Среди них как стандартные компоненты *NET Framework*, так и сторонние библиотеки, в частности *OpenTK.dll*, обеспечивающая взаимодействие с графическим *API OpenGL*. Логика самого игрового приложения, включая ядро механики, может быть инкапсулирована в одном или нескольких *DLL*-файлах, например, *RingRaceLab.dll*. Помимо исполняемых и библиотечных файлов, дистрибутив включает файлы ресурсов – игровые активы. Это в основном текстуры изображений в формате *.png*, используемые для визуального представления всех игровых элементов: трасс, автомобилей, призов и элементов пользовательского интерфейса. Также могут присутствовать специализированные файлы данных, например, для описания карт коллизий, которые критичны для игровой логики. Файлы ресурсов, как правило, располагаются в определенных поддиректориях

относительно местоположения основного исполняемого файла. Структура программы на уровне исходного кода, включающая разделение логики на *UI*, игровое ядро и вспомогательные модули, подробно описывается в Руководстве программиста, тогда как для системного программиста актуальна именно файловая структура дистрибутива. Программа не имеет явных прямых связей с другими автономными программами на уровне их вызова или межпроцессного взаимодействия, функционируя как изолированное приложение.

3. Настройка программы.

Программное обеспечение «Кольцевые гонки» не предусматривает выполнения сложных или специфических процедур настройки на системном уровне или модификации внешних конфигурационных файлов для своего стандартного запуска и функционирования. Все основные параметры игры, касающиеся игрового процесса или внешнего вида игровых сущностей (например, выбор конкретной трассы для гонки или цветовой схемы автомобилей), определяются пользователем непосредственно через графический интерфейс, доступный в главном меню приложения, и сохраняются только в рамках текущей игровой сессии. Программа не использует традиционные внешние файлы конфигурации, такие как *.ini*, *.XML* или записи в реестре операционной системы *Windows*, для хранения своих основных настроек. Единственными требованиями на уровне операционной системы, которые можно отнести к настройке среды выполнения, являются стандартные процедуры установки или обновления *NET Framework* до версии, совместимой с приложением, если это необходимо, и обеспечение корректной установки актуальных драйверов для графического адаптера. Прочие аспекты настройки на условия конкретного применения, такие как выбор функций или адаптация к специфическому составу технических средств, не предусмотрены функционалом данного приложения.

4. Проверка программы.

Проверка работоспособности программы на системном уровне направлена на подтверждение готовности операционной среды и успешности запуска основного исполняемого модуля. Системному программисту для верификации следует выполнить ряд шагов. В первую очередь, необходимо убедиться в полном соответствии программно-аппаратной среды системным требованиям, а именно: наличие подходящей версии операционной системы *Windows*, установленного *NET Framework* требуемой версии и графического адаптера с поддержкой *OpenGL* и корректными драйверами. Далее следует проверить полноту дистрибутива, убедившись в наличии всех необходимых файлов – исполняемого файла *.exe*, сопутствующих *.dll* библиотек (включая *OpenTK* и *RingRaceLab*), а также файлов ресурсов (*.png* и других). Основным методом прогона для первичной проверки является попытка запуска исполняемого файла приложения. Успешным результатом на этом этапе считается запуск программы без возникновения критических ошибок, приводящих к ее аварийному завершению, и отображение главного меню приложения. Возможность выбрать опции в меню и инициировать старт гонки дополнительно подтверждает базовое функционирование графической подсистемы и основных компонентов

приложения. Более глубокая проверка игровой логики, обработки коллизий или корректности начисления бонусов относится к функциональному тестированию и выходит за рамки системной проверки.

5. Дополнительные возможности.

Данное программное обеспечение разработано как узкоспециализированное приложение, предназначенное исключительно для проведения локальных гонок между двумя игроками на одном компьютере. В силу своей специфики и назначения как учебного/демонстрационного проекта, оно не обладает расширенными функциями системного уровня, которые могли бы представлять интерес для системного программиста за рамками обеспечения базового запуска. В частности, приложение не поддерживает ведение детализированных системных логов, которые могли бы помочь в диагностике неигровых проблем. Отсутствуют функции сетевого взаимодействия для многопользовательской игры по сети. Программа не принимает параметров командной строки для изменения поведения или настроек при запуске, что исключает возможность автоматизированной настройки или интеграции в сложные сценарии развертывания. Приложение функционирует как максимально автономный модуль, минимизируя свое влияние на операционную систему и используя стандартные *API* для взаимодействия с графической подсистемой и устройствами ввода без реализации специфических системных служб или компонентов. Соответственно, разделы, описывающие выбор и использование таких дополнительных функциональных возможностей, в данном руководстве не приводятся по причине их отсутствия в программе.

