

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 «Программирование сетевых приложений»

на тему: «Сетевое *RESTful* приложение, реализующее игру «Кольцевые  
гонки»»

Исполнитель: студент группы ИТИ-42  
Крук М.Д.

Руководитель: зав. кафедрой ИТ  
Курочка К.С.

Дата проверки: \_\_\_\_\_

Дата допуска к защите: \_\_\_\_\_

Дата защиты: \_\_\_\_\_

Оценка работы: \_\_\_\_\_

Подписи членов комиссии  
по защите курсового проекта: \_\_\_\_\_

Гомель 2022

## СОДЕРЖАНИЕ

Введение .....	4
1 Теоретические основы программных средств реализации игрового приложения .....	5
1.1 Технология <i>OpenGL</i> и ее особенности .....	5
1.2 Отличительные особенности технологии <i>OpenGL</i> в сравнении с .....	8
<i>DirectX</i> .....	8
1.3 Основы оболочки <i>OpenTK</i> .....	9
1.4 <i>RESTful</i> службы .....	10
1.5 Пример игрового приложения на тему «Кольцевые гонки» .....	13
2 Программная реализация приложения «Кольцевые гонки» .....	15
2.1 Отличительные особенности разработки игрового приложения.....	15
«Кольцевые гонки» .....	15
2.2 Архитектура игрового приложения .....	16
2.3 Структура игрового приложения «Кольцевые гонки».....	17
2.4 Структура сетевой части приложения «Кольцевые гонки» .....	21
3 Реализация игрового приложения .....	23
3.1 Принцип работы игрового приложения «Кольцевые гонки».....	23
3.2 Верификация игрового приложения «Кольцевые гонки» .....	25
3.3 Результаты тестирования игрового приложения.....	29
Заключение .....	30
Список использованных источников .....	31
Приложение А_Листинг программы «Кольцевые гонки».....	32
Приложение Б_Руководство пользователя.....	69
Приложение В_Руководство программиста.....	71
Приложение Г_Руководство системного программиста .....	72
Приложение Д_Схема паттерна «Декоратор» .....	73

## ВВЕДЕНИЕ

Игра является неперенным спутником развития человечества. Еще на ранних этапах формирования человечества игры выполняли важные функции. Они использовались для социализации подрастающего поколения, для подготовки к коллективной охоте, для тренировки.

В XX веке развернулась индустрия досуга, которая оккупировала все коммуникационные каналы и средства: газетно-журнальное и книжное дело, театр и кино, радиовещание и телевидение. Игровая индустрия не стала исключением. Все большее распространение получили примитивные игровые автоматы, компьютерные клубы и т.д.

Глобальная и повсеместная компьютеризация позволила сделать компьютерные игры массовым и легкодоступным способом проведения досуга. С развитием компьютерных и сетевых технологий развиваются и сами компьютерные игры: усложняется технический процесс производства игр, существенно увеличивается качество программного продукта, повышается максимальный порог вхождения в индустрию производства компьютерных игр. На сегодняшний день существует огромное количество разнообразных игр с прекрасным графическим и звуковым оформлением, максимально имитирующим реальную жизнь.

Повсеместное распространение данного вида досуга привело к созданию множества компаний, социализирующихся на производстве программных решений, связанных с индустрией компьютерных игр. Игровые студии тратят большое количество ресурсов на разработку игровых приложений, способных удовлетворить даже самого искушенного пользователя. Как правило, продукты и маркетинговая политика данных компаний настроены на платежеспособную аудиторию, что делает данный вид деятельности очень прибыльным, а специалистов данной отрасли – востребованными на рынке труда. Одно из ключевых задач специалистов в области разработки игровых решений – умение грамотно и качественно настроить сетевое взаимодействие компонентов. Данный навык может стать ключевым, при выборе специалиста данной отрасли.

Таким образом, представленные данные отражают актуальность выбранной темы курсового проектирования: разработка игрового приложения. Основные цели и задачи курсового проектирования: создание качественного игрового приложения с помощью высокоуровневой графической библиотеки и подготовка его сетевой составляющей. Предметная область, выбранная в качестве основной темы курсового проектирования, позволит получить качественные и полезные в будущем навыки разработки программного обеспечения.

# 1 ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ПРОГРАММНЫХ СРЕДСТВ РЕАЛИЗАЦИИ ИГРОВОГО ПРИЛОЖЕНИЯ

## 1.1 Технология *OpenGL* и ее особенности

Первая версия технологии появилась в 1992 году, будучи продуктом компании *Silicon Graphics (SGI)*, которая специализировалась на создании высокотехнологического графического оборудования и программных средств. В этом же году *SGI* стал главой *The OpenGL Architecture Review Board (OpenGL ARB)*, группы компаний, разрабатывающих спецификацию *OpenGL (Open Graphics Library)*.

Рассматриваемая технология произошла от 3D-интерфейса *SGI – IRIS GL*. Одним из ограничений предшественника было то, что он разрешал использовать только функции, поддерживаемые аппаратным обеспечением. Если функция не была реализована аппаратным способом, приложение не могло её использовать. *OpenGL* решает эту проблему путем программной реализации функций, не предоставляемых аппаратным обеспечением, что позволяет приложениям использовать этот интерфейс в системах с относительно низким энергопотреблением.

*OpenGL* в основном рассматривается как *Application Programming Interface (API* – описание способов взаимодействия одной компьютерной программы с другими приложениями), который предоставляет нам большой набор функций, которые мы можем использовать для управления графикой и изображениями. Однако *OpenGL* сам по себе не является *API*, а всего лишь спецификация, разработанная и поддерживаемая *Khronos Group*. На рисунке 1.1 представлено изображение логотипа технологии.

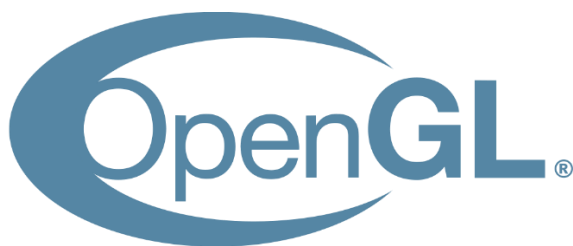


Рисунок 1.1 – Изображение логотипа технологии

**1.1.1** На базовом уровне, *OpenGL* – это просто спецификация, то есть документ, описывающий набор функций и их точное поведение. Спецификация *OpenGL* точно определяет, каким должен быть результат или же вывод каждой функции и как она должна выполняться. Затем разработчики программного продукта, реализующие данную спецификацию, должны придумать решение о том, как должна работать эта функция. Поскольку спецификация *OpenGL* не

дает нам подробностей реализации, разработанным версиям *OpenGL* разрешено иметь разные реализации, если их результаты соответствуют спецификации (и, следовательно, одинаковы для конечного пользователя).

Специалисты, разрабатывающие библиотеки *OpenGL*, обычно являются производителями видеокарт. Каждый видеоадаптер, который приобретается пользователем, поддерживает определенные версии *OpenGL*, являющиеся версиями, разработанными специально для данного устройства (серии). Эффективные реализации *OpenGL* существуют для *Windows*, *Unix*-платформ и *MacOS*. Эти реализации обычно предоставляются изготовителями видеоадаптеров и активно используют возможности последних. Существуют также открытые реализации спецификации *OpenGL*, одной из которых является библиотека *Mesa*. Из лицензионных соображений *Mesa* является «неофициальной» реализацией *OpenGL*, хотя полностью с ней совместима на уровне кода и поддерживает как программную эмуляцию, так и аппаратное ускорение при наличии соответствующих драйверов.

### 1.1.2 *OpenGL* фокусируется на следующих двух задачах:

- а) Скрыть сложности адаптации различных 3D-ускорителей, предоставляя разработчику единый API;
- б) Скрыть различия в возможностях аппаратных платформ, требуя реализации недостающей функциональности с помощью программной эмуляции. [1]

Основным принципом работы *OpenGL* является получение наборов векторных графических примитивов в виде точек, линий и треугольников с последующей математической обработкой полученных данных. Векторные трансформации и растеризация выполняются при использовании графического конвейера (*graphics pipeline*). Схематическое изображение этапов работы конвейера представлено на рисунке 1.2.

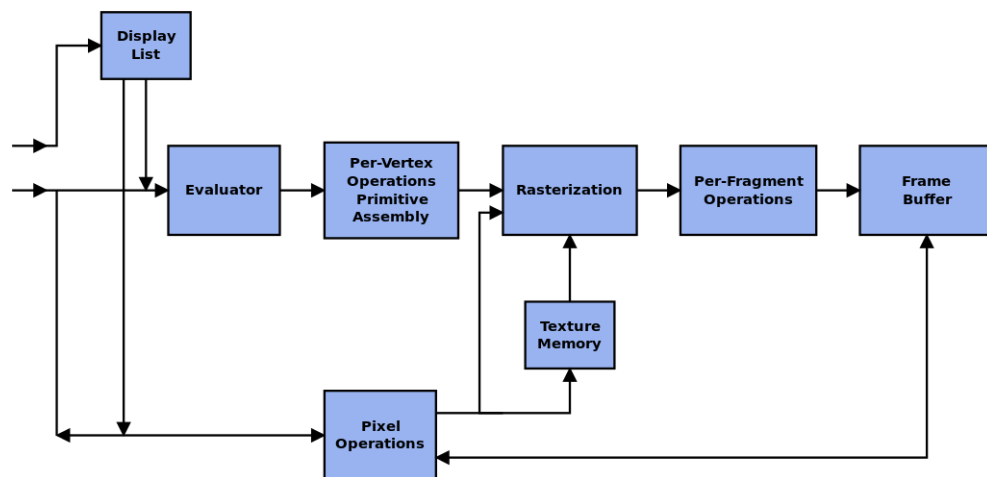


Рисунок 1.2 – Процесс графического конвейера

Абсолютное большинство команд *OpenGL* попадает в одну из двух групп: либо они добавляют графические примитивы на вход в конвейер, либо конфигурируют конвейер на различное исполнение трансформаций.

Особенности функционирования *OpenGL* требуют от разработчика графического приложения диктовать точную последовательность шагов, чтобы построить результирующую растровую графику (императивный подход). Это является основным отличием от дескрипторных подходов, когда вся сцена передается в виде структуры данных (чаще всего дерева), которое обрабатывается и строится на экране. С одной стороны, императивный подход требует от разработчика программного продукта глубокого знания законов трёхмерной графики и математических моделей, с другой стороны – даёт свободу внедрения различных инноваций.

**1.1.3** Отличительной особенностью *OpenGL* является поддержка им расширений. Всякий раз, когда графическая компания предлагает новую технику или новую масштабную оптимизацию для рендеринга, это часто встречается в расширении, реализованном в драйверах. Если аппаратное обеспечение, на котором работает приложение, поддерживает такое расширение, разработчик графического решения может использовать функциональность, предоставляемую расширением, для более продвинутых или эффективных графических возможностей.

Каждый производитель имеет аббревиатуру, которая используется при именовании его новых функций и констант. Например, компания *NVIDIA* имеет аббревиатуру *NV*, которая используется при именовании её новых функций, как, например, *glCombinerParameterfvNV*, констант: *GL\_NORMAL\_MAP\_NV*. Может случиться так, что определённое расширение могут реализовать несколько производителей. В этом случае используется аббревиатура *EXT*, например, *glDeleteRenderbuffersEXT*.

Часто, когда расширение популярно или очень полезно, оно в конечном итоге становится частью будущих версий *OpenGL*.

**1.1.4** Библиотеки *OpenGL* написаны на языке *C* и допускают множество производных на других языках, но по своей сути они остаются *C*-библиотекой. Поскольку многие языковые конструкции представленного языка не так хорошо переводятся на другие языки более высокого уровня, *OpenGL* был разработан с учетом нескольких абстракций. Одной из таких абстракций являются объекты в *OpenGL*.

Объект в *OpenGL* – это набор параметров, который представляет подмножество состояния *OpenGL*. Например, возможно создать объект, представляющий настройки окна рисования. В таком случае предполагается

возможность установить его размер, количество поддерживаемых цветов и другие настройки.

Особенность использования объектов заключается в том, что разработчик вправе определить более одного объекта в разрабатываемом графическом приложении, задать их параметры, и всякий раз, когда будет запускаться операция, использующая состояние *OpenGL*, объект привязывается к предпочтительным настройкам. Например, есть объекты, которые действуют как объекты-контейнеры для данных 3D-модели (дом или персонаж), и всякий раз, когда существует необходимость нарисовать один из них, разработчик привязывает объект, содержащий данные модели, которые необходимо отобразить (модели и параметры для этих объектов устанавливаются предварительно).

## 1.2 Отличительные особенности технологии *OpenGL* в сравнении с *DirectX*

Технологии *DirectX* и *OpenGL* являются конкурирующими интерфейсами. Они оба могут использоваться для визуализации 3D и 2D компьютерной графики. Отсюда можно выделить одно из основных преимуществ *OpenGL*: наличие множества различных расширений, которые позволяют пользователю работать не только с графикой, но и со звуком, текстом, вычислениями с использованием графических процессоров.

Представленные интерфейсы прикладного программирования имеют множество различий. Одним из них можно считать поддержку расширений. Так как *DirectX* является проприетарным продуктом, то и изменения в него вносятся исключительно самой компанией разработчиком – *Microsoft*. *OpenGL*, в свою очередь, является свободным API, распространяющийся под лицензией. *OpenGL* представляет собой открытый стандарт, однако некоторые его функции запатентованы. Правки в интерфейс и расширение *OpenGL* осуществляются всем сообществом, в которое входят и ключевые производители видеокарт: *AMD* и *NVIDIA*. В конечном итоге наиболее важные расширения становятся частью основной спецификации. Каждая новая версия *OpenGL* представляет из себя старую версию с добавлением некоторых новых расширений. В то же время новые функции по-прежнему доступны в виде расширений. [2]

Еще одно существенное различие двух технологий – кроссплатформенность. Проприетарный *DirectX* реализован официально лишь в семействе операционных систем *Windows*, включая те версии, которые используются в семействе игровых консолей *Xbox*.

*OpenGL* имеет реализации, доступные на многих платформах, включая *Microsoft Windows*, *Unix*-системы, такие как *MacOS*, *Linux*. *Nintendo* и *Sony* разработали свои собственные библиотеки, которые похожи, но не идентичны

*OpenGL*. *OpenGL* было выбрано в качестве основной графической библиотеки для *Android*, *BlackBerry*, *iOS* и *Symbian* в форме *OpenGL ES*.

*DirectX* основан на технологии *COM*, одним из преимуществ которой является то, что *API* можно использовать на любом *COM*-совместимом языке, в частности: *Object Pascal (Delphi)*, *C++*, *C#* и др. [3]

*OpenGL* – спецификация, реализованная на языке программирования *C*, которая может быть использована и на других языках. Как *API*, *OpenGL* не зависит ни от одной функции языка программирования и может быть вызван практически из любого языка. В *OpenGL* используется так называемая машина состояний (конечный автомат). Результат вызовов функций *OpenGL* зависит от внутреннего состояния, и может изменять его. В *OpenGL*, чтобы получить доступ к конкретному объекту (например, текстуре), нужно сначала выбрать его в качестве текущего функцией *glBindTexture*, а затем уже можно влиять на объект, например, задание содержимого текстуры осуществляется вызовом *glTexImage2D*.

На рынке профессиональной графики более востребованным считался *OpenGL*, нежели *DirectX*, в то время как набор интерфейсов прикладного программирования от *Microsoft* считался наиболее пригодным в основном для компьютерных игр. Однако, на данный момент как *OpenGL*, так и *DirectX* имеют достаточно большое перекрытие в функциональности, которое может быть использовано для большинства общих целей, причем операционная система часто является главным критерием, диктующим, что используется: *DirectX* является общим выбором для *Windows*, а *OpenGL* – почти для всего остального.

При рассмотрении игровой индустрии можно с уверенностью убедиться, что большинство производителей игровых приложений отдают свое предпочтение технологии *DirectX* благодаря гибкости и расширенному набору библиотек. Игр, где присутствует полноценная поддержка *OpenGL* не так уж и много. Производители игровых приставок оснащают свои продукты собственными *API* для максимизации производительности, что делает сравнение *OpenGL* и *DirectX* актуальным лишь для платформы ПК.

Таким образом, менее продвинутый и реже обновляемый *DirectX* является наиболее предпочтительным для тех, кто использует операционную систему *Windows*. Но если игры необходимо запускать на разных платформах с разными аппаратными требованиями, *OpenGL* является более предпочтительным решением.

### 1.3 Основы оболочки *OpenTK*

*OpenTK (The Open Toolkit Library)* – это библиотека, которая обеспечивает высокоскоростной доступ к *OpenGL*, *OpenCL* (спецификация



параллельных вычислений) и *OpenAL* (используется во многих играх для создания звуковых эффектов и музыки) для *.NET* приложений. *OpenTK* стремится сделать работу с *OpenGL*, *OpenCL* и *OpenAL* на *.NET*-поддерживаемых языках, таких как *C#*, похожей как по производительности, так и по ощущениям на эквивалентный код *C*, при этом все еще работая в управляемой среде.

