**ВВЕДЕНИЕ**

История компьютерных игр очень обширна и разнообразна. Свое начало, компьютерные игры берут в конце 1970-х годах. Одной из первых коммерчески доступных игр в истории развития индустрии компьютерных игр стала игра под названием «*Pong*». Однако в скором времени она заполучила огромную популярность.

Компьютерные игры стали популярными из-за широкого спектра интересов, которые они обслуживают. Каждая игра состоит из комбинации нескольких различных элементов. Хорошие игры, как правило, являются теми, в которых эти элементы органично сочетаются друг с другом и в то же время достаточно хороши, чтобы их оценили сами по себе. Самым главным смыслом создания компьютерной игры является получение возможности опробовать разные характеристики, которые игрок в идеале хотел бы иметь. Хорошо разработанная компьютерная игра может способствовать вовлеченности и создавать эффективную среду обучения. В некоторых отношениях компьютерные игры могут преподавать ценные уроки и могут даже помочь улучшить умственные способности игрока. Примером такого воздействия на пользователей служат гоночные игры. Они дебютировали в середине 1970-х годов. За свою более чем 40-летнюю историю гоночный жанр разветвлялся на несколько различных направлений. Суть жанра «Линейные гонки» – заключается в управлении любым видом транспортного средства либо от первого, либо от третьего лица, в котором игрок соревнуется с компьютером или с таким же игроком. Первой игрой в этом жанре стала аркадная игра «*Space Race*», которая была выпущена в 1973 году компанией «*Atari*». В ней игрок управляет космическим кораблем и участвует в гонках с другими кораблями, уклоняясь от комет и метеоритов [1].

Актуальность данного жанра объясняется тем, что основная часть любителей поиграть в игры-гонки, пока не может сесть за руль настоящего автомобиля. Игры жанра «гонки» дают эту возможность автолюбителям почувствовать себя в роли настоящего автогонщика. Так же, так как столкновения крайне опасны, в игровых приложениях они не смертельны.

Интерес игроков к этому жанру со временем нисколько не ослабевает, ведь те, кто играют продолжительное время, начинают повышать степень реализма: приобретают специальные контроллеры, кресла и даже устанавливаю системы с большим количеством мониторов, чтобы видеть перед собой не только дорогу, но и окружающий пейзаж.

Из вышеперечисленного материала можно сделать вывод, что реализация игрового приложения в жанре «Линейные гонки» является актуальной темой, так как данный жанр разрабатывается на всех платформах и остается популярным для игроков вне зависимости от их возраста.

1. **ОБЗОР ТЕХНОЛОГИЙ ДЛЯ РАЗРАБОТКИ ИГРОВОГО ПРИЛОЖЕНИЯ**
   1. **Обзор предметной области для игрового приложения**

Линейные гонки – это один из самых первых и древних жанров среди всех существующих компьютерных игр. Данный жанр игр появился не так давно: в начале 1970-х. В это время игровые консоли не были достаточно мощными, чтобы отображать трёхмерную графику или видео, вследствие чего, игры разрабатывались преимущественно в *2D*. История авто-симуляторов началась в клубах с игровыми автоматами. Именно там зародились такие характеристики современных игр, как вид из кабины или сверху сзади, игра вдвоём и против компьютера, тактильный отклик при авариях. Игровой объект, которым могло быть любое транспортное средство, передвигался либо влево, либо вправо по дороге или другим конструкциям. Однако в основном это была игра про лавирование в потоке машин на бесконечном шоссе. В некоторых из них была возможность собирать предметы, улучшающие характеристики или зарабатывая бонусы и восполняя «жизнь» игрока [2].

В какой-то момент гонки были самым популярным жанром видеоигр. Первой игрой такого типа стала космическая гонка «*Space Race*» (рисунок 1.1).



Рисунок 1.1 – Игра *Spacе Race*

Для победы нужно было пройти сквозь астероиды, не потеряв корабль. Игра предусматривала одиночный режим или одновременную борьбу двух игроков.

Эталонной игрой, породившей множество подражателей, стала «*Pole Position*» от «*Atari*» 1982 года. Здесь использован вид сверху, который стал преимущественно использоваться для гонок в будущем (рисунок 1.2).

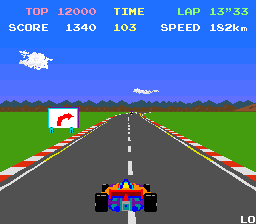


Рисунок 1.2 – Игра *Pole Position*

Вскоре процесс прохождения уровней перестал быть в основном в 2*D* и в 1989 году «*Atari*» выпустила новый авто-симулятор для игрового автомата – «*Hard Drivin’*». Она стала первой гоночной игрой с 3*D*-полигональной графикой. До этого в играх использовали масштабированные двумерные спрайты (рисунок 1.3).



Рисунок 1.3 – Игра *Wolfenstein 3D*

Таким образом, можно сделать вывод, что игры данного жанра стали довольно популярными, так как имеют богатую историю и самую разнообразный функционал.

**1.2 Технология *WPF* для реализации приложения**

Сейчас все операционные системы для персональных компьютеров реализуют взаимодействие с пользователем с помощью графического интерфейса. Это позволяет даже начинающему пользователю компьютера смело работать в среде операционной системы: проводить операции с файлами, запускать программы и так далее.Технология *WPF* представляет собой подсистему для построения графических интерфейсов.

Стандартное *Windows*-приложение при создании пользовательского интерфейса полагается на две основополагающие части операционной системы *Windows*: *User32* и *Graphical Device Interface /GDI+. User32* обеспечивает внешний вид и поведение таких элементов, как окна, кнопки, текстовые поля и т.п. *GDI/GDI+* предоставляет поддержку рисования фигур, текста и изображений за счет дополнительного усложнения, а так же часто не очень хорошей производительности. Немного позднее в *Microsoft* был разработан путь для преодоления ограничений, присущих библиотекам *User32* и *GDI/GDI+*. Этим путем является *DirectX*. Приложения *WPF* используют *DirectX* независимо от создаваемого типа пользовательского интерфейса. Это значит, что независимо от типа отображения элементов, которые применяют трехмерную графику, или просто рисуются кнопки и простой текст – вся эта работа реализуется и проходит через конвейер *DirectX*.

*WPF* позволяет разрабатывать приложения, используя как разметку, так и программный код. Основное поведение приложения заключается в реализации функциональности, которая реагирует на взаимодействие с пользователем, включая обработку событий и логики доступа к данным в ответ [3].

Такое разделение внешнего вида и поведения имеет следующие преимущества:

* затраты на разработку и обслуживание снижаются, так как разметка, которая зависит от внешнего вида, не тесно связана с кодом, специфичным для поведения;
* в основе *WPF* лежит мощная инфраструктура, основанная на *DirectX*.

*WPF* предоставляет широкий спектр возможностей по созданию интерактивных настольных приложений:

* рисования примитивов: базовые фигуры, блоки текста, также прозрачные элементы управления, возможность укладывания друг на друга множества уровней с разной степенью прозрачности и встроенная поддержка трехмерной графики;
* возможность реализации анимации;
* наличие стилей и шаблонов: стили позволяют сделать форматирование стандартным и использовать его по всему приложению, а шаблоны в свою очередь дают возможность изменить путь отображения элементов [4].

**1.3 Сравнение моделей взаимодействия сетевых игр**

Для осуществления взаимодействия по сети необходимым пунктом при выполнении работы стал выброр модели по которой будет функционировать приложение.

Существуют две основные модели взаимодействия сетевых игр: *peer-to-peer* и *client-server*. Первая модель – это *peer-to-peer* – то есть, архитектура, основанная на взаимодействии равноправных программ. Такая модель взаимодействия предусматривает равноправие каждой из запущенных копий игры.

При подобном способе взаимодействия каждая из запущенных копий игры равноправна, она обязана отслеживать состояние других копий, отправлять каждой из них сообщения. Такая модель взаимодействия подходит лишь для игр, предусматривающих небольшое количество игроков – при росте количества лавинообразно увеличивается поток сообщений, которыми они вынуждены обмениваться для поддержания игрового процесса. Как правило, в такой модели взаимодействия одна из игр назначается хостом – именно к ней должны обратиться новые игроки для того, чтобы принять участие в игре. Однако хост не управляет обменом сообщениями между играми. Схематически модель взаимодействия *peer-to-peer* представлена на рисунке 1.4.

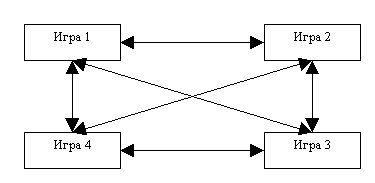


Рисунок 1.4 – Схема взаимодействия *peer-to-peer*

Достоинствами такого подхода являются: простота реализации, минимальная нагрузка на ведущего (равняющаяся нагрузке на ведомых), минимальные задержки передачи. Недостатками служат следующие: большой трафик при большом количестве игроков, практически невозможно реализовать противодействие некачественной связи, высокая нагрузка на устройство [5].

Вторая модель сетевого взаимодействия игр – это модель *client-server*. При таком взаимодействии выделяется два вида программ. Первая – это программа-сервер, на которую ложатся все заботы по организации игры и по взаимодействию с программами-клиентами. При таком подходе клиенты обмениваются сообщениями лишь с сервером, не взаимодействия друг с другом напрямую. Этот подход позволяет создавать игры, эффективно использующие пропускную способность каналов связи и поддерживающие одновременное участие в игре множества игроков. Схематически модель взаимодействия *client-server* представлена на рисунке 1.5.

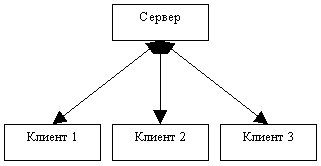


Рисунок 1.5 – Схема взаимодействия *client-server*

Каждая из игр одноранговой архитектуры является самостоятельной единицей, содержащей в себе все необходимое для полноценной организации игрового процесса. Фактически, каждая из копий игры работает самостоятельно, отсылая другим копиям информацию о своем состоянии и получая от других игр данные об их состоянии. При клиент-серверной схеме взаимодействия возможны несколько подходов. При первом каждый клиент является полноценной программой, которая выполняет все необходимые вычисления и отсылает серверу лишь их результаты. В итоге сервер превращается в инструмент для рассылки информации о состоянии игр между подключенными к нему клиентами. Второй подход заключается в большей загруженности сервера – так, например, все игровые вычисления могут вестись на сервере, а клиенты используются лишь для приема ввода пользователя и отправки его на сервер, а так же для визуализации принятых от сервера данных. Достоинствами данного подхода являются: простота реализации; малая нагрузка на сеть; реализация устранения задержек. Недостатками можно назвать следующее: высокая вычислительная нагрузка на устройство; более высокие задержки передачи.

Таким образом из вышеперечисленного материала можно сделать вывод, что наиболее удобным решением при создании игрового приложения для 2-ух игроков служит модель *peer-to-peer*.

**1.4 Сравнение протоколов *TCP* и *UDP* для сетевых коммуникаций**

Выбор протокола передачи данных был выполнен из следующих альтернатив: взаимодействие клиент-клиент – протокол *User Datagram Protocol* и взаимодействие клиент-сервер – протокол *Transmission Control Protocol*.

Протокол управления передачей данных (*TCP*) является наиболее распространенным протоколом, используемым в интернете. Он ориентирован на соединение, т.е. данные могут быть отправлены двунаправленно, как только соединение установлено. Протокол управления передачей данных используется чаще, чем протокол *UDP*. В отличие от *UDP*, она включает в себя автоматическую систему проверки ошибок, чтобы убедиться, что каждый пакет доставлен в соответствии с запросом. Протокол *UDP*, с другой стороны, не предоставляет такую систему, которая требует подтверждения того, что пакетная передача прошла успешно. Эта встроенная проверка ошибок, хотя и более эффективна, но замедляет работу по сравнению с *UDP*, так как она определяет приоритеты точной передачи данных между системами.

Протокол управления передачей работает, когда пользователи загружают веб-страницу в свой веб-браузер, что подразумевает отправку *TCP*-пакетов с компьютера на адрес хост-сервера веб-сайта. Эти пакеты посылают запрос на получение содержимого страницы сайта с хост-сервера для доступа пользователя. Поток *TCP*-пакетов отправляется пользователю или оконечному устройству в качестве ответа веб-сервера. Затем веб-браузер отвечает за склеивание этих пакетов для считывания отображаемых данных, чтобы пользователь мог их увидеть. Когда пользователи регистрируются на определенном веб-сайте или размещают комментарии в интернете, они полагаются на ПТС. Система подразумевает не только отправку пакетов, но и подтверждение того, что эти пакеты также получены. Это гарантировано системой нумерации.

Через систему, получение неверного ответа вызовет повторное получение пакетов, чтобы убедиться, что запрашивающие получатели получат правильные данные, запрошенные для получения. Пакеты, отправляемые в рамках *TCP*, делают соединение более надежным, так как эти данные отслеживаются, т.е. данные не повреждаются и не теряются при передаче. Вот почему пользователи все еще могут получить полный и бесперебойный файл даже в периоды, когда сеть испытывает икоту или помехи. В некоторых случаях, когда приемник находится в автономном режиме, компьютер посылает сообщение об ошибке, в котором говорится, что он не смог связаться с удаленным узлом.

Преимуществами данного протокола служит надежность, так как получение и подтверждение пакетов гарантировано. Он также ориентирован на подключение, т.е. требует от конечных точек создания защищенного канала перед передачей сообщений. Пакеты не будут повреждены или утеряны во время транзита. Недостаткам является следующее: он имеет низкую производительность в беспроводных сетях. Это в первую очередь потому, что беспроводные линии обычно падают пакеты во время транзита из-за «шума» на радиоканале, и что перегруженность сети является индикатором потери пакетов. Поскольку для доставки сообщения требуется полный набор пакетов, это может вызвать короткую задержку в передаче, когда пакеты теряются по пути, что приведет к увеличению времени ожидания загрузки и быстрому времени запаздывания.

*User Datagram Protocol* (*UDP*) – один из ключевых элементов набора сетевых протоколов для интернета. С *UDP* компьютерные приложения могут посылать сообщения (датаграммы) другим хостам по *IP*-сети без необходимости предварительного сообщения для установки специальных каналов передачи или путей данных. *UDP*, в отличие от *TCP*, посылает пакеты получателю независимо от того, могут ли они получить их полностью или нет. Каждый из пакетов отправляется отправителем получателю напрямую и индивидуально, без установления и подтверждения наличия надежного канала передачи данных. Пользователям не предоставляется возможность запрашивать недостающие пакеты данных после того, как они потеряны при транспортировке. Данный тип протокола используется в основном в тех случаях, когда скорость передачи данных имеет более высокий приоритет, чем надежность успешной передачи данных. Нет внутреннего порядка передачи пакетов данных, и все пакеты передаются по сети независимо друг от друга [6].

Онлайн игры реализуют аналогичную концепцию. Символы проигрывателя могут появляться при телепортации по картам, когда вы получаете новые *UDP*-пакеты при пропуске некоторых из предыдущих передач данных. Игра продолжается, и пользователям не нужно извлекать старые и потерянные пакеты. Отмена коррекции ошибок *TCP* снижает задержки и улучшает скорость игрового соединения. Отсутствие *UDP* пакетов во время игры приведет к незначительным сбоям, но не обязательно изменит ее производительность. В то время как игра продолжается в *UDP*, *TCP* зависимые игры будут иметь другой результат, который является целым замораживанием игры. В онлайн-играх важно то, что происходит в режиме реального времени.

Преимуществом данного протокола является наличие относительно более высокой скорости передачи данных благодаря небольшому весу пакетов с минимальными заголовками. Так как он не требует ответа, он подходит для видеоконференций, трансляций и игр. Недостатком служит то, что так как последовательность и подтверждение во время передачи данных отсутствуют, *UDP* считается ненадежной и небезопасной. Поврежденные пакеты удаляются, но не запрашиваются для повторной передачи, после того как они утеряны.

Таким образом из вышеперечисленного материала можно сделать вывод, что наиболее удобным решением при создании игрового приложения для 2-ух игроков служит протокол *TCP*.

**1.5 Обзор графической библиотеки *DirectX***

Достоинства библиотек становятся очевиднее при их использовании в разных, но пересекающихся, прикладных областях. *DirectX* подходит для профессиональной разработки игр и мультимедийных приложений на платформе *Windows*. Данная технология более эффективна, поскольку оперирует высокоуровневыми конструкциями вроде текстур и градиентов, которые могут отображаться непосредственно видеокартой. Таким образом, в процессе создания и написания игры графический интерфейс был реализован по средствам *DirectX*. Как было описано ранее, *DirectX* представляет собой набор классов, функций и структур, обеспечивающих высокоэффективное выполнение задач, возникающих при обработке компьютерных игр и мультимедийных приложений. Компьютерные игры, которые используют *DirectX*, эффективнее используют встроенные функции акселерации мультимедиа, благодаря чему повышается производительность выполнения мультимедийных задач.

Сам *DirectX* состоит из нескольких компонентов:

* *DirectSound* – отвечает за работу со звуком, так же используется при разработке программ для воспроизведения или захвата звукового сигнала;
* *Direct2D* – отвечает за вывод двумерной графики;
* *DirectInput* – организует работу с устройством вывода, к данным устройствам относят клавиатуру, мышь, джойстики;
* *DirectMusic* – отвечает за воспроизведение музыки, может воспроизвести файлы *Musical Instrument Digital Interface* и *Waveform Audio File Format*;
* *DirectPlay* – позволяет создавать многопользовательские игры, отвечает за обмен данными по сети.

Преимущества использования *DirectX* заключаются в следующем:

* доступность многопользовательских игр;
* изображение в играх вышло на новый уровень, *DirectX* поддерживает *Video Graphics Array* видеорежим;
* возможность доступа к *3D*-играм [7].

В ходе подбора литературных источников и изучения технологии использования *DirectX* в приложениях *WPF* была выбрана управляемая .*NET*-оболочка с открытым исходным кодом *SharpDX.*

*SharpDX* – это проект с открытым исходным кодом, который предоставляет полные *DirectX* интерфейсы прикладного программирования на платформе *.Net*. Другими словами можно сказать, что *SharpDX –* это низкоуровневая оболочка для интерфейсов *DirectX*. Платформа *.Net* позволяет разрабатывать высокопроизводительные игры для *2D* рендеринга и *3D* графики. *SharpDX* построен с помощью специального инструмента под названием *SharpGen*, который может генерироваться автоматически самой платформой непосредственно из заголовков *DirectX Software Development Kit*.

Из вышеперечисленного материала следует, что *SharpDX* служит удобным и доступным проектом с исходным кодом, так как он предоставляет все необходимые для реализации игрового приложения функции.