6. Сообщения системному программисту.

В процессе развертывания и поддержания работоспособности игрового приложения «Кольцевые гонки» системному программисту следует учитывать несколько критических моментов. Наиболее частой причиной проблем, являются ошибки, связанные с графической подсистемой. При невозможности инициализировать *OpenGL* контекст или при ошибках рендеринга могут возникать исключения в коде, работающем с *OpenTK*. Стандартное сообщение об ошибке *NET Framework* в этом случае часто указывает на сбой при вызове нативной *DLL* или проблему с графическим устройством. Действие: проверьте наличие актуальных драйверов для видеокарты и поддержку *OpenGL* требуемой версии. Убедитесь, что *OpenTK.dll* и другие нативные библиотеки, необходимые для *OpenGL*, находятся в корректной директории рядом с исполняемым файлом. Другим аспектом является модульность ядра игровой логики, представленного, например, в *RingRaceLab.dll*. Хотя это и облегчает потенциальное повторное использование логики, текущий пользовательский интерфейс программы жестко привязан к *WindowsForms*. Возможные ошибки при взаимодействии *UI* и игрового ядра могут указывать на проблемы с синхронизацией потоков или передачей данных между ними, проявляясь как зависания или исключения при попытке доступа к элементам *UI* из игрового потока. Действие: в случае таких ошибок следует проверить логику взаимодействия *UI* потока и игрового потока на предмет корректного использования механизмов синхронизации *WindowsForms* для обращения к элементам управления.

ПРИЛОЖЕНИЕ Д (обязательное)

Внешний вид окон интерфейса программы

Игровое приложение "Кольцевые гонки" функционирует в рамках единого основного окна, которое изменяет свое содержимое в зависимости от текущего состояния программы (главное меню или игровой процесс). Окно реализовано на базе технологии *Windows Forms*, при этом отрисовка игрового мира осуществляется с использованием графической библиотеки *OpenGL* через элемент управления *OpenTK GLControl*, интегрированный в соответствующую панель интерфейса.

При запуске приложения пользователь видит главное меню, предназначенное для выбора параметров гонки. На этом экране представлены элементы управления для выбора игровой трассы и визуального оформления (текстур) для каждого из двух гоночных автомобилей, а также кнопка для старта игры.

На рисунке Д.1 представлен общий вид окна пользовательского интерфейса приложения при его запуске, демонстрирующий состояние главного меню с доступными опциями выбора.



Рисунок Д.1 – Главное меню приложения «Кольцевые гонки»

После выбора параметров и начала гонки, содержимое окна переключается на отображение непосредственно игрового процесса. На экране появляется выбранная трасса, автомобили игроков, расположенные на стартовых позициях, и бонусные призы. Также в игровом режиме отображается внутриигровой

интерфейс (*HUD*), предоставляющий актуальную информацию о состоянии автомобилей, такую как уровень топлива.

Вид окна пользовательского интерфейса в момент активной игровой сессии, показывающий гоночную трассу и игровые элементы, представлен на рисунке Д.2.