Перед началом разработки следует учесть, что *OpenTK* не является библиотекой высокого уровня: это не игровой движок, или фреймворк, или полноценный рендерер, или полноценная аудиосистема сама по себе. Вместо этого это низкоуровневый фундамент, на котором разработчик игрового продукта имеет возможность строить системы подобного рода.

*OpenTK* также включает в себя удобную математическую библиотеку для распространенных типов графики, таких как векторы и матрицы, так что разработчику нет необходимости реализовывать их самостоятельно или искать другую библиотеку для их имплементации. Данные графические элементы разработаны так, чтобы быть очень быстрыми и гибкими, и они также включают в себя значительную функциональность "прямо из коробки".

Представленную оболочку можно использовать без фреймворка пользовательского интерфейса: она способна создавать окно и принимать входные данные напрямую, чего часто бывает достаточно для многих программ, таких как игры.

Отдельного внимания заслуживает тот факт, что для библиотеки *.NET OpenTK* работает очень быстро. *OpenTK 3* и *4* используют оптимизированную ручную сборку *IL* для минимизации накладных расходов при вызове функций *OpenGL*, а в *OpenTK 5* планируется использовать новые указатели на функции *C#* для достижения той же цели.

Эти и многие другие возможности позволяют отдать преимущество данной оболочке при выборе технологии доступа к *OpenGL* для *.NET* приложений.

## 1.4 *RESTful* службы

Передача репрезентативного состояния (*REST*) – это программная архитектура, которая определяет условия работы *API*. Изначально *REST* был создан как руководство по управлению взаимодействиями в сети Интернет. Архитектура, основанная на *REST*, может быть использована для поддержки высокопроизводительной и надежной связи в требуемом масштабе. *REST* может быть легко внедрен и модифицирован, обеспечивая прозрачность и кроссплатформенную переносимость любой системы *API*. [4]

Разработчики имеют возможность создавать *API*-интерфейсы, используя несколько архитектур. *API*, соответствующие архитектурному стилю *REST*,

называются *REST API*. Веб-службы, реализующие архитектуру *REST*, называются веб-службами *RESTful*. Как правило, термин *RESTful API* относится к сетевым *RESTful API*. Однако *REST API* и *RESTful API* являются взаимозаменяемыми терминами.

#### **1.4.1** Существует несколько принципов архитектурного стиля *RESTful*.

Единый интерфейс (*Uniform Interface*) – сервер передает информацию в стандартном формате. Ресурс, подвергшийся форматированию в *REST* называется представлением. Этот формат может отличаться от внутреннего представления ресурса в серверном приложении. Например, сервер может хранить данные в виде текста, но отправлять их в формате представления *HTML*.

Сервер не имеет сохраненного состояния (*Stateless*) – это метод связи, при котором сервер выполняет каждый клиентский запрос независимо от всех предыдущих запросов. Клиенты могут запрашивать ресурсы в любом порядке, и каждый запрос либо изолирован от других запросов, либо его состояние не сохраняется. Это конструктивное ограничение подразумевает, что сервер может полностью понимать и выполнять запрос каждый раз.

Отделение клиента от сервера (*Client-Server*). Клиент – это пользовательский интерфейс веб-сайта или приложения. В *REST API* код запроса остается на стороне клиента, а код для доступа к данным остается на стороне сервера. Это упрощает организацию *API*, облегчает миграцию пользовательского интерфейса на другую платформу и позволяет лучше масштабировать серверное хранилище данных.

Кэшируемость (*Cacheable*) означает, что в данных запроса должно быть указано, нужно ли кэшировать данные (хранить в специальном буфере для частых запросов). Если есть такое указание, клиент будет иметь право получить доступ к этому буферу при необходимости.

Понятие «многоуровневая система (*Layered System*)» свидетельствует о том, что серверы могут располагаться на разных уровнях, при этом каждый сервер взаимодействует только с ближайшими уровнями и не связан запросами с другими.

Предоставление кода по требованию (*Code on Demand*) означает, что серверы могут отправлять код клиенту (например, скрипт для запуска видео). Таким образом, общий код приложения или веб-сайта усложняется только при необходимости. [5]

#### **1.4.2** Важно отметить преимущества, которыми обладает служба, реализующая *RESTful* принципы.

Системы, реализующие *REST API*, могут эффективно масштабироваться за счет оптимизации взаимодействия между сервером и клиентом. Отсутствие

сохранения состояния снимает нагрузку с сервера: серверу не нужно сохранять информацию о предыдущих запросах клиента. Отложенное кэширование частично или полностью устраняет некоторые задержки на этапе взаимодействия между клиентом и сервером.

Веб-службы *RESTful* отличаются своей «гибкостью» и поддерживают полное разделение клиента и сервера. Они упрощают и разделяют различные серверные компоненты, так что каждая часть может развиваться независимо от клиентской. Изменения платформы или технологии в серверном приложении не влияют на клиентское приложение. Возможность разделения функций приложения на уровни еще больше повышает гибкость. Например, разработчики программного продукта могут вносить изменения в уровень базы данных, не беспокоясь о логике основного приложения.

*REST API* не зависит от стека используемых технологии. Разработчик способен создавать как клиентские, так и серверные приложения на разных языках программирования, не влияя на структуру *API*. Также возможно изменить базовую технологию, с любой стороны, не влияя на обмен данными.

**1.4.3** Основной принцип работы *RESTful API* построен на взаимодействии клиента с сервером. Клиент взаимодействует с сервером с помощью *API*, когда ему нужен ресурс.

Первым делом клиент отправляет запрос на сервер. Руководствуясь документацией *API*, клиент форматирует запрос таким образом, чтобы сервер его понял. Клиентский запрос содержит уникальный идентификатор ресурса, присваиваемый сервером. В случае служб *REST* сервер идентифицирует ресурсы с помощью универсального указателя ресурса (*URL*). *URL*-адрес указывает путь к ресурсу и четко указывает серверу, что нужно клиенту. Также в запрос включены и методы. Метод *HTTP* сообщает серверу, что ему нужно сделать с ресурсом. Наиболее распространенные методы это (*GET* – получение информации о данных или списка объектов, *PUT* – обновление данных, *POST* – добавление данных, *DELETE* – удаление данных). К информации, передаваемой с запросом, также относятся заголовки запросов. Заголовки запросов – это метаданные, которыми обмениваются клиент и сервер. Например, заголовок запроса определяет формат запроса и ответа, предоставляет информацию о статусе запроса и так далее.

На следующем этапе после отправки запроса происходит этап аутентификации. Сервер аутентифицирует клиента и подтверждает, что клиент имеет право сделать этот запрос. После чего сервер получает запрос и обрабатывает его.

После обработки запроса сервер возвращает ответ клиенту. Как правило ответ в первую очередь содержит строку состояния. Строка состояния содержит трехзначный код состояния, который указывает на успешное или

неуспешное выполнение запроса. Например, коды 2XX указывают на успешное выполнение, а коды 4XX и 5XX указывают на ошибки. Коды 3XX указывают на перенаправление URL-адреса. Помимо строки состояния в ответе также содержится текст ответа. Текст ответа содержит представление ресурса. Сервер выбирает подходящий формат представления на основе содержания заголовков запроса.

### 1.5 Пример игрового приложения на тему «Кольцевые гонки»

В качестве примера игрового приложения, реализующего 2D игру на тему «Кольцевые гонки» прекрасно подойдет, достаточно распространенная и знакомая каждому пользователю сети Интернет, игра *Ultimate Racing 2D* (рисунок 1.3).



Рисунок 1.3 – Игровое пространство игры «*Ultimate Racing 2D*»

Игровые приложения в формате 2D и жанре «Гонки» с видом «сверху» хоть и не могут похвастаться таким же уровнем глубины проработки геймплейных и графических деталей, как их 3D-аналоги, однако привлекают своей доступностью и легким порогом входа в динамичный игровой процесс. Они, как правило, придают аркадный оттенок тому, что в случае сложных гоночных симуляторов часто является сложным игровым процессом для новичков, позволяя практически любому пользователю, независимо от опыта, сразу погрузиться в игру. *Ultimate Racing 2D* преуспевает в этом отношении,

обладая упрощенным игровым процессом, с которым невероятно легко разобраться.

На игровом уровне (трассе) пользователь управляет одним из многих классов транспортных средств, включая очевидные варианты, такие как суперкары, картинги и мотоциклы, а также менее подходящие для гонок транспортные средства, такие как лимузины и вилочные погрузчики. Трасса является кольцевой дорогой, с различными поворотами, усложняющими процесс прохождения уровня. Выбор треков является довольно широким.

Сам игровой процесс невероятно прост, но этого нельзя сказать об искусственном интеллекте противника. Даже при сниженной сложности коллеги-гонщики быстры и, похоже, никогда не ошибаются. Если пользователю случится выехать на гравий или врезаться в барьер, не исключено, что остаток гонки он проведет в нижней части таблицы лидеров. Только используя свой импульс и срезая достаточное количество поворотов, игрок может надеяться снова подняться на вершину турнирной таблицы и выиграть гонку.

Игрок имеет возможность настроить сложность игры перед гонкой и изменить погодные условия локаций вместе с типом используемых шин.

По умолчанию камера игры находится довольно далеко, что дает пользователю хороший обзор трассы. Однако существует реальный риск потерять свой автомобиль среди множества похожих на него противников.

Таким образом, представленная в качестве примера игра *Ultimate Racing 2D* отражает основные механики и функции, которые возможно реализовать при проектировании игрового приложения на тему «Кольцевые гонки». Для сетевого взаимодействия интерфейсов удобным решением, согласно упомянутой выше информации, представляется применение принципов *RESTful* при имплементации веб-служб. Упомянуты выше инструменты отображения графических объектов (технология элементов *OpenGL* и библиотека *OpenTK*) возможно использовать для представления графической составляющей игрового приложения.

## 2 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ «КОЛЬЦЕВЫЕ ГОНКИ»

### 2.1 Отличительные особенности разработки игрового приложения «Кольцевые гонки»

Необходимо разработать игровое приложение, реализующее многопользовательскую игру (между собой соревнуются несколько игроков) «Кольцевые гонки». Пользователи соревнуются между собой на разных устройствах. Сетевое взаимодействие пользователей описывается принципами *RESTful*. Область видимости контроллера ввода при использовании одного устройства разделена и активна при выборе соответствующего окна приложения. Для реализации данной функции, были выбраны средства языка программирования *C# Windows Presentation Foundation (WPF)*.

*WPF* – система, основная задача которой – построение клиентских приложений для *OS Windows* с визуально привлекательными возможностями взаимодействия вычислительной системы и пользователя.

Каждый из игроков управляет автомобилем, который должен преодолевать появляющиеся препятствия и различного рода опасности на пути к финишу, передвигаясь по кольцевой автодороге. В качестве препятствий, возникающих на пути игрока, могут быть ограждения и вражеский автомобиль.

Во время игры игроки получают возможность подбирать различного рода бонусы и призы. Для генерации призов рационально использовать шаблон проектирования «фабричный метод».

Бонусы способны увеличивать различные характеристики игрока: увеличивать здоровье, восстанавливать пробитые соперником шины и изменять количество патронов у игрока. Для задания изменений характеристик игрока грамотным решением будет использовать шаблон проектирования «декоратор», обеспечивающий надлежащую гибкость и масштабируемость при изменении характеристик игрока.

Для графического отображения объектов на экране была выбрана спрайтовая графика и графические возможности интерфейса *OpenGL*.

Одной из характерных особенностей программного обеспечения является то, что для использования графических возможностей *OpenGL* в паре с языком программирования *C#* необходимо использовать низкоуровневую библиотеку *OpenTK*, так как большинство ресурсов, доступных пользователю, в основном предназначены для *C++*, а не для *C#*. Данная оболочка позволяет решить проблему использования языка *C#* и технологии *OpenGL*. Для успешного внедрения оболочки в проект необходимо подключить некоторые библиотеки: *OpenTK*, *OpenTK.GLWpfControl*.

## 2.2 Архитектура игрового приложения

Проектирование различных алгоритмов и программных продуктов может основываться на различных подходах и методах, среди которых можно выделить:

- а) структурное проектирование программных продуктов;
- б) информационное моделирование предметной области и связанных с ней приложений;
- с) объектно-ориентированное проектирование программных продуктов.

Для реализации игрового приложения был выбран подход структурного проектирования, в основе которого лежит процесс последовательной декомпозиции: научного метода, который позволяет разбить решение объемной задачи на более мелкие ее части.

В качестве основного принципа структурного подхода был выбран принцип нисходящего проектирования (метод пошаговой детализации).

При использовании данного принципа задача разделяется на связанные между собой подзадачи.

Визуально, процесс разбиения на подзадачи можно представить с помощью функциональной схемы. На рисунке 2.1 изображена функциональная схема приложения:

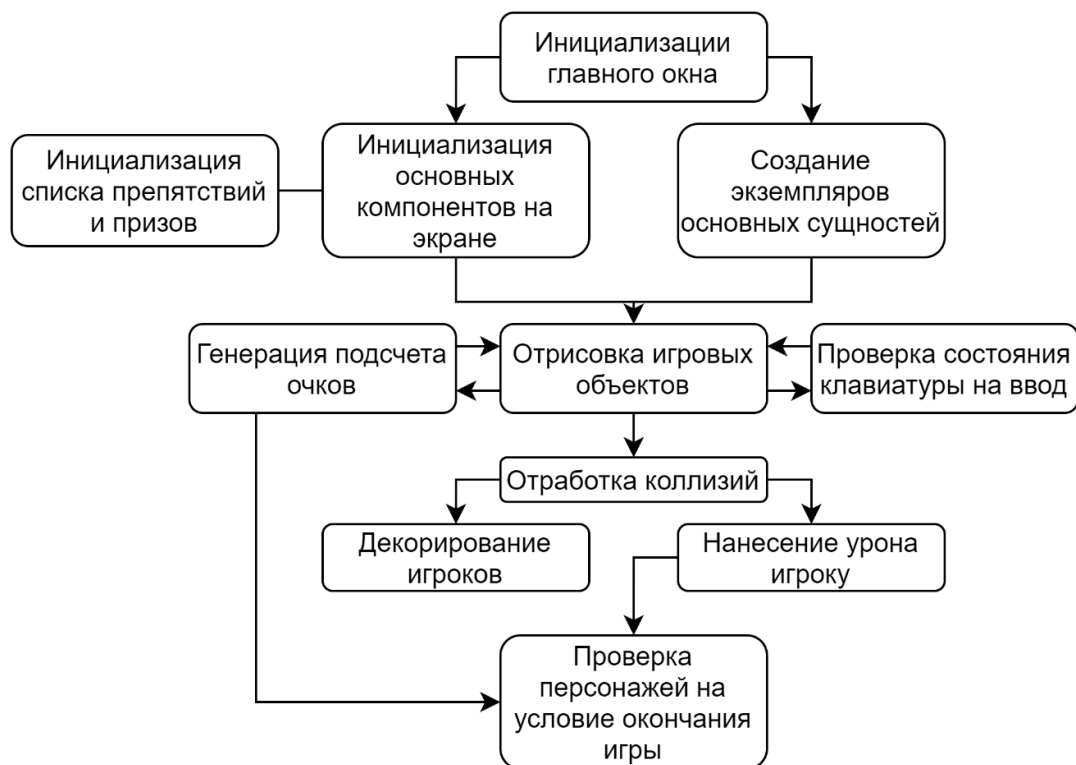


Рисунок 2.1 – Функциональная схема приложения «Кольцевые гонки»

Приложение включает в себя несколько проектов:

- a) Библиотека классов *PSP.GameCommon*;
- b) Проект *WebAPI PSP.GameApi*;
- c) Клиентское *WPF* приложение *PSP.GameClient*.

Проект *WebAPI PSP.GameApi* представляет собой основной сервер приложения, к которому осуществляется подключение игрока. Логика взаимодействия клиентского приложения и сервера определена принципами *RESTful*.

Клиентское *WPF* приложение *PSP.GameClient* инкапсулирует основные методы библиотеки *OpenTK*, для отрисовки графики и пользовательского интерфейса приложения.

Библиотека классов *PSP.GameCommon* содержит в себе классы с описанием основных сущностей приложения, игровых объектов, «декораторов», «фабричных методов» для генерации объектов.

Основу приложения представляют интерфейсы и методы реализации серверной части. Класс *CarService* инициализирует все основные объекты на сцене, и совмещает в себе основной функционал приложения. Класс использует другие, совершенные обособленные экземпляры классов для реализации игровой механики.

Конечной точкой обращения клиента к серверу служат эндпоинты (*Endpoint*). Данный тип взаимодействия представляет собой особый шлюз, который соединяет серверные процессы приложения с внешним интерфейсом (проектом клиента).

Реализации интерфейсов серверной части совмещают в себе все аспекты имплементации поведения объектов на сцене, собирая в общий набор отдельные механизмы программы, обеспечивая большую гибкость для последующих модификаций и доработок приложения.