**1.6 Методология проектирования игрового приложения**

При разработке игрового приложения необходимым и важным пунктом стал выбор методологии проектирования программного обеспечения. Под методологией в широком смысле понимается совокупность идей, понятий, принципов, способов и средств, определяющая стиль написания, отладки и сопровождения программ.

Для реализации данного курсового проекта, из всех существующих методологий проектирования была выбрана функционально-ориентированная разработка или *Feature driven development*. *FDD* – это методология, которая объединяет лучшие практики и сосредотачивает внимание разработчика на функциональных элементах, которые являются полезными с точки зрения клиента [8].

Общая схема методологии представлена на рисунке 1.6.

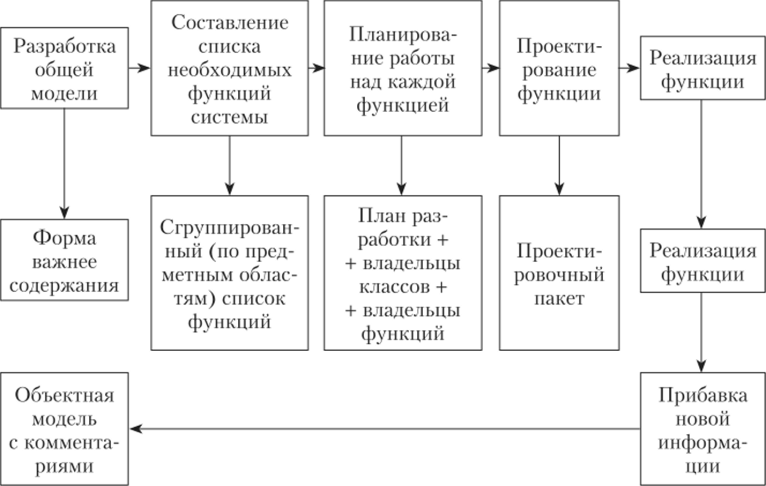


Рисунок 1.6 – Схема методологии *FDD*

Суть данного подхода заключается в разделении процесса реализации приложения на пять частей:

* разработка общей модели;
* создание списка функций, которые должна реализовывать программа;
* планирование времени;
* дизайн и разработка;
* реализация.

Общая модель предполагает собой полноценное игровое приложение, которое будет реализовать игру жанра «Линейные гонки». Разработка осуществляется на *WPF* с использованием спрайтовой графики и средств *DirectX*. Также игровое поле разделяется на две части, что делает приложение многопользовательским.

Далее следует описание списка функций, которые планируются при реализации игры:

* меню игры;
* интуитивный и удобный в эксплуатации интерфейс пользователя;
* возможность получения призов, за счет столкновения с ними;
* реализация столкновения игрока с противниками;
* возможность передвижения главных игроков, по средствам нажатия определенных клавиш;
* уведомление об окончании игры при столкновении, окончании топлива, либо прохождении игроком всей трассы.

Планирование времени заключается в разделении задачи на пункты. Первым пунктом стало создание приватного репозитория в системе контроля версий *Git*, для удобства реализации и возможности следить за изменениями в проекте. Вторым пунктом стал подбор литературы, для изучения интеграции технологий *DirectX* в приложение *WPF*. Затем следует создание структуры приложения – его основных классов, далее идет разработка графического интерфейса. Заключительным пунктом является создание модульных тестов для отладки и опытной эксплуатации приложения.

Данная методология была выбрана из-за простоты реализации и удобства эксплуатации. Главным достоинством *FDD* является возможность в любой момент оценить: отстаёт ли проект от графика или продвигается быстрее.

Из данной главы, в которой были рассмотрены предметная область приложения, а так же средства его реализации следует, что наиболее оптимальными технологиями для создания игры в жанре «Линейные гонки» служит приложение *WPF* с использованием *DirectX* технологий. Для удобства разработки была выбрана методология проектирования *FDD*.

1. **ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ «ЛИНЕЙНЫЕ ГОНКИ»**
   1. **Функциональные требования для игрового приложения**

Игровое приложение жанра «Линейные гонки» должно выполнять следующие функции: экран игрового пространства разделен на две части: игровое поле первого и второго игрока. Таким образом, пользователи будут играть друг против друга, также игровое поле представляет собой прямую автодорогу, по которой пользователи могут перемещаться влево и вправо. Дорога для игроков одна и та же. Так же в игре реализуется увеличение и уменьшение скорости по средствам переключения передач у машины. Расход топлива зависит от скорости движения. Изначально топлива недостаточно для прохождения всей трассы, поэтому в ходе прохождения игры на дороге случайным образом появляются «призы». Так же существуют и препятствия для игрока: другие машины, в которые он может врезаться и игра закончится. Победителем считается игрок, который первым доберется до финиша.

Для реализации необходимо использовать шаблоны проектирования:

* «фабричный метод» – для реализации генерации «призов»;
* «декоратор» – для изменения характеристик автомобиля.

Для отображения объектов использовать спрайтовую графику и средства *DerectX*. Для сетевых коммуникаций использовать протокол *TCP*.

* 1. **Исходные данные для разработки игрового приложения**

Создание программы осуществлялось при помощи среды разработки *Visual Studio* 2019. Для реализации данного игрового приложения были загружены графические библиотеки для работы с *DirectX* в *WPF*. Библиотеки были загружены при помощи установки и загрузки менеджера пакетов .*Net –* пакетов *NuGet*. Данные пакеты содержат повторно используемый код, который другие разработчики предоставляют другим разработчикам для использования в своих проектах. Пакеты устанавливаются в проект *Visual Studio* с помощью диспетчера пакетов *NuGet* или консоли диспетчера пакетов.

В качестве оболочки для *DirectX* в *C#* использовалась библиотека *SharpDx*. Для работы так же были использованы такие библиотеки, как *ShatpDx.Direct2D1*, для отображения двумерной графики, *SharpDx.DirectWrite* для работы с текстом, а так же *SharpDx.DXGI*, *SharpDx.Direct3D11* и *SharpDx.Direct3D.*

Для реализации сетевого взаимодействия по протоколу *TCP* использовался класс *Socket*, а для осуществления работы приложения без задержек – класс *Threading.*

* 1. **Разработка архитектуры программного приложения**

Для того, чтобы проект был грамотно реализован и выполнял все свои функции правильно, в ходе разработки приложения был применен принцип разделения крупных элементов проекта на более мелкие и легко управляемые задачи – принцип декомпозиции. С использованием методов декомпозиции определяется так называемая иерархическая структура работ. Данная структура разделяет проект на иерархически связанные, управляемые и контролируемые подзадачи.

Принцип декомпозиции имеет следующие характеристики:

* разделение проекта на подзадачи;
* описание с необходимой точностью содержание каждой подзадачи по проекту;
* формирование проекта с его задачами в виде иерархической структуры;
* оценка реалистичности выполнения проекта;
* оценка ресурсов.

В ходе применения принципа декомпозиции приложение было разделено на следующие блоки (рисунок 2.1).

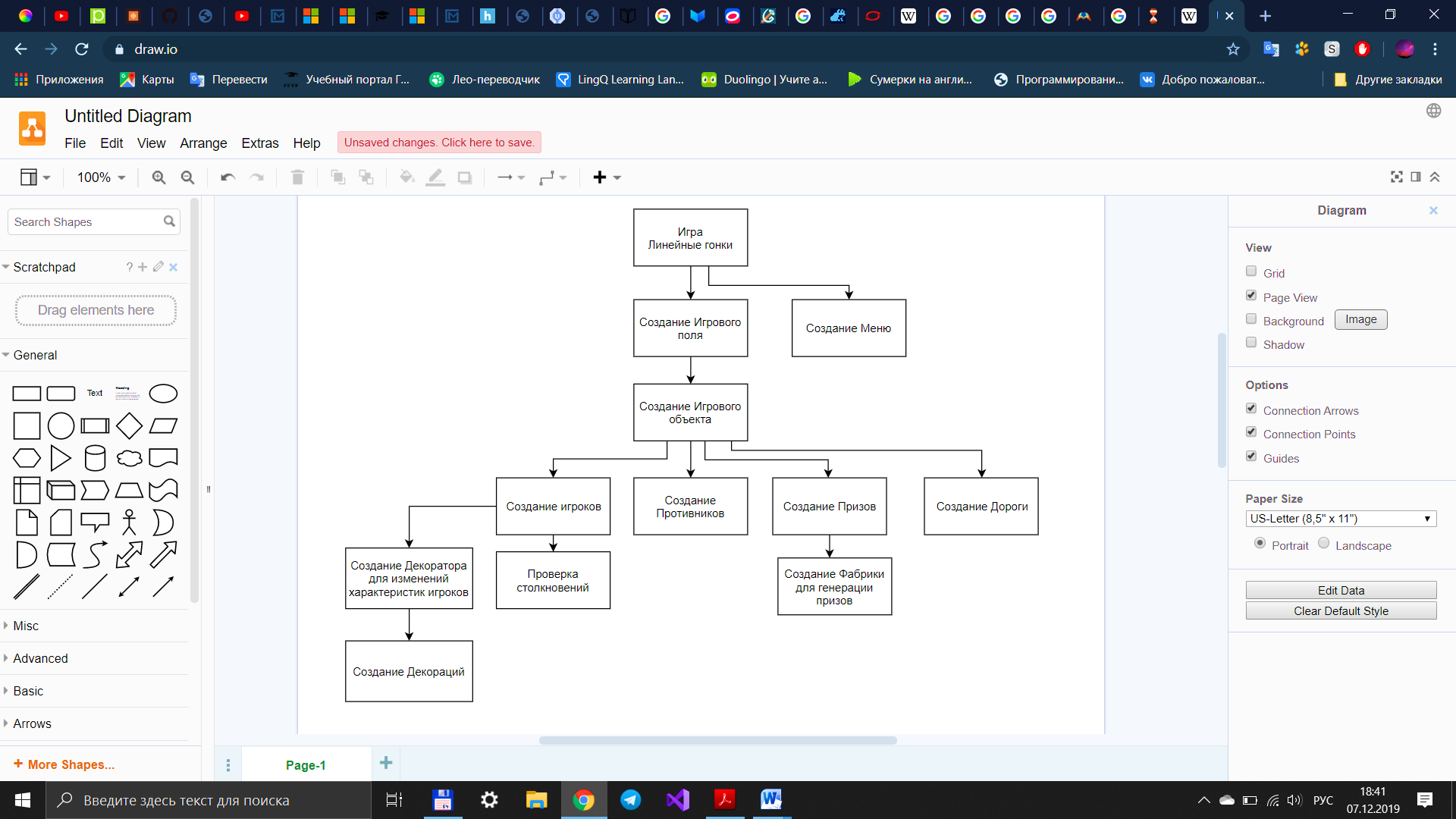


Рисунок 2.1 – Результат разделения проекта на отдельные блоки по принципу декомпозиции

Из рисунка 2.1 видно, что для расширения функционала приложения необходимо применить паттерны проектирования «фабричный метод» для удобного создания призов на игровом поле и «декоратор», для изменений характеристик игроков.

Паттерны проектирования описывают типичные способы решения часто встречающихся проблем при проектировании любого рода программ. Выбор и использование шаблонов среди различных паттернов проектирования зависит от индивидуальных потребностей и проблем. Прежде всего, при решении какой-нибудь проблемы надо выделить все используемые сущности и связи между ними и абстрагировать их от конкретной ситуации. Затем надо посмотреть, вписывается ли абстрактная форма решения задачи в определенный паттерн [8].

Первый паттерн имеет название «фабричный метод». Суть паттерна заключается в использовании механизма полиморфизма. Данный паттерн предоставляет интерфейс для создания объектов без указания их конкретных классов. Он определяет метод, который мы можем использовать для создания объекта вместо использования его конструктора.

Паттерн «фабричный метод» применяется в курсовом проекте так как:

* заранее нет информации, какие призы предстоит создавать на игрвовом поле;
* система создания призов не имеет зависимости от процесса генерации объектов, она является масштабируемой;
* создание новых видов призов нужно делегировать из базового класса классам наследникам.

Паттерн был разработан следующим образом: во-первых был создан интерфейс *IPrize*, во-вторых сама фабрика – интерфейс *IPrizeFactory*, который имеет метод *CreatePrize().* Именно от этих интерфейсов наследуются классы, которые в последующем будут создавать объекты-«призы». От интерфейса IPrizeFactory наследуются два класса: класс *FuelPrizeFactory*, который реализует поставку объектов класса *FuelPrize*, дающие игроку дополнительное топливо в течение всего игрового процесса, а так же класс *DeleteFuelPrizeFactory*, который реализует поставку объектов класса *DeleteFuelPrize*, дающие игроку возможность отнимать некоторое количество топлива у соперника.

Такая структура позволяет создать масштабируемую систему, что является важным аспектом объектно-ориентированного программирования.

Схема паттерна для игрового приложения представлена на рисунке 2.2.

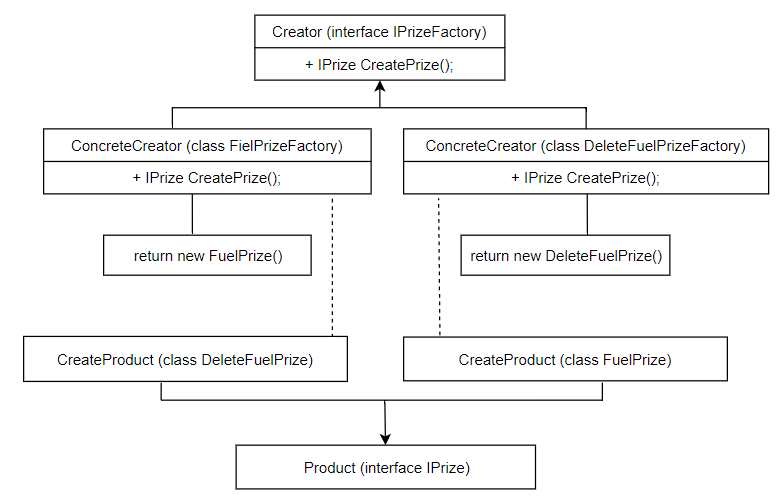


Рисунок 2.2 – Схема паттерна «фабричный метод»

Второй паттерн имеет название «декоратор». Суть паттерна заключается в том, что он позволяет динамически добавлять объектам новую функциональность, по средствам добавления им новых возможностей. Создаётся абстрактный класс, представляющий как исходный класс. Затем создаются новые классы, которые реализуют функции абстрактного класса, далее идет базовый класс, для декораторов который должен содержать объект исходного класса. И последний шаг – реализация самих классов-декораций.

Данный паттерн применяется в курсовом проекте так как:

* новые характеристики для машины имеют одинаковый интерфейс;
* нет необходимости использовать принцип наследования, так как классы имеют схожий функционал.

Данный шаблон проектирования отвечает за добавление игрокам нового функционала, таких, как: другой цвет, расход топлива для управляемого устройства или объем бензобака. Для этого был создан абстрактный класс *Player*, который имеет наследника *PlayerCar*. Данный класс является одной из возможных реализаций абстрактного класса, так как управляемое средство может быть, например трактором, лимузином или другой разновидностью транспортных устройств. Далее идет абстрактный класс *DecoratorPlayerCar*, который представляет собой декоратор. А затем идет разный функционал, который добавляется как декорация на класс. Классы *FerrariCar*, *FordCar* и *PorsheCar* декорируют стандартную машину под свои модели.

Такая структура позволяет динамически добавлять объектам новый функционал.

Схема паттерна представлена на рисунке 2.3.

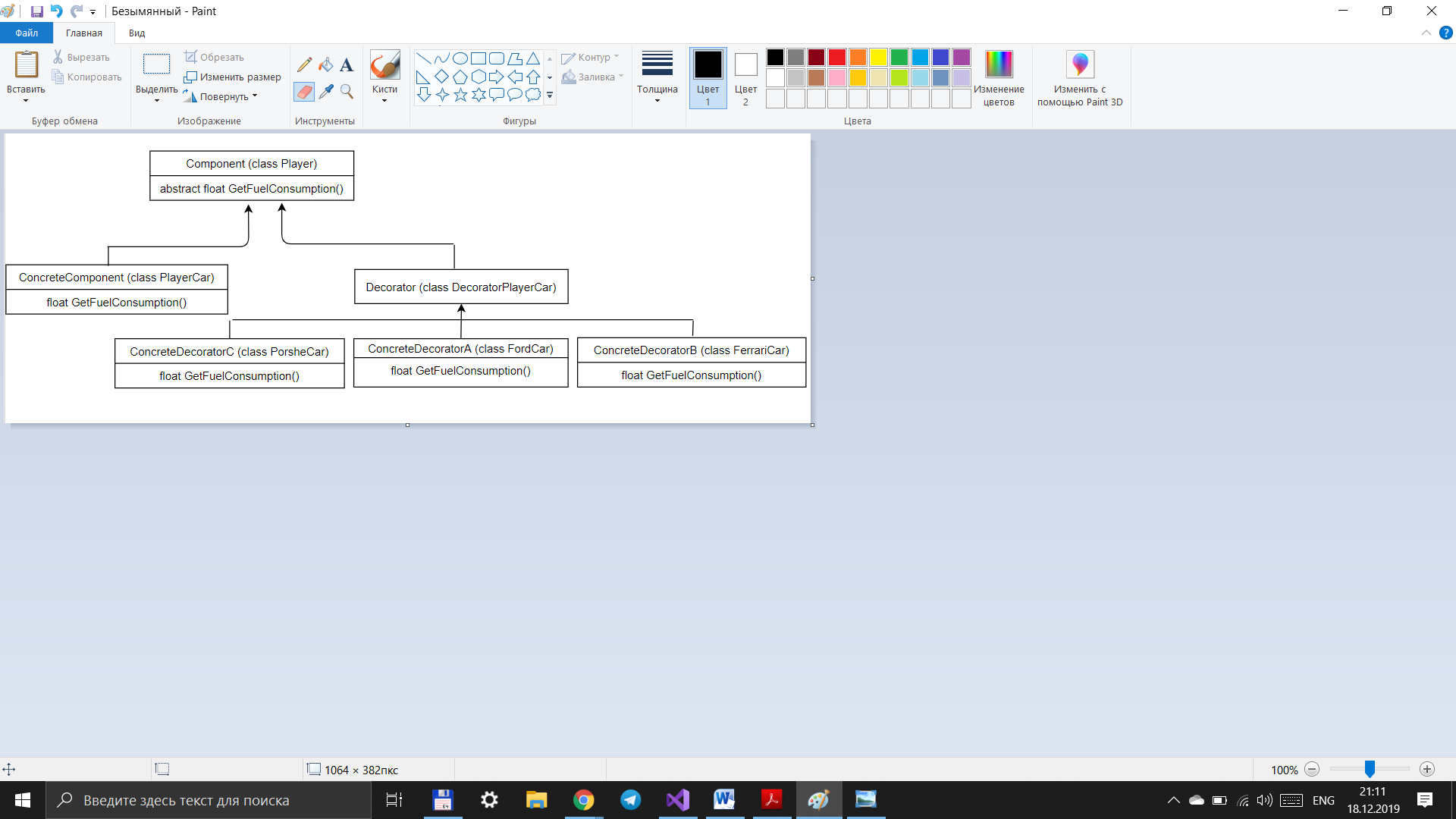


Рисунок 2.3 – Схема паттерна «декоратор»

Из ранее изложенного материала можно сделать вывод, что паттерны проектирования очень важны для решения поставленной задачи, так как они помогают сэкономить время для разработки игрового приложения, а так же код становится более стандартизированным.

