# ВВЕДЕНИЕ

На данный момент, видеоигры являются очень популярным и легкодоступным источником удовольствия. В них можно играть как одному, так и в компании друзей. Видеоигры доступны на большинстве современных гаджетов: смартфонах, компьютерах, консолях, телевизорах и даже на некоторых электронных наручных часах. В видеоигры играют люди всех возрастных категорий потому что это весело и они, как правило, имеют низкий порог вхождения, удобное управление и красивую графику.

В настоящее время компьютерные видеоигры помимо развлечения также несут обучающий, расслабляющий и даже лечебный характер. Ведь ни для кого не секрет, что всего лишь пара часов проведённые за любимой игрой может поднять настроение и избавить игрока от депрессии. Видеоигры сближают людей и помогают им лучше узнать друг друга и найти общий язык. Некоторые игры так быстро набирают популярность и захватывают молодую аудиторию, что по ним проводят соревнования, в которых любой желающий игрок может принять участие и побороться за главный приз и всеобщую славу.

Бурное развитие игровой индустрии привело к появлению различных жанров и игровых механик, которые были необходимы для того, чтобы сделать игру уникальной среди большого многообразия похожих друг на друга продуктов. Одним из таких представителей жанров в видеоиграх являются гонки.

Гонки – достаточно древний жанр видеоигр. Он зародился со времён игровых автоматов и развивается до сих пор, уживаясь в смешанных жанрах. Этот жанр включает в себя огромный спектр видов этого жанра: от простой аркадной гоночной игрой до авто-симулятором с уклоном на реализм. В данном случае, мы будем рассматривать кольцевые гонки – классический вид игр-гонок.

Данный вид гонок не сложный в реализации благодаря своей сути, – игрок должен доехать первым на протяжении одного или нескольких кругов. Кольцевые гонки являются хорошим примером для изучения тем, кто хочет понимать основы игр, что, как я думаю, и является целью данной работы.

# ИГРОВЫЕ ПРИЛОЖЕНИЯ И СРЕДСТВА ИХ РАЗРАБОТКИ

## Особенности игрового жанра «Аркада»

Компьютерные игры появились почти сразу же с самим началом развития ИТ отрасли. Первой компьютерной игрой считается *SpaceWar*. Она была создана в далёком шестьдесят первом году двадцатого века в Массачусетском Технологическом университете на одном из мэйнфреймов. Геймплей этой игры был весьма примитивен, и заключался в том, что по экрану летали два небольших кораблика и стреляли друг в друга снарядами. Так мы приходим к выводу, что первый игровой жанр – это аркада.

По началу в аркадные игры можно было поиграть только в аркадных игровых автоматах, расцвет которых охарактеризован с конца семидесятых, до середины восьмидесятых, его часто называют золотым веком аркадных игр. Через время из-за технического прогресса комплектующие становились дешевле и доступнее для обычного человека, что способствовало появлению игровых приставок. По итогу к концу девяностых игровые автоматы, не выдержав конкуренции покинут индустрию.

Аркада – жанр, характеризующийся динамическим игровым процессом, но коротким по времени.

Классические аркады характеризуются следующими пунктами:

* Действия игры происходят на одной карте. Это связано с техническими ограничением железа того времени, но в тоже время это значительно повлияло на геймдизайн.
* Бесконечная игра. У игры нет конца, это означает, что игроки потенциально могли играть в неё бесконечно. Геймдизайнеры проектировали игры так, чтобы со временем играть было всё сложнее и сложнее пока не проиграешь.
* Множество попыток. Игра даёт множество попыток для того, чтобы игрок смог быстро адаптироваться к игровым механикам, и быстро не забросить игру.
* Игровой счёт. Он показывает игроку как хорошо играл. На основании этого в играх как правило есть таблица рекордов, где игрок может сравнивать себя с другими.
* Отсутствует сюжет. Игре нужно, чтобы игрок быстро осознал, что происходит – это научная фантастика, война, спорт или что-то ещё.

## Особенности спрайтовой графики для игровых приложений

В наше время спрайт – растровый объект, который легко двигается по игровому экрану, способен динамично менять форму и цвет не искажая фон.

Слово «спрайт» придумал в 1970-е г в *Texas* *Instruments*: их новая микросхема *TMS*9918 могла отображать небольшие картинки поверх неподвижного фона

Спрайт чаще всего состоит из нескольких растровых изображений собранных в специальную библиотеку, из которой их можно поочерёдно и быстро вызывать, создавая эффект анимации объекта. Благодаря этому утки летают, трава колышется, собака ходит и т.д.

Для того что бы получить хороший спрайт, необходимо придерживаться нескольких основополагающих принципов.

