

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

## ВВЕДЕНИЕ

Современные компьютерные технологии предоставляют множество возможностей для создания интерактивных приложений, таких как игры, которые одновременно развлекают и способствуют развитию навыков программирования, алгоритмизации и работы с графикой. Разработка игровых приложений сохраняет свою актуальность благодаря востребованности в сфере образования и индустрии развлечений. Особое внимание привлекают игры, включающие элементы стратегии и взаимодействия между игроками, поскольку они помогают развивать логическое мышление и умение принимать решения в динамичных условиях.

Эта курсовая работа сосредоточена 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.

Созданное 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.

## 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».

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.

## 2.2 Структура классов приложения «Кольцевые гонки»

**2.2.1** Разрабатываемое приложение «Кольцевые гонки» представляет собой двухмерную гоночную игру для двух игроков, разработанную на платформе C# с использованием *Windows Forms* для создания пользовательского интерфейса и библиотеки *OpenTK* для рендеринга графики посредством *OpenGL*. 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 (*Controls*). Переключение между меню и игрой осуществляется методами *ShowMenu()* и *ShowGame()*, которые отвечают за скрытие одной панели и отображение другой. Форма также устанавливает *KeyPreview = true*, что позволяет ей перехватывать события клавиатуры до того, как они достигнут дочерних элементов управления, что важно для глобальных обработчиков, например, выхода из игры по нажатию *Escape*.

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*, находящийся в *RingRaceLab*, 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 (текстура и карта коллизий), стартовые позиции машин, координаты финишной линии, текстуры для автомобилей обоих игроков и размеры игрового окна. Основные обязанности *GameManager*:

- управление игровым циклом: хотя фактический вызов обновления и перерисовки инициируется событиями *GLControl* в *GameController*, основная логика каждого кадра выполняется в методе *GameManager.Update()*. Этот метод вычисляет время, прошедшее с предыдущего кадра (*deltaTime*), используя объект

*Stopwatch*, и на основе этого времени обновляет состояние всех игровых объектов;

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

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. Он определяет единственный абстрактный метод *Draw()*, который должны реализовать все его наследники (например, *Track* и *Car*);

- *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 (*SpawnPositions*), координаты для определения финишной линии (*FinishLine*) и размеры игрового поля. Он загружает текстуру с помощью *TextureLoader* и создает объект *FinishLine*. 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. Он загружает специальное изображение – карту коллизий (черно-белую версию трассы). На основе этой карты создается двумерный массив булевых значений (*\_drivablePixels*), где *true* соответствует проезжей части (пиксели, у которых все цветовые компоненты *RGB* меньше 100). 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. Это позволяет реализовать физику взаимодействия машин с окружением (например, отскок при столкновении со стеной);

- *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 (вперед или назад). Эта информация используется в *GameManager* для подсчета пройденных кругов.

Описанные классы задают основу приложения: структуру окон, переключение состояний, централизованное управление игровым процессом, базовые сущности и механизмы взаимодействия с игровым миром, такие как отрисовка трассы, определение столкновений и пересечение финишной линии.

**2.2.2** 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 (*Vector2*), путь к файлу текстуры, которая будет использоваться для отображения машины, и объект *CarConfig*, содержащий параметры физики и поведения. 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, изначально равный 100. 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 (*deltaTime*) и коэффициента расхода (*FuelConsumptionRate*). Уровень топлива уменьшается, но не может упасть ниже нуля;
- делегирование движения: если у машины еще есть топливо (*Fuel > 0*), 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 (например, ускорение), вызывается метод *Update* объекта *\_currentDecorator*.

Класс *Car* также предоставляет методы *ApplyDecorator* и *RemoveDecorator* для управления этими временными эффектами, о которых подробнее будет сказано в следующем разделе. Отрисовка машины (*Car.Draw*) полностью делегируется компоненту *\_renderer*, а расчет координат углов (*Car.GetCorners*) – компоненту *\_physics*.

Класс *CarConfig* (Приложение А, код программы *CarConfig.cs*) представляет собой простой контейнер данных, который хранит набор параметров, определяющих характеристики автомобиля. Сюда входят:

- физические параметры: ускорение при движении вперед и назад (*ForwardAcceleration*, *ReverseAcceleration*), максимальные скорости (*ForwardMaxSpeed*, *ReverseMaxSpeed*), скорость замедления при отсутствии тяги (*Deceleration*), базовая скорость поворота (*TurnSpeed*).
- размер: *Size* (тип *Vector2*) определяет половину ширины и высоты машины, используется для отрисовки и расчетов физики. В *GameManager* при создании машин эти размеры вычисляются пропорционально разрешению экрана.

– расход топлива: *FuelConsumptionRate* – коэффициент, влияющий на интенсивность расхода топлива.

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. Он хранит текущую позицию (*Position*), угол поворота (*Angle* в градусах) и текущую скорость (*CurrentSpeed*, положительная – вперед, отрицательная – назад). Класс также содержит ссылку на объект *CarConfig* для доступа к параметрам физики.

Основная работа происходит в методе *Update*. Скорость (*CurrentSpeed*) 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 (*ForwardMaxSpeed*, *ReverseMaxSpeed*) из конфигурации. Если игрок не нажимает ни "вперед", ни "назад", вызывается метод *ApplyDeceleration*, который плавно уменьшает абсолютное значение скорости к нулю, используя параметр *Deceleration* из *CarConfig*. Метод *UpdateRotation* отвечает за поворот машины. Машина может поворачивать, только если она движется ( $\text{Math.Abs}(\text{CurrentSpeed}) > 0.1f$ ). Скорость поворота (*effectiveTurnSpeed*) не постоянна, она зависит от текущей скорости машины относительно максимальной – чем выше скорость, тем медленнее поворот (используется *speedFactor*). Угол (*Angle*) изменяется на основе нажатых клавиш "влево"/"вправо" и времени кадра. Метод *UpdatePosition* отвечает за обновления позиции машины. На основе текущего угла (*Angle*) вычисляется единичный вектор направления движения. Позиция (*Position*) обновляется путем прибавления вектора направления, умноженного на *CurrentSpeed* и *deltaTime*.

Компонент *CarRenderer* (Приложение А, код программы *CarRenderer.cs*) отвечает исключительно за отрисовку спрайта машины в правильном месте и под нужным углом. В конструкторе он загружает текстуру по указанному пути с помощью *TextureLoader* и сохраняет ее OpenGL идентификатор (*\_textureId*), а также размеры машины. Метод *Draw* использует стандартные функции OpenGL: он перемещает систему координат в позицию машины (*GL.Translate*), поворачивает ее на нужный угол (*GL.Rotate*), активирует использование текстуры (*GL.Enable*, *GL.BindTexture*) и рисует прямоугольник (*PrimitiveType.Quads*), на который натягивается текстура машины. Использование *GL.PushMatrix* и *GL.PopMatrix* гарантирует, что трансформации (перенос, поворот), примененные для отрисовки этой машины, не повлияют на отрисовку других объектов сцены.

Класс *CarPhysics* (Приложение А, код программы *CarPhysics.cs*) предоставляет информацию о физических границах машины, необходимую для системы определения столкновений. Его основной метод *GetCorners* на основе текущей позиции, угла поворота и размеров машины (полученных из *CarConfig*)

вычисляет и возвращает список координат (*List<Vector2>*) четырех углов прямоугольника, описывающего машину в мировых координатах. Этот список используется классом *CollisionMask* для проверки столкновений с границами трассы.

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. Восполнить запас топлива можно с помощью специального бонуса *FuelPrize*. Уровень топлива постоянно отображается на экране с помощью *HUDRenderer*.

Подсчет кругов реализован в *GameManager*. После каждого обновления позиции машины (*Car.Update*), менеджер проверяет, пересекла ли машина финишную линию за этот кадр, используя метод *FinishLine.CheckCrossing()*. Этот метод возвращает 1 при пересечении вперед, -1 при пересечении назад и 0, если пересечения не было. В соответствии с результатом обновляется счетчик *Car.lapsComplete*. Когда счетчик достигает установленного значения (5 кругов), *GameManager* генерирует событие *OnCarFinished*, что приводит к завершению гонки.

Упомянутые классы формируют полное представление об автомобиле в игре: его настраиваемые параметры, сложную логику движения и поворотов, визуальное отображение, физическую модель для столкновений и взаимодействие с основными игровыми механиками, такими как топливо и подсчет кругов.

**2.2.3** 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. Он случайным образом выбирает координаты внутри игрового поля и проверяет их валидность с помощью метода *IsValidPosition*. Валидная позиция должна находиться на проезжей части трассы (проверяется через *\_collisionSystem.IsDrivable*) и не должна быть слишком близко к уже существующим бонусам. Если позиция подходит, случайным образом выбирается одна из доступных фабрик бонусов (*PrizeFactory*), и с ее помощью создается экземпляр конкретного бонуса, который добавляется в список активных бонусов (*\_activePrizes*);



– 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. В этом случае вызывается метод *ApplyEffect* подобранного бонуса (передавая ему объект *Car*, который его подобрал), и сам бонус удаляется из списка активных. Для потокобезопасности при доступе к списку *\_activePrizes* используется *lock*, так как список может изменяться и из другого потока (таймера респавна);

– 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*, чтобы добавить недостающее количество, стремясь к максимальному лимиту (*MAX\_PRIZES*). Это гарантирует, что на трассе всегда будет достаточно бонусов для игроков;

– 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. Он создает и управляет внутренним таймером (*System.Timers.Timer*), который отсчитывает время действия эффекта. Метод *Apply()* запускает таймер, записывает время начала действия эффекта (*timerStartTime*) 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*. Например, *SpeedBoostDecorator* в *ApplyEffect* запоминает текущую максимальную скорость машины и увеличивает ее на заданный множитель, а в *RevertEffect* восстанавливает исходное значение максимальной скорости. *SlowDownDecorator* действует аналогично, но уменьшает максимальную скорость.

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.

Такая система с использованием фабрик для создания бонусов и декораторов для временных эффектов обеспечивает гибкость и расширяемость. Добавление нового типа бонуса с временным эффектом потребует создания трех новых классов (конкретный бонус, его фабрика, конкретный декоратор) и минимальных изменений в инициализации *GameManager* (добавление новой фабрики в массив), не затрагивая при этом основную логику *PrizeManager* или *Car*.

**2.2.4** 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 (*Form1*) взаимодействует с логикой меню и игры через эти интерфейсы, что уменьшает связанность компонентов. Каждый контроллер управляет своей панелью (*System.Windows.Forms.Panel*) – *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* выступает в роли посредника между *Form1* и детальной логикой построения меню. Он реализует интерфейс *IMenuController*, управляет видимостью *MenuPanel* и предоставляет *Form1* доступ к выбору, сделанному пользователем (выбранная трасса, текстуры машин для первого и второго игрока) через свойства *SelectedTrack*, *Player1CarTexture*, *Player2CarTexture*. В конструкторе *MenuController* создается экземпляр *MenuBuilder*, которому делегируется фактическое создание элементов управления меню.

*MenuBuilder* – это класс, который конструирует все визуальные элементы меню внутри *MenuPanel*. Он использует компоненты *TableLayoutPanel* и *FlowLayoutPanel* для структурирования интерфейса на три основные колонки: зона выбора машины для первого игрока, центральная зона с выбором трассы и кнопкой старта, и зона выбора машины для второго игрока. Фоновое изображение задается для всей панели меню, а для предотвращения мерцания включается двойная буферизация.

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*.

Метод *StartGame* контроллера вызывается формой *Form1*. Он делает *GamePanel* видимой, устанавливает фокус на *GLControl* (чтобы он мог получать события клавиатуры), и, что самое важное, создает экземпляр *GameManager*, передавая ему всю необходимую информацию, выбранную в меню (трассу, машины). *GameController* также подписывается на событие *OnCarFinished*, чтобы узнать о завершении гонки.

Когда гонка завершена (сработало событие *OnCarFinished*), *GameController* уведомляет об этом объект *GameBuilder* (Приложение А, код программы *GameBuilder.cs*). *GameBuilder* отвечает за построение небольшого интерфейса, отображаемого поверх игрового экрана после финиша. Он создает панель *GameFinishedPanel*, которая содержит изображение победившего игрока и кнопку "Exit to Menu". Нажатие кнопки "Exit to Menu" возвращает пользователя в главное меню, скрывая *GameFinishedPanel* и *GamePanel*.

Во время гонки игрокам необходимо видеть актуальную информацию о состоянии их машины. За это отвечает *HUDRenderer* (Приложение А, код программы *HUDRenderer.cs*). Он отрисовывает так называемый *Heads-Up Display (HUD)* – элементы интерфейса, наложенные поверх основной игровой сцены. *GameManager* вызывает его метод *DrawHUD* в конце каждого цикла отрисовки.