## 2.3 Структура игрового приложения «Кольцевые гонки»

Для реализации генерации призов и препятствий, встречающихся на пути игрока, а также для задания изменений характеристик игрока были выбраны шаблоны проектирования «декоратор» и «фабричный метод», как наиболее функциональные, гибкие и масштабируемые инструменты реализации программного обеспечения.

Определим понятие шаблона проектирования.

Шаблон проектирования – конструкция, позволяющая решить проблему проектирования в пределах часто возникающей задачи.

Шаблоны, представленные в техническом задании, можно отнести к шаблонам генерации объектов («фабричный метод»), и к шаблонам программирования гибких объектов («декоратор»).



«Декоратор» (*Decorator*) – один из шаблонов проектирования, основная задача которого – динамическое расширение возможностей объекта, расширение функциональности базового объекта.

Применение «декоратора» предполагает динамическое изменение возможностей объектов, что является более гибким функционалом, нежели механизм наследования.

Реализация шаблона декорирования в общем виде может быть следующей:

- а) определение и реализация некоего общего интерфейса, например интерфейс *Component*;
- б) разработка базового «декоратора», наследуемого от общего интерфейса *Component*: подключение и хранение исходной версии базового объекта, переопределение всех методов и свойств родителя;
- с) создание конкретных «декораторов», которые являются наследниками базового;
- д) использование декоратора в клиентском коде, вместо конкретного компонента;
- е) создание, при необходимости, цепочки «декораторов» последовательным «переодеванием» одного в другой, что позволяет увеличивать количество новых возможностей для компонента декорирования;
- ф) обращение к базовому компоненту декорирования в тот момент, когда функции декоратора больше не нужны и последующее возвращение компонента «в оригинальное состояние».

Таким образом, можно сделать следующие выводы:

- а) при использовании шаблона «декоратор» пользователь не замечает подмены компонента за счет реализации одного и того же интерфейса;
- б) шаблон не имеет какого-либо ограничения на количество новых методов и свойств;
- с) шаблон в состоянии работать не только с исходным компонентом, но также и с наследниками исходного компонента;
- д) добавление функциональности компоненту достигается путем подмены оригинального компонента, который можно восстановить, если извлечь его из «декоратора»;
- е) разработчик имеет возможность вкладывать «декораторы» друг в друга, создавая таким образом «цепочки» декораторов, что по праву можно считать альтернативой множественному наследованию.

Следующим шаблоном, активно используемым в курсовом проекте, является «фабричный метод».

«Фабричный метод» (*Factory Method*) – один из порождающих шаблонов, определяющий некий интерфейс для создания объектов производного класса, но какой именно класс будет создан заранее неизвестно [6].

Шаблон используется в том случае, когда необходимо создать большое количество объектов различных классов, но класс создатель заранее не знает какой именно объект необходимо создать.

Реализация шаблона «фабрики» в общем виде может быть следующей:

а) определение и реализация некоего общего абстрактного класса для порождаемых объектов, например класс *Product*;

б) создание базового класса, наследуемого от *Product*, который описывает некий метод, например *Product FabricGenerateMethod()*, для последующего создания конкретных объектов;

с) создание определенных «фабрик» порождаемых объектов, наследуемых от базового класса «фабрики» и переопределение метода создания конкретных объектов;

д) на финальном этапе клиентский код и абстрактный базовый класс используют лишь общий абстрактный класс (*Product*), не обращаясь к реализациям конкретной «фабрики» самостоятельно.

Таким образом, несмотря на все преимущества, которые получает разработчик при использовании шаблона «фабричный метод», очевидными недостатком остается тот факт, что при каждом добавлении нового вида порождаемого объекта необходимо создавать уникальный для этого объекта класс «создателя».

Рассмотрим один из вариантов реализации описанных выше шаблонов проектирования на конкретной реализации их в игровом приложении «Кольцевые гонки».

Метод проектирования «декоратор» предполагает наличие некоторого абстрактного класса, отражающего базовую сущность декорируемого объекта, абстрактного класса «декоратора» и наследуемых от него классов, реализующих декорирование определенного свойства.

Исходя из всего вышеупомянутого была выбрана модель шаблона декоратор, которая представлена на рисунке 2.2

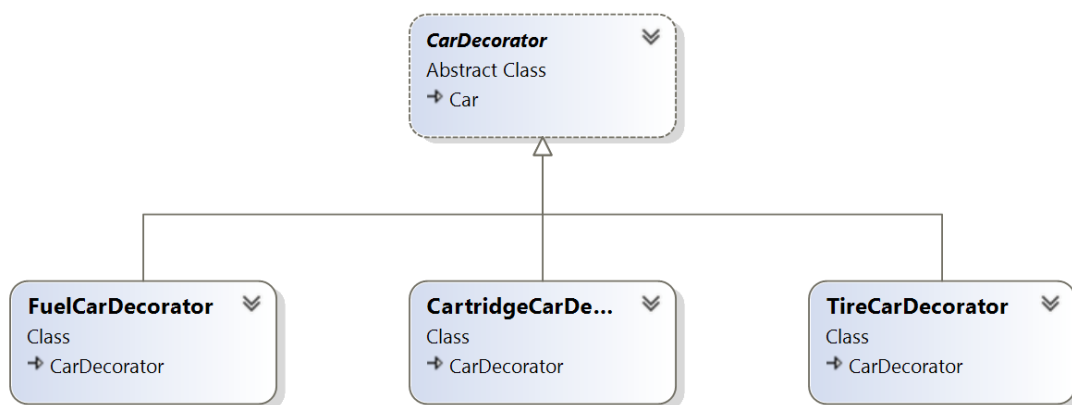


Рисунок 2.2 – Схема шаблона «декоратор» в приложении «Кольцевые гонки»

Из приведенной выше схемы (рисунок 2.2) можно выделить классы, характерные для шаблона «декоратор»: *CarDecorator*, *FuelCarDecorator*, *CartridgeCarDecorator*, *TireCarDecorator*.

Класс *Car* является родительским для класса *CarDecorator*, который, в свою очередь, представляет собой «сущность» игрока. Данный абстрактный класс определяет некоторые поля и свойства для последующего переопределения и декорирования сущности *Car*.

Класс *CarDecorator*, реализован как абстрактный класс и имеет тот же базовый класс, что и декорируемые объекты. Класс хранит ссылку на базовый объект декорирования.

Классы *FuelCarDecorator*, *CartridgeCarDecorator*, *TireCarDecorator* предоставляют дополнительную функциональность декорируемому объекту (*Car*), представляя собой конкретные «декораторы».

Конструктор объекта в классе *FuelCarDecorator* принимает в качестве параметра декорируемую сущность и количество топлива, которое необходимо добавить. С помощью метода *GetCar()* «декоратор» снимается с объекта, возвращая экземпляр базового класса в состояние до момента внесения изменений.

Конструктор объекта в классе *CartridgeCarDecorator* принимает в качестве параметра декорируемую сущность и количество патронов, которое необходимо добавить. С помощью метода *GetCar()* «декоратор» снимается с объекта.

Конструктор объекта в классе *TireCarDecorator* изменяет свойство пробитой шины в зависимости от параметров инициализации, переданных в указанный конструктор. Используя метод *GetCar()* декорируемый экземпляр возвращается в исходное состояние.

Следующим шаблоном, применяемым в приложении, является «фабричный метод». На рисунке 2.3 представлены основные сущности приложения, генерация которых происходит при помощи «фабричного метода».

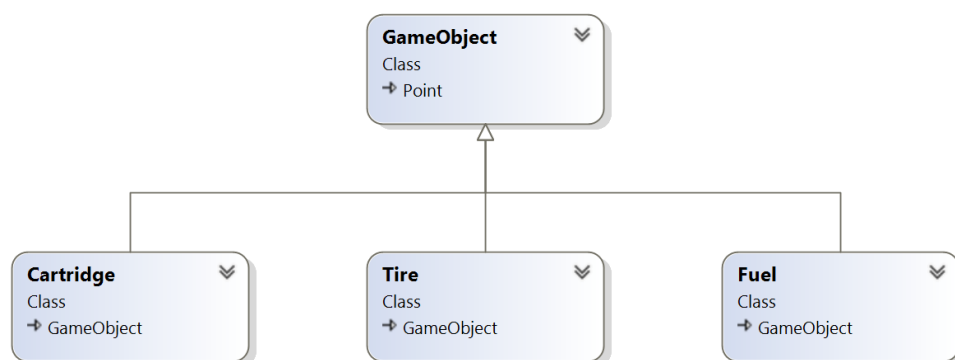


Рисунок 2.3 – Иерархия фабрично генерируемых призов

Родительский класс *GameObject* содержит несколько наследников призов:

- a) *Cartridge* – бонус увеличения количества патронов игрока;
- b) *Fuel* – бонус увеличения уровня топлива игрока;
- c) *Tire* – бонус, восстанавливающий целостность шины.

Каждый из классов содержит инициализацию типа приза в конструкторе объекта.

На рисунке 2.4 представлена схема реализации «фабричного метода» для генерации призов.

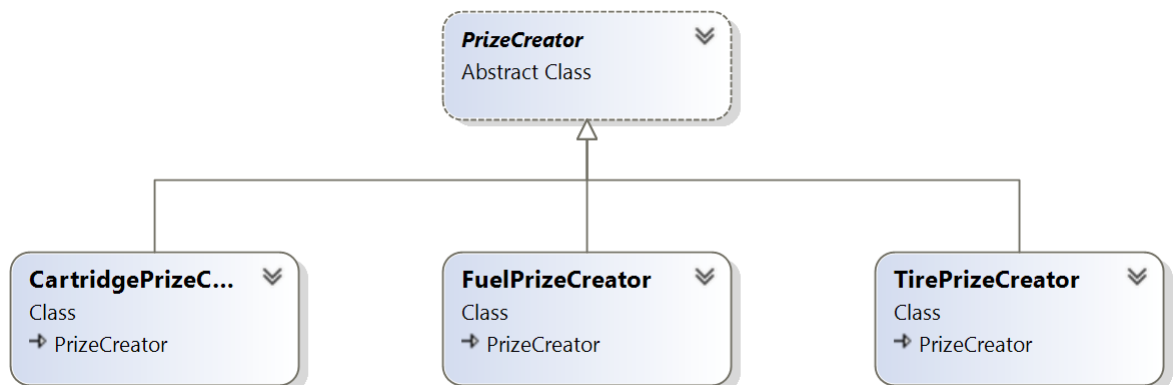


Рисунок 2.4 – Схема «фабричного метода» для генерации призов

Шаблон содержит родительский класс *PrizeCreator*, определяющий абстрактный метод *GetObject()*, который, после переопределения в каждой конкретной фабрике: *CartridgePrizeCreator*, *TirePrizeCreator*, *FuelPrizeCreator*, возвращает конкретный приз, в случайном порядке генерирующийся на игровом поле. В качестве параметров данного метода предаются координаты места инициализации необходимого объекта на игровой сцене.

## 2.4 Структура сетевой части приложения «Кольцевые гонки»

При разработке архитектуры приложения предполагается руководствоваться принципами *RESTful* для построения веб-службы. Исходя из данных принципов была выбрана модель сетевого взаимодействия клиент/сервер – то есть архитектура, состоящая из двух типов программ.

Первая – это серверная программа, которая берет на себя всю организацию игры и взаимодействие с клиентскими программами, а вторая – это сами клиенты, подключаемые к веб-службе.

При таком подходе клиенты будут обмениваться сообщениями только с сервером, не взаимодействуя друг с другом напрямую.

*REST API* основывается на протоколе передачи гипертекста *HTTP* (*Hypertext Transfer Protocol*). Это стандартный протокол в интернете, созданный для передачи гипертекста.

Согласно первой главе документа, в которой определялись правила для сетевого взаимодействия, в *REST API* есть четыре метода *HTTP*, которые используют для действий с объектами на серверах: *GET* – получение информации о данных или списка объектов, *PUT* – обновление данных, *POST* – добавление данных, *DELETE* – удаление данных.

Как было упомянуто в параграфе о сетевом взаимодействии, для обращения к серверу необходимо определить на его стороне *endpoint* (конечную точку/маршрут), который способен реагировать на вызовы со стороны клиента. Для разграничений вызова клиента, целесообразным будет разделить конечные точки согласно особенностям обращений к каждой из них. Таким образом, лучшим решением будет создать несколько классов «контроллеров» [7], которые будут инкапсулировать логику конечных маршрутов, а также применять атрибут маршрутизации, указывающий на то, как контроллер будет сопоставляться с запросами.

Схема для сетевого взаимодействия клиента с веб-службой представлена на рисунке 2.5.

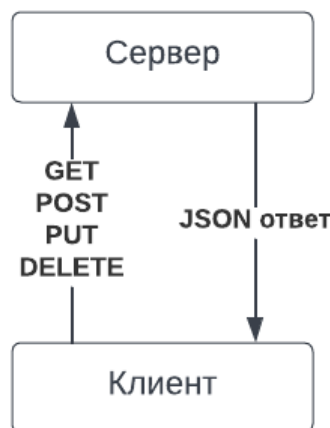


Рисунок 2.5 – Схема сетевого взаимодействия

Ответ в данном случае возвращается в виде формата *JSON* (*JavaScript Object Notation*). Объект *JSON* – это неупорядоченный набор пар имя-значение. Синтаксис объекта *JSON* определяет имена как строки, которые всегда заключены в двойные кавычки. Это удобны и легко читаемый формат, преимущества использования которого по достоинству оценены многими разработчиками.

## 3 РЕАЛИЗАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ

### 3.1 Принцип работы игрового приложения «Кольцевые гонки»

При запуске клиентской версии игрового приложения происходит открытие проекта *WPF* и инициализация основных элементов формы. Следует отметить, что для корректной работы игрового приложения «Кольцевые гонки», перед запуском клиентской части необходимо запустить проект веб-службы. Необходимость поэтапного запуска приложений обусловлена тем, что при старте программы и подготовке к отрисовке главного окна приложения, в методе *OpenTkControl\_OnReady()* (приложение А, код класса *MainWindow.xaml.cs*) происходит обращение к серверной части игрового приложения. В зависимости от результата данного обращения будет ясно: может ли пользователь продолжить дальнейшее взаимодействие с игровым приложением, или же ему необходимо дождаться решения проблемы.

Параметры старта серверной части приложения обусловлены спецификацией, определенной в файле *launchsettings.json* (приложение А, код файла *launchsettings.json*). Данный файл определяет различные переменные среды. Например, адрес запускаемого приложения, активация элементов документирования и иные настройки.

После успешного запуска веб-службы происходит процесс инициализации и запуска клиентского приложения.

В разметку главного окна приложения включен специальный элемент управления: *GLWpfControl* (приложение А, код класса *MainWindow.xaml*), который позволяет отображать графические элементы *OpenGL* в области окна *WPF* формы. В конструкторе инициализатора формы настраивается форма управления графическим интерфейсом, после чего следует запуск и отслеживание событий на форме (приложение А, код класса *MainWindow.xaml.cs*). Произведенные манипуляции необходимы по причине того, что для использования графической библиотеки *OpenGL* в паре с языком программирования *C#* необходимо использовать низкоуровневую оболочку *OpenTK*, так как большинство ресурсов, доступных пользователю, в основном предназначены для *C++*, а не для *C#*. Таким образом, выполняется одно из главных условий технического задания: использование элемента *C# WPF*.

*GLWpfControl* имеет несколько наиболее значимых обработчиков событий, один из которых (*OnReady*) уже был упомянут. Метод *OpenTkControl\_OnReady()* подключает игрока к серверу, обрабатывает полученный ответ, а также загружает карту и все игровые объекты. Загрузка текстур для создаваемых объектов происходит при помощи метода *LoadSprite()*

(приложение А, код класса *DrawService.cs*), который определен в интерфейсе *IdrawService* (приложение А, код класса *IdrawService.cs*), и имеет реализацию соответствующем классе *DrawService*. Отрисовка игровых объектов на сцене приложения осуществляется также с использованием класса *DrawService* и метода *Draw* (приложение А, код класса *DrawService.cs*).

После инициализации формы происходит обращение к веб службе для создания объекта класса *Car*, инкапсулирующего в себе логику клиента (приложение А, код класса *Car.cs*). Обращение к серверной части происходит с помощью интерфейса *INetworkService* (приложение А, код класса *INetworkService.cs*), который определяет ряд методов, впоследствии реализованных одноименным классом, для доступа к конечным маршрутам. Обращение реализовано с помощью создания экземпляра объекта *HttpClient* (приложение А, код класса *NetworkService.cs*).

Событие начала игры срабатывает при подключении двух участников игрового процесса. В случае отсутствия исключения при подключении последнего игрока запускается метод *GetGameObjects()* определенный в интерфейсе *ClientService* (приложение А, код класса *IClientService.cs*). В данном методе происходит обращение к удаленному серверу посредством вышеупомянутого интерфейса сервиса *INetworkService*. Обращение к веб-службе позволяет получить разметку уровня игры *GetLevel()* (приложение А, код класса *INetworkService.cs*), ключевые точки прохождения трассы *GetLevelRightSequence()* (приложение А, код класса *INetworkService.cs*), а также сгенерированные в случайном порядке призы *GetPrizes()* (приложение А, код класса *INetworkService.cs*).

После инициализации сцены запускается зацикленная инфраструктуру рендеринга, которая формирует каждый кадр игрового окна соответствующим методом *OpenTkControl\_OnRender()* (приложение А, код класса *MainWindow.xaml.cs*). Данный метод позволяет игровому приложению каждый кадр обновлять состояние компонентов клавиатуры и времени, а затем выводить изображение текущего кадра с помощью рендеринга игровых объектов сцены. Игровые объекты требуется проинициализировать нужными для его цели компонентами, после чего игровой объект каждый кадр выводится на экран методом *Draw* (приложение А, код класса *DrawServices.cs*), пока игровой объект существует на сцене. Рендеринг объекта заключается в создании матрицы перемещения, вращения и размера, и рисования изображения на основе это матрицы.

Класс *Timer* (приложение А, код класса *GameService.cs*) является таймером для генерации призов на экране пользователя. Через определенный интервал реального времени происходит вызов методов обновления позиции призов с

помощью метода *RefreshPrizes()* (приложение А, код класса *GameService.cs*). Данный метод оперирует всеми объектами на игровой сцене и вызывает метод *RefreshPrizes()* (приложение А, код класса *PrizeService.cs*) из специализированного интерфейса. Последний оперирует методом *RandomNoCollisionPosition()* (приложение А, код класса *RandomNoCollisionPosition.cs*), который, используя список объектов на сцене, определяет место для обновленного положения призов.

Движение игроков по игровому пространству осуществляется с использованием клавиш клавиатуры. Основная механика игры – передвижение автомобиля по кольцевой автодороге. Движение и логика взаимодействия игроков с внутриигровыми объектами описана с помощью метода *MoveGamer()* (приложение А, код класса *CarService.cs*).

Столкновение игрока и препятствий (призов) обрабатывается с помощью метода *CheckAndUpdateWithPrizeCollision()* класса *CarService* (приложение А, код класса *CarService.cs*). В качестве параметра метод принимает объект игрока и массив объектов призов на игровой сцене. Внутренний метод *CheckCollision()* (приложение А, код класса *CollisionHelper.cs*) возвращает значение «истинно», если столкновение существовало, и значение «ложно», если столкновения не случилось. Метод работает по принципу сравнения координат игрового объекта и координат игрока на игровой сцене. В случае столкновения игрока с призом вызывается метод *Decorate()* класса *CarService*. При подборе приза игрок получает бонус к своим характеристикам. Наложение призов осуществляется с использованием паттерна «декоратор», принцип работы которого был описан ранее.

В ходе выполнения, игровое приложение постоянно проверяется на окончание сеанса игрового процесса. Все проверки игры осуществляются в методе *IsGameEnd()* (приложение А, код класса *ClientService.cs*). Финальным результатом игры является победа одного из участников гонки. Результаты состязания выводятся каждому участнику с помощью отдельного окна отображения.

При завершении игры закрывается основное окно формы отображения программы, освобождаются ресурсы приложения.

### **3.2 Верификация игрового приложения «Кольцевые гонки»**

При запуске игрового приложения открывается окно пользовательского интерфейса, представляющее собой основную игровую область приложения.

Игровая сцена приложения «Кольцевые гонки» представлена на рисунке 3.1.



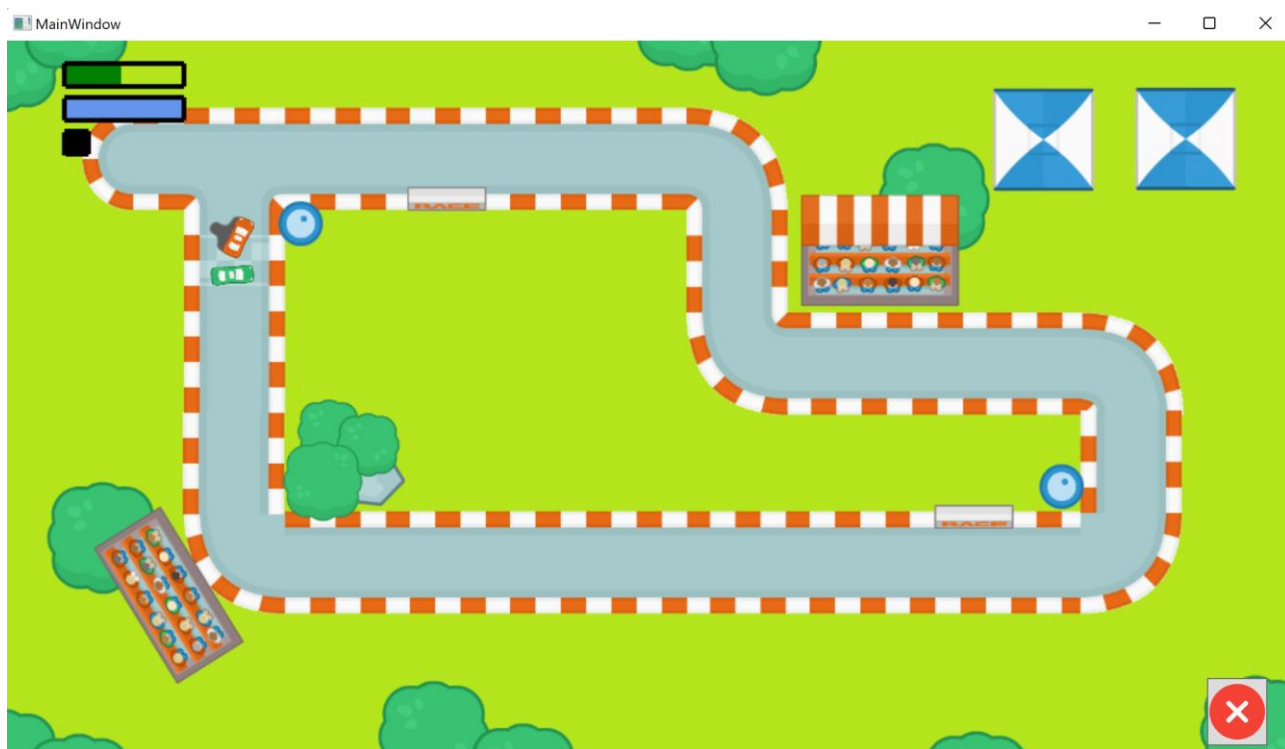


Рисунок 3.1 – Игровая сцена приложения

В отрисованной сцене игрового приложения «Кольцевые гонки» можно заметить, что автомобиль игрока имеет зеленый цвет. Рядом с ним, при инициализации с небольшим смещением отображается автомобиль соперника, который, в свою очередь, имеет текстуру автомобиля красного цвета.

Для передвижения автомобиля по трассе игроку необходимо использовать стандартные и интуитивно понятные для рассматриваемого жанра сочетания клавиш. Так, например, нажав нажать на клавиатуре клавишу *W*, пользователь может двигаться вперед, увеличивая свою скорость с течением времени. Соответственно, нажатие клавиши *S* будет означать движение игрока в обратном направлении. При нажатии клавиш *A* и *D* осуществляется поворот пользователя на определенное значение угла поворота. Клавиша *SPACE* используется для стрельбы по противнику.

В главной сцене игрового приложения у пользователя также есть возможность закрыть форму нажатием на кнопку «Выход», которая представлена соответствующим изображением в правом углу формы. В таком случае программа клиента прекратит свое выполнение для этого пользователя, игрок будет удален и произойдёт освобождение ресурсов игры.

Пользователи игрового приложения соревнуются на совмещенном экране, который представлен отдельным окном клиентской формы для каждого из них. На пути игроков в случайном порядке генерируются призы, увеличивающие характеристики каждого отдельного игрока.

Виды доступных генерируемых призов:

- d) приз повышения уровня здоровья;
- e) приз увеличения скорости движения игрока;
- f) приз увеличения высоты прыжка.

Графическое представление генерируемых призов продемонстрировано на рисунке 3.2.



Рисунок 3.2 – Призы игрового приложения

При движении автомобиль каждого игрока имеет некоторую скорость, которая изменяется со временем. Уровень топлива игрового объекта уменьшается на модуль скорости движения объекта, тем самым затормаживая автомобиль. Приз «топливо» (рисунок 3.2) помогает игроку восполнить количества топлива в баке и продолжить движение с повышенной скоростью.

Как упоминалось ранее, пользователь программного продукта в процессе геймплея способен стрелять по вражескому автомобилю. В процессе «перестрелки» колесо соперника может быть пробито. При условии, что колесо противника пробито, автомобиль начинает двигаться с постоянно низкой скоростью. Спасением для пользователя в таком случае становится бонус «шина» (рисунок 3.2). Данный приз позволяет восстановить целостность колес автомобиля и продолжить движение в нормальном режиме.

С возможностью стрельбы тесно связан еще один приз: патроны. Механика взаимодействия схожа с предыдущими призами, а уникальное свойство данного бонуса заключается в увеличении количества патронов для игрока.

Максимальный уровень топлива автомобиля на начальном этапе игрового процесса равен 300 условным единицами. В левом верхнем углу графически отображается уровень топлива игрока, количество доступных патронов и идентификатор пробитой шины. В зависимости от уровня топлива автомобиля отображение динамически изменяется, отражая текущее значение. Количество

патронов по умолчанию равно нулю, однако, как было отмечено, это значение всегда можно изменить, воспользовавшись определенным бонусом.

Графическое представление элементов игрового интерфейса для автомобиля пользователя продемонстрировано на рисунке 3.3.



Рисунок 3.3 – Элемент игрового интерфейса пользователя

Финальным этапом служит окончание игры, которое варьируется в зависимости от того, одержал ли победу пользователь или нет. По нажатию на кнопку положительного выбора основная сцена игрового приложения закроется, а пользователь станет недоступен для отрисовки на сцене «соперника». Примеры окон окончания игры представлены на рисунке 3.4.

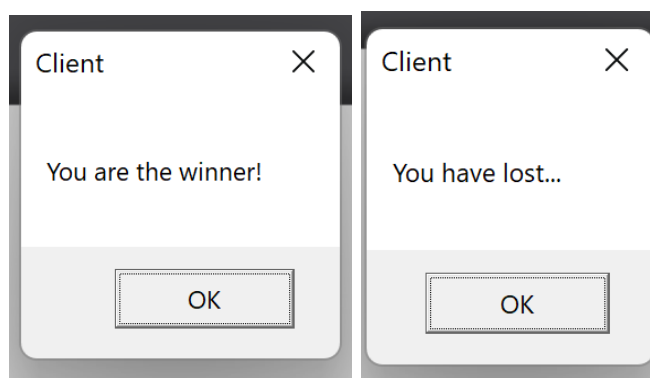


Рисунок 3.4 – Окна окончания игры

Главная цель игры – первым прийти к финишу. Первым пришедшим к финишу игроком считается тот игрок, который первый преодолел необходимое для победы количество кругов. Для достижения этой цели игроку необходимо избегать столкновения с препятствиями и противником, а также стараться поддерживать максимальную скорость движения автомобиля и не быть «подстреленным» противником.

Как только проверка игры на окончание получает положительный результат, отрисовка и генерация игровых объектов прекращается, после чего открывается окно окончания игры (рисунок 3.4).

### 3.3 Результаты тестирования игрового приложения

Специально для тестирования разработанного игрового приложения был создан проект модульных тестов *PSP.GameTest*. Для тестирования используются статические методы класса *Assert* и классы модульного тестирования, которые находятся в пространстве имен *Microsoft.VisualStudio.TestTools.UnitTesting*.

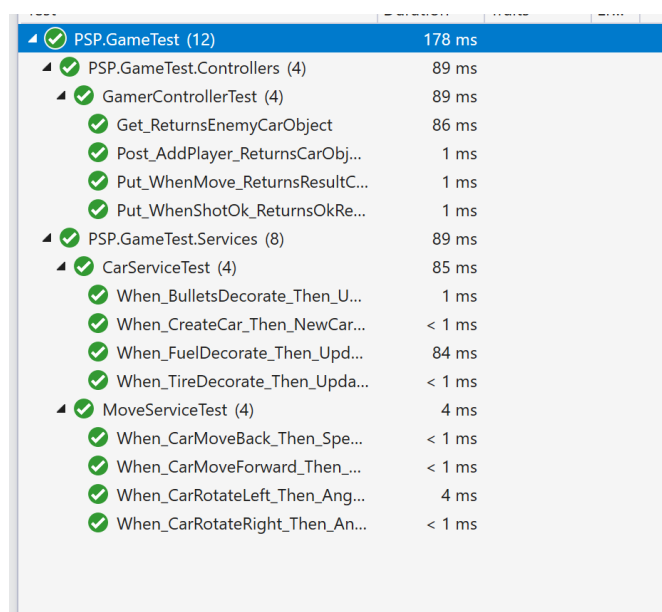
В ходе подготовки модульных тестов был выделен ряд классов, позволяющих протестировать разработанное игровое приложение.

*CarServiceTest* (приложение А, код класса *CarServiceTest.cs*), методы которого позволяют протестировать классы *CarService* (приложение А, код класса *CarService.cs*), *FuelCarDecorator* (приложение А, код класса *FuelCarDecorator.cs*), *TireCarDecorator* (приложение А, код класса *TireCarDecorator.cs*), *CartidgeCarDecorator* (приложение А, код класса *CartidgeCarDecorator.cs*), на соответствие объекта, подвергнувшегося декорированию, базовому объекту, а также создание нового объекта.

*MoveServiceTest* (приложение А, код класса *MoveServiceTest.cs*), методы которого позволяют протестировать направление движения объекта, а также изменение свойств автомобиля при движении в различные стороны.

*GamerControllerTest* (приложение А, код класса *GamerControllerTest.cs*) определяет тестовые случаи для контролера и его конечных точек: создание объекта, получение, выстрел и изменение позиции.

На рисунке 3.5 представлены результаты тестирования классов игрового приложения.



✓ PSP.GameTest (12)	178 ms
✓ PSP.GameTest.Controllers (4)	89 ms
✓ GamerControllerTest (4)	89 ms
✓ Get_ReturnsEnemyCarObject	86 ms
✓ Post_AddPlayer_ReturnsCarObj...	1 ms
✓ Put_WhenMove_ReturnsResultC...	1 ms
✓ Put_WhenShotOk_ReturnsOkRe...	1 ms
✓ PSP.GameTest.Services (8)	89 ms
✓ CarServiceTest (4)	85 ms
✓ When_BulletsDecorate_Then_U...	1 ms
✓ When_CreateCar_Then_NewCar...	< 1 ms
✓ When_FuelDecorate_Then_Upd...	84 ms
✓ When_TireDecorate_Then_Upda...	< 1 ms
✓ MoveServiceTest (4)	4 ms
✓ When_CarMoveBack_Then_Spe...	< 1 ms
✓ When_CarMoveForward_Then_...	< 1 ms
✓ When_CarRotateLeft_Then_Ang...	4 ms
✓ When_CarRotateRight_Then_An...	< 1 ms

Рисунок 3.5 – Окно результатов модульного тестирования

## ЗАКЛЮЧЕНИЕ

В результате проделанной работы было создано сетевое игровое приложение «Кольцевые гонки», где каждый игрок управляет персонажем на совмещенной игровой сцене, разделенной для каждого пользователя клиентской формой. На пути игрока случайным образом генерируются призы, которые способны увеличить характеристики игрока (уровень топлива, количество патронов, восстановление шины). Пользователи имеют возможность стрелять друг в друга, тем самым вызывая повреждение колеса и замедление автомобиля соперника.

Для реализации приложения были использованы средства языка программирования *C# WPF* с подключением библиотеки *OpenGL* для отображения графических объектов. Реализована сетевая часть приложения для взаимодействия нескольких пользователей.

Проведен аналитический анализ программных средств реализации игровых приложений, разработана иерархия классов проектируемого приложения. Были рассмотрены основные компоненты и особенности библиотеки отображения компьютерной графики *OpenGL*, приведены основные отличительные особенности и различия в технологиях отображения графики *DirectX* и *OpenGL*, рассмотрены основные принципы сетевого взаимодействия при применении *RESTful* подхода. На этапе аналитического обзора был представлен пример схожего программного продукта.

Во время проектирования приложения была разработана и представлена функциональная схема приложения, общая структура программного продукта. Изучены и применены шаблоны проектирования, такие как «декоратор», «фабричный метод». Представлена схема сетевого взаимодействия клиентской и серверной части приложения.

В процессе создания игрового приложения был создан, интуитивно понятный пользователю, пользовательский интерфейс для запуска приложения, выхода из игрового приложения.

С помощью библиотеки модульных тестов протестированы участки кода игрового приложения «Кольцевые гонки». Функции протестированных методов:

- а) декорирование игровых объектов;
- б) правильность изменения характеристик игровых персонажей.

Без корректной работы данных методов игровое приложение будет работать некорректно.