* Во-первых, для получения плавной анимации, схожим с движением реальных тел в живой природе, необходимо стараться делать отдельные фазы движения с минимальными изменениями. Глаз сам поможет обмануть наблюдателя.
* Во-вторых, если спрайт замкнутый т.е. после последнего кадра идёт первый, то различия между первым и последним кадром должно быть минимальным.
* В-третьих, на первых парах нельзя перемещать спрайты по экрану, а нужно ограничиться печатью всей последовательности попеременно в одной позиции. Это даёт возможность легко обнаружить ошибки.

В двух мерной графике изначально под спрайтами понимали небольшие изображения, которые выводились на экран при аппаратном ускорении. На некоторых машинах (*MSX* 1, *NES*) программная прорисовка приводила к определённым ограничениям, а аппаратные спрайты этого ограничения не имели. С увеличением мощности центрального процессора понятие «спрайт» стало шире и распространилось на всех двухмерных персонажей. Например, в таких играх как *Super* *Mario* и *Heroes* *of* *Might* *and* *Magic* вся графика спрайтовая.

В первых трёх мерных играх, например, *Wolfenstein* 3*D* (1992) и *Doom* (1993), были спрайтовые персонажи. Они изменяли размер в зависимости от расстояния (Рисунок 1.1).



Рисунок 1.1 – Игра *Wolfenstein* 3*D*

## Движки для создания игр

Существует множество игровых движков. Среди современных выделим:

* *Unity* является самым популярным игровым движком в отрасли. Существует бесплатная версия, которая может использоваться для создания 2*D* и 3*D* игр, но если вы зарабатываете на игре больше ста тысяч долларов в год, то придётся отчислять часть разработчикам движка. Он поддерживает почти все платформы это: мобильные устройства, *VR*, десктоп, консоли и веб платформы. Среди известных игр, сделанных в *Unity*: *Pok*é*mon* *Go*, *Cuphead*, *The* *Long* *Dark*.
* *Unreal* *Engine* 4 является вторым по популярности движок для разработки игр. На этом движке выходили такие игры, как: *Final* *Fantasy* *VII* *Remake* и *Tekken* 7. На *Unreal* *Engine* 4 можно бесплатно создавать игры, но как только ваш доход превышает порог в 3000$, вам необходимо выплачивать 5% роялти. Можно не платите за роялти если вы создаёте: проект для кино, архитектурные проекты и визуализации.
* *CryEngine* стал популярен у разработчиков благодаря выходу *Crysis*. Игра показала новый уровень графики, которая впечатляет по сей день. Следом вышли множество версий *CryEngine* с поддержкой новых технологий. Сейчас движок известен такими масштабными играми, как: *Prey*, *Sniper*: *Ghost* *Warrior* 3, *Heathen*, *Far* *Cry*, *Ryse*: *Son* *of* *Rome*.

## Языки программирования

Существует гигантское количество языков программирования, для разработки игры подойдут любые, начиная от *C* и *Python* до *Java* и *Pascal*. Разница только во времени, которое уходит на разработку игры и для какой платформы. Также язык влияет и на производительность.

Самые распространённые и популярные в этом направлении являются следующие языки:

* С++. Очень функциональный и сложный язык, изучение потребует много времени, усидчивости и реальной заинтересованности. Это статически типизированный компилируемый язык программирования, поддерживающий основные парадигмы: объектно-ориентированное, обобщенное, процедурное программирование. Сильная сторона С++ является совместимость с языком С, что означает наличие совместимости библиотек, написанных на С, и очень объемной стандартной библиотеки, включающей массу полезных функций: ввод/вывод, многопоточность, удобные алгоритмы и контейнеры. Язык подходит для написания собственных игровых движков, способен функционировать на разных платформах, он отлично сочетается с другими средствами разработки, что делает его универсальным и беспроигрышным вариантом для создания игр с нуля.
* *C*#. Си Шарп был создан как язык для приложений и хорошо подходит для разработки игр. Как игровой язык он имеет много важных характеристик и преимуществ. Это полностью объектно-ориентированный язык, разработанный в недрах компании *Microsoft*. Синтаксис Си Шарп очень близок к синтаксису всего семейства языков С, а также *Java*. В основном на нем разрабатываются игры на ПК, а если более конкретно – то на платформу .*NET* *Framework*. Язык поддерживает полиморфизм, перегрузку операторов и имеет статическую типизацию. Возможностей *C*# вполне достаточно, чтобы разработать полноценную игру: ее логику, архитектуру и другие важные элементы. Языки *C* для программирования игр используются уже давно и в целом очень успешно [4, c.120].
* *Java*. Хорошо подходит для разработки игр, особенно под мобильные, в частности на операционную систему *Android*. Как и Си Шарп он способен справиться с созданием логики игр, ее механики и других важных нюансов. *Java* помогает работать с многими потоками, что очень важно для игр и для самