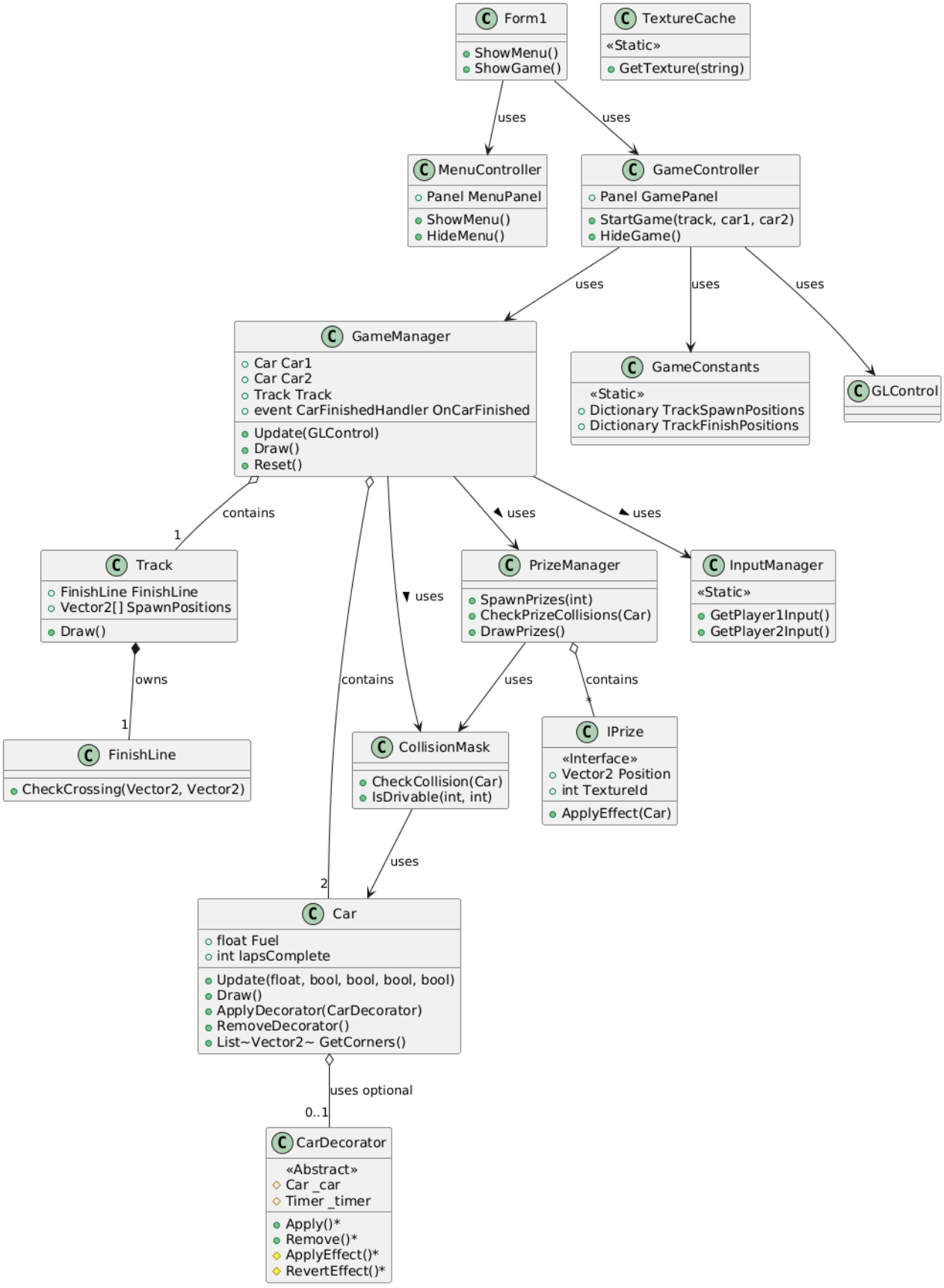

Рисунок 2.3 – Высокоуровневая архитектура приложения

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* обсчитывает 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. Машина может поворачивать, только если она движется (*Math.Abs*(*CurrentSpeed*) > 0.1*f*). 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.