* 1. **Принцип взаимодействия классов в программе**

В пункте 2.3 был рассмотрен процессе разделения задачи на подзадачи, из чего следует, что проект стал проще для понимания и реализации. Таким образом, в ходе применения принципа декомпозиции, а также методологии по проектированию программного обеспечения была разработана следующая система взаимодействия для классов проекта (рисунок 2.3).

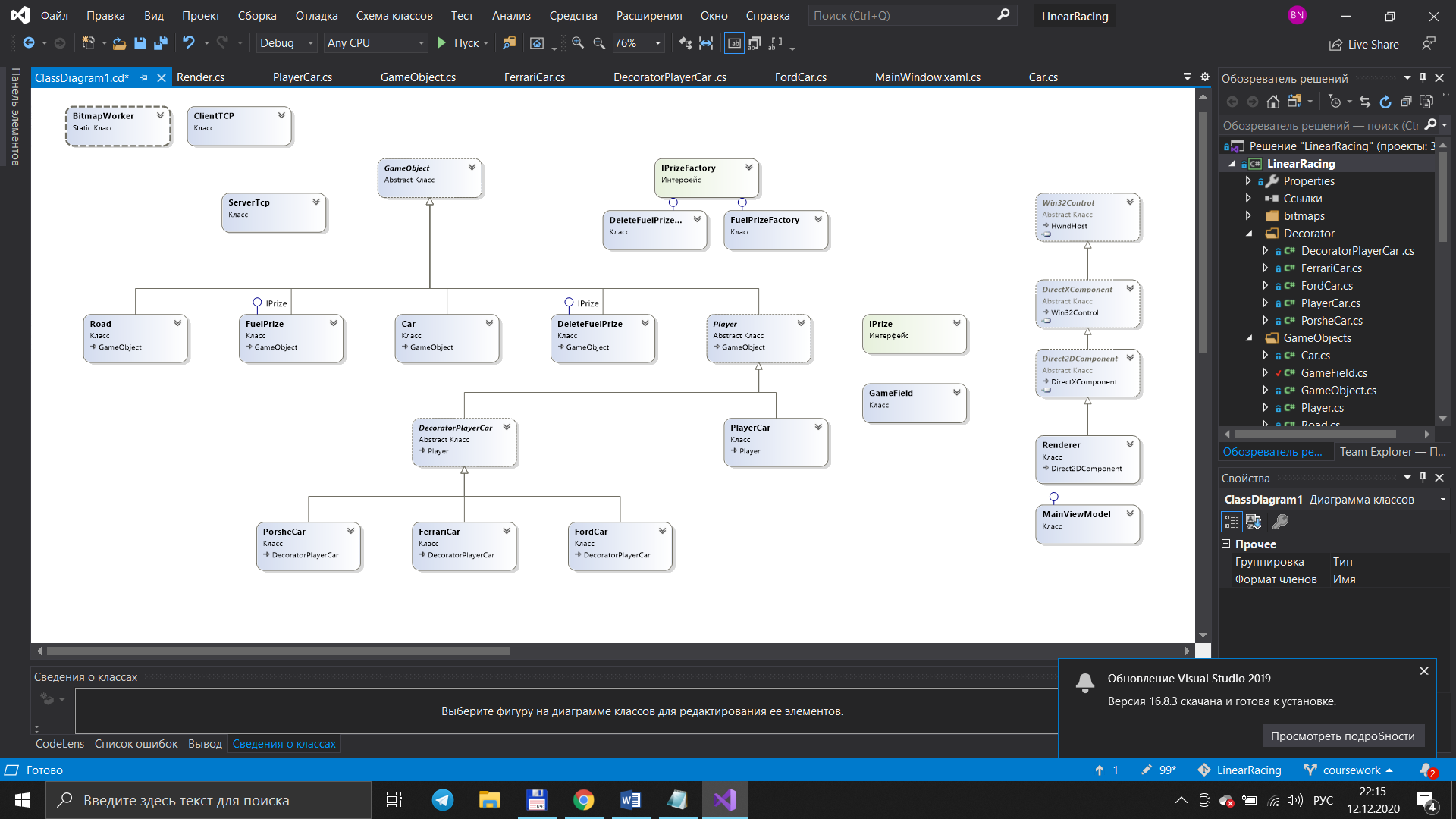


Рисунок 2.3 – Схема взаимодействия классов в программе

Первым и основным классом стал абстрактный класс *GameObject*, от которого впоследствии наследуются практически все игровые объекты. В нем существуют такие методы как *CheckCollision()*, который проверяет столкновение двух объектов, и абстрактный метод *Move()*, который реализуется в каждом наследуемом классе по-разному, в зависимости от типа объекта и его расположения.

Далее были созданы все имеющиеся на игровом поле объекты. Класс Road, отвечает за дорогу, и в себе имеет реализованный метод движения *Move()*. Видимость движения дороги на протяжении всего игрового процесса осуществляется путем перемещения блока дороги по циклу. Затем идет класс, который описывает противников, он имеет название *Car*. Перемещение данных объектов реализовано случайным образом, что делает игру каждый раз новой и непредсказуемой. И, наконец, абстрактный класс *Player*, объекты которого, в свою очередь будет выступать за игроков. Метод передвижения игроков зависит от того, на какие клавиши нажимает тот, или иной пользователь.

Для реализации паттерна «декоратор» был создан абстрактный класс *Player*, который имеет наследника *PlayerCar*. Данный класс является одной из возможных реализаций абстрактного класса, так как управляемое средство может быть, например трактором, лимузином или другой разновидностью транспортных устройств. Далее идет абстрактный класс *DecoratorPlayerCar*, который представляет собой декоратор. А затем идет разный функционал, который добавляется как декорация на класс. Классы *FerrariCar*, *FordCar* и *PorsheCar* декорируют стандартную машину под свои модели.

Паттерн «фабричный метод» создан с целью генерации призов на игровом поле. Метод был разработан следующим образом: во-первых был создан интерфейс *IPrize*, во-вторых сама фабрика – интерфейс *IPrizeFactory*, который имеет метод *CreatePrize().* Именно от этих интерфейсов наследуются классы, которые в последующем будут создавать объекты-«призы». От интерфейса IPrizeFactory наследуются два класса: класс *FuelPrizeFactory*, который реализует поставку объектов класса *FuelPrize*, дающие игроку дополнительное топливо в течение всего игрового процесса, а так же класс *DeleteFuelPrizeFactory*, который реализует поставку объектов класса *DeleteFuelPrize*, дающие игроку возможность отнимать некоторое количество топлива у соперника. Призы генерируется по мере прохождения определенных точек на карте дороги. Таким образом, у игроков равное количество шансов на прохождение игры.

Класс *GameField* содержит в себе все объекты игрового поля, а так же методы которые отвечают за состояние игрового процесса – *GameProcess().*

После создание всех имеющихся в игре сущностей, возникла необходимость отображения их на экране. Если в приложении используется несколько технологий презентации, таких как *WPF*, *Win32* или *DirectX*, они должны совместно использовать области рендеринга в общем окне верхнего уровня. Для любого дескриптора окна нельзя смешивать рендеринг *WPF* с рендерингом *DirectX*, поэтому для того, чтобы отобразить окно, с отрисованными в нем компонентами была создана библиотека классов, имеющая название *SharpDxControl*. В ее состав входит класс *Win32Control*, который наследуется от класса *HwndHost*. Этот класс оборачивает окно в элемент *WPF*, который можно добавить в дерево элементов *WPF*.

Для начала были задействованы неуправляемые функции *CreateWindowEx* и *DestroyWindow* из *Dynamic Link Library* при помощи специального механизма *Platform Invoke*. Затем были переопределлены следующие методы: *BuildWindowCore()* – с целью создания окна *Win32*, и *DestroyWindowCore()* – который необходим для уничтожения окна. Чтобы получить доступ к очереди сообщений и оконному дескриптору, переопределяется метод *WndProc* из родительского класса *HwndHost*. Также был создан делегат, который используется для разрешения обратных вызовов из собственного кода в управляемый код.

Далее был создан класс *DirectXComponent*, наследуемый от *Win32Control*, который размещается в родительском элементе управления созданного окна. В данном классе было создано все необходимое для отрисовки на экран:

* *device*, данный объект производит отрисовку кадров в указанный буфер визуализации;
* *swapchain*, цепочка переключений которая содержит в себе первичный и вторичный буфер, она отвечает за отображение на экране содержимого буфера, в которое устройство выводит изображение;
* *render* *target* *view*, объект представления данных, он служит связующим между вторичным буфером и устройством;
* *back* *buffer*, задний буфер.

Затем был создан класс *Direct2DComponent* для работы с библиотекой *Direct2D*. *Direct2D* – это аппаратно-ускоренный *2D* графический *API* для непосредственного режима, который обеспечивает высокую производительность и высокое качество рендеринга двумерной графики. Он предлагает наличие фабрики, которая была создана в данном классе и непосредственно программный рендеринг. Далее был создан класс *Render*, который наследуется от предыдущего. В нем содержится отрисовка всех объектов, которые участвуют в игровом процессе.

Следующим пунктом становится осуществление взаимодействия двух объектов: источника и приемник данных. Привязка – это механизм, позволяющий связывать некоторые свойства приемника с некоторыми свойствами объекта источника и выполнять взаимное обновление этих свойств при изменении одной из частей связки.

Игровое поле отображается в элементе типа *ContentControl*, который в свою очередь находится в *StackPanel*. Реализовывается это соединение по средствам паттерна *Model-View-ViewModel* и использования интерфейса *INotifyPropertyChanged*. Класс *MainViewModel*, который наследуется от интерфейса, описанного выше, служит связующим звеном между интерфейсом пользователя и логикой приложения. Интерфейс *INotifyPropertyChanged* служит для оповещения изменений в каждой части привязки.

Для реализации сетевого взаимодействия были созданы классы *Server* и *Client*. Класс *Server* отвечает за серверную часть, которая будет принимать подключение от клиента. На нем устанавливался порт для прослушивания входящих соединений. В классе *Client*, в свою очередь происходит подключение к серверу. Для создания данных классов были использованы сокеты. Так как передача данных идет по протоколу *TCP*, использовались потоковые сокеты. Поддержку сокетов в *.NET* обеспечивают классы в пространстве имен *System.Net.Sockets*. В классе *Server* использовались следующие методы класса *Socket*:

* *Accept()*, создание нового сокета для обработки входящего запроса на соединение с клиентом;
* *Bind()*, связывание сокета с локальной конечной точкой для ожидания входящих запросов на соединение;
* *Close()*, закрытие сокета;
* *Listen()*, помещение сокета в режим прослушивания;
* *Receive()*, получение данных от клиента;
* *Send()*, отправление данных клиенту;
* *Shutdown()*, запрещение операций отправки и получения данных на сокете.

В классе *Client*, в свою очередь, использовались следующие методы класса *Socket*:

* *Connect()*, устанавливает соединение с удаленным хостом;
* *Close()*, закрытие сокета;
* *Receive()*, получение данных от сервера;
* *Send()*, отправление данных серверу;
* *Shutdown()*, запрещение операций отправки и получения данных на сокете.

Так как оба класса равноправны между собой, в каждом из них имеются методы для передачи и получения положение игрока и машин-противников.

Передача и прием данных осуществляется в отдельном потоке для того, чтобы игра не останавливалась и работала исправно.

В заключение данной главы можно сделать вывод, что использование паттернов проектирования, а также применения принципа декомпозиции при разработке программного обеспечения оказали значительное влияние на разработку проекта в сторону создания корректной иерархии классов приложения.

1. **ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ. ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРИЛОЖЕНИЯ**
   1. **Описание работы игрового приложения**

Для реализации данного игрового приложения необходимо было загрузить графические библиотеки для работы с *DirectX* в *WPF*. Оболочкой для *DirectX* в языке программирования *C#* стала библиотека *SharpDx*. Для работы так же были использованы такие библиотеки, как *ShatpDx.Direct2D1*, для отоборажения двумерной графики, *SharpDx.DirectWrite* для работы с текстом, а так же *SharpDx.DXGI*, *SharpDx.Direct3D11* и *SharpDx.Direct3D.*

Далее была создана библиотека классов *SharpDxControl* для создания окна с игровым полем. Так же были созданы основные сущности для игрового процесса, такие как игрок – абстрактный класс *Player*. Его реализация – класс *PlayerCar*, декоратор – класс *DecoratorPlayerCar*. И ряд декораций: классы *FerrariCar*, *PorsheCar*, *FordCar*. Для реализации противников был создан класс *Car*. Затем класс *Road*, отвечающий за дорогу и классы, реализующие призы: *FuelPrize* – приз-топливо, и *DeleteFuelPrize*  – приз, который забирает топливо у противника.

Создание призов было сделано при помощи паттерна «фабричный метод», в котором, каждая из фабрик *FuelPrizeFactory* и *DeleteFuelPrizeFactory*, которые наследуются от интерфейса *IPrizeFactory.* Данные классы реализуют метод *CreatePrize()*, который поставляет призы на игровое поле. Все сущности, которые входят в игровой процесс наследуются от класса *GameObject*.

Сам процесс игры происходит в классе *GameField*. Сначала создаются все необходимые объекты: массив из противников, дорога, лист призов для возможности динамического создания и удаления каждого приза, фабрики, для поставки призов и сами игроки.

Затем следует блок инициализации, в котором всем ранее созданным объектам присваиваются начальные значения. Установкой типа игрока, в зависимости от его выбора в начале игры происходит в методе *SetPlayerCar()*. Данный метод принимает тип машины в форме строки, игрока и его индекс на игровом поле. В данном методе, в зависимости от выбора типа машины управляемому объекту назначается определенная функциональность.

Далее идет непосредственно сам игровой процесс, за это отвечает метод *GameFieldProcess()*, который принимает все сущности для каждого игрока: противники, призы, дорога, сам игрок, его индекс и выходной параметр для определения состояния игры. В данном методе запускается цикл, в процессе которого, идет движение всех противников, в нем происходит проверка на столкновение игрока и противоположной машиной, в случае столкновения игровое поле перестает функционировать и выходной параметр принимает значение 1. Если игрок достиг конца трассы, выходной параметр равен 2.

Если игрок не столкнулся и не дошел до конца трассы, то игровой процесс продолжается и запускается движение дороги, на определенных точках которой появляются призы. В зависимости от расстояния пройденной трассы, создаются разного типа призы.

Далее следует проверка столкновения с призами. В зависимости от типа приза, игрок получает разные возможности, однако если приз не сталкивается с игроком и доходит до конца игрового поля, то он автоматический удаляется с поля. В случае, когда приз находится в пределах трассы и не соприкасается с игроком, он продолжает движение

Если игрок не сталкивается с противниками и у него не заканчивается топливо, то он продолжает свое движение. Схема метода *GameFieldProcess()* представлена на рисунке 3.1.

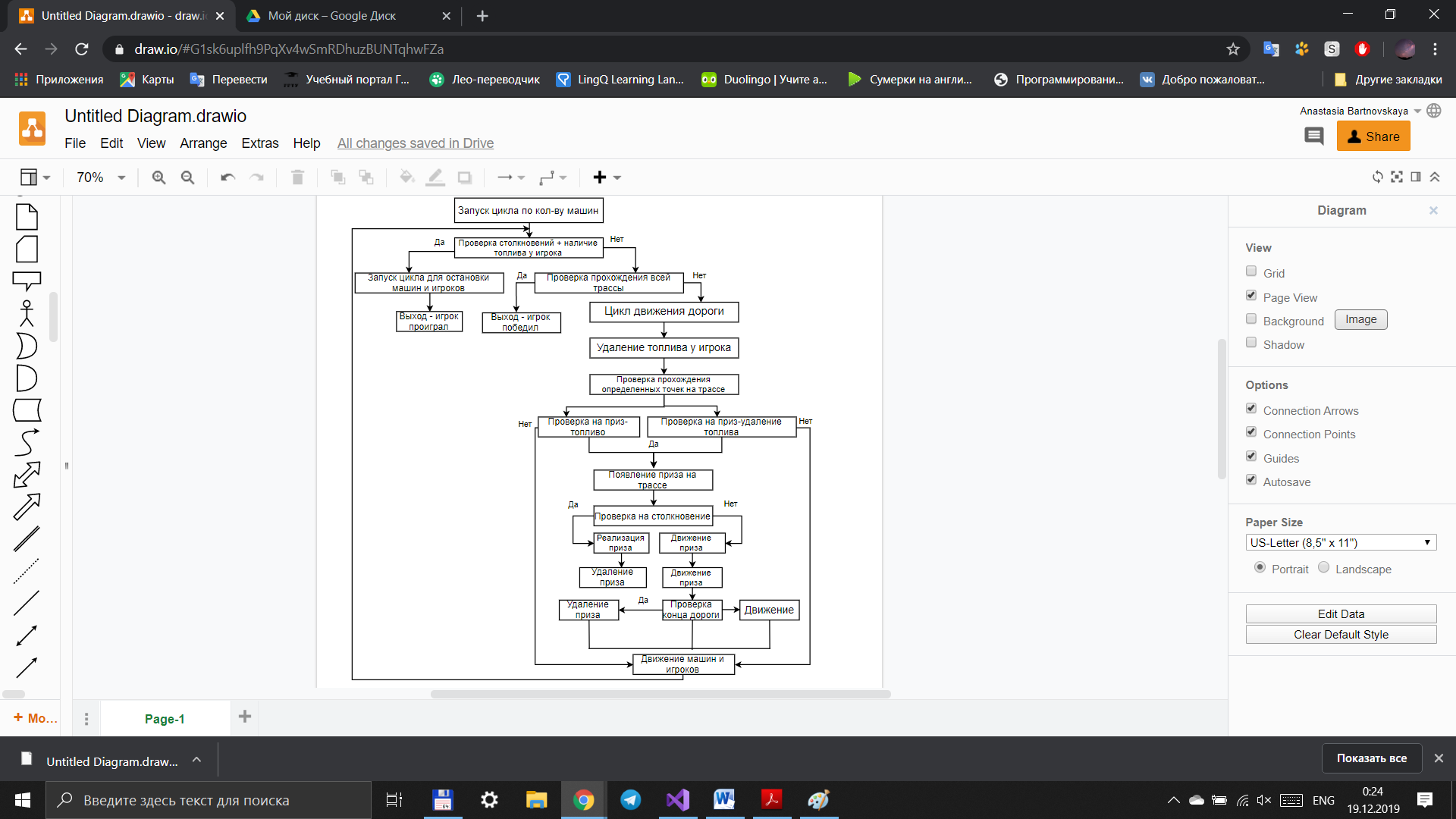


Рисунок 3.1 – Блок-схема метода *GameFieldProcess()*

После того, как была создана логика игры, она была отрисована. В классе *Render* был создан объект класса *GameField*, а так же инициализированы все основные цвета для игровых объектов.