Курсовая работа выполнена самостоятельно, проверена в системе «Антиплагиат». Процент оригинальности составляет 95%. Цитирования обозначены ссылками на публикации, указанные в «Списке использованных источников».

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *J. Vries, Learn OpenGL: Learn modern OpenGL graphics programming / Joey de Vries – USA: Kendall & Welling, 2020. – 11 с.*
2. Игнатенко, А. *OpenGL и DirectX: взгляд изнутри / А. Игнатенко / Компьютерная графика и мультимедиа. – 2004. – №2. – С.1 – 2.*
3. Гончаров, Д. А. *DirectX 7.0 для программистов / Д. А. Гончаров, Т. Т. Салихов. – СПб: Питер, 2001. – 26 с.*
4. *Massé, REST API Design Rulebook / Mark Massé – United States of America: O'Reilly Media Inc., 2012. – 5 с.*
5. *L. Richardson, M. Amundsen, RESTful Web APIs / Leonard Richardson and Mike Amundsen – United States of America: O'Reilly Media Inc., 2013. – 28 с.*
6. Приемы объектно-ориентированного программирования. Паттерны проектирования / Э. Гамма – СПб.: Питер, 2015. – 111 с.
7. *S. Allamaraju, M. Amundsen, RESTful Web Services Cookbook / Subbu Allamaraju – United States of America: O'Reilly Media Inc., 2010. – 39 с.*

## ПРИЛОЖЕНИЕ А

(обязательное)

### Листинг программы «Кольцевые гонки»

#### Код программы для CarDecorator.cs

```
namespace PSP.GameCommon.Decorator
{
    public abstract class CarDecorator : Car
    {
        protected Car car;
        public bool IsDecorate { get; protected set; }

        public CarDecorator(Car car) : base()
        {
            this.car = car;
            IsDecorate = true;
        }

        public override bool[] LevelsSequence { get => car.LevelsSequence; set => car.LevelsSequence = value; }
        public override int RightLevelsSequence { get => car.RightLevelsSequence; set => car.RightLevelsSequence = value; }
        public override bool IsFailingTire { get => car.IsFailingTire; set => car.IsFailingTire = value; }
        public override float MaxFuel { get => car.MaxFuel; set => car.MaxFuel = value; }
        public override int Cartridges { get => car.Cartridges; set => car.Cartridges = value; }
        public override int MaxCartridges { get => car.MaxCartridges; set => car.MaxCartridges = value; }
        public override bool IsCollision { get => car.IsCollision; set => car.IsCollision = value; }
        public override string PrizeId { get => car.PrizeId; set => car.PrizeId = value; }
        public override bool Tire { get => car.Tire; set => car.Tire = value; }

        public override float Speed { get => car.Speed; set => car.Speed = value; }
        public override float MaxSpeed { get => car.MaxSpeed; set => car.MaxSpeed = value; }
        public override float Fuel { get => car.Fuel; set => car.Fuel = value; }
        public override float SpeedChange { get => car.SpeedChange; set => car.SpeedChange = value; }
        public override float OldPositionX { get => car.OldPositionX; set => car.OldPositionX = value; }
        public override float OldPositionY { get => car.OldPositionY; set => car.OldPositionY = value; }

        public override string Id { get => car.Id; set => car.Id = value; }
        public override float SizeX { get => car.SizeX; set => car.SizeX = value; }
        public override float SizeY { get => car.SizeY; set => car.SizeY = value; }
        public override float Angle { get => car.Angle; set => car.Angle = value; }
        public override GameObjectType Name { get => car.Name; set => car.Name = value; }
        public override int SpriteId { get => car.SpriteId; set => car.SpriteId = value; }
        public override float SpriteSizeX { get => car.SpriteSizeX; set => car.SpriteSizeX = value; }
        public override float SpriteSizeY { get => car.SpriteSizeY; set => car.SpriteSizeY = value; }
        public override bool IsDeactivate { get => car.IsDeactivate; set => car.IsDeactivate = value; }

        public override float PositionX { get => car.PositionX; set => car.PositionX = value; }
        public override float PositionY { get => car.PositionY; set => car.PositionY = value; }

        /// <summary>
        /// Return car instance
        /// </summary>
        /// <returns>Car instance</returns>
        public Car GetCar()
        {
            return car;
        }
    }
}
```

### **Код программы для CartridgeCarDecorator.cs**

```
namespace PSP.GameCommon.Decorator
{
    public class CartridgeCarDecorator : CarDecorator
    {
        public CartridgeCarDecorator(Car car, int cartridges) : base(car)
        {
            car.Cartridges = cartridges;
            IsDecorate = false;
        }
    }
}
```

### **Код программы для FuelCarDecorator.cs**

```
namespace PSP.GameCommon.Decorator
{
    public class FuelCarDecorator : CarDecorator
    {
        public FuelCarDecorator(Car car, float fuel) : base(car)
        {
            car.Fuel = fuel;
            IsDecorate = false;
        }
    }
}
```

### **Код программы для TireCarDecorator.cs**

```
namespace PSP.GameCommon.Decorator
{
    public class TireCarDecorator : CarDecorator
    {
        public TireCarDecorator(Car car, bool tire) : base(car)
        {
            car.Tire = tire;
            IsDecorate = false;
        }
    }
}
```

### **Код программы для PrizeCreator.cs**

```
namespace PSP.GameCommon.Factory
{
    public abstract class PrizeCreator
    {
        public abstract GameObject GetObject(float w, float h);
    }
}
```

### **Код программы для CartridgePrizeCreator.cs**

```
namespace PSP.GameCommon.Factory
{
    public class CartridgePrizeCreator : PrizeCreator
```



```

{
    public override GameObject GetObject(float w, float h)
    {
        var obj = new Cartridge
        {
            SizeX = w,
            SizeY = h
        };

        return obj;
    }
}

```

### **Код программы для FuelPrizeCreator.cs**

```

namespace PSP.GameCommon.Factory
{
    public class FuelPrizeCreator : PrizeCreator
    {
        public override GameObject GetObject(float w, float h)
        {
            var obj = new Fuel
            {
                SizeX = w,
                SizeY = h
            };

            return obj;
        }
    }
}

```

### **Код программы для TirePrizeCreator.cs**

```

namespace PSP.GameCommon.Factory
{
    public class TirePrizeCreator : PrizeCreator
    {
        public override GameObject GetObject(float w, float h)
        {
            var tire = new Tire
            {
                SizeX = w,
                SizeY = h
            };

            return tire;
        }
    }
}

```

### **Код программы для Bullet.cs**

```

namespace PSP.GameCommon.GameObjects
{
    public class Bullet : MovableGameObject
    {
        public string OwnerId { get; set; }
    }
}

```

## Код программы для Car.cs

```
namespace PSP.GameCommon.GameObjects
{
    public class Car : MovableGameObject
    {
        public virtual bool[] LevelsSequence { get; set; }
        public virtual int RightLevelsSequence { get; set; }
        public virtual bool IsFailingTire { get; set; }
        public virtual float MaxFuel { get; set; }
        public virtual int Cartridges { get; set; }
        public virtual int MaxCartridges { get; set; }
        public virtual bool IsCollision { get; set; }
        public virtual string PrizeId { get; set; }
    }
}
```

## Код программы для Cartridge.cs

```
namespace PSP.GameCommon.GameObjects
{
    public class Cartridge : GameObject
    {
        public Cartridge()
        {
            Name = GameObjectType.Cartridge;
        }
    }
}
```

## Код программы для Fuel.cs

```
namespace PSP.GameCommon.GameObjects
{
    public class Fuel : GameObject
    {
        public Fuel()
        {
            Name = GameObjectType.Fuel;
        }
    }
}
```

## Код программы для Tire.cs

```
namespace PSP.GameCommon.GameObjects
{
    public class Tire : GameObject
    {
        public Tire()
        {
            Name = GameObjectType.Tire;
        }
    }
}
```

## Код программы для ItemsScope.cs

```
namespace PSP.GameCommon.Utility
{
    public static class ItemsScope
    {
        public const int RightLevelsSequence = 1;
    }
}
```

```

    }
}

```

### **Код программы для GameObject.cs**

```

namespace PSP.GameCommon
{
    public class GameObject : Point
    {
        public virtual string Id { get; set; }
        public virtual float SizeX { get; set; }
        public virtual float SizeY { get; set; }
        public virtual float Angle { get; set; }
        public virtual GameObjectType Name { get; set; }
        public virtual int SpriteId { get; set; }
        public virtual float SpriteSizeX { get; set; }
        public virtual float SpriteSizeY { get; set; }
    }
}

```

### **Код программы для GameObjectType.cs**

```

namespace PSP.GameCommon
{
    public enum GameObjectType
    {
        Fuel,
        Cartridge,
        Tire
    }
}

```

### **Код программы для MovableGameObject.cs**

```

namespace PSP.GameCommon
{
    public class MovableGameObject : GameObject
    {
        public virtual float Speed { get; set; }
        public virtual float MaxSpeed { get; set; }
        public virtual float Fuel { get; set; }
        public virtual bool Tire { get; set; }
        public virtual float SpeedChange { get; set; }
        public virtual float OldPositionX { get; set; }
        public virtual float OldPositionY { get; set; }
    }
}

```

### **Код программы для Point.cs**

```

namespace PSP.GameCommon
{
    public class Point
    {
        public virtual float PositionX { get; set; }
        public virtual float PositionY { get; set; }
        public virtual bool IsDeactivate { get; set; }
    }
}

```

### **Код программы для GamerControllerTest.cs**

```

namespace PSP.GameTest.Controllers
{
    public class GamerControllerTest
    {
        [Fact]
        public void Post_AddPlayer_ReturnsCarObject()
        {
            // Arrange
            var mockGameService = new Mock<IGameService>();

            var carId = Guid.NewGuid().ToString();
            mockGameService.Setup(exp => exp.AddGamer(carId))
                .Returns(new Car());
            var mockCarService = new Mock<ICarService>();

            var controller = new GamerController(mockGameService.Object, mockCarService.Object);

            // Act
            var result = controller.Post(carId);

            // Assert
            Assert.IsType<Car>(result);
        }

        [Fact]
        public void Get_ReturnsEnemyCarObject()
        {
            // Arrange
            var mockGameService = new Mock<IGameService>();
            var carId = Guid.NewGuid().ToString();
            var mockCarService = new Mock<ICarService>();
            mockCarService.Setup(exp => exp.GetEnemyCar(carId))
                .Returns(new Car { Id = string.Empty });

            var controller = new GamerController(mockGameService.Object, mockCarService.Object);

            // Act
            var result = controller.Get(carId);

            // Assert
            Assert.IsType<Car>(result);
            Assert.NotEqual(result.Id, carId);
        }

        [Fact]
        public void Put_WhenShotOk_ReturnsOkResult()
        {
            // Arrange
            var mockGameService = new Mock<IGameService>();
            var carId = Guid.NewGuid().ToString();
            var mockCarService = new Mock<ICarService>();
            mockCarService.Setup(exp => exp.GetShot(carId))
                .Returns(new Bullet());

            var controller = new GamerController(mockGameService.Object, mockCarService.Object);

            // Act
            var result = controller.PutGetShot(carId);

            // Assert
            Assert.IsType<OkResult>(result);
        }
    }
}

```

```

[Fact]
public void Put_WhenMove_ReturnsResultCarObject()
{
    // Arrange
    var mockGameService = new Mock<IGameService>();

    var carId = Guid.NewGuid().ToString();
    var direction = 0;

    var mockCarService = new Mock<ICarService>();
    mockCarService.Setup(exp => exp.MoveGamer(carId, direction))
        .Returns(new Car());

    var controller = new GamerController(mockGameService.Object, mockCarService.Object);

    // Act
    var result = controller.PutMove(carId, direction);

    // Assert
    Assert.IsType<Car>(result);
}
}
}

```

### **Код программы для CarServiceTest.cs**

```

namespace PSP.GameTest.Services
{
    public class CarServiceTest
    {
        [Fact]
        public void When_CreateCar_Then_NewCarObjectReturned()
        {
            // Arrange
            var mockLevelService = new Mock<ILevelService>();
            var mockPrizeService = new Mock<IPrizeService>();
            var mockMoveService = new Mock<IMoveService>();

            var carService = new CarService(mockMoveService.Object, mockPrizeService.Object, mockLevelService.Object);
            var carId = Guid.NewGuid().ToString();

            // Act
            var result = carService.CreateCar(carId);

            // Assert
            Assert.IsType<Car>(result);
        }

        [Fact]
        public void When_FuelDecorate_Then_UpdatedCarReturned()
        {
            // Arrange
            var mockLevelService = new Mock<ILevelService>();
            var mockPrizeService = new Mock<IPrizeService>();
            var mockMoveService = new Mock<IMoveService>();

            var carService = new CarService(mockMoveService.Object, mockPrizeService.Object, mockLevelService.Object);
            var carId = Guid.NewGuid().ToString();
            var car = carService.CreateCar(carId);
            car.Fuel = 0;

            var prizeType = GameObjectType.Fuel;

```

```

        // Act
        var result = carService.Decorate(car, prizeType);

        // Assert
        Assert.IsType<Car>(result);
        Assert.Equal(50, car.Fuel);
    }

    [Fact]
    public void When_TireDecorate_Then_UpdatedCarReturned()
    {
        // Arrange
        var mockLevelService = new Mock<ILevelService>();
        var mockPrizeService = new Mock<IPrizeService>();
        var mockMoveService = new Mock<IMoveService>();

        var carService = new CarService(mockMoveService.Object, mockPrizeService.Object, mockLevelService.Object);
        var carId = Guid.NewGuid().ToString();
        var car = carService.CreateCar(carId);
        car.Tire = false;

        var prizeType = GameObjectType.Tire;

        // Act
        var result = carService.Decorate(car, prizeType);

        // Assert
        Assert.IsType<Car>(result);
        Assert.True(car.Tire);
    }

    [Fact]
    public void When_BulletsDecorate_Then_UpdatedCarReturned()
    {
        // Arrange
        var mockLevelService = new Mock<ILevelService>();
        var mockPrizeService = new Mock<IPrizeService>();
        var mockMoveService = new Mock<IMoveService>();

        var carService = new CarService(mockMoveService.Object, mockPrizeService.Object, mockLevelService.Object);
        var carId = Guid.NewGuid().ToString();
        var car = carService.CreateCar(carId);
        car.Cartridges = 0;

        var prizeType = GameObjectType.Cartridge;

        // Act
        var result = carService.Decorate(car, prizeType);

        // Assert
        Assert.IsType<Car>(result);
        Assert.Equal(2, car.Cartridges);
    }
}
}

```

## Код программы для MoveServiceTest.cs

```

namespace PSP.GameTest.Services
{
    public class MoveServiceTest
    {
        [Fact]
    }
}

```

```

public void When_CarMoveForward_Then_SpeedChanged()
{
    // Arrange
    var moqMovableGameObject = new MovableGameObject
    {
        Speed = 0.04f,
        MaxSpeed = 1f,
        Fuel = 100,
        Tire = true
    };

    var moveService = new MoveService();

    // Act
    var result = moveService.MoveForward(moqMovableGameObject);

    // Assert
    Assert.IsType<MovableGameObject>(result);
    Assert.Equal(0.08f, result.Speed);
}

[Fact]
public void When_CarRotateLeft_Then_AngleChanged()
{
    // Arrange
    var moqMovableGameObject = new MovableGameObject
    {
        Speed = 0.04f,
        MaxSpeed = 1f,
        Fuel = 100,
        Tire = true,
        Angle = 0
    };

    var moveService = new MoveService();

    // Act
    var result = moveService.RotateLeft(moqMovableGameObject);

    // Assert
    Assert.IsType<MovableGameObject>(result);
    Assert.Equal(0.08f, result.Angle);
}

[Fact]
public void When_CarRotateRight_Then_AngleChanged()
{
    // Arrange
    var moqMovableGameObject = new MovableGameObject
    {
        Speed = 0.04f,
        MaxSpeed = 1f,
        Fuel = 100,
        Tire = true,
        Angle = 0
    };

    var moveService = new MoveService();

    // Act
    var result = moveService.RotateRight(moqMovableGameObject);

    // Assert

```

```

        Assert.IsType<MovableGameObject>(result);
        Assert.Equal(-0.08f, result.Angle);
    }

    [Fact]
    public void When_CarMoveBack_Then_SpeedChanged()
    {
        // Arrange
        var mockMovableGameObject = new MovableGameObject
        {
            Speed = 0.04f,
            MaxSpeed = 1f,
            Fuel = 100,
            Tire = true,
            Angle = 0
        };

        var moveService = new MoveService();

        // Act
        var result = moveService.MoveBack(mockMovableGameObject);

        // Assert
        Assert.IsType<MovableGameObject>(result);
        Assert.Equal(0, result.Speed);
    }
}
}