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*. *InputManager*

предоставляет два простых метода: *GetPlayer1Input()* и *GetPlayer2Input()*. Каждый метод проверяет состояние специфических клавиш (*W*, *A*, *S*, *D* для первого игрока и стрелки для второго) с помощью *Keyboard.GetState()* из *OpenTK* и возвращает кортеж из четырех булевых значений, указывающих, какие из клавиш управления (вперед, назад, влево, вправо) нажаты в данный момент.

Описанные классы обеспечивают полное взаимодействие пользователя с игрой: от навигации в меню и выбора настроек до управления машиной во время гонки и получения визуальной обратной связи через *HUD* и экран завершения игры.

**2.2.5** Визуализация игры осуществляется с помощью графической библиотеки *OpenGL*. Взаимодействие с *OpenGL* из среды *.NET* (C#) происходит через библиотеку-обертку *OpenTK*. Она предоставляет доступ к функциям *OpenGL*, позволяя использовать аппаратное ускорение для 2D-графики.

Центральным элементом для рендеринга в игровом режиме является *GLControl*, расположенный на *GamePanel* в *GameController*. При инициализации этого элемента управления (*Load event*) настраиваются базовые параметры *OpenGL*, такие как цвет фона и режим смешивания цветов (*Blending*), необходимый для корректного отображения полупрозрачных текстур.

Для проецирования 2D-координат на экран используется ортографическая проекция (*GL.Ortho*), настраиваемая в методе *SetupViewport*. Эта проекция устанавливает систему координат, где левый верхний угол соответствует координатам (0, 0), а правый нижний – ширине и высоте *GLControl* в пикселях. Это значительно упрощает позиционирование объектов на экране.

Загрузка текстур (изображений для трассы, машин, бонусов, элементов *HUD*) выполняется с помощью статического класса *TextureLoader* (Приложение А, код программы *TextureLoader.cs*). Он использует *System.Drawing.Bitmap* для чтения данных из файлов изображений (преимущественно *PNG*), создает текстурные объекты *OpenGL* (*GL.GenTextures*, *GL.BindTexture*) и загружает в них пиксельные данные (*GL TexImage2D*). Также он настраивает параметры фильтрации текстур (*GL.TexParameter*), влияющие на качество изображения при масштабировании. Для оптимизации и избежания повторной загрузки одних и тех же текстур из файла используется *TextureCache* (Приложение А, код программы *TextureCache.cs*). Этот статический класс хранит словарь, сопоставляющий пути к файлам с уже загруженными идентификаторами текстур *OpenGL*. При запросе текстуры он сначала проверяет наличие ее в кэше и только если ее там нет, обращается к *TextureLoader*.

Отрисовка всех игровых спрайтов (машины, призы, элементы *HUD*, фон трассы) выполняется путем рисования текстурированных четырехугольников (*PrimitiveType.Quads*). Стандартный процесс включает активацию 2D-текстурирования, привязку нужной текстуры, определение вершин четырехугольника (*GL.Vertex2*) и соответствующих им текстурных координат (*GL.TexCoord2*) внутри блока *GL.Begin/GL.End*. Позиционирование и ориентация спрайтов достигается с помощью матричных преобразований *OpenGL* (*GL.Translate*, *GL.Rotate*), изолированных с помощью *GL.PushMatrix/GL.PopMatrix*. Структура *Vector2* из библиотеки *OpenTK* активно

используется во всем проекте для представления 2D-координат, векторов направления и размеров объектов.

Для обеспечения плавности рендеринга и устранения мерцания используются стандартные техники: двойная буферизация включена как для главной формы *Form1*, так и для панели меню (*MenuPanel*), а отрисовка в *GLControl* завершается вызовом *SwapBuffers*. Дополнительно, в *Form1* используется расширенный стиль окна *WS\_EX\_COMPOSITED*, который также способствует уменьшению мерцания.

Проект включает несколько полезных вспомогательных статических классов:

- *GameConstants* (Приложение А, код программы *GameConstants.cs*): содержит константные данные, специфичные для игровых трасс, такие как координаты точек старта (*TrackSpawnPositions*) и линии финиша (*TrackFinishPositions*). Данные хранятся в словарях, где ключом является путь к файлу текстуры трассы. Это упрощает добавление новых трасс и изменение существующих.

- *LineIntersection* (Приложение А, код программы *LineIntersection.cs*): предоставляет статический метод *CheckLineCrossing* для определения факта пересечения двух отрезков на плоскости. Этот метод используется классом *FinishLine* для точного определения момента пересечения машиной линии финиша.

Схема данных проекта приложения «Кольцевые гонки» представлена на рисунке 2.3



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*).