* 1. **Тестирование созданных классов**

В ходе разработки курсового проекта возникла необходимость проверки правильности взаимодействия созданных классов.

Класс *Player* взаимодействует с противниками и призами, которые генерируются на игровом поле. Таким образом, можно сделать вывод, что тестирование необходимо провести с классами противников – *Car* и классами призов *FuelPrize* и *DeleteFuelPrize*.

Класс *Car* тестируется с классом *Player* на столкновение. Для реализации проверки на столкновение были созданы объекты и заданы им начальные позиции. В первом тесте игрок и противоположная машина сталкиваются по верхней стороне машины игрока. Проверка осуществляется через метод *Assert.IsTrue()*, куда попадает функция проверки коллизии, которая выдает *true* – в случае если машины соприкоснулись и *false*, если нет.

Далее идут тесты по проверке столкновений с разных сторон игрового персонажа. Проверка осуществляется как на столкновение, так и на его отсутствие.

Затем идет тестирование класса *Player*, который сталкивается с классами *FuelPrize* и *DeleteFuelPrize*.

Для проверки столкновения и правильности начисления приза для игрока инициализируются все необходимые объекты для правильного взаимодействия сущностей на игровом поле: создается массив для дороги, массив противников, лист призов, фабрика для генерации призов и противоположный игрок. Для проверки столкновения с объектом класса *FuelPrize*,игроку задается начальное количество топлива. Затем создается приз с координатами, при которых он соприкоснется с игроком. Приз добавляется в лист с призами и запускается метод для реализации игрового процесса. Таким образом, в данном методе происходит начисление топлива для игрока и тест проходит проверку.

При столкновении с классом *DeleteFuelPrize* проделывается тот же алгоритм, но проверка количества топлива осуществляется у противоположного игрока, так как при сборе данного приза игрок имеет возможность забрать некоторое количество у своего врага на трассе.

Так же был реализован метод проверки прохождения игроком всей трассы. Игроку было поставлено значение пройденного пути равного длине трасы, а затем был запущен цикл игрового процесса, который выдал состояние игрока, как победителя.

Результаты тестов приведены в рисунке 3.2.

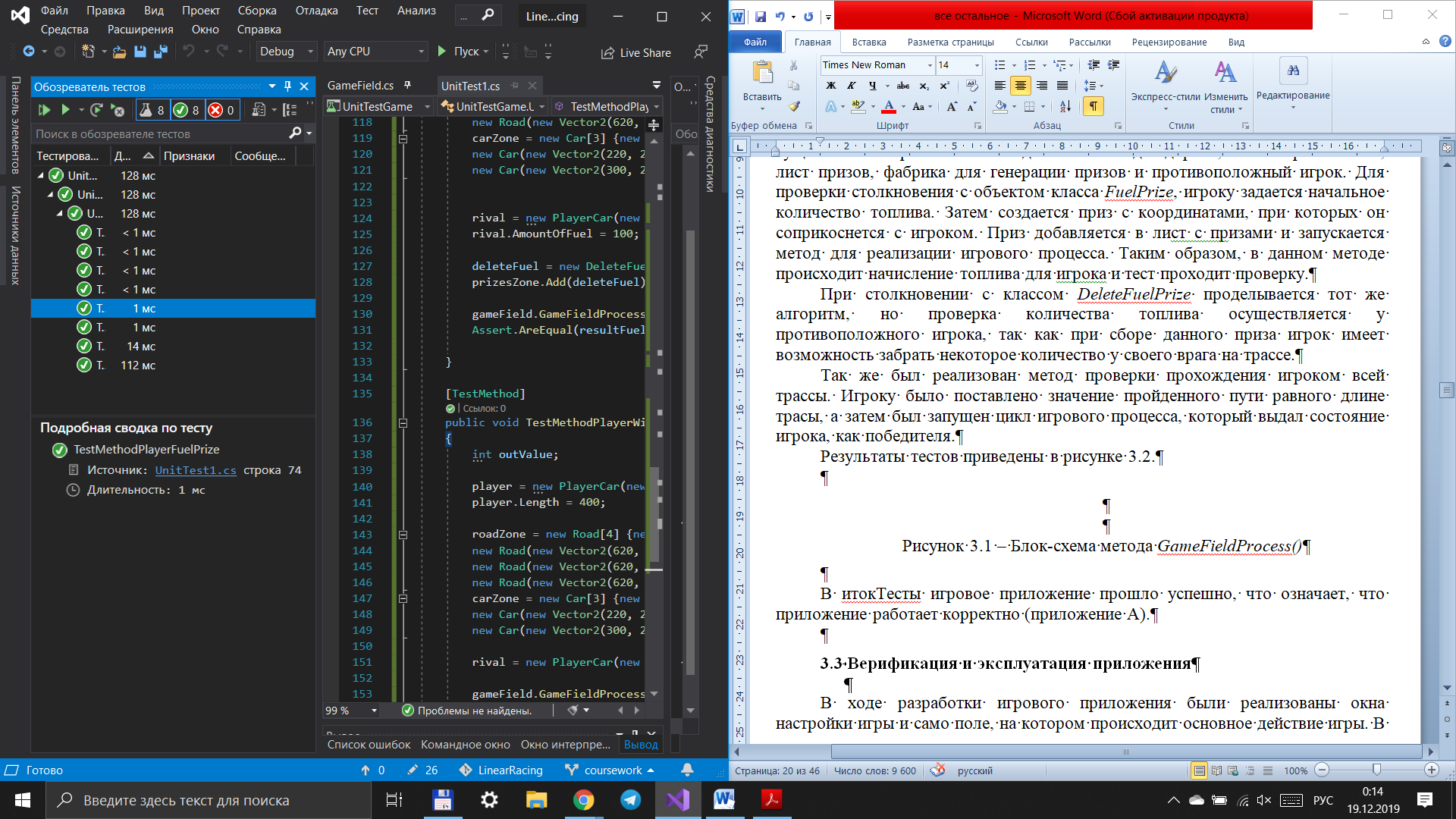


Рисунок 3.2 – Результаты тестов для приложения

Тесты игровое приложение прошло успешно, что означает, что приложение работает корректно.

* 1. **Верификация и эксплуатация приложения**

В ходе разработки игрового приложения были реализованы окна настройки игры и само поле, на котором происходит основное действие игры. В окне меню игрок имеет возможность выбрать транспортное средство, которым будет управлять в течение всего игрового процесса. Имеется три вида машины: Форд, Феррари и Порше. Каждая из них отличается от других по ряду характеристик: расход топлива, цвет и начальное количество топлива. Все эти характеристики очень влияют на ход игры, поэтому игроку придется с осторожностью выбирать машину для прохождения трассы. Выбор машины реализуется через выпадающий список. Ниже идет кнопка старта, при нажатии на которую открывается окно, в котором происходит главное действие игры (рисунок 3.3).

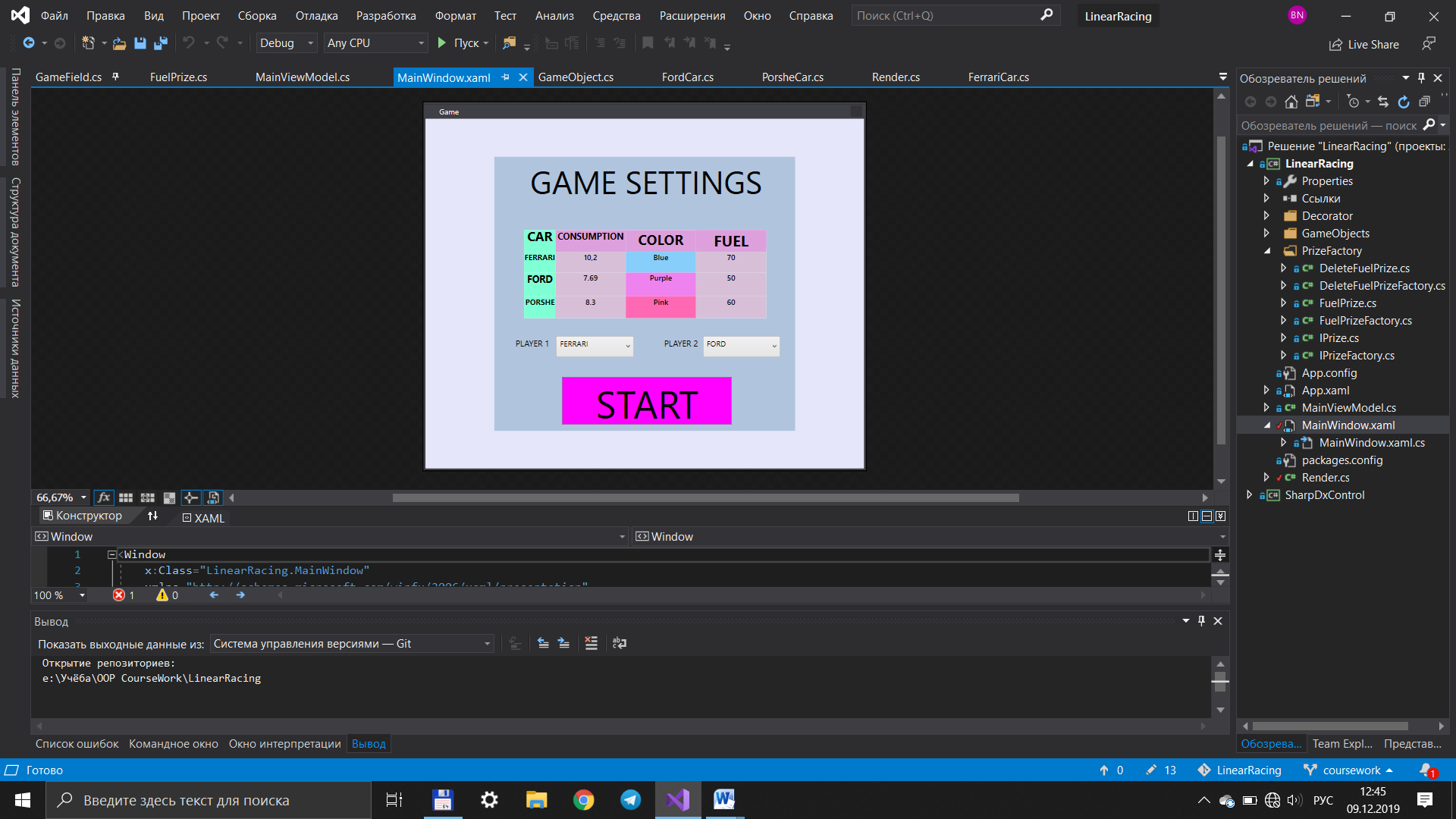


Рисунок 3.3 – Окно настроек игры

Окно с игровым полем представляет собой разделенный на две части экран. Для каждого игрока свое поле, однако, дорога для них одна и та же (рисунок 3.4).

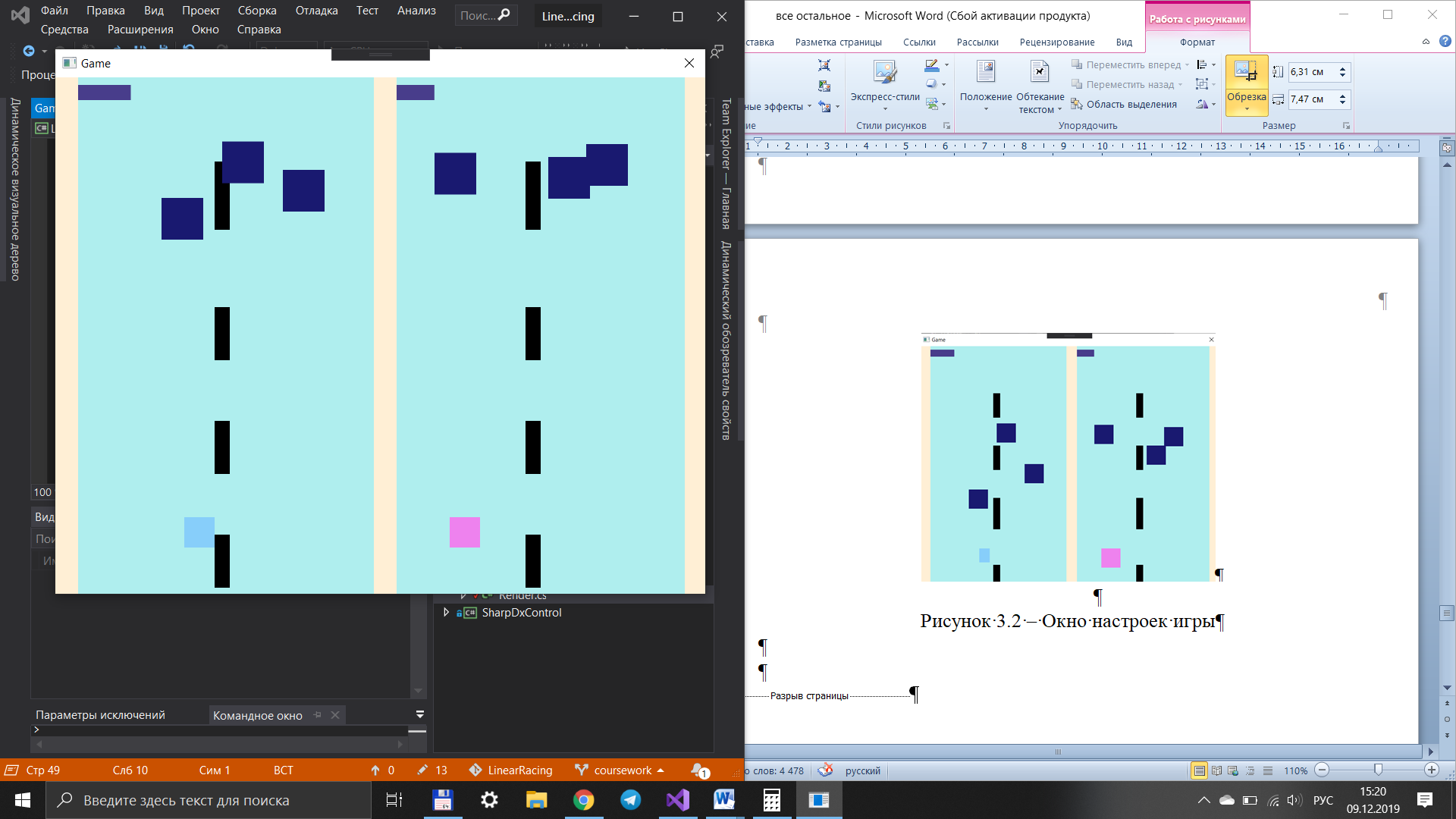


Рисунок 3.4 – Игровое поле приложения

На дороге случайным образом появляются препятствия в виде других машин. Противники окрашены в темно-синий цвет, они больше по размеру и имеют скорость, отличную от игрового персонажа.

При столкновении игрока с противником игра заканчивается проигрышем (рисунок 3.5).

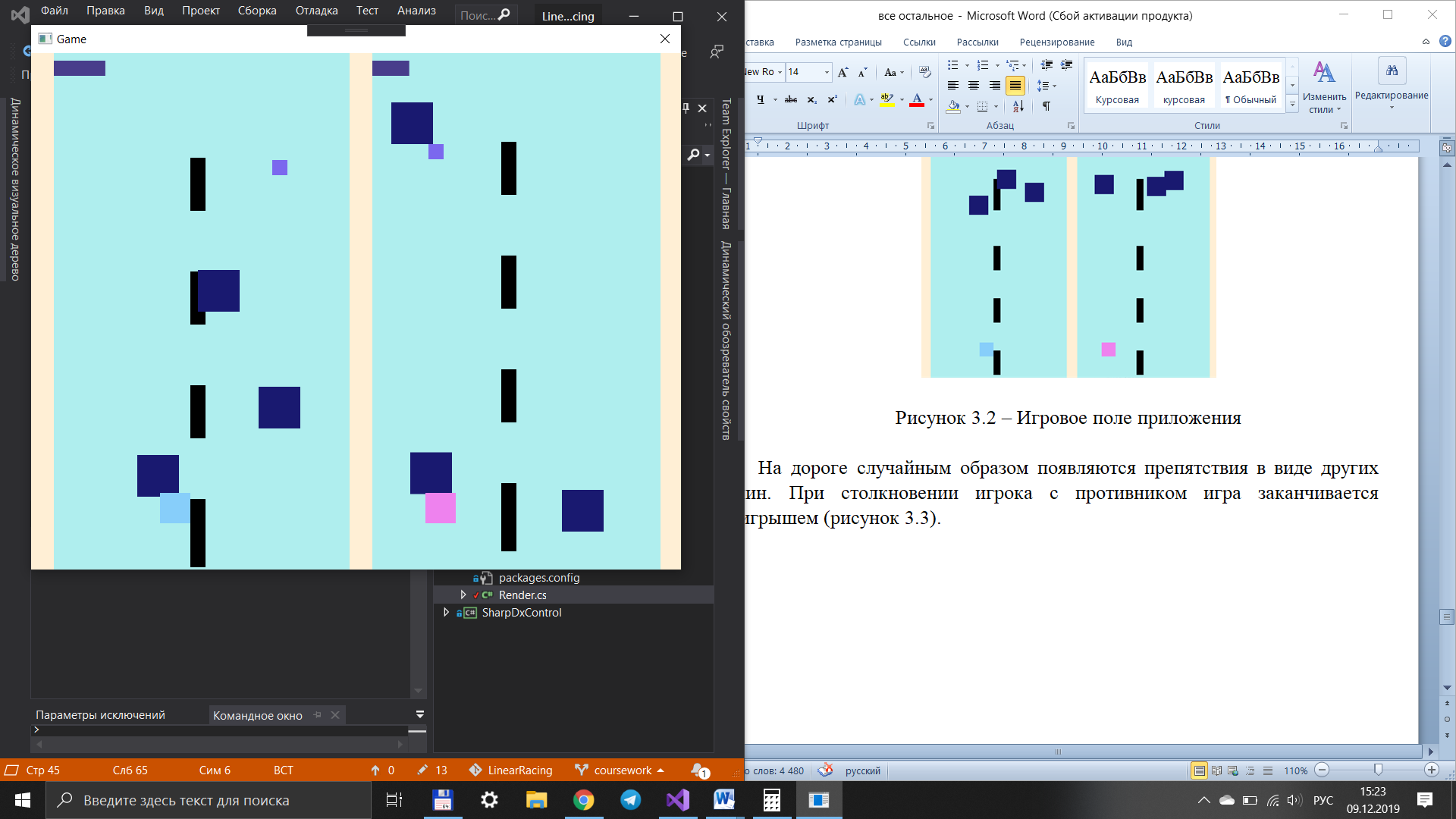


Рисунок 3.5 – Столкновение игрока с препятствием

В левом верхнем углу каждого игрока есть индикатор наличия топлива, так как при его окончании игра так же заканчивается. Расход топлива, по заданию, зависит от скорости движения игрока, поэтому, чем выше скорость передвижения по дороге, тем больше расход топлива. Игроку необходимо следить за этим индикатором, чтобы успевать подбирать топливо. Подборка топлива осуществляется за счет сбора призов. Призы реализованы двух видов – фиолетовый приз – приз-топливо и красный приз – приз, при сборе которого отнимается топливо у соперника (рисунок 3.6).