```

## Код программы для MainWindow.xaml

```

<Window x:Class="PSP.GameClient.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:glWpfControl="clr-namespace:OpenTK.Wpf;assembly=GLWpfControl"
        xmlns:local="clr-namespace:PSP.GameClient"
        mc:Ignorable="d"
        KeyDown="OpenTkControl_OnKeyDown"
        Title="MainWindow" Height="630" Width="1080">
    <Grid>
        <glWpfControl:GLWpfControl
            x:Name="OpenTkControl"
            Render="OpenTkControl_OnRender"
            Ready="OpenTkControl_OnReady"/>
        <Button x:Name="button1" Click="EndGame_Button_Click" Margin="995,528,22,10.5">
            <Image Source="D:\Studies\4_kurs_1_sem\KursWork\PSP\Game\Game\PSP.GameClient\Sprite\exitGame.png"
                Height="50" Width="46"/></Image>
        </Button>
    </Grid>
</Window>

```

## Код программы для MainWindow.xaml.cs

```

namespace PSP.GameClient
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
    }
}

```



```

{
    private readonly IClientService _clientService = new ClientService();

    public MainWindow()
    {
        InitializeComponent();

        var settings = new GLWpfControlSettings
        {
            MajorVersion = 3,
            MinorVersion = 6
        };
        OpenTkControl.Start(settings);
    }

    /// <summary>
    /// Drawing game objects and checking the player for victory
    /// </summary>
    /// <param name="delta">Time</param>
    private void OpenTkControl_OnRender(TimeSpan delta)
    {
        #region GL finctions

        GL.ClearColor(Color.CornflowerBlue);
        GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);
        GL.MatrixMode(MatrixMode.Projection);
        GL.LoadIdentity();
        GL.Ortho(0, Width, Height, 0, 0d, 1d);

        #endregion

        _clientService.Update();
        _clientService.Draw();

        if (_clientService.IsGameEnd())
        {
            {
                var message = _clientService.IsCurrentClientWinner() ? "You are the winner!" : "You have lost...";
                var result = MessageBox.Show(message, "Client", MessageBoxButton.OK);
                switch (result)
                {
                    {
                        case MessageBoxResult.OK:
                        {
                            _clientService.EndGame();
                            Close();
                            break;
                        }
                    }
                }
            }
        }
    }

    /// <summary>
    /// Connects the player to the server, gets the current state of the game
    /// </summary>
    private void OpenTkControl_OnReady()
    {
        #region GL functions

        GL.Enable(EnableCap.Blend);
        GL.Enable(EnableCap.Texture2D);
        GL.BlendFunc(BlendingFactor.SrcAlpha, BlendingFactor.OneMinusSrcAlpha);

        #endregion
    }
}

```

```

        var isCreated = false;

        try
        {
            isCreated = _clientService.ConnectClient();
        }
        catch
        {
            MessageBox.Show("Some problems with the server. Please try to connect later.", "Client",
                MessageBoxButton.OK);
        }

        if (!isCreated)
        {
            var result = MessageBox.Show("Game already started. Please try to connect later.", "Client",
                MessageBoxButton.YesNo);
            switch (result)
            {
                case MessageBoxResult.Yes:
                    OpenTkControl_OnReady();
                    break;
                case MessageBoxResult.No:
                    Close();
                    break;
            }
        }

        _clientService.GetGameObjects();
    }

    /// <summary>
    /// Updates the state of the game through the player's action
    /// </summary>
    /// <param name="sender">Client</param>
    /// <param name="e">Key param</param>
    private void OpenTkControl_OnKeyDown(object sender, KeyEventArgs e)
    {
        if (e.Key == Key.A || e.Key == Key.W || e.Key == Key.S || e.Key == Key.D || e.Key == Key.E)
        {
            _clientService.ClientAction(e.Key);
        }
    }

    /// <summary>
    /// End game and close window
    /// </summary>
    /// <param name="sender">Client</param>
    /// <param name="e">Route param</param>
    private void EndGame_Button_Click(object sender, RoutedEventArgs e)
    {
        _clientService.EndGame();
        Close();
    }
}
}

```

## Код программы для NetworkService.cs

```

namespace PSP.GameClient.ApiCaller
{
    public class NetworkService : INetworkService
    {
        private HttpClient _httpClient;
    }
}

```

```

public NetworkService()
{
    _httpClient = new HttpClient();
    _httpClient.BaseAddress = new Uri("http://localhost:5000");

    _httpClient.DefaultRequestHeaders.Accept.Clear();
    _httpClient.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json"));
}

public void GetShot(string clientId)
{
    _ = _httpClient.PutAsJsonAsync<object>($"api/gamer/{clientId}/shot", null).Result;
}

public Bullet[] GetBullets()
{
    var response = _httpClient.GetAsync($"api/gamer/bullets").Result;
    var content = response.Content.ReadAsStringAsync().Result;

    return JsonConvert.DeserializeObject<Bullet[]>(content);
}

public void ResetGame()
{
    _ = _httpClient.PutAsJsonAsync<object>($"api/game-object/reset", null).Result;
}

public Car GetEnemyGamer(string clientId)
{
    var response = _httpClient.GetAsync($"api/gamer/{clientId}/enemy").Result;
    var content = response.Content.ReadAsStringAsync().Result;

    return JsonConvert.DeserializeObject<Car>(content);
}

public List<GameObject> GetGameObjects(string gameId)
{
    var response = _httpClient.GetAsync($"api/game-object/{gameId}/all").Result;
    var content = response.Content.ReadAsStringAsync().Result;

    return JsonConvert.DeserializeObject<List<GameObject>>(content);
}

public List<GameObject> GetLevelRightSequence()
{
    var response = _httpClient.GetAsync($"api/game-object/level/right").Result;
    var content = response.Content.ReadAsStringAsync().Result;

    return JsonConvert.DeserializeObject<List<GameObject>>(content);
}

public List<GameObject> GetLevel()
{
    var response = _httpClient.GetAsync($"api/game-object/level").Result;
    var content = response.Content.ReadAsStringAsync().Result;

    return JsonConvert.DeserializeObject<List<GameObject>>(content);
}

public List<Car> GetCars()
{

```

```

        var response = _httpClient.GetAsync($"api/gamer").Result;
        var content = response.Content.ReadAsStringAsync().Result;

        return JsonConvert.DeserializeObject<List<Car>>(content);
    }

    public GameObject[] GetPrizes()
    {
        var response = _httpClient.GetAsync($"api/game-object/prizes").Result;
        var content = response.Content.ReadAsStringAsync().Result;

        return JsonConvert.DeserializeObject<GameObject[]>(content);
    }

    public Point[] GetPrizesState()
    {
        var response = _httpClient.GetAsync($"api/game-object/prizes/state").Result;
        var content = response.Content.ReadAsStringAsync().Result;

        return JsonConvert.DeserializeObject<Point[]>(content);
    }

    public Car CreateGamer(string clientId)
    {
        var response = _httpClient.PostAsJsonAsync($"api/gamer", clientId).Result;
        var content = response.Content.ReadAsStringAsync().Result;

        Car result = JsonConvert.DeserializeObject<Car>(content);

        return result;
    }

    public Car MoveGamer(string gameId, int direction)
    {
        var response = _httpClient
            .PutAsJsonAsync($"api/gamer/{gameId}/move/{direction}", "").Result;
        var content = response.Content.ReadAsStringAsync().Result;

        return JsonConvert.DeserializeObject<Car>(content);
    }

    public void DeleteGamer(string gameId)
    {
        _ = _httpClient.DeleteAsync($"api/gamer/{gameId}").Result;
    }

    public void UpdateGamerTexture(Car car)
    {
        _ = _httpClient.PutAsJsonAsync<Car>($"api/gamer/texture", car).Result;
    }
}

```

### **Код программы для INetworkService.cs**

```

namespace PSP.GameClient.ApiCaller
{
    public interface INetworkService
    {
        List<GameObject> GetGameObjects(string gameId);
        Car CreateGamer(string clientId);
        Car GetEnemyGamer(string currentGamerId);
        Car MoveGamer(string gameId, int direction);
    }
}

```

```

        void DeleteGamer(string clientId);
        void UpdateGamerTexture(Car car);
        List<GameObject> GetLevel();
        GameObject[] GetPrizes();
        List<Car> GetCars();
        Point[] GetPrizesState();
        void ResetGame();
        List<GameObject> GetLevelRightSequence();
        Bullet[] GetBullets();
        void GetShot(string clientId);
    }
}

```

### **Код программы для IClientService.cs**

```

namespace PSP.GameClient.Client
{
    public interface IClientService
    {
        void GetGameObjects();
        bool ConnectClient();
        void ClientAction(Key key);
        void Update();
        void Draw();
        void EndGame();
        bool IsGameEnd();
        bool IsCurrentClientWinner();
        void ResetGame();
    }
}

```

### **Код программы для ClientService.cs**

```

namespace PSP.GameClient.Client
{
    public class ClientService : IClientService
    {
        private readonly INetworkService _networkService;
        private readonly IDrawService _drawService;

        private GameObject _bg;
        private GameObject[] _gamePrizes;
        private Bullet[] _bullets;
        private List<GameObject> _level;
        private List<GameObject> _levelRightSequence;
        private Car _gamer;
        private Car _enemyGamer;
        private bool isGamerCreated;

        private int _enemyTextureId;
        private int _tireTextureId;
        private int _fuelTextureId;
        private int _cartridgeTextureId;
        private int _bgTextureId;

        private int _explosionTextureId;

        Timer timer;

        public ClientService()
        {
            _networkService = new NetworkService();
            _drawService = new DrawService();

```

```

        _bg = new GameObject()
        {
            PositionY = 315,
            PositionX = 540,
            SizeX = 1080,
            SizeY = 630,
        };
    }

    public bool IsGameEnd()
    {
        if (_enemyGamer == null)
        {
            return false;
        }

        return _gamer.RightLevelsSequence >= ItemsScope.RightLevelsSequence
            || _enemyGamer.RightLevelsSequence >= ItemsScope.RightLevelsSequence;
    }

    public bool IsCurrentClientWinner()
    {
        return _gamer.RightLevelsSequence >= ItemsScope.RightLevelsSequence;
    }

    public void ResetGame()
    {
        _networkService.ResetGame();
    }

    private void UpdatePrizes(object obj)
    {
        var state = _networkService.GetPrizesState();
        for (int i = 0; i < _gamePrizes.Length; i++)
        {
            _gamePrizes[i].PositionX = state[i].PositionX;
            _gamePrizes[i].PositionY = state[i].PositionY;
            _gamePrizes[i].IsDeactivate = state[i].IsDeactivate;
        }
    }

    public void GetGameObjects()
    {
        _level = _networkService.GetLevel();
        _levelRightSequence = _networkService.GetLevelRightSequence();
        _gamePrizes = _networkService.GetPrizes();
    }

    /// <summary>
    /// Since client starts
    /// </summary>
    /// <returns></returns>
    public bool ConnectClient()
    {
        var clientId = Guid.NewGuid().ToString();

        var carsCount = _networkService.GetCars().Count;
        if (carsCount == 2)
        {
            return false;
        }
    }

```

```

_gamer = _networkService.CreateGamer(clientId);

float height = 0;
float width = 0;
_gamer.SpriteId = _drawService.LoadSprite("carGreen.png", out height, out width);
_gamer.SpriteSizeX = width;
_gamer.SpriteSizeY = height;
_networkService.UpdateGamerTexture(_gamer);

_enemyTextureId = _drawService.LoadSprite("carRed.png", out height, out width);
_cartriggeTextureId = _drawService.LoadSprite("bullet.png", out height, out width);
_tireTextureId = _drawService.LoadSprite("tire.png", out height, out width);
_fuelTextureId = _drawService.LoadSprite("fuel.png", out height, out width);
_bg.SpriteId = _drawService.LoadSprite("RACE2.png", out height, out width);

isGamerCreated = _gamer != null;

return isGamerCreated;
}

public void ClientAction(Key key)
{
    var keyValue = KeyToCode(key);

    if (keyValue == 5)
    {
        _networkService.GetShot(_gamer.Id);
    }
    else
    {
        _gamer = _networkService.MoveGamer(_gamer.Id, keyValue);
    }
}

public void Update()
{
    if(isGamerCreated)
    {
        _gamer = _networkService.MoveGamer(_gamer.Id, 0);
        _enemyGamer = _networkService.GetEnemyGamer(_gamer.Id);
        UpdatePrizes(null);
        _bullets = _networkService.GetBullets();
    }
}

public void Draw()
{
    if (isGamerCreated)
    {
        _drawService.Draw(_bg, Color.White, _bg.SpriteId);

        foreach (var obj in _gamePrizes)
        {
            if (!obj.IsDeactivate)
            {
                if (obj.Name == GameObjectType.Cartridge)
                {
                    _drawService.Draw(obj, Color.White, _cartriggeTextureId);
                }
                else if (obj.Name == GameObjectType.Tire)
                {
                    _drawService.Draw(obj, Color.White, _tireTextureId);
                }
            }
        }
    }
}

```

```

        else if (obj.Name == GameObjectType.Fuel)
        {
            _drawService.Draw(obj, Color.White, _fuelTextureId);
        }
        else
        {
            _drawService.Draw(obj, Color.Yellow, 0);
        }
    }
}

if (_enemyGamer != null)
{
    _drawService.Draw(_enemyGamer, Color.White, _enemyTextureId);
}

if (_gamer.IsCollision)
{
    _drawService.Draw(_gamer, Color.Red, _gamer.SpriteId);
}
else
{
    _drawService.Draw(_gamer, Color.White, _gamer.SpriteId);
}

for (int i = 0; i < _bullets.Length; i++)
{
    if (!_bullets[i].IsDeactivate)
    {
        _drawService.DrawCircle(_bullets[i].PositionX,
            _bullets[i].PositionY, _bullets[i].SizeX / 2, Color.Black);
    }
}

_drawService.DrawState(_gamer);
}
}

/// <summary>
/// Player movement
/// </summary>
/// <param name="key">Key code</param>
/// <returns>Move code</returns>
private int KeyToCode(Key key)
{
    int code;
    switch (key)
    {
        case Key.W:
            code = 1;
            break;
        case Key.S:
            code = 2;
            break;
        case Key.A:
            code = 3;
            break;
        case Key.D:
            code = 4;
            break;
        case Key.E:
            code = 5;
            break;
    }
}

```



```

        default:
            code = 1;
            break;
    }

    return code;
}

public void EndGame()
{
    _networkService.DeleteGamer(_gamer.Id);
}
}
}

```

### **Код программы для IDrawService.cs**

```

namespace PSP.GameClient.Render
{
    public interface IDrawService
    {
        /// <summary>
        /// Rendering of player indicators
        /// </summary>
        /// <param name="obj"></param>
        void DrawState(Car obj);

        void Draw(GameObject gameObject, Color color, int textureId);

        void DrawRectangle(Vector2 position, Vector2 size, Color color);

        int LoadSprite(string filePath, out float height, out float width);

        void DrawCircle(float x, float y, float radius, Color color);
    }
}

```

### **Код программы для DrawService.cs**

```

namespace PSP.GameClient.Render
{
    public class DrawService : IDrawService, IDisposable
    {
        private readonly List<int> _ids;

        public DrawService()
        {
            _ids = new List<int>();
        }

        public void DrawCircle(float x, float y, float radius, Color Color)
        {
            GL.BindTexture(TextureTarget.Texture2D, 0);

            GL.Begin(PrimitiveType.TriangleFan);
            GL.Color3(Color);

            GL.Vertex2(x, y);
            for (int i = 0; i < 360; i++)
            {
                GL.Vertex2(x + Math.Cos(i) * radius, y + Math.Sin(i) * radius);
            }
        }
    }
}

```

```

    GL.End();
}

public void DrawRectangle(Vector2 Position, Vector2 Size, Color Color)
{
    GL.Begin(PrimitiveType.Quads);

    GL.Color3(Color);
    GL.Vertex3(Position.X, Position.Y, 0);
    GL.Vertex3(Position.X + Size.X, Position.Y, 0);
    GL.Vertex3(Position.X + Size.X, Position.Y + Size.Y, 0);
    GL.Vertex3(Position.X, Position.Y + Size.Y, 0);

    GL.End();
}

public void DrawEmptyRectangle(Vector2 Position, Vector2 Size, Color color)
{
    GL.LineWidth(3.5f);

    GL.Begin(PrimitiveType.LineStrip);

    GL.Color3(color);
    GL.Vertex3(Position.X, Position.Y, 0);
    GL.Vertex3(Position.X + Size.X, Position.Y, 0);
    GL.Vertex3(Position.X + Size.X, Position.Y + Size.Y, 0);
    GL.Vertex3(Position.X, Position.Y + Size.Y, 0);
    GL.Vertex3(Position.X, Position.Y, 0);

    GL.End();
}

public void DrawState(Car obj)
{
    GL.BindTexture(TextureTarget.Texture2D, 0);

    DrawRectangle(new Vector2(50, 20), new Vector2(obj.Fuel*100/obj.MaxFuel, 20), Color.Green);
    DrawEmptyRectangle(new Vector2(50, 20), new Vector2(100, 20), Color.Black);

    DrawRectangle(new Vector2(50, 50), new Vector2(obj.Cartridges*100/obj.MaxCartridges, 20),
Color.CornflowerBlue);
    DrawEmptyRectangle(new Vector2(50, 50), new Vector2(100, 20), Color.Black);

    DrawRectangle(new Vector2(50, 80), new Vector2(obj.Tire ? 20 : 0, 20), Color.Black);
    DrawEmptyRectangle(new Vector2(50, 80), new Vector2(20, 20), Color.Black);
}

public void Draw(GameObject obj, Color color, int textureId)
{
    var position = new Vector2(obj.PositionX - obj.SizeX/2, obj.PositionY - obj.SizeY/2);
    var size = new Vector2(obj.SizeX, obj.SizeY);
    var Size = size; //new Vector2(obj.SpriteSizeX, obj.SpriteSizeY);

    Vector2 center = position + size / 2;

    Vector2[] vertices = new Vector2[4]
    {
        new Vector2(0, 0),
        new Vector2(1, 0),
        new Vector2(1, 1),
        new Vector2(0, 1),
    };
};