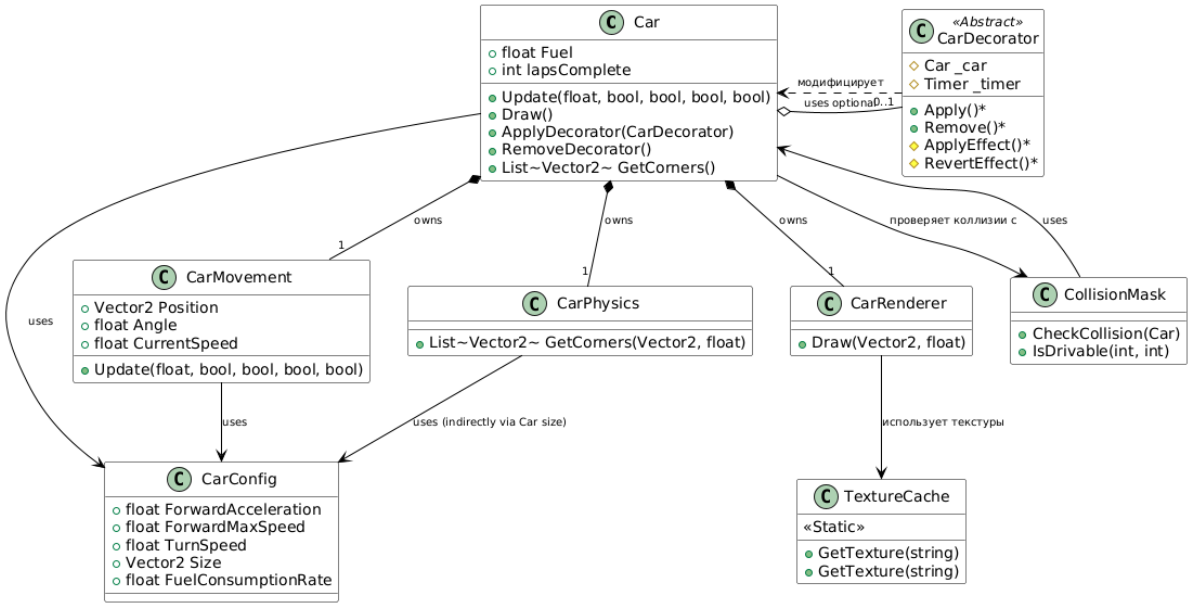

Рисунок 2.4 – Диаграмма класса *Car*, его компонентов и связей

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.