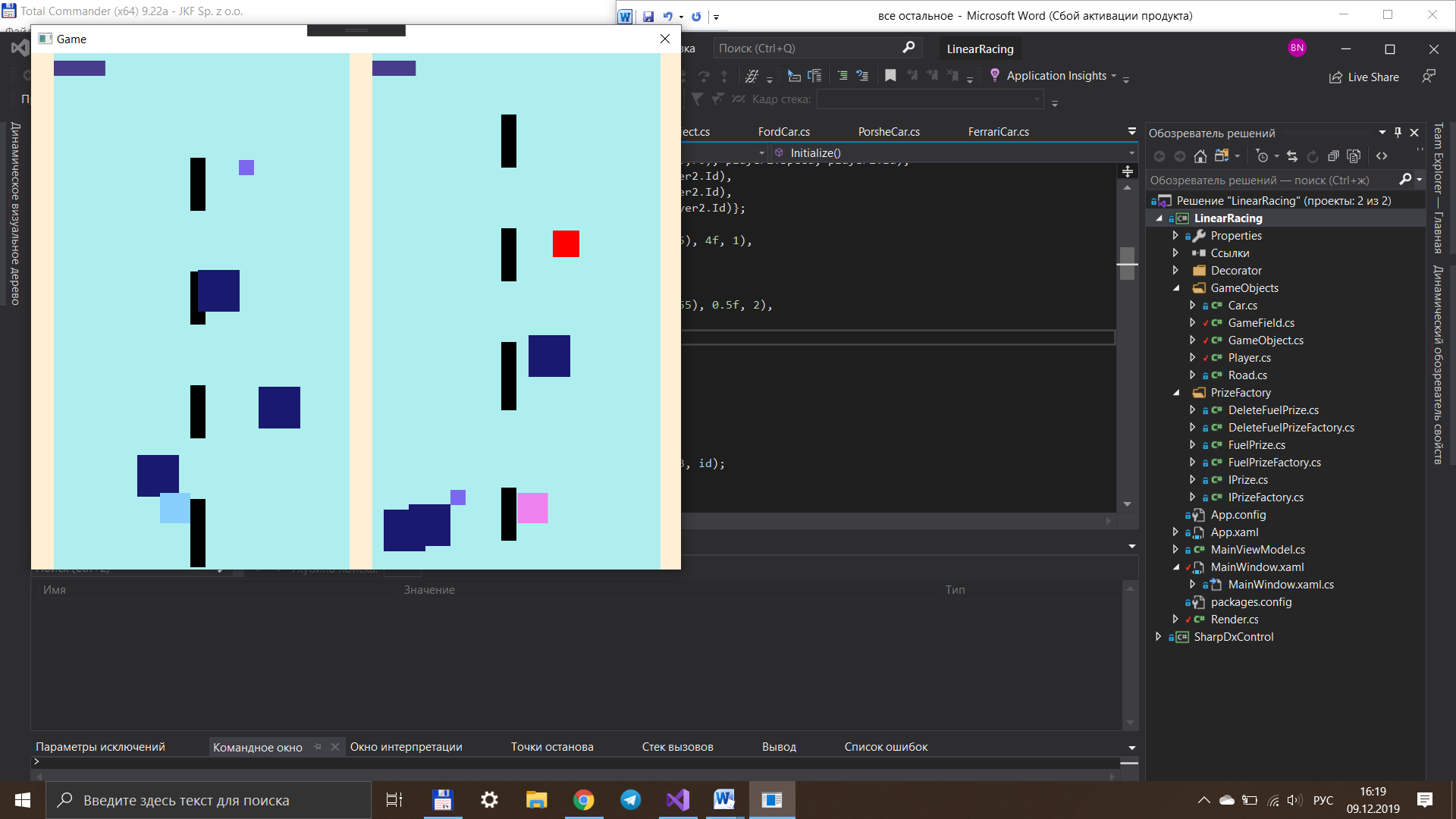


Рисунок 3.6 – Призы на игровом поле

Первый из них, как было сказано выше – это непосредственно топливо. Подбирая его, игрок может продолжить движение, за счет увеличения собственного топлива. Данный приз имеет меньший размер, чем сам игрок, что повышает сложность его сбора и делает игровое приложение интереснее.

Второй тип приза имеет красный цвет. Принцип его действия заключается в отнятии топлива у соперника, который играет на противоположном поле. Данный вид приза появляется реже, чем первый, однако имеет больший размер, что повышает шансы игрока его собрать.

Управление игроком осуществляется при помощи клавиш *W* или *Up* – для увеличения скорости, *S* или *Down* – для уменьшения скорости, *A* или *Left* – для перемещения игрока влево, *D* или *Right* – для перемещения игрока вправо соответственно.

Чтобы победить, необходимо пройти всю трассу. В случае проигрыша обоих игроков игра заканчивается, и никто не выигрывает. Результат игрового процесса представлен на рисунке 3.7.

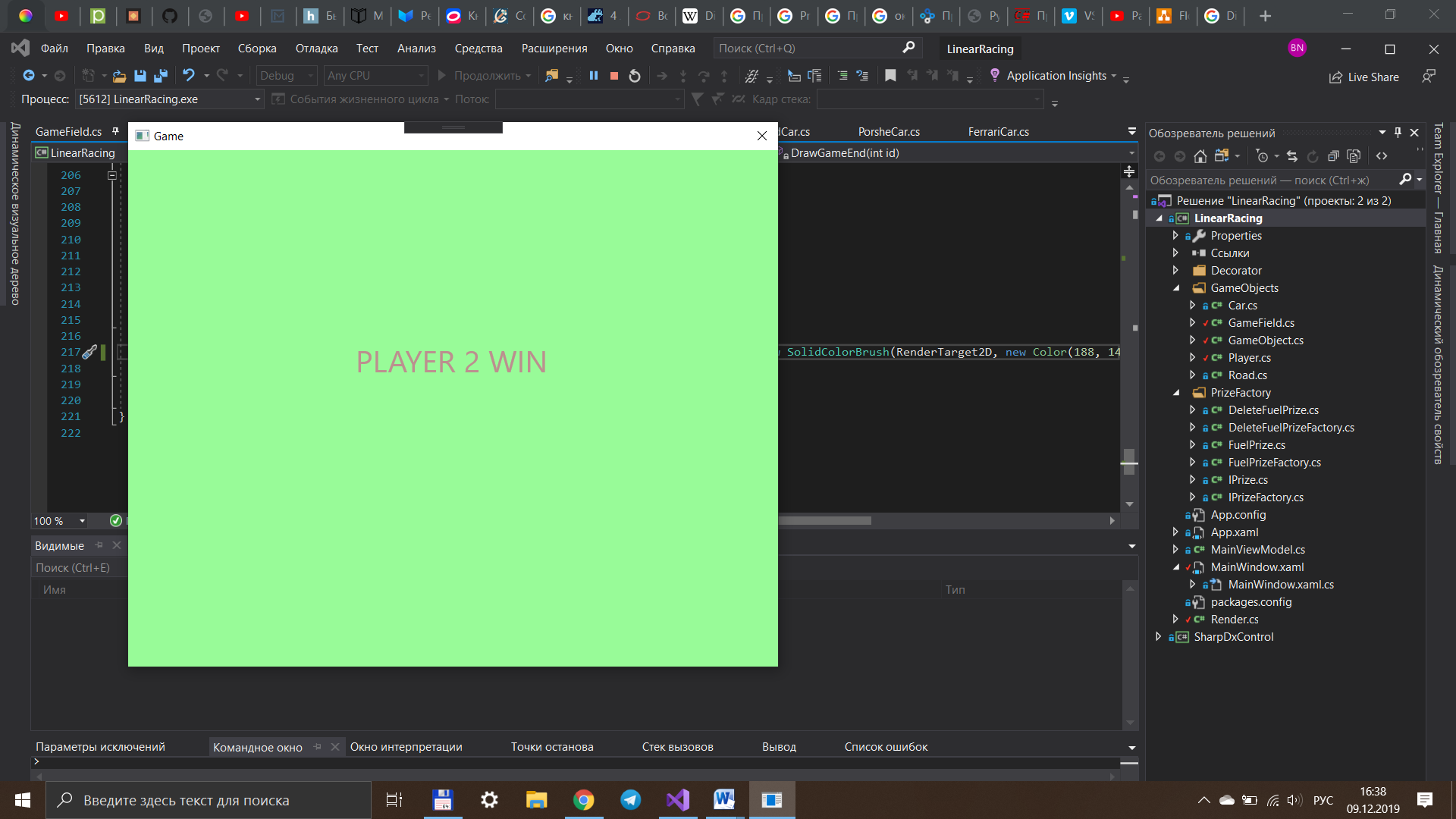


Рисунок 3.7 – Результат игрового процесса

Таким образом, приложение состоит из 3 областей: окно с настройками игры, для выбора типа игрока, само игровое поле и конец игры. Из вышеизложенного разбора приложения можно сделать вывод, что игра реализована с доступным и интуитивно понятным интерфейсом, так как клавиатура для управления игроками используется та же, что и в стандартных играх-гонках. Так же она имеет соревновательные элементы, что дает игрокам возможность почувствовать себя в роли гонщиков. В игре реализован нестандартный для гонок сбор призов в виде топлива, которое позволяет игрокам приблизиться к реальности, и в виде приза, который отнимает топливо у соперника. Данный вид бонуса реализует соревновательный элемент.

Данное игровое приложение имеет возможность добавления нового функционала по средствам увеличения типов машин или смена видов призов, что позволяет разрабатывать и улучшать его в будущем.

**ЗАКЛЮЧЕНИЕ**

При выполнении поставленной задачи первоначально был проведен аналитический обзор литературы в области разработки игровых приложений жанра «Линейные гонки». Выявлено, что в успешной игре данного жанра должен быть соревновательный элемент, возможность выбора разных характеристик машины, а так же удобный и интуитивно понятный пользовательский интерфейс.

Также были изучены и в дальнейшем выбраны средства разработки программного обеспечения и интерфейсы по обработке графики, которые в совокупности дают неплохой результат для разработки игровых приложений.

В соответствии с заданием была разработана 2*D*-игра жанра «Линейные гонки» под операционную систему *Windows*. Игры данного жанра были первыми компьютерными играми, в которые играли абсолютно все поколения, поэтому данное приложение актуально для широкого круга пользователей: как детей, так и взрослых. Игры такого типа развивают скорость реакции, а так же дают возможность ощутить себя в роли гонщика.

Достоинства данного приложения состоят в необычном для гонок сборе призов, которые дают возможность приближения игрового процесса к реальности, а также простоте управления, что помогает сосредоточиться на игровом процессе, не задумываясь о сложности взаимодействия с устройством. Также данное приложение реализует грамотный сетевой режим: чтобы победить в игре, нужно обдумывать каждое свое решение, что добавляет интерес пользователя к данному приложению.

**СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ**

1. Самые знаменитые компьютерные игры / Под ред. Р. М. Анатольевича. – СПб: Питер, 2018. – 426 с.
2. Кровь, пот и пиксели. Обратная сторона индустрии видеоигр / Под ред. Л. Степанова, Александра Голубева. – СПб: Питер, 2018. – 286 с.
3. Microsoft Windows Presentation Foundation. Базовый курс / Чарльз Петцольд. – СПб: Питер, 2017. – 365 с.
4. WPF: Windows Presentation Foundation в .NET 4.5 с примерами на C# 5.0 для профессионалов, 4-е изд. / Вильямс, 2013. – 1024 с.
5. OpenGL Superbible: всеобъемлющее и справочное руководство, 7-е издание / Аддисон-Уэсли, 2017. – 880 с.
6. Программирование игр для Windows, 3-е изд. / Вильямс, 2018. – 880 с.
7. DirectX и С#. Искусство программирования / БХВ-Петербург, 2018. – 1207 с.
8. Методы проектирования / М: Мир, 2016. – 326 с.
9. Паттерны проектирования на платформе .NET. – СПб: Питер, 2015. – 320 с.

**ПРИЛОЖЕНИЕ А**

(обязательное)

**Листинг программы**

**Код программы для класса *GameObject*:**

using SharpDX;

namespace LinearRacing

{

public abstract class GameObject

{

protected Vector2 position;

//size.X = weigth size.Y = heigth

private Vector2 size;

/// <summary>

/// Constructor for class GameObject

/// </summary>

/// <param name="position">Position of game object</param>

/// <param name="size">Size of game object</param>

/// <param name="speed">Speed of game object</param>

/// <param name="id">Id of field game object</param>

public GameObject(Vector2 position,Vector2 size, float speed, int id)

{

this.Id = id;

this.position = position;

this.size = size;

this.Speed = speed;

}

/// <summary>

/// Size

/// </summary>

public Vector2 Size

{

get

{

return size;

}

set

{

size = value;

}

}

/// <summary>

/// Position

/// </summary>

public Vector2 Position

{

get

{

return position;

}

set

{

position = value;

}

}

/// <summary>

/// Speed

/// </summary>

public float Speed { get; set; }

/// <summary>

/// Id

/// </summary>

public int Id { get; set; }

/// <summary>

/// Method for moving

/// </summary>

/// <param name="speed">Speed for moving</param>

/// <param name="id">Id of field for game object</param>

public abstract void Move(float speed, int id);

/// <summary>

/// Method for cheaking collision

/// </summary>

/// <param name="object1">object1 for cheaking collision</param>

/// <param name="object2">object2 for cheaking collision</param>

/// <returns></returns>

public static bool CheckCollision(GameObject object1, GameObject object2)

{

if ( object1.position.X + object1.size.X >= object2.position.X &&

object1.position.X <= object2.position.X + object2.size.X &&

object1.position.Y + object1.size.Y >= object2.position.Y &&

object1.position.Y <= object2.position.Y + object2.size.Y)

{

return true;

}

else

{

return false;

}

}

}

}

**Код программы для класса *Car*:**

using System;

using SharpDX;

namespace LinearRacing

{

public class Car : GameObject

{

Random random = new Random(Guid.NewGuid().GetHashCode());

public bool stop = false;

/// <summary>

/// Constructor for class Car

/// </summary>

/// <param name="position">Position of car</param>

/// <param name="size">Size of car</param>

/// <param name="speed">Speed of car</param>

/// <param name="id">Id of field for car</param>

public Car(Vector2 position, Vector2 size, float speed, int id) : base(position, size, speed, id)

{

}

/// <summary>

/// Method for moving Car

/// </summary>

/// <param name="speed">Speed for moving</param>

/// <param name="id">Id of field for car</param>

public override void Move(float speed, int id)

{

if (Position.Y > 645 )

{

if (id == 1)

{

Position = new Vector2(random.Next(30, 363), 0);

}

else

{

Position = new Vector2(random.Next(448, 775), 0);

}

}

if (!stop)

{

position.Y += speed;

}

}

}

}

**Код программы для класса *Player*:**

using System.Windows.Input;

using SharpDX;

namespace LinearRacing

{

public abstract class Player : GameObject

{

public float AmountOfFuel { get; set; }

public int Length { get; set; }

public int gear = 1;

public int lengthFuelPrize;

public int lengthDeletePrize;

public int IdCar { get; set; }

public float fuelConsumption = 0;

/// <summary>

/// Constructor for class Player

/// </summary>

/// <param name="position">Position of player</param>

/// <param name="size">Size of player</param>

/// <param name="speed">Speed of player</param>

/// <param name="id">Id of field for player</param>

public Player(Vector2 position, Vector2 size, float speed, int id) : base (position, size, speed, id)

{

}

/// <summary>

/// Method for getting fuel consumtion for car

/// </summary>

/// <returns>fuel consumtion for car</returns>

public abstract float GetFuelConsumption();

/// <summary>

/// Method for moving Player

/// </summary>

/// <param name="speed">Speed for moving</param>

/// <param name="id">Id of field for car</param>

public override void Move(float speed, int id)

{

if (id == 1)

{

if (Keyboard.IsKeyDown(Key.A) && position.X > 30)

{

position.X -= Speed / 2;

}

if (Keyboard.IsKeyDown(Key.D) && position.X < 363)

{

position.X += Speed / 2;

}

if (Keyboard.IsKeyDown(Key.W) && Speed > 0 && gear <= 2)

{

gear++;

Speed += 1.5f;

position.Y -= 0.1f;

}

if (Keyboard.IsKeyDown(Key.S) && Speed > 1 && gear > 1)

{

gear--;

position.Y += 0.1f;

Speed -= 1.5f;

}

}

else

{

if (Keyboard.IsKeyDown(Key.Left) && position.X > 448)

{

position.X -= Speed / 2;

}

if (Keyboard.IsKeyDown(Key.Right) && position.X < 775)

{

position.X += Speed / 2;

}

if (Keyboard.IsKeyDown(Key.Up) && Speed > 0 && gear <= 2)

{

gear++;

Speed += 1.5f;

position.Y -= 0.1f;

}

if (Keyboard.IsKeyDown(Key.Down) && Speed > 1 && gear > 1)

{

gear--;

position.Y += 0.1f;

Speed -= 1.5f;

}

}

}

}

}

**Код программы для класса *Road*:**

using SharpDX;

namespace LinearRacing

{

public class Road : GameObject

{

/// <summary>

/// Constructor for class Road

/// </summary>

/// <param name="position">Position of road</param>

/// <param name="size">Size of road</param>

/// <param name="speed">Speed of road</param>

/// <param name="id">Id of field for road</param>

public Road (Vector2 position, Vector2 size, float speed,int id): base(position, size, speed, id)