```

```

GL.BindTexture(TextureTarget.Texture2D, textureId);
GL.Begin(PrimitiveType.Quads);

GL.Color3(color);
for (int i = 0; i < 4; i++)
{
    GL.TexCoord2(vertices[i]);
    vertices[i].X *= Size.X;
    vertices[i].Y *= Size.Y;
    vertices[i] *= new Vector2(size.X / Size.X, size.Y / Size.Y);
    vertices[i] += position;
    vertices[i] = center + Vector2.Transform(vertices[i] - center,
        Quaternion.FromEulerAngles(0, 0, obj.Angle));
    GL.Vertex2(vertices[i]);
}

GL.End();
}

public int LoadSprite(string filePath, out float height, out float width)
{
    filePath = @"D:\Studies\4_kurs_1_sem\KursWork\PSP\Game\Game\PSP.GameClient\Sprite\" + filePath;
    if (!File.Exists(filePath)) throw new Exception("File not found!");
    Bitmap bmp = new Bitmap(filePath);
    BitmapData data = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
        ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb);

    int tex;
    GL.Hint(HintTarget.PerspectiveCorrectionHint, HintMode.Nicest);

    GL.GenTextures(1, out tex);
    GL.BindTexture(TextureTarget.Texture2D, tex);

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

    bmp.UnlockBits(data);

    GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter,
        (int)TextureMinFilter.Linear);
    GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter,
        (int)TextureMagFilter.Linear);
    GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapS,
        (int)TextureWrapMode.Repeat);
    GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapT,
        (int)TextureWrapMode.Repeat);

    _ids.Add(tex);

    height = bmp.Height;
    width = bmp.Width;

    return tex;
}

public void Dispose()
{
    foreach (var id in _ids)
    {
        GL.DeleteTexture(id);
    }
}

```

```
}  
}
```

## Код программы для GameObjectController.cs

```
namespace PSP.GameAPI.Controllers  
{  
    [Route("api/game-object")]  
    [ApiController]  
    public class GameObjectController : ControllerBase  
    {  
        private readonly IGameService _gameService;  
        private readonly ILevelService _levelService;  
        private readonly IPrizeService _prizeService;  
  
        public GameObjectController(  
            IGameService gameService,  
            ILevelService levelService,  
            IPrizeService prizeService)  
        {  
            _gameService = gameService;  
            _levelService = levelService;  
            _prizeService = prizeService;  
        }  
  
        [HttpPut("reset")]  
        public IActionResult ResetGame()  
        {  
            _gameService.ResetGame();  
  
            return Ok();  
        }  
  
        [HttpGet("all")]  
        public List<GameObject> GetAllGameObjects()  
        {  
            return _gameService.GetAllObjects();  
        }  
  
        [HttpGet("prizes")]  
        public GameObject[] GetPrizes()  
        {  
            return _prizeService.GetGamePrizes();  
        }  
  
        [HttpGet("prizes/state")]  
        public Point[] GetPrizesState()  
        {  
            return _prizeService.GetPrizesState();  
        }  
  
        [HttpGet("level")]  
        public List<GameObject> GetLevel()  
        {  
            return _levelService.GetLevel().ToList();  
        }  
  
        [HttpGet("level/right")]  
        public List<GameObject> GetLevelRightSequence()  
        {  
            return _levelService.GetLevelRightSequence().ToList();  
        }  
    }  
}
```

## Код программы для GamerController.cs

```
namespace PSP.GameAPI.Controllers
{
    [Route("api/gamer")]
    [ApiController]
    public class GamerController : ControllerBase
    {
        private readonly IGameService _gameService;
        private readonly ICarService _carService;

        public GamerController(
            IGameService gameService,
            ICarService carService)
        {
            _gameService = gameService;
            _carService = carService;
        }

        [HttpPost]
        public Car Post([FromBody] string clientId)
        {
            var resultCar = _gameService.AddGamer(clientId);

            return resultCar;
        }

        [HttpGet("{clientId}/enemy")]
        public Car Get(string clientId)
        {
            return _carService.GetEnemyCar(clientId);
        }

        [HttpPut("{clientId}/shot")]
        public IActionResult PutGetShot(string clientId)
        {
            _carService.GetShot(clientId);

            return Ok();
        }

        [HttpGet("bullets")]
        public Bullet[] GetBullets()
        {
            return _carService.GetBullets();
        }

        [HttpGet]
        public List<Car> Get()
        {
            return _carService.GetCars();
        }

        [HttpPut("{clientId}/move/{direction}")]
        public Car PutMove(string clientId, int direction)
        {
            var resultCar = _carService.MoveGamer(clientId, direction);

            return resultCar;
        }

        [HttpPut("texture")]
    }
```

```

public IActionResult PutTexture([FromBody] Car car)
{
    _carService.UpdateCarTexture(car);

    return Ok();
}

[HttpDelete("{clientId}")]
public void Delete(string clientId)
{
    _carService.DeleteCar(clientId);
}
}
}

```

### **Код программы для CarService.cs**

```

namespace PSP.GameAPI.Services.CarService
{
    public class CarService : ICarService
    {
        private int maxFuel;
        private int startCartridges;
        private float maxSpeed;

        private List<Car> gamers;
        private List<Bullet> _shorts;

        private readonly IMoveService _moveService;
        private readonly IPrizeService _prizeService;
        private readonly ILevelService _levelService;

        public CarService(IMoveService moveService,
            IPrizeService prizeService, ILevelService levelService)
        {
            _moveService = moveService;
            _prizeService = prizeService;
            _levelService = levelService;

            gamers = new List<Car>();
            _shorts = new List<Bullet>();

            maxSpeed = 100;
            maxFuel = 300;
            startCartridges = 10;
        }

        public Bullet[] GetBullets()
        {
            BulletsStep();
            return _shorts.ToArray();
        }

        public Bullet GetShot(string carId)
        {
            var gamer = GetCar(carId);

            if (gamer.Cartridges <= 0)
            {
                return null;
            }

            var shot = new Bullet

```

```

    {
        Id = Guid.NewGuid().ToString(),
        SizeX = 10,
        SizeY = 10,
        Speed = 0.9f,
        SpeedChange = gamer.SpeedChange * 2,
        PositionX = gamer.PositionX,
        PositionY = gamer.PositionY,
        Angle = gamer.Angle,
        OwnerId = carId
    };

    var shotOld = _shots.FirstOrDefault(s => s.IsDeactivate);

    _shots.Remove(shotOld);

    _shots.Add(shot);

    var car = gamers.FirstOrDefault(c => c.Id.Equals(carId));
    car.Cartridges -= 1;

    return shot;
}

public void ResetCars()
{
    try
    {
        gamers[0] = CreateCar(gamers[0].Id);
        gamers[1] = CreateCar(gamers[1].Id);
    }
    catch
    {
    }
}

public Car CreateCar(string clientId)
{
    float positionX = 110; //150
    float positionY = 210; //90

    float sizeX = 36;
    float sizeY = 16;

    if (gamers.Count > 0)
    {
        positionY = gamers[0].PositionY + sizeY + 10;
    }

    var car = new Car()
    {
        Id = clientId,
        PositionX = positionX,
        PositionY = positionY,
        SizeX = sizeX,
        SizeY = sizeY,
        Speed = 0,
        Tire = true,
        MaxSpeed = maxSpeed,
        MaxFuel = maxFuel,
        SpeedChange = 2,
        Angle = 0,
    }
}

```

```

        Fuel = maxFuel / 2,
        Cartridges = 0,
        MaxCartridges = startCartridges,
        IsFailingTire = false,
    };

    var rs = _levelService.GetLevelRightSequence();
    car.LevelsSequence = new bool[rs.Length];

    return car;
}

public Car GetEnemyCar(string clientId)
{
    return gamers.FirstOrDefault(c => !c.Id.Equals(clientId));
}

public Car GetCar(string clientId)
{
    return gamers.FirstOrDefault(c => c.Id.Equals(clientId));
}

public int AddCar(Car car)
{
    gamers.Add(car);

    return gamers.Count();
}

public void UpdateCar(Car car)
{
    var originCar = gamers.FirstOrDefault(c => c.Id.Equals(car.Id));
    if (originCar != null)
    {
        originCar = car;
    }
}

public void UpdateCarTexture(Car car)
{
    var originCar = gamers.FirstOrDefault(c => c.Id.Equals(car.Id));
    if (originCar != null)
    {
        originCar.SpriteId = car.SpriteId;
        originCar.SpriteSizeX = car.SpriteSizeX;
        originCar.SpriteSizeY = car.SpriteSizeY;
    }
}

public List<Car> GetCars()
{
    return gamers;
}

public void DeleteCar(string id)
{
    var removedGamer = gamers.First(g => g.Id.Equals(id));
    gamers.Remove(removedGamer);
}

public Car MoveGamer(string clientId, int direction)
{
    var car = GetCar(clientId);

```



```

switch (direction)
{
    case 0:
    {
        {
            car = (Car)_moveService.UpdatePosition(car);
            break;
        }
    }
    case 1:
    {
        {
            car = (Car)_moveService.MoveForward(car);
            break;
        }
    }
    case 2:
    {
        {
            car = (Car)_moveService.MoveBack(car);
            break;
        }
    }
    case 3:
    {
        {
            car = (Car)_moveService.RotateRight(car);
            break;
        }
    }
    case 4:
    {
        {
            car = (Car)_moveService.RotateLeft(car);
            break;
        }
    }
}

car = CheckAndUpdateWithLevelRightSequenceCollision(car, _levelService.GetLevelRightSequence());
car = CheckAndUpdateWithPrizeCollision(car, _prizeService.GetGamePrizes());
car = CheckAndUpdateWithBulletCollision(car, _shorts.ToArray());

var isLevelCollision = CheckAndUpdateWithLevelCollision(ref car, _levelService.GetLevel());
if (isLevelCollision)
{
    UpdateCar(car);
    return car;
}

var enemy = GetEnemyCar(car.Id);
var isEnemyCollision = CheckAndUpdateWithEnemyCollision(ref car, enemy);
if (isLevelCollision)
{
    UpdateCar(car);
    return car;
}

UpdateCar(car);

return car;
}

private void BulletsStep()
{
    for (int i = 0; i < _shorts.Count; i++)
    {
        {
            if (!_shorts[i].IsDeactivate)
            {
                string collisionObjId = null;
                var isCollision = CollisionHelper.CheckCollision(_shorts[i],
                    out collisionObjId, _levelService.GetLevel());
            }
        }
    }
}

```

```

        if (isCollizion)
        {
            _shorts[i].IsDeactivate = true;
        }
        else
        {
            _shorts[i] = (Bullet)_moveService.UpdatePosition(_shorts[i]);
        }
    }
}

private bool CheckAndUpdateWithEnemyCollision(ref Car car, Car enemy)
{
    string collisionObjId = null;

    var IsCollision = CollisionHelper.CheckCollision(car, out collisionObjId, enemy);
    if (IsCollision)
    {
        car = (Car)_moveService.ReturnPreviousState(car);
    }

    car.IsCollision = IsCollision;
    return IsCollision;
}

private Car CheckAndUpdateWithBulletCollision(Car car, Bullet[] bullets)
{
    string collisionObjId = null;

    var isCollizion = CollisionHelper.CheckCollision(car, out collisionObjId, bullets);

    if (isCollizion)
    {
        var bullet = _shorts.FirstOrDefault(s => s.Id.Equals(collisionObjId));

        if (bullet != null && !bullet.OwnerId.Equals(car.Id) && !bullet.IsDeactivate)
        {
            car.IsCollision = isCollizion;
            car.Tire = false;
            car.Speed = car.Speed < 0.4f ? car.Speed : 0.4f;

            bullet.IsDeactivate = true;
        }
    }

    return car;
}

private Car CheckAndUpdateWithLevelRightSequenceCollision(Car car, GameObject[] levelRight)
{
    string collisionObjId = null;

    var isCollision = CollisionHelper.CheckCollision(car, out collisionObjId, levelRight);
    if (isCollision)
    {
        var id = int.Parse(collisionObjId);

        if (id == 1)
        {
            car.LevelsSequence[4] = false;
        }
    }
}

```

```

        if (car.LevelsSequence.Where(l => l == false).Count() == 0)
        {
            car.RightLevelsSequence += 1;

            car.LevelsSequence = new bool[levelRight.Length];
        }
        else
        {
            car.LevelsSequence[id] = true;
        }
    }

    return car;
}

private bool CheckAndUpdateWithLevelCollision(ref Car car, GameObject[] levels)
{
    string collisionObjId = null;

    var IsCollision = CollisionHelper.CheckCollision(car, out collisionObjId, levels);
    if (IsCollision)
    {
        car.Speed = 0.1f;
        car = (Car)_moveService.ReturnPreviousState(car);
    }

    car.IsCollision = IsCollision;
    return IsCollision;
}

private Car CheckAndUpdateWithPrizeCollision(Car car, GameObject[] prizes)
{
    string collisionObjId = null;

    var isPrizeCollizion = CollisionHelper.CheckCollision(car, out collisionObjId, prizes);
    if (isPrizeCollizion)
    {
        if (car.PrizeId == null || !car.PrizeId.Equals(collisionObjId))
        {
            car.PrizeId = collisionObjId;

            var prize = _prizeService.GetGamePrize(collisionObjId);
            if (prize != null && !prize.IsDeactivate)
            {
                car = Decorate(car, prize.Name);
            }

            _prizeService.UpdateGamePrize(collisionObjId, true);
        }
    }

    return car;
}

public Car Decorate(Car car, GameObjectType type)
{
    switch (type)
    {
        case GameObjectType.Fuel:
        {
            car = (car.Fuel + 50) <= car.MaxFuel ? new FuelCarDecorator(car, car.Fuel + 50).GetCar() : new
FuelCarDecorator(car, car.MaxFuel).GetCar();
        }
    }
}

```

```

        break;
    }
    case GameObjectType.Cartridge:
    {
        car = (car.Cartridges + 2) <= car.MaxCartridges ? new CartridgeCarDecorator(car, car.Cartridges +
2).GetCar() : new CartridgeCarDecorator(car, car.MaxCartridges).GetCar();
        break;
    }
    case GameObjectType.Tire:
        car = new TireCarDecorator(car, true).GetCar();
        break;
    default:
        break;
    }
    return car;
}
}
}

```

### **Код программы для ICarService.cs**

```

namespace PSP.GameAPI.Services.CarService
{
    public interface ICarService
    {
        public Car CreateCar(string clientId);
        public int AddCar(Car car);
        public Car GetCar(string clientId);
        public Car GetEnemyCar(string clientId);
        public List<Car> GetCars();
        public void UpdateCar(Car car);
        public void UpdateCarTexture(Car car);
        public void DeleteCar(string id);
        public Car MoveGamer(string clientId, int direction);
        public void ResetCars();
        public Bullet[] GetBullets();
        public Bullet GetShot(string carId);
    }
}

```

### **Код программы для IGameService.cs**

```

namespace PSP.GameAPI.Services.GameService
{
    public interface IGameService
    {
        public Car AddGamer(string clientId);
        public List<GameObject> GetAllObjects();
        public void ResetGame();
    }
}

```

### **Код программы для GameService.cs**

```

namespace PSP.GameAPI.Services.GameService
{
    public class GameService : IGameService
    {
        private bool _isGameStarted;
        private Timer prizeTimer;

        private readonly ICarService _carService;
        private readonly IPrizeService _prizeService;
    }
}