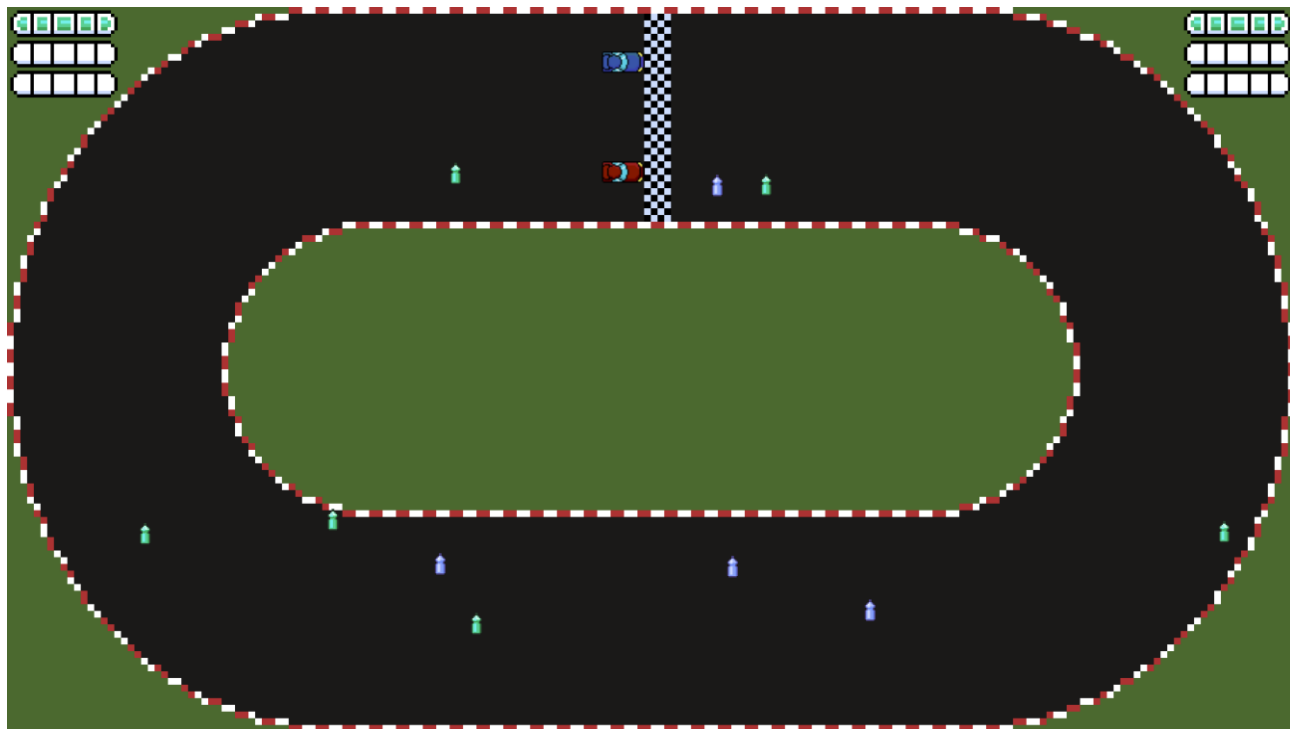


Рисунок Д.2 – Вид окна в момент игры в момент игры

ПРИЛОЖЕНИЕ Ж

(обязательное)

Результаты опытной эксплуатации

Игровое приложение «Кольцевые гонки» успешно функционирует при следующей минимальной конфигурации вычислительной техники, использованной в ходе опытной эксплуатации:

- процессор: класса *Intel Core 2 Duo* или аналогичный с тактовой частотой от 2.0 ГГц;
- оперативная память: не менее 2 ГБ;
- графический адаптер: с поддержкой *OpenGL 2.0* и выше, при наличии актуальных драйверов производителя;
- периферийные устройства: стандартная клавиатура и цветной монитор с разрешением не ниже 1024x768 пикселей;
- операционная система: семейства *Windows*, начиная с версии *Windows 7*.

Опытная эксплуатация приложения проводилась группой пользователей на нескольких персональных компьютерах с различным аппаратным обеспечением, соответствующим или превышающим минимальные требования. Длительность периода опытной эксплуатации составила суммарно около двух часов активного использования. В рамках тестирования проверялись такие ключевые сценарии, как запуск приложения, навигация по меню, выбор параметров гонки, полное прохождение нескольких игровых сессий от старта до финиша каждым из игроков, а также корректное завершение работы приложения.

По результатам проведенной опытной эксплуатации значимых ошибок, приводящих к нарушению логики игрового процесса или аварийному завершению программы, выявлено не было. Функциональность приложения в рамках заявленных возможностей подтверждена. Доработка или исправление критических дефектов по итогам опытной эксплуатации не потребовались.

ПРИЛОЖЕНИЕ И
(обязательное)

Схема использования паттерна «декоратор»