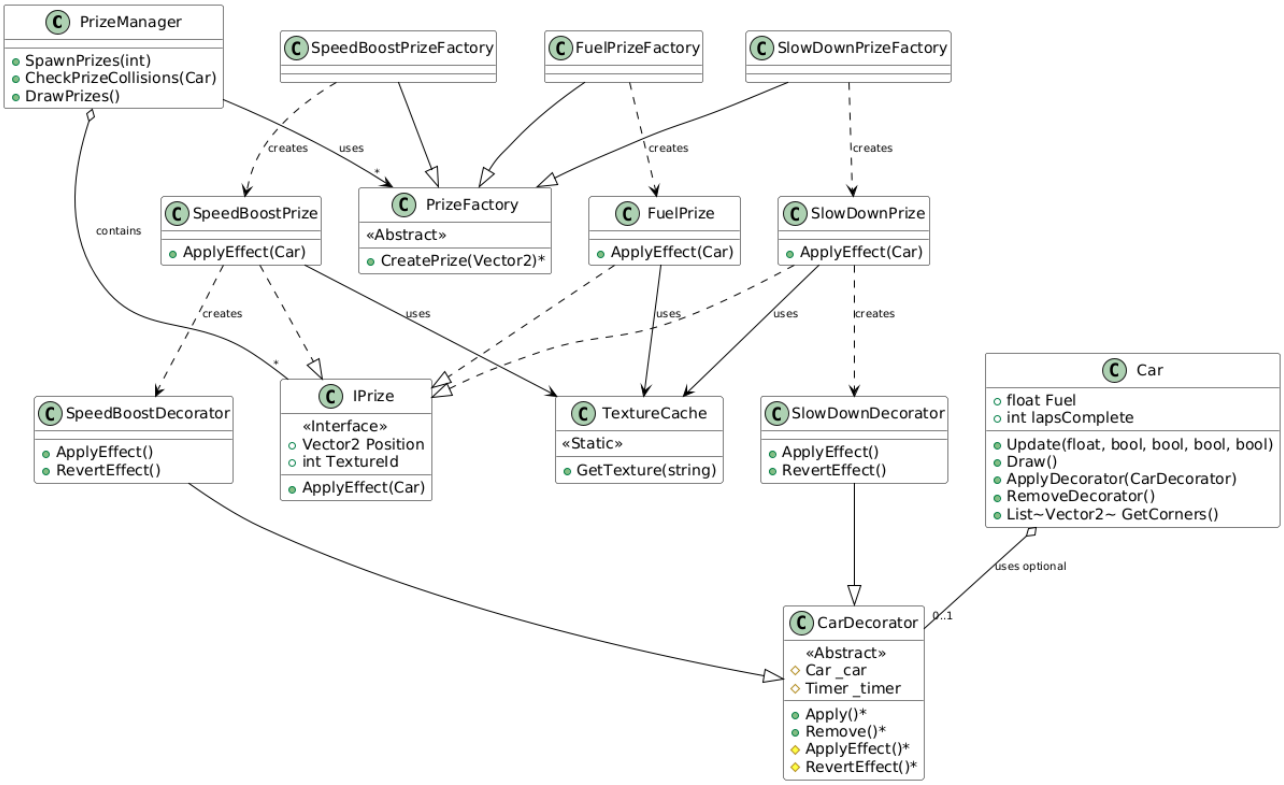

Рисунок 2.4 – Структура классов, реализующих систему бонусов

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. *Form*1 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 *Form*1 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.

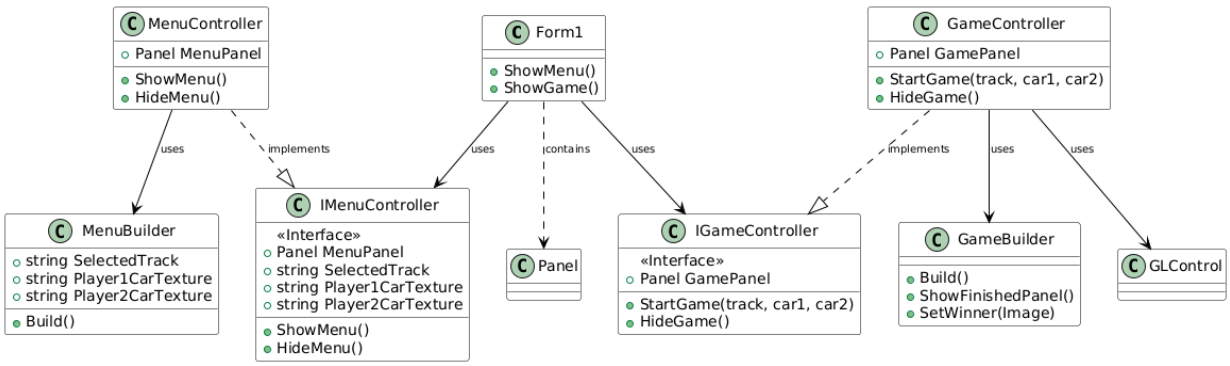

Рисунок 2.5 – Пользовательский интерфейс и управление переключением между экранами приложения

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 2*D*-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.

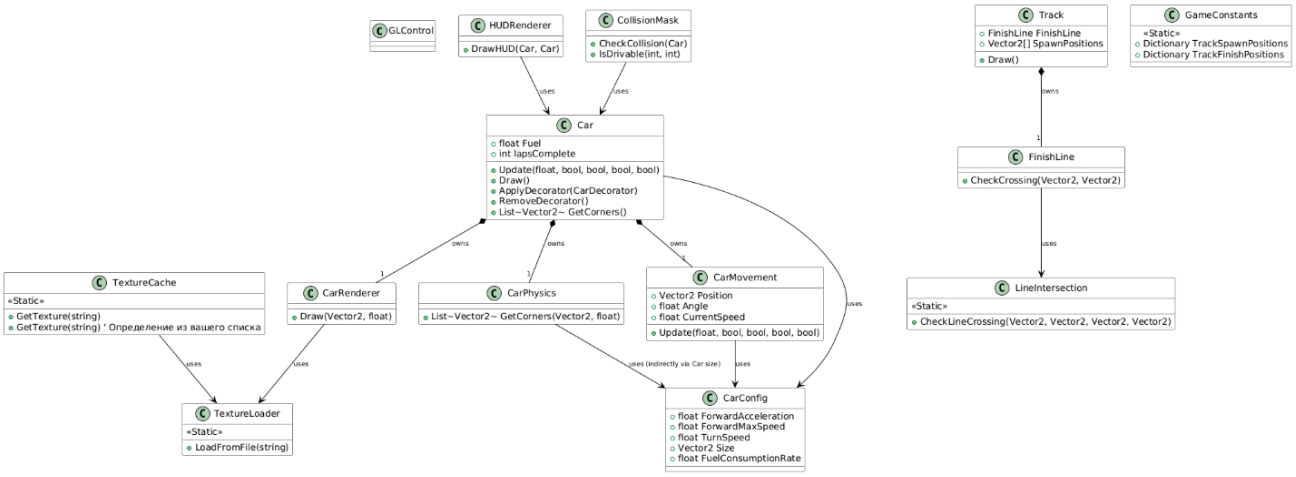

Рисунок 2.6 – Структура классов, отвечающих за графический вывод, обработку коллизий и набор общих вспомогательных утилит

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*).