```

```

private readonly ILevelService _levelService;

public GameService(ICarService carService,
    IPrizeService prizeService, ILevelService levelService)
{
    _carService = carService;
    _prizeService = prizeService;
    _levelService = levelService;

    _isGameStarted = false;

    prizeTimer = new Timer(new TimerCallback(RefreshPrizes), null, 0, 60000);
    RefreshPrizes(null);
}

public List<GameObject> GetAllObjects()
{
    var gameObjects = new List<GameObject>(_prizeService.GetGamePrizes());
    gameObjects = gameObjects.Concat(_levelService.GetLevel()).ToList();
    gameObjects = gameObjects.Concat(_carService.GetCars()).ToList();

    return gameObjects;
}

private void RefreshPrizes(object obj)
{
    var gameobj = GetAllObjects().ToArray();
    _prizeService.RefreshPrizes(gameobj);
}

public Car AddGamer(string clientId)
{
    var count = _carService.GetCars().Count;

    Car car;
    if (count < 2)
    {
        car = _carService.CreateCar(clientId);
        _ = _carService.AddCar(car);
    }
    else
    {
        car = _carService.GetCars().FirstOrDefault();
    }

    if (count == 1)
    {
        StartGame();
    }

    return car;
}

public void ResetGame()
{
    RefreshPrizes(null);

    _carService.ResetCars();
}

private void StartGame() { }
}

```

## Код программы для PositionHelper.cs

```
namespace PSP.GameAPI.Services.GameService
{
    public static class PositionHelper
    {
        private static Random _random = new Random();

        public static GameObject RandomNoCollisionPosition(GameObject currentObj, GameObject[] gameObjects)
        {
            string collisionObjId = null;

            do
            {
                currentObj = RandomPosition(currentObj);
            }
            while (CollisionHelper.CheckCollision(currentObj, out collisionObjId, gameObjects));

            return currentObj;
        }

        private static GameObject RandomPosition(GameObject obj)
        {
            var position = RandomPosition(new Vector2(50, 50), new Vector2(1100, 600));

            obj.PositionX = position.X;
            obj.PositionY = position.Y;

            return obj;
        }

        private static Vector2 RandomPosition(Vector2 min, Vector2 max)
        {
            Vector2 cord = new Vector2();

            cord.X = (float)(_random.NextDouble() * (max.X - min.X) + min.X);
            cord.Y = (float)(_random.NextDouble() * (max.Y - min.Y) + min.Y);

            return cord;
        }
    }
}
```

## Код программы для IMoveService.cs

```
namespace PSP.GameAPI.Services.MoveService
{
    public interface IMoveService
    {
        public MovableGameObject RotateLeft(MovableGameObject moveObject);
        public MovableGameObject RotateRight(MovableGameObject moveObject);
        public MovableGameObject MoveBack(MovableGameObject moveObject);
        public MovableGameObject MoveForward(MovableGameObject moveObject);
        public MovableGameObject UpdatePosition(MovableGameObject moveObject);
        public MovableGameObject ReturnPreviousState(MovableGameObject moveObject);
    }
}
```

## Код программы для MoveService.cs

```
namespace PSP.GameAPI.Services.MoveService
{
    public class MoveService : IMoveService
```

```

{
    public MovableGameObject RotateLeft(MovableGameObject moveObject)
    {
        moveObject.Angle += 0.08f;

        return moveObject;
    }

    public MovableGameObject RotateRight(MovableGameObject moveObject)
    {
        moveObject.Angle -= 0.08f;

        return moveObject;
    }

    public MovableGameObject MoveBack(MovableGameObject moveObject)
    {
        if (moveObject.Speed > -moveObject.MaxSpeed / 2 && moveObject.Fuel > 0 && moveObject.Tire) //вниз,
движение назад
        {
            moveObject.Speed -= 0.04f;
            moveObject.Fuel -= Math.Abs(moveObject.Speed);
        }

        return moveObject;
    }

    public MovableGameObject MoveForward(MovableGameObject moveObject)
    {
        if (moveObject.Speed < moveObject.MaxSpeed && moveObject.Fuel > 0 && moveObject.Tire) //вверх,
вперед
        {
            moveObject.Speed += 0.04f;
            moveObject.Fuel -= Math.Abs(moveObject.Speed);
        }

        return moveObject;
    }

    public MovableGameObject UpdatePosition(MovableGameObject moveObject)
    {
        var vector = Vector2.Transform(Vector2.UnitX,
            Quaternion.FromEulerAngles(0, 0, moveObject.Angle)) * (moveObject.Speed * moveObject.SpeedChange);

        moveObject.OldPositionX = moveObject.PositionX;
        moveObject.OldPositionY = moveObject.PositionY;

        moveObject.PositionX += vector.X;
        moveObject.PositionY += vector.Y;

        return moveObject;
    }

    public MovableGameObject ReturnPreviousState(MovableGameObject moveObject)
    {
        moveObject.PositionX = moveObject.OldPositionX;
        moveObject.PositionY = moveObject.OldPositionY;

        return moveObject;
    }
}

```

## Код программы для IPrizeService.cs

```
namespace PSP.GameAPI.Services.PrizeService
{
    public interface IPrizeService
    {
        GameObject[] GetGamePrizes();
        Point[] GetPrizesState();
        void RefreshPrizes(GameObject[] objects);
        void UpdateGamePrize(string prizeId, bool isDeactivate);
        GameObject GetGamePrize(string id);
    }
}
```

## Код программы для PrizeService.cs

```
namespace PSP.GameAPI.Services.PrizeService
{
    public class PrizeService : IPrizeService
    {
        private GameObject[] _gamePrizes;
        private int _prizeSize;
        private int _prizeCount;

        private readonly FuelPrizeCreator _fuelPrizeFactory;
        private readonly CartridgePrizeCreator _cartridgePrizeFactory;
        private readonly TirePrizeCreator _tirePrizeFactory;

        public PrizeService()
        {
            _tirePrizeFactory = new TirePrizeCreator();
            _cartridgePrizeFactory = new CartridgePrizeCreator();
            _fuelPrizeFactory = new FuelPrizeCreator();

            _prizeSize = 20;
            CreatePrizes();
        }

        private void CreatePrizes()
        {
            _gamePrizes = new GameObject[]
            {
                _fuelPrizeFactory.GetObject(_prizeSize, _prizeSize),
                _fuelPrizeFactory.GetObject(_prizeSize, _prizeSize),
                _fuelPrizeFactory.GetObject(_prizeSize, _prizeSize),
                _cartridgePrizeFactory.GetObject(_prizeSize, _prizeSize),
                _cartridgePrizeFactory.GetObject(_prizeSize, _prizeSize),
                _tirePrizeFactory.GetObject(_prizeSize, _prizeSize),
                _tirePrizeFactory.GetObject(_prizeSize, _prizeSize),
                _tirePrizeFactory.GetObject(_prizeSize, _prizeSize),
            };

            _prizeCount = _gamePrizes.Length;

            for (int i = 0; i < _gamePrizes.Length; i++)
            {
                _gamePrizes[i].IsDeactivate = true;
                _gamePrizes[i].Id = Guid.NewGuid().ToString();
            }
        }

        public void UpdateGamePrize(string prizeId, bool isDeactivate)
```



```

{
    var prize = _gamePrizes.FirstOrDefault(p => p.Id.Equals(priseId));

    if(prize != null)
    {
        prize.IsDeactivate = isDeactivate;
    }
}

public GameObject[] GetGamePrizes()
{
    return _gamePrizes;
}

public Point[] GetPrizesState()
{
    Point[] points = new Point[_prizeCount];

    for (int i = 0; i < _prizeCount; i++)
    {
        points[i] = new Point()
        {
            PositionX = _gamePrizes[i].PositionX,
            PositionY = _gamePrizes[i].PositionY,
            IsDeactivate = _gamePrizes[i].IsDeactivate,
        };
    }

    return points;
}

public void RefreshPrizes(GameObject[] objects)
{
    for (int i = 0; i < _gamePrizes.Length; i++)
    {
        _gamePrizes[i].IsDeactivate = false;
        _gamePrizes[i] = PositionHelper.RandomNoCollisionPosition(_gamePrizes[i], objects);
    }
}

public GameObject GetGamePrize(string id)
{
    return _gamePrizes.FirstOrDefault(p => p.Id.Equals(id));
}
}

```

### **Код программы для ILevelService.cs**

```

namespace PSP.GameAPI.Services.LevelService
{
    public interface ILevelService
    {
        public void CreateLevel();
        public GameObject[] GetLevel();
        public GameObject[] GetLevelRightSequence();
    }
}

```

### **Код программы для LevelService.xaml**

```

namespace PSP.GameAPI.Services.LevelService
{

```

```

public class LevelService : ILevelService
{
    private GameObject[] _levelObjects;
    private GameObject[] _levelsRightSequence;

    public LevelService()
    {
        CreateLevel();
        CreateLevelRightSequence();
    }

    public void CreateLevelRightSequence()
    {
        Vector2[] points = new Vector2[]
        {
            new Vector2(200,200),
            new Vector2(600,200),
            new Vector2(200,500),
            new Vector2(600,500),
            new Vector2(100,300),
        };

        _levelsRightSequence = new GameObject[points.Length];
        for (int i = 0; i < points.Length-1; i++)
        {
            _levelsRightSequence[i] = new GameObject()
            {
                Id = i.ToString(),
                PositionX = points[i].X,
                PositionY = points[i].Y,
                SizeX = 1,
                SizeY = 100
            };
        }

        int ii = points.Length - 1;
        _levelsRightSequence[ii] = new GameObject()
        {
            Id = ii.ToString(),
            PositionX = points[ii].X,
            PositionY = points[ii].Y,
            SizeX = 100,
            SizeY = 1
        };
    }

    public void CreateLevel()
    {
        Vector2[] positions =
        {
            new Vector2(0,0),
            new Vector2(0,100),
            new Vector2(0,200),
            new Vector2(0,300),
            new Vector2(0,400),
            new Vector2(0,500),
            new Vector2(0,600),
            new Vector2(100,0),
            new Vector2(200,0),
            new Vector2(300,0),
            new Vector2(400,0),
            new Vector2(500,0),
            new Vector2(600,0),
        }
    }
}

```

```

        new Vector2(700,0),
        new Vector2(800,0),
        new Vector2(900,0),
        new Vector2(1000,0),
        new Vector2(1100,0),
        new Vector2(1100,100),
        new Vector2(1100,200),
        new Vector2(1100,300),
        new Vector2(1100,400),
        new Vector2(1100,500),
        new Vector2(1100,600),
        new Vector2(100,600),
        new Vector2(200,600),
        new Vector2(300,600),
        new Vector2(400,600),
        new Vector2(500,600),
        new Vector2(600,600),
        new Vector2(700,600),
        new Vector2(800,600),
        new Vector2(900,600),
        new Vector2(1000,600),
        new Vector2(1100,600),

        new Vector2(100,100),
        new Vector2(200,100),
        new Vector2(600,100),

};

_levelObjects = new GameObject[positions.Length];
for (int i = 0; i < positions.Length; i++)
{
    _levelObjects[i] = new GameObject()
    {
        Id = Guid.NewGuid().ToString(),
        PositionX = positions[i].X,
        PositionY = positions[i].Y,
        SizeX = 100, //sizes[i].X,
        SizeY = 100 //sizes[i].Y
    };
}
}

public GameObject[] GetLevel()
{
    return _levelObjects;
}

public GameObject[] GetLevelRightSequence()
{
    return _levelsRightSequence;
}
}
}

```

## **ПРИЛОЖЕНИЕ Б**

(обязательное)

### **Руководство пользователя**

#### **1. Введение.**

Программный продукт является игровым приложением для компьютеров на базе операционной системы *ОС Windows*.

Игровое приложение обладает следующим функционалом:

- игровая область, разделенная для каждого из игроков на несколько обособленных клиентов;
- передвижение игроков кольцевой автотрассе;
- случайная генерация призов;
- отслеживание хода окончания игры.

Для использования игрового приложения пользователю не обязательно иметь специальные навыки.

Для использования программного приложения пользователь должен быть ознакомлен с:

- настоящим руководством пользователя;
- правилами использования ЭВМ.

#### **2. Назначение и условия применения.**

Разработанное программное приложение предназначено для совместной игры двух игроков с помощью сети Интернет. Игровое приложение развивает скорость реакции, концентрацию и внимание.

Для успешного запуска программы в соответствии с назначением необходимо соблюдение следующих условий:

- операционная система *Windows 7* и выше;
- наличие источника ввода информации (клавиатуры), подключенной к ЭВМ;
- наличие источника вывода графической информации (монитора), подключенного к ЭВМ.

#### **3. Подготовка к работе.**

Приложение запускается путём открытия файла *PSP.GameApi.exe*. Затем запускается файл *PSP.GameClient.exe*. Также на компьютере должны быть установлены драйвера для видеокарты. Если все инструкции соблюдены и приложение не выдаёт никаких сообщений об ошибках, значит программа работает исправно.

#### **4. Описание операций.**

При запуске веб-службы инициализируются основные сервисы обработки игрового процесса в игре.

При запуске клиентского приложения инициализируется и запускается главное меню основного интерфейса программы. Из окна главного меню, пользователь может закрыть игровое приложения.

Игроки управляют персонажами с помощью клавиатуры. Для взаимодействия определены несколько функциональных клавиш. Клавиша *W* предназначена для движения игрока вперед и увеличения его скорости. Клавиши *A* и *D* необходимы для поворота игрока в пространстве. Клавиша *SPACE* отвечает за процесс стрельбы по противнику.

экрane в случайном порядке генерируются призы, увеличивающие характеристики игрока. При соприкосновениях с призами игрок увеличивает показатели, зависящие от типа бонуса.

Виды случайно генерируемых призов:

- приз повышения уровня топлива;
- приз увеличения количества боеприпасов для стрельбы;
- приз восстановления поврежденного колеса.

#### 5. Аварийные ситуации

Чтобы избежать ошибок при использовании программы, необходимо соблюдать порядок действий и условия пользования, описанные в пункте 3 данного руководства пользователя.

В случае непредвиденного «зависания» программы рекомендуется завершить процесс в диспетчере задач и запустить снова.

#### 6. Рекомендации по освоению.

Приложение не требует определенных навыков или предварительной подготовки. Все этапы работы приложения, управление игроками и навигация в графическом интерфейсе интуитивно понятны и легки в освоении.

## **ПРИЛОЖЕНИЕ В**

(обязательное)

### **Руководство программиста**

#### **1. Назначение и условия применения программы.**

Разработанное программное приложение предназначено для совместной игры двух игроков на совмещенном экране. Игровое приложение развивает скорость реакции, концентрацию и внимание.

Для успешного запуска программы в соответствии с назначением необходимо соблюдение следующих условий:

- операционная система *Windows 7* и выше;
- наличие источника ввода информации (клавиатуры), подключенной к ЭВМ;
- наличие источника вывода графической информации (монитора), подключенного к ЭВМ.

#### **2. Характеристики программы.**

Запуск игрового приложения не требует специальных настроек. Приложение работает в атематическом режиме. При запуске веб-службы инициализируются основные сервисы обработки игрового процесса в игре.

При запуске клиентского приложения инициализируется и запускается главное меню основного интерфейса программы.

#### **3. Обращение к программе.**

Приложение запускается путём открытия файла *PSP.GameApi.exe*. Затем запускается файл *PSP.GameClient.exe*. Также на компьютере должны быть установлены драйвера для видеокарты. Если все инструкции соблюдены и приложение не выдаёт никаких сообщений об ошибках, значит программа работает исправно.

#### **4. Входные и выходные данные.**

В данной программе в качестве входных данных используется ввод с клавиатуры кнопок управления игровым процессом. В качестве выходных выступает окно отображения игры.

#### **5. Сообщения.**

При проверки приложением условий на окончание игры приложение останавливает и появляется окно, отображающее победителя.

# ПРИЛОЖЕНИЕ Г

(обязательное)

## Руководство системного программиста

### 1. Общие сведения о программе.

Программный продукт является игровым приложением для компьютеров на базе операционной системы *ОС Windows*.

Игровое приложение обладает следующим функционалом:

- игровая область, разделенная для каждого из игроков на несколько обособленных клиентов;
- передвижение игроков кольцевой автотрассе;
- случайная генерация призов;
- отслеживание хода окончания игры.

Для использования игрового приложения пользователю не обязательно иметь специальные навыки.

### 2. Структура программы.

Игровое приложение логически можно разбить на несколько составляющих:

- графический интерфейс игрового приложения;
- сервис отрисовки объектов, инкапсулирующий основные методы и классы для отображения спрайтовой графики;
- библиотеку основных сущностей игры;
- веб-служба (сервер) как контроллер основных взаимодействий внутриигровых объектов.

### 3. Настройка программы.

Приложение запускается путём открытия файла *PSP.GameApi.exe*. Затем запускается файл *PSP.GameClient.exe*. Также на компьютере должны быть установлены драйвера для видеокарты. Если все инструкции соблюдены и приложение не выдаёт никаких сообщений об ошибках, значит программа работает исправно.

### 4. Проверка программы.

Проверка основных методов и компонентов программы осуществляется с помощью модульных тестов.

### 5. Дополнительные возможности.

Приложение реализует узконаправленный функционал, однако имеет большой потенциал для расширения и дальнейшей масштабируемости.

### 6. Сообщение системному программисту.

В случае непредвиденного «зависания» программы рекомендуется завершить процесс в диспетчере задач и запустить снова.

**ПРИЛОЖЕНИЕ Д**  
**(обязательное)**  
**Схема паттерна «Декоратор»**