{

}

/// <summary>

/// Method for moving Road

/// </summary>

/// <param name="speed">Speed for moving</param>

/// <param name="id">Id of field for road</param>

public override void Move(float speed, int id)

{

if (position.Y > 640)

{

if (id == 1)

{

position = new Vector2(position.X, 0);

}

else

{

position = new Vector2(position.X, 0);

}

}

position.Y += speed;

}

}

}

**Код программы для класса *GameField***:

using System.Collections.Generic;

using SharpDX;

namespace LinearRacing

{

public class GameField

{

/// <summary>

/// Objects for game

/// </summary>

public Car[] carZone1;

public Car[] carZone2;

public Road[] roadZone1;

public Road[] roadZone2;

public List<IPrize> prizesZone1 = new List<IPrize>();

public List<IPrize> prizesZone2 = new List<IPrize>();

private IPrizeFactory fuelFactory;

private IPrizeFactory deleteFuelFactory;

public Player player1;

public Player player2;

/// <summary>

/// Initialization for objects

/// </summary>

public void Initialize()

{

fuelFactory = new FuelPrizeFactory();

deleteFuelFactory = new DeleteFuelPrizeFactory();

roadZone1 = new Road[4] { new Road(new Vector2(210, 150), new Vector2(20,70), player1.Speed,player1.Id),

new Road(new Vector2(210, 300), new Vector2(20,70), player1.Speed,player1.Id),

new Road(new Vector2(210, 450), new Vector2(20,70), player1.Speed,player1.Id),

new Road(new Vector2(210, 600), new Vector2(20,90), player1.Speed, player1.Id)};

roadZone2 = new Road[4] {new Road(new Vector2(620, 150), new Vector2(20,70), player2.Speed, player2.Id),

new Road(new Vector2(620, 300), new Vector2(20,70), player2.Speed,player2.Id),

new Road(new Vector2(620, 450), new Vector2(20,70), player2.Speed,player2.Id),

new Road(new Vector2(620, 600), new Vector2(20,90), player2.Speed, player2.Id)};

carZone1 = new Car[3] {new Car(new Vector2(140, 40), new Vector2(55, 55), 4f, 1),

new Car(new Vector2(220, 20), new Vector2(55, 55), 0.8f, 1),

new Car(new Vector2(300, 20), new Vector2(55, 55), 3f, 1)};

carZone2 = new Car[3] { new Car(new Vector2(500, 40), new Vector2(55, 55), 0.5f, 2),

new Car(new Vector2(650, 20), new Vector2(55, 55), 2f, 2),

new Car(new Vector2(700, 20), new Vector2(55, 55), 1f, 2)};

}

/// <summary>

/// Method for setting types of car for players

/// </summary>

/// <param name="carType">Type for car</param>

/// <param name="player">Player for setting type of player car</param>

/// <param name="id">Id of field for player</param>

/// <returns></returns>

public static Player SetPlayerCar(string carType, Player player, int id)

{

if(id == 1)

{

player = new PlayerCar(new Vector2(170, 580), new Vector2(40, 40),3, id);

}

else if (id == 2)

{

player = new PlayerCar(new Vector2(520, 580), new Vector2(40, 40), 3, id);

}

switch (carType)

{

case "FERRARI": player = new FerrariCar(player, 70);

break;

case "PORSHE": player = new PorsheCar(player, 60, 0.1f);

break;

case "FORD": player = new FordCar(player, 0.2f);

break;

}

return player;

}

/// <summary>

/// Method for start game process

/// </summary>

/// <param name="player1Game">Out parametr for result of game for player1</param>

/// <param name="player2Game">Out parametr for result of game for player2</param>

public void GameProcess(out int player1Game, out int player2Game )

{

GameFieldProcess(carZone1, player1, player2, roadZone1, prizesZone1, 1, out player1Game);

GameFieldProcess(carZone2, player2, player1, roadZone2, prizesZone2, 2, out player2Game);

}

/// <summary>

/// Method for start game process on some field

/// </summary>

/// <param name="cars">Array of cars</param>

/// <param name="player">Payer for game</param>

/// <param name="rival">Rival for player</param>

/// <param name="roads">Array of roads</param>

/// <param name="prizes">List of prizes</param>

/// <param name="id">Id of field for player</param>

/// <param name="gameResult">result of game for player</param>

public void GameFieldProcess(Car[] cars, Player player, Player rival, Road[] roads, List<IPrize> prizes, int id, out int gameResult)

{

gameResult = 0;

for (int k = 0; k < cars.Length; k++)

{

if (GameObject.CheckCollision(cars[k], player) || player.AmountOfFuel < 0)

{

for (int l = 0; l < cars.Length; l++)

{

cars[l].stop = true;

}

player.Speed = 0;

gameResult = 1;

}

else if(player.Length == 400)

{

gameResult = 2;

}

else

{

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

{

roads[i].Move(player.Speed, id);

if (roads[i].Position.Y > 640)

{

player.lengthFuelPrize++;

player.lengthDeletePrize++;

player.AmountOfFuel -= player.GetFuelConsumption();

player.Length++;

}

if (player.lengthFuelPrize == 4)

{

prizes.Add(fuelFactory.CreatePrize(id, player.Speed));

player.lengthFuelPrize = 0;

}

if (player.lengthDeletePrize == 50)

{

prizes.Add(deleteFuelFactory.CreatePrize(id, player.Speed));

player.lengthDeletePrize = 0;

}

}

for (int i = 0; i < prizes.Count; i++)

{

if (GameObject.CheckCollision(player, (GameObject)prizes[i]))

{

if (prizes[i] is FuelPrize)

{

player.AmountOfFuel += prizes[i].GetPrize();

}

else if (prizes[i] is DeleteFuelPrize)

{

rival.AmountOfFuel -= prizes[i].GetPrize();

}

prizes.RemoveAt(i);

}

else if (prizes[i].GetPosition().Y > 630)

{

prizes.RemoveAt(i);

}

else

{

prizes[i].Move(player.Speed, id);

}

}

cars[k].Move(cars[k].Speed + player.Speed, id);

player.Move(player.Speed, id);

gameResult = 3;

}

}

}

}

}

**Код программы для класса *IPrize*:**

using SharpDX;

namespace LinearRacing

{

public interface IPrize

{

/// <summary>

/// Method for getting position

/// </summary>

/// <returns>position of prize</returns>

Vector2 GetPosition();

/// <summary>

/// Method for getting size

/// </summary>

/// <returns>size of prize</returns>

Vector2 GetSize();

/// <summary>

/// Method for getting speed

/// </summary>

/// <returns>speed of prize</returns>

float GetSpeed();

/// <summary>

/// Method for getting id

/// </summary>

/// <returns>id of prize</returns>

int GetId();

/// <summary>

/// Method for getting prize

/// </summary>

/// <returns>amount of prize</returns>

int GetPrize();

/// <summary>

/// Method for moving

/// </summary>

/// <param name="speed">Speed for moving</param>

/// <param name="id">Id of field for game object</param>

void Move(float speed, int id);

}

}

Код программы для класса IPrizeFactory:

namespace LinearRacing

{

public interface IPrizeFactory

{

/// <summary>

/// Method for creating prize

/// </summary>

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

/// <param name="speed">speed of prize</param>

/// <returns>prize</returns>

IPrize CreatePrize(int id, float speed);

}

}

Код программы для класса FuelPrizeFactory:

using System;

using SharpDX;

namespace LinearRacing

{

public class FuelPrizeFactory : IPrizeFactory

{

readonly Random random = new Random();

/// <summary>

/// Method for creating FuelPrize

/// </summary>

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

/// <param name="speed">speed of prize</param>

/// <returns>prize</returns>

public IPrize CreatePrize(int id, float speed)

{

if (id == 1)

{

return new FuelPrize(new Vector2(random.Next(30, 363), 0), new Vector2(20, 20), speed, 1);

}

else

{

return new FuelPrize(new Vector2(random.Next(448, 775), 0), new Vector2(20, 20), speed, 2);

}

}

}

}

**Код программы для класса *DeleteFuelPrizeFactory*:**

using System;

using SharpDX;

namespace LinearRacing

{

public class DeleteFuelPrizeFactory : IPrizeFactory

{

readonly Random random = new Random();

/// <summary>

/// Method for creating DeleteFuelPrize

/// </summary>

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

/// <param name="speed">speed of prize</param>

/// <returns>prize</returns>

public IPrize CreatePrize(int id, float speed)

{

if (id == 1)

{

return new DeleteFuelPrize (new Vector2(random.Next(30, 363), random.Next(0, 50)), new Vector2(35, 35), speed, 1);

}

else

{

return new DeleteFuelPrize(new Vector2(random.Next(448, 775), 0), new Vector2(35, 35), speed, 2);

}

}

}

}

**Код программы для класса *FuelPrize***:

using System;

using SharpDX;

namespace LinearRacing

{

public class FuelPrize : GameObject, IPrize

{

readonly Random random = new Random();

/// <summary>

/// Constructor for class FuelPrize

/// </summary>

/// <param name="position">Position of prize</param>

/// <param name="size">Size of prize</param>

/// <param name="speed">Speed of prize</param>

/// <param name="id">Id of field for prize</param>

public FuelPrize(Vector2 position, Vector2 size, float speed, int id) : base(position, size, speed, id)

{

}

public Vector2 GetSize()

{

return Size;

}

public float GetSpeed()

{

return Speed;

}

public int GetId()

{

return Id;

}

public Vector2 GetPosition()

{

return Position;

}

/// <summary>

/// Method for moving FuelPrize

/// </summary>

/// <param name="speed">Speed for moving</param>

/// <param name="id">Id of field for prize</param>

public override void Move(float speed, int id)

{

if (Position.Y > 645)

{

if (id == 1)

{

Position = new Vector2(random.Next(30, 363), 0);

}

else

{

Position = new Vector2(random.Next(448, 775), 0);

}

}

position.Y += speed;

}

/// <summary>

/// Method for getting prize

/// </summary>

/// <returns>amount of fiel for add</returns>

public int GetPrize()

{

return 10;

}

}

}

**Код программы для класса *DeleteFuelPrize*:**

using System;

using SharpDX;

namespace LinearRacing

{

public class DeleteFuelPrize : GameObject, IPrize

{

readonly Random random = new Random();

/// <summary>

/// Constructor for class DeleteFuelPrize

/// </summary>

/// <param name="position">Position of prize</param>

/// <param name="size">Size of prize</param>

/// <param name="speed">Speed of prize</param>

/// <param name="id">Id of field for prize</param>

public DeleteFuelPrize(Vector2 position, Vector2 size, float speed, int id) : base(position, size,speed, id)

{

}

public Vector2 GetSize()

{

return Size;

}

public float GetSpeed()

{

return Speed;

}

public int GetId()

{

return Id;

}

public Vector2 GetPosition()

{

return Position;

}

/// <summary>

/// Method for moving DeleteFuelPrize

/// </summary>

/// <param name="speed">Speed for moving</param>

/// <param name="id">Id of field for prize</param>

public override void Move(float speed, int id)

{

if (Position.Y > 645)

{

if (id == 1)

{

Position = new Vector2(random.Next(30, 363), 0);

}

else

{

Position = new Vector2(random.Next(448, 775), 0);

}

}

position.Y += speed;

}

/// <summary>

/// Method for getting prize

/// </summary>

/// <returns>amount of fiel for delete</returns>

public int GetPrize()

{

return 15;

}

}

}

**Код программы для класса *MainViewModel*:**

using System.ComponentModel;

namespace LinearRacing

{

class MainViewModel : INotifyPropertyChanged

{

private object content;

public object Content

{

get

{

return content;

}

set

{

content = value;

NotifyPropertyChanged("Content");

}

}

public event PropertyChangedEventHandler PropertyChanged;

/// <summary>

/// Method for tracking changes in binding

/// </summary>

/// <param name="propertyName">mame of component in binding</param>

private void NotifyPropertyChanged(string propertyName)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}

}

}

**Код программы для класса *Render*:**

using SharpDX;

using SharpDX.Direct2D1;

using SharpDxControl;

using SharpDX.DirectWrite;

namespace LinearRacing

{

class Renderer : Direct2DComponent

{

TextFormat textGameOver;

int field1;

int field2;

GameField gameField = new GameField();

SharpDX.DirectWrite.Factory factory = new SharpDX.DirectWrite.Factory();

private SolidColorBrush colorFerrari;

private SolidColorBrush colorFord;

private SolidColorBrush colorPorshe;

private SolidColorBrush colorCar;

private SolidColorBrush colorRoad;

private SolidColorBrush colorBoardRoad;

private SolidColorBrush colorFuelPrize;

private SolidColorBrush colorDeletePrize;

private SolidColorBrush colorFuel;

private SolidColorBrush colorEndGame;

/// <summary>

/// Constructor for class Renderer

/// </summary>

/// <param name="player1">type of Player1</param>

/// <param name="player2">type of Player2</param>

public Renderer(string player1, string player2)

{

gameField.player1 = GameField.SetPlayerCar(player1, gameField.player1, 1);

gameField.player2 = GameField.SetPlayerCar(player2, gameField.player2, 2);

}

/// <summary>

/// Initialize for colors and game fields

/// </summary>

public override void InternalInitialize()

{

base.InternalInitialize();

gameField.Initialize();

colorFerrari = new SolidColorBrush(RenderTarget2D, new Color(135, 206, 250));

colorFord = new SolidColorBrush(RenderTarget2D, new Color(238, 130, 238));

colorPorshe = new SolidColorBrush(RenderTarget2D, new Color(255, 105, 180));

colorCar = new SolidColorBrush(RenderTarget2D, new Color(25, 25, 112));

colorRoad = new SolidColorBrush(RenderTarget2D, new Color(0, 0, 0));

colorFuelPrize = new SolidColorBrush(RenderTarget2D, new Color(123, 104, 238));

colorDeletePrize = new SolidColorBrush(RenderTarget2D, new Color(255, 0, 0));

colorFuel = new SolidColorBrush(RenderTarget2D, new Color(72, 61, 139));

colorBoardRoad = new SolidColorBrush(RenderTarget2D, new Color(255, 239, 213));

colorEndGame = new SolidColorBrush(RenderTarget2D, new Color(152, 251, 152));

textGameOver = new TextFormat(factory, "Calibry", 40);

}

/// <summary>

/// Clean data

/// </summary>

protected override void InternalUninitialize()

{

Utilities.Dispose(ref colorBoardRoad);

Utilities.Dispose(ref colorCar);

Utilities.Dispose(ref colorDeletePrize);

Utilities.Dispose(ref colorEndGame);

Utilities.Dispose(ref colorFerrari);

Utilities.Dispose(ref colorFuel);

Utilities.Dispose(ref colorPorshe);

Utilities.Dispose(ref colorRoad);

base.InternalUninitialize();

}

/// <summary>

/// Method for painting all objects in game

/// </summary>

protected override void Render()

{

RenderTarget2D.Clear(new Color(175, 238, 238));

Draw(new Vector2(0, 0), new Vector2(30, 703), colorBoardRoad);

Draw(new Vector2(420, 0), new Vector2(30, 703), colorBoardRoad);

Draw(new Vector2(830, 0), new Vector2(30, 703), colorBoardRoad);

DrawRoads();

DrawPrizes();

DrawCars();

DrawPlayer(gameField.player1);

DrawPlayer(gameField.player2);

DrawPlayerFuel();

gameField.GameProcess(out field1, out field2);

if (field1 == 2)

{

DrawGameEnd(1);

}

if (field2 == 2)

{

DrawGameEnd(2);

}

}

/// <summary>

/// Method for painting

/// </summary>

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

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

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

public void Draw(Vector2 position, Vector2 size, SolidColorBrush color)

{

RectangleF rectangle = new RectangleF(position.X, position.Y, size.X, size.Y);

RenderTarget2D.FillRectangle(rectangle, color);

}

/// <summary>

/// Method for painting prizes

/// </summary>

private void DrawPrizes()

{

for (int i = 0; i < gameField.prizesZone1.Count; i++)

{

if (gameField.prizesZone1[i] is FuelPrize )

{

Draw(gameField.prizesZone1[i].GetPosition(),gameField.prizesZone1[i].GetSize(), colorFuelPrize);

}

else if (gameField.prizesZone1[i] is DeleteFuelPrize)

{

Draw(gameField.prizesZone1[i].GetPosition(), gameField.prizesZone1[i].GetSize(), colorDeletePrize);

}

}

for (int i = 0; i < gameField.prizesZone2.Count; i++)

{

if (gameField.prizesZone2[i] is FuelPrize )

{

Draw(gameField.prizesZone2[i].GetPosition(), gameField.prizesZone2[i].GetSize(), colorFuelPrize);

}

else if (gameField.prizesZone2[i] is DeleteFuelPrize )

{

Draw(gameField.prizesZone2[i].GetPosition(), gameField.prizesZone2[i].GetSize(), colorDeletePrize);

}

}

}

/// <summary>

/// Method for cars

/// </summary>

private void DrawCars()

{

for (int i = 0; i < gameField.carZone2.Length; i++)

{

Draw(gameField.carZone1[i].Position, gameField.carZone1[i].Size, colorCar);

Draw(gameField.carZone2[i].Position, gameField.carZone2[i].Size, colorCar);

}

}

/// <summary>

/// Method for painting player

/// </summary>

/// <param name="player">player for painting</param>

private void DrawPlayer(Player player)

{

SolidColorBrush colorCar;

if(player is FerrariCar)

{

colorCar = colorFerrari;

}

else if (player is FordCar)

{

colorCar = colorFord;

}

else if (player is PorsheCar)

{

colorCar = colorPorshe;

}

else

{

colorCar = colorPorshe;

}

Draw(player.Position, player.Size, colorCar);

}

/// <summary>

/// Method for painting roads

/// </summary>

private void DrawRoads()

{

for (int i = 0; i < gameField.roadZone1.Length; i++)

{

Draw(gameField.roadZone1[i].Position,gameField.roadZone1[i].Size, colorRoad);

Draw(gameField.roadZone2[i].Position, gameField.roadZone2[i].Size, colorRoad);

}

}

/// <summary>

/// Method for painting fuel

/// </summary>

private void DrawPlayerFuel()

{

if (gameField.player1.AmountOfFuel > 0 )

{

Draw(new Vector2(30, 10), new Vector2(gameField.player1.AmountOfFuel, 20), colorFuel);

}

if (gameField.player2.AmountOfFuel > 0)

{

Draw(new Vector2(450, 10), new Vector2(gameField.player2.AmountOfFuel, 20), colorFuel);

}

}

/// <summary>

/// Method for painting end game

/// </summary>

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

private void DrawGameEnd(int id)

{

Draw(new Vector2(0, 0), new Vector2(900, 900), colorEndGame);

string result;

switch (id)

{

case 1: result = "PLAYER 1 WIN";

break;

case 2: result = "PLAYER 2 WIN";

break;

default: result = "ERROR";

break;

}

renderTarget2D.DrawText(result, textGameOver, new RectangleF(300, 250, 400, 400), new SolidColorBrush(RenderTarget2D, new Color(188, 143, 143)));

}

}

}

**Код программы для класса *DecorartorPlayerCar*:**

using System;

namespace LinearRacing

{

/// <summary>

/// Class decorator

/// </summary>

abstract class DecoratorPlayerCar : Player

{

protected Player player;

/// <summary>

/// Constructor for Decoratot

/// </summary>

/// <param name="player">class object Player for decoration</param>

public DecoratorPlayerCar(Player player) : base(player.Position, player.Size, player.Speed, player.Id)

{

this.player = player;

}

}

}

**Код программы для класса *FerrariCar*:**

namespace LinearRacing

{

/// <summary>

/// Class decoration for PlayerCar

/// </summary>

class FerrariCar : DecoratorPlayerCar

{

/// <summary>

/// Constructor for class FerrariCar

/// </summary>

/// <param name="player">Class object Player for decoration</param>

/// <param name="volume">Start amount os fuel</param>

public FerrariCar(Player player, int volume) : base(player)

{

AmountOfFuel = volume;

}

/// <summary>

/// Method for getting fuel consumtion for car

/// </summary>

/// <returns>fuel consumtion for car</returns>

public override float GetFuelConsumption()

{

return player.GetFuelConsumption() ;

}

}

}

**Код программы для класса *FordCar*:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using SharpDX;

namespace LinearRacing

{

/// <summary>

/// Class decoration for PlayerCar

/// </summary>

class FordCar : DecoratorPlayerCar

{

/// <summary>

/// Constructor for class FordCar

/// </summary>

/// <param name="player">Class object Player for decoration</param>

/// <param name="fuelConsumption">New fuel consumption</param>

public FordCar(Player player, float fuelConsumption) : base(player)

{

this.fuelConsumption = fuelConsumption;

AmountOfFuel = 50;

}

/// <summary>

/// Method for getting fuel consumtion for car

/// </summary>

/// <returns>fuel consumtion for car</returns>

public override float GetFuelConsumption()

{

return player.GetFuelConsumption() - fuelConsumption;

}

}

}

**Код программы для класса *PorcheCar*:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LinearRacing

{

/// <summary>

/// Class decoration for PlayerCar

/// </summary>

class PorsheCar : DecoratorPlayerCar

{

/// <summary>

/// Constructor for class PorsheCar

/// </summary>

/// <param name="player">Class object Player for decoration</param>

/// <param name="volume">Start amount of fuel</param>

/// <param name="fuelConsumption">New fuel consumption</param>

public PorsheCar(Player player, int volume, float fuelConsumption) : base(player)

{

this.fuelConsumption = fuelConsumption;

AmountOfFuel = volume;

}

/// <summary>

/// Method for getting fuel consumtion for car

/// </summary>

/// <returns>fuel consumtion for car</returns>

public override float GetFuelConsumption()

{

return player.GetFuelConsumption() - fuelConsumption;

}

}

}

**Код программы для класса *PlayerCar*:**

using SharpDX;

namespace LinearRacing

{

/// <summary>

/// Concrete component for Player

/// </summary>

public class PlayerCar : Player

{

/// <summary>

/// Constructor for class PlayerCar

/// </summary>

/// <param name="position">Position of car</param>

/// <param name="size">Size of acr</param>

/// <param name="speed">Speed of car</param>

/// <param name="id">Id of field for car</param>

public PlayerCar(Vector2 position, Vector2 size, float speed, int id) : base(position, size, speed, id)

{

AmountOfFuel = 30;

Speed = 3;

}

/// <summary>

/// Method for getting fuel consumtion for car

/// </summary>

/// <returns>fuel consumtion for car</returns>

public override float GetFuelConsumption()

{

return (AmountOfFuel \* Speed) / 400;

}

}

}

**Код программы для класса *DirectXComponent*:**

using System;

using System.Windows;

using System.Windows.Media;

using SharpDX;

using SharpDX.Direct3D;

using SharpDX.Direct3D11;

using SharpDX.DXGI;

using Device = SharpDX.Direct3D11.Device;

using Resource = SharpDX.Direct3D11.Resource;

namespace SharpDxControl

{

/// <summary>

/// Create SharpDx swapchain hosted in the controls parent Hwnd

/// Resources created on Loaded, destroyed on Unloaded.

/// </summary>

public abstract class DirectXComponent : Win32Control

{

private Device device;

// buffer exchange chain

private SwapChain swapChain;

// back buffer

private Texture2D backBuffer;

//renderTargetView

private RenderTargetView renderTargetView;

protected Texture2D BackBuffer => backBuffer;

protected int SurfaceWidth { get; private set; }

protected int SurfaceHeight { get; private set; }

protected DirectXComponent()

{

}

/// <summary>

/// Method for initialize

/// </summary>

protected override sealed void Initialize()

{

InternalInitialize();

//поверхность на котором происходит отрисовка

CompositionTarget.Rendering += OnCompositionTargetRendering;

}

/// <summary>

/// Method for uninitialize

/// </summary>

protected override sealed void Uninitialize()

{

CompositionTarget.Rendering -= OnCompositionTargetRendering;

InternalUninitialize();

}

/// <summary>

/// Method for rendering

/// </summary>

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

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

private void OnCompositionTargetRendering(object sender, EventArgs eventArgs)

{

BeginRender();

Render();

EndRender();

}

private double GetDpiScale()

{

PresentationSource source = PresentationSource.FromVisual(this);

return source.CompositionTarget.TransformToDevice.M11;

}

public virtual void InternalInitialize()

{

var dpiScale = GetDpiScale();

SurfaceWidth = (int)(ActualWidth < 0 ? 0 : Math.Ceiling(ActualWidth \* dpiScale));

SurfaceHeight = (int)(ActualHeight < 0 ? 0 : Math.Ceiling(ActualHeight \* dpiScale));

//SwapChainDescription

SwapChainDescription swapChainDescription = new SwapChainDescription

{

//window in which the display will take place

OutputHandle = Hwnd,

//BufferCount

BufferCount = 1,

IsWindowed = true,

//back buffer

ModeDescription = new ModeDescription(SurfaceWidth, SurfaceHeight, new Rational(60, 1), Format.B8G8R8A8\_UNorm),

SampleDescription = new SampleDescription(1, 0),

SwapEffect = SwapEffect.Discard,

Usage = Usage.RenderTargetOutput

};

Device.CreateWithSwapChain(DriverType.Hardware, DeviceCreationFlags.BgraSupport, swapChainDescription, out device, out swapChain);

// New RenderTargetView from the backbuffer

backBuffer = Resource.FromSwapChain<Texture2D>(swapChain, 0);

renderTargetView = new RenderTargetView(device, backBuffer);

}

/// <summary>

/// Destory all DirectX resources.

/// Derived methods should end with base.InternalUninitialize();

/// </summary>

protected virtual void InternalUninitialize()

{

Utilities.Dispose(ref renderTargetView);

Utilities.Dispose(ref backBuffer);

Utilities.Dispose(ref swapChain);

Utilities.Dispose(ref device);

}

/// <summary>

/// sets the rendering object to the one that was created earlier

/// </summary>

protected virtual void BeginRender()

{

device.ImmediateContext.OutputMerger.SetRenderTargets(renderTargetView);

}

/// <summary>

/// replaces the back with a front buffer (1 - vertical sync)

/// </summary>

protected virtual void EndRender()

{

swapChain.Present(1, PresentFlags.None);

}

protected abstract void Render();

}

}

**Код программы для класса *Win32Control*:**

using System;

using System.Runtime.InteropServices;

using System.Windows;

using System.Windows.Interop;

namespace SharpDxControl

{

public abstract class Win32Control : HwndHost

{

protected IntPtr Hwnd { get; private set; }

private const string WindowClass = "HwndWrapper";

/// <summary>

/// Constructor for win32control

/// </summary>

protected Win32Control()

{

Loaded += OnLoaded;

Unloaded += OnUnloaded;

}

/// <summary>

/// Onload window

/// </summary>

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

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

private void OnLoaded(object sender, RoutedEventArgs routedEventArgs)

{

Initialize();

Loaded -= OnLoaded;

}

/// <summary>

/// OnUnload window

/// </summary>

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

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

private void OnUnloaded(object sender, RoutedEventArgs routedEventArgs)

{

Uninitialize();

Unloaded -= OnUnloaded;

Dispose();

}

protected abstract void Initialize();

protected abstract void Uninitialize();

/// <summary>

/// win32 child window handle

/// </summary>

/// <param name="hwndParent">parent window handle</param>

/// <returns></returns>

protected override HandleRef BuildWindowCore(HandleRef hwndParent)

{

var wndClass = new NativeMethods.WndClassEx();

wndClass.cbSize = (uint)Marshal.SizeOf(wndClass);

wndClass.hInstance = NativeMethods.GetModuleHandle(null);

wndClass.lpfnWndProc = NativeMethods.DefaultWindowProc;

wndClass.lpszClassName = WindowClass;

wndClass.hCursor = NativeMethods.LoadCursor(IntPtr.Zero, NativeMethods.IDC\_ARROW);

NativeMethods.RegisterClassEx(ref wndClass);

Hwnd = NativeMethods.CreateWindowEx(

0, WindowClass, "", NativeMethods.WS\_CHILD | NativeMethods.WS\_VISIBLE,

0, 0, (int)Width, (int)Height, hwndParent.Handle, IntPtr.Zero, IntPtr.Zero, 0);

return new HandleRef(this, Hwnd);

}

/// <summary>

/// Delete window

/// </summary>

/// <param name="hwnd">structure containing window elements</param>

protected override void DestroyWindowCore(HandleRef hwnd)

{

NativeMethods.DestroyWindow(hwnd.Handle);

Hwnd = IntPtr.Zero;

}

}

internal class NativeMethods

{

public const int WS\_CHILD = 0x40000000;

public const int WS\_VISIBLE = 0x10000000;

public const int IDC\_ARROW = 32512;

[StructLayout(LayoutKind.Sequential)]

public struct WndClassEx

{

public uint cbSize;

public uint style;

[MarshalAs(UnmanagedType.FunctionPtr)]

public WndProc lpfnWndProc;

public int cbClsExtra;

public int cbWndExtra;

public IntPtr hInstance;

public IntPtr hIcon;

public IntPtr hCursor;

public IntPtr hbrBackground;

public string lpszMenuName;

public string lpszClassName;

public IntPtr hIconSm;

}

public delegate IntPtr WndProc(IntPtr hWnd, uint msg, IntPtr wParam, IntPtr lParam);

public static readonly WndProc DefaultWindowProc = DefWindowProc;

[DllImport("user32.dll", EntryPoint = "CreateWindowEx", CharSet = CharSet.Auto)]

public static extern IntPtr CreateWindowEx(

int exStyle,

string className,

string windowName,

int style,

int x, int y,

int width, int height,

IntPtr hwndParent,

IntPtr hMenu,

IntPtr hInstance,

[MarshalAs(UnmanagedType.AsAny)] object pvParam);

[DllImport("user32.dll", EntryPoint = "DestroyWindow", CharSet = CharSet.Auto)]

public static extern bool DestroyWindow(IntPtr hwnd);

[DllImport("user32.dll")]

public static extern IntPtr DefWindowProc(IntPtr hWnd, uint uMsg, IntPtr wParam, IntPtr lParam);

[DllImport("kernel32.dll")]

public static extern IntPtr GetModuleHandle(string module);

[DllImport("user32.dll")]

[return: MarshalAs(UnmanagedType.U2)]

public static extern short RegisterClassEx([In] ref WndClassEx lpwcx);

[DllImport("user32.dll")]

public static extern IntPtr LoadCursor(IntPtr hInstance, int lpCursorName);

// ReSharper restore InconsistentNaming

}

}

**Код программы для класса *Direct2DComponent*:**

using SharpDX;

using SharpDX.Direct2D1;

using SharpDX.DXGI;

using AlphaMode = SharpDX.Direct2D1.AlphaMode;

using Factory = SharpDX.Direct2D1.Factory;

namespace SharpDxControl

{

public abstract class Direct2DComponent : DirectXComponent

{

private Factory factory2D;

public static RenderTarget renderTarget2D;

protected RenderTarget RenderTarget2D => renderTarget2D;

/// <summary>

/// Method for initialize factory and rander target

/// </summary>

public override void InternalInitialize()

{

base.InternalInitialize();

factory2D = new Factory();

using (var surface = BackBuffer.QueryInterface<Surface>())

{

renderTarget2D = new RenderTarget(factory2D, surface, new RenderTargetProperties(new PixelFormat(Format.Unknown, AlphaMode.Premultiplied)));

}

}

/// <summary>

/// Method for Uninitialize factory and rander target

/// </summary>

protected override void InternalUninitialize()

{

Utilities.Dispose(ref renderTarget2D);

Utilities.Dispose(ref factory2D);

base.InternalUninitialize();

}

/// <summary>

/// Method for begin render

/// </summary>

protected override void BeginRender()

{

base.BeginRender();

renderTarget2D.BeginDraw();

renderTarget2D.Clear(Color.Orange);

}

/// <summary>

/// Method for end render

/// </summary>

protected override void EndRender()

{

renderTarget2D.EndDraw();

base.EndRender();

}

}

}

**Код программы для класса *UnitTestsGame*:**

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using LinearRacing;

using SharpDX;

using SharpDX.Direct2D1;

using System.Windows.Input;

using System.Collections.Generic;

namespace UnitTestGame

{

[TestClass]

public class UnitTest1

{

GameField gameField = new GameField();

Car[] carZone;

Road[] roadZone;

List<IPrize> prizesZone = new List<IPrize>();

Player player;

Player rival;

Car car;

/// <summary>

/// Test Method for cheaking Collision Player with car on top

/// </summary>

[TestMethod]

public void TestMethodPlayerCollisionTopCarTrue()

{

player = new PlayerCar(new Vector2(100, 50), new Vector2(40, 40), 3, 1);

car = new Car(new Vector2(100, 65), new Vector2(40, 40), 3, 1);

Assert.IsTrue(GameObject.CheckCollision(player, car));

}

/// <summary>

/// Test Method for cheaking Collision Player with car on top

/// </summary>

[TestMethod]

public void TestMethodPlayerCollisionTopCarFalse()

{

player = new PlayerCar(new Vector2(100, 50), new Vector2(40, 40), 3, 1);

car = new Car(new Vector2(160, 65), new Vector2(40, 40), 3, 1);

Assert.IsFalse(GameObject.CheckCollision(player, car));

}

/// <summary>

/// Test Method for cheaking Collision Player with car on left

/// </summary>

[TestMethod]

public void TestMethodPlayerCollisionLeftCar()

{

player = new PlayerCar(new Vector2(100, 50), new Vector2(40, 40), 3, 1);

car = new Car(new Vector2(60, 50), new Vector2(40, 40), 3, 1);

Assert.IsTrue(GameObject.CheckCollision(player, car));

}

/// <summary>

/// Test Method for cheaking Collision Player with car on right

/// </summary>

[TestMethod]

public void TestMethodPlayerCollisionRightCar()

{

player = new PlayerCar(new Vector2(100, 50), new Vector2(40, 40), 3, 1);

car = new Car(new Vector2(100, 85), new Vector2(40, 40), 3, 1);

Assert.IsTrue(GameObject.CheckCollision(player, car));

}

/// <summary>

/// Test Method for cheaking Collision Player with car on bottom

/// </summary>

[TestMethod]

public void TestMethodPlayerCollisionBottomCar()

{

player = new PlayerCar(new Vector2(100, 50), new Vector2(40, 40), 3, 1);

car = new Car(new Vector2(139, 55), new Vector2(40, 40), 3, 1);

Assert.IsTrue(GameObject.CheckCollision(player, car));

}

/// <summary>

/// Test Method for cheaking Collision Player with prize

/// </summary>

[TestMethod]

public void TestMethodPlayerFuelPrize()

{

int outValue;

FuelPrize fuelPrize;

int resultFuelPlayer = 60;

int startFuel = 50;

player = new PlayerCar(new Vector2(100, 50), new Vector2(40, 40), 3, 1);

player.AmountOfFuel = startFuel;

roadZone = new Road[4] {new Road(new Vector2(620, 150), new Vector2(20,70), player.Speed, player.Id),

new Road(new Vector2(620, 300), new Vector2(20,70), player.Speed,player.Id),

new Road(new Vector2(620, 450), new Vector2(20,70), player.Speed,player.Id),

new Road(new Vector2(620, 600), new Vector2(20,90), player.Speed, player.Id)};

carZone = new Car[3] {new Car(new Vector2(140, 40), new Vector2(55, 55), 4f, 1),

new Car(new Vector2(220, 20), new Vector2(55, 55), 0.8f, 1),

new Car(new Vector2(300, 20), new Vector2(55, 55), 3f, 1)};

fuelPrize = new FuelPrize(new Vector2(100, 50), new Vector2(40, 40), 3, 1);

prizesZone.Add(fuelPrize);

gameField.GameFieldProcess(carZone, player, rival, roadZone, prizesZone, 1, out outValue);

Assert.AreEqual(resultFuelPlayer, player.AmountOfFuel);

}

/// <summary>

/// Test Method for cheaking Collision Player with delete fuel prize

/// </summary>

[TestMethod]

public void TestMethodPlayerDeleteFuelPrize()

{

int outValue;

DeleteFuelPrize deleteFuel;

int resultFuelRival = 85;

player = new PlayerCar(new Vector2(100, 50), new Vector2(40, 40), 3, 1);

roadZone = new Road[4] {new Road(new Vector2(620, 150), new Vector2(20,70), player.Speed, player.Id),

new Road(new Vector2(620, 300), new Vector2(20,70), player.Speed,player.Id),

new Road(new Vector2(620, 450), new Vector2(20,70), player.Speed,player.Id),

new Road(new Vector2(620, 600), new Vector2(20,90), player.Speed, player.Id)};

carZone = new Car[3] {new Car(new Vector2(140, 40), new Vector2(55, 55), 4f, 1),

new Car(new Vector2(220, 20), new Vector2(55, 55), 0.8f, 1),

new Car(new Vector2(300, 20), new Vector2(55, 55), 3f, 1)};

rival = new PlayerCar(new Vector2(400, 50), new Vector2(40, 40), 3, 2);

rival.AmountOfFuel = 100;

deleteFuel = new DeleteFuelPrize(new Vector2(100, 80), new Vector2(40, 40), 3, 1);

prizesZone.Add(deleteFuel);

gameField.GameFieldProcess(carZone, player, rival, roadZone, prizesZone, 1, out outValue);

Assert.AreEqual(resultFuelRival, rival.AmountOfFuel);

}

/// <summary>

/// Test Method for cheaking Player wining

/// </summary>

[TestMethod]

public void TestMethodPlayerWin()

{

int outValue;

player = new PlayerCar(new Vector2(100, 50), new Vector2(40, 40), 3, 1);

player.Length = 400;

roadZone = new Road[4] {new Road(new Vector2(620, 150), new Vector2(20,70), player.Speed, player.Id),

new Road(new Vector2(620, 300), new Vector2(20,70), player.Speed,player.Id),

new Road(new Vector2(620, 450), new Vector2(20,70), player.Speed,player.Id),

new Road(new Vector2(620, 600), new Vector2(20,90), player.Speed, player.Id)};

carZone = new Car[3] {new Car(new Vector2(140, 40), new Vector2(55, 55), 4f, 1),

new Car(new Vector2(220, 20), new Vector2(55, 55), 0.8f, 1),

new Car(new Vector2(300, 20), new Vector2(55, 55), 3f, 1)};

rival = new PlayerCar(new Vector2(400, 50), new Vector2(40, 40), 3, 2);

gameField.GameFieldProcess(carZone, player, rival, roadZone, prizesZone, 1, out outValue);

Assert.AreEqual(outValue, 2);

}

}

}

**ПРИЛОЖЕНИЕ Б**

(обязательное)

**Руководство пользователя**

1. Введение.

Руководство пользователя обеспечивает получение пользователем базовых навыков по эксплуатации игрового приложения.

Разработанное программное приложение предназначено для организации игрового процесса между двумя игроками. Оно предполагает наличие двух игроков, которые соревнуются в прохождении всей трассы, встречая на своем пути преграды и призы. Управление игроком осуществляется с помощью клавиатуры.

Программное средство обладает следующим функционалом:

* наличие окна для игровых настроек;
* игровое поле;
* управление игроком при помощи клавиатуры;
* наличие преград и призов.

Ход использования приложения:

Для использования программного средство пользователь должен быть ознакомлен с:

* настоящим руководством пользователя;
* правилами использования ЭВМ.

1. Назначение и условия применения.

Разработанное программное приложение предназначено для симуляции управления транспортным средством, помогающим почувствовать себя за рулем автомобиля на гоночной трассе.

Для корректной работы программного средства необходимо соблюдение следующих требований:

* поддерживаемые операционные системы *Windows XP* и выше;
* наличие стандартной клавиатуры;
* наличие следующих устройств вывода: экран, подключаемый по *HDMI*.

1. Подготовка к работе.

Для установки приложения необходимо загрузить на компьютер каталог *LinearRacing*.После этого в папке с программой зайти в каталог *bin/Debag* и запустить файл *LinearRacing.exe*.

Если все инструкции соблюдены и приложение не выдает никаких сообщений об ошибках, значит, программа работает исправно.

1. Описание операций

В ходе разработки игрового приложения были реализованы окна настройки игры и само поле, на котором происходит основное действие игры. В окне меню игрок имеет возможность выбрать транспортное средство, которым будет управлять в течение всего игрового процесса. Выбор машины реализуется через выпадающий список. Чтобы выбрать машину нужно выбрать элемент из выпадающего списка. Ниже идет кнопка старта, при нажатии на которую открывается окно, в котором происходит главное действие игры (рисунок Б.1).

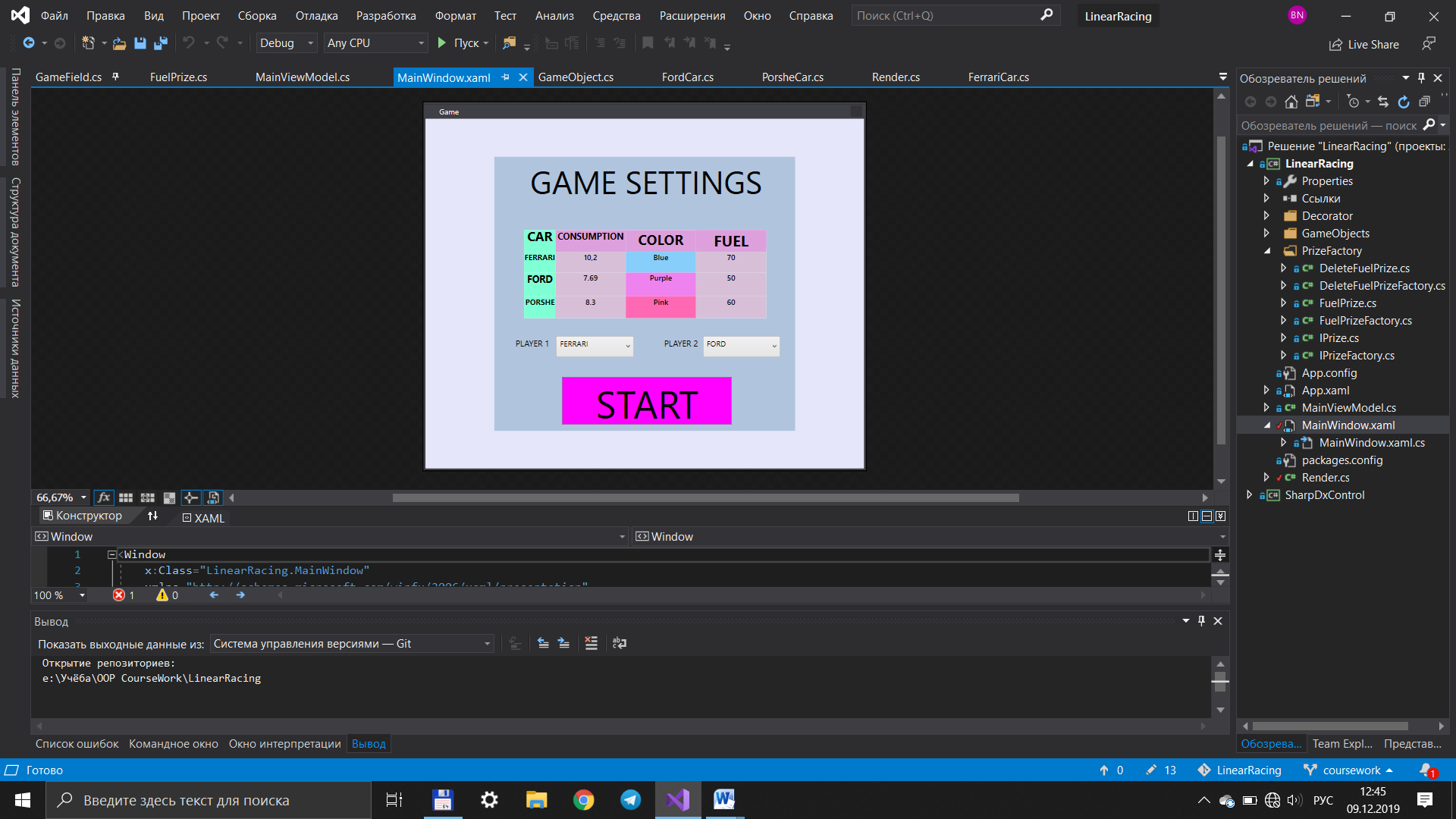


Рисунок Б.1 – Окно настроек игры

Окно с игровым полем представляет собой разделенный на две части экран (рисунок Б.2).

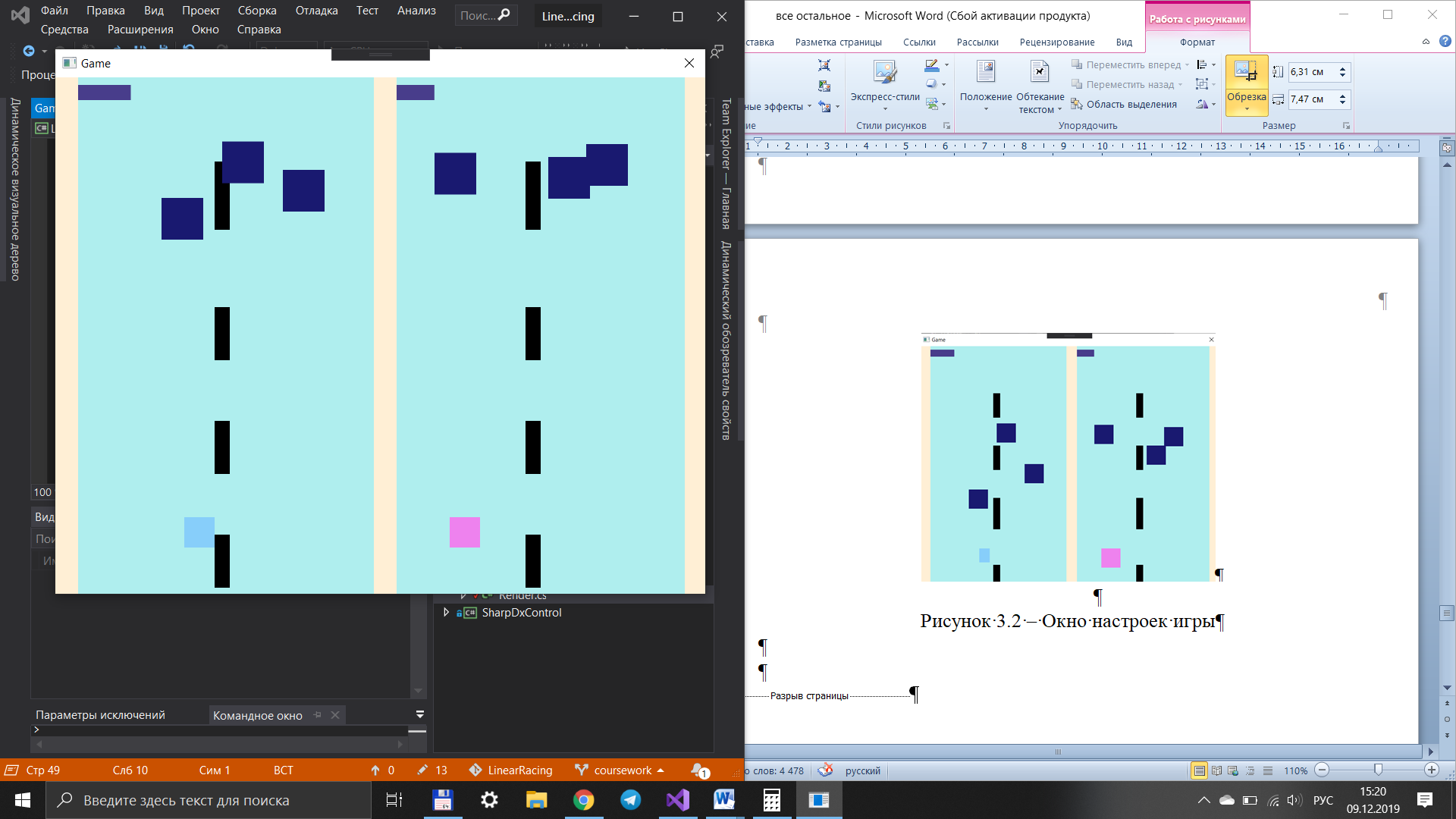


Рисунок Б.2 – Игровое поле приложения

На дороге случайным образом появляются препятствия в виде других машин. Противники окрашены в темно-синий цвет, они больше по размеру и имеют скорость, отличную от игрового персонажа. Игроку необходимо следить за этим индикатором, чтобы успевать подбирать топливо. Подборка топлива осуществляется за счет сбора призов (рисунок Б.3).

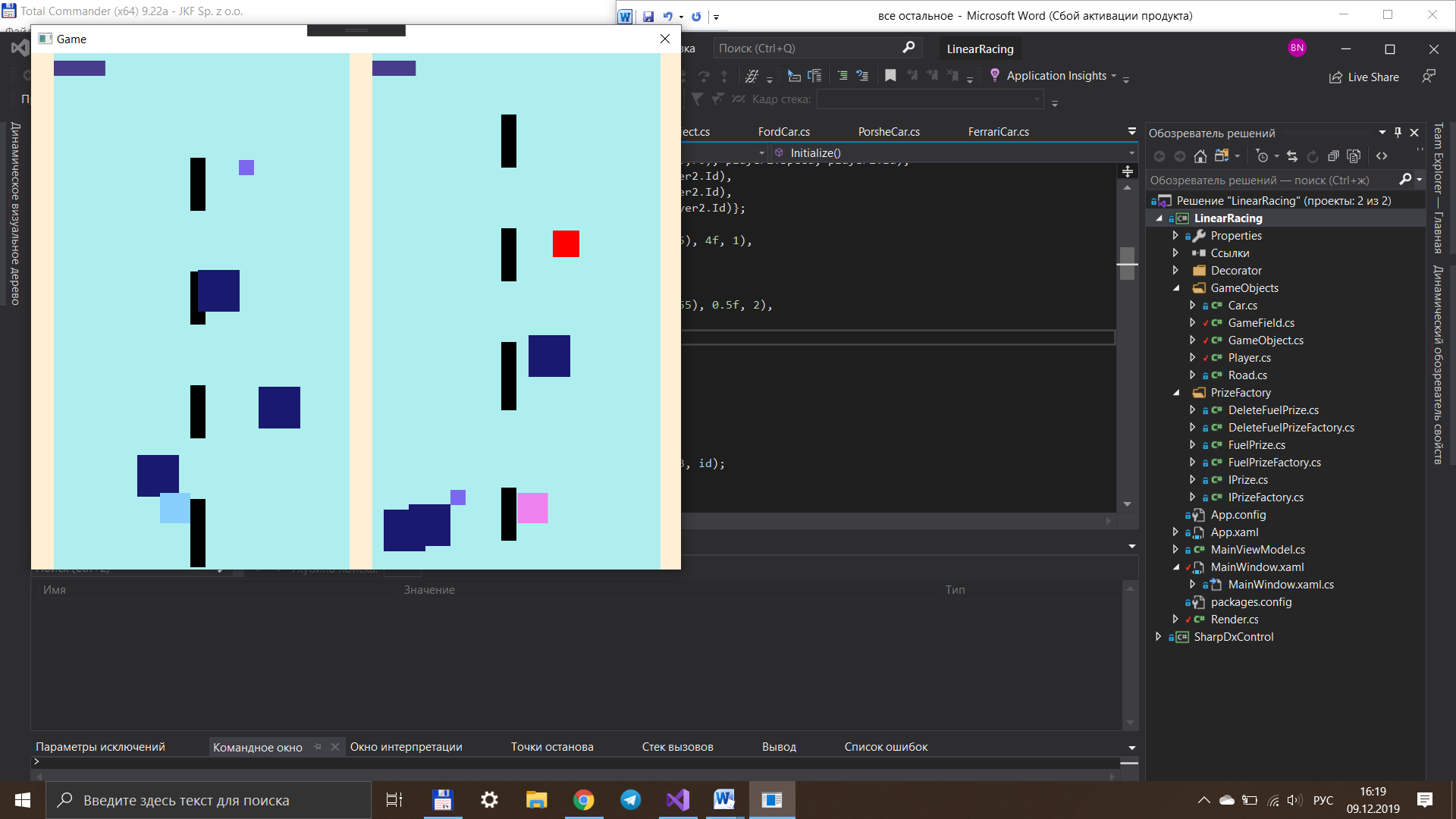


Рисунок Б.3 – Призы на игровом поле

Управление игроком осуществляется при помощи клавиш *W* или *Up* – для увеличения скорости, *S* или *Down* – для уменьшения скорости, *A* или *Left* – для перемещения игрока влево, *D* или *Right* – для перемещения игрока вправо соответственно.

1. Аварийные ситуации.

Чтобы избежать ошибок при использовании данного игрового приложения, необходимо соблюдать порядок действий и условия использования, описанные в главе 3 данного руководства пользователя.

В случае непредвиденного «зависания» программы рекомендуется завершить процесс в диспетчере задач и запустить снова. Или можно подождать некоторое время, так как существует возможность зависания программы за счет наличия библиотек, используемых в игровом приложении.

6.Рекомендации по освоению.

Игровое приложение рекомендуется запускать при наличии двух игроков для правильного функционирования.

**ПРИЛОЖЕНИЕ В**

(обязательное)

**Руководство программиста**

1. Назначение и условия применения.

Разработанное программное приложение предназначено для организации игрового процесса между двумя игроками. Жанр игры – «Линейные гонки». Он предполагает наличие двух игроков, которые соревнуются в прохождении всей трассы, встречая на своем пути преграды и призы. Управление игроком осуществляется с помощью клавиатуры.

Для корректной работы программного средства необходимо соблюдение следующих требований:

* поддерживаемые операционные системы *Windows XP* и выше;
* наличие стандартной клавиатуры;
* наличие следующих устройств вывода: экран, подключаемый по *HDMI*.

1. Характеристики программы.

При запуске открывается окно настроек для игрового приложения, в котором выбирается тип игрока, затем нажимается кнопка старт и игроки имеют возможность управлять выбранными устройствами.

1. Обращение к программе

Приложение запускается путем открытия файла *LinearRacing.exe*, находящимся в каталоге *bin/Debag*.

1. Входные и выходные данные.

Входной информацией являются данные, получаемые с окна настроек, которые определяют тип игрового объекта. Выходной информацией является экран с результатом игрового процесса.

1. Сообщения.

В ходе выполнения приложения никаких сообщений не предусмотрено.

**ПРИЛОЖЕНИЕ Г**

(обязательное)

**Руководство системного программиста**

1. Общие сведения о программе.

Разработанное программное приложение предназначено для организации игрового процесса между двумя игроками. Жанр игры – «Линейные гонки». Он предполагает наличие двух игроков, которые соревнуются в прохождении всей трассы, встречая на своем пути преграды и призы. Управление игроком осуществляется с помощью клавиатуры.

Для корректной работы программного средства необходимо соблюдение следующих требований:

* поддерживаемые операционные системы *Windows XP* и выше;
* наличие стандартной клавиатуры;
* наличие следующих устройств вывода: экран, подключаемый по *HDMI*.

2. Структура программы.

Структурно приложение разделено на две рабочие области – область настроек для игры и область игрового поля.

Окно с настройками игры состоит из:

* таблицы характеристик для разных типов машин;
* выпадающих списков для выбора игрока;
* кнопки старт для начала игры.

Окно с игровым полем состоит из двух разделенных экранов для каждого пользователя.

1. Настройка программы

Настройка программы осуществляется путем запуска исполняемого файла.

1. Проверка программы.

Отсутствие сообщения об ошибках свидетельствует, о корректной работе приложения.

5.Дополнительные возможности

Дополнительных возможностей программа не имеет

6.Сообщения системному программисту.

В ходе выполнения программы необходимо выбрать тип машины и нажать кнопку старта. Других действий программа не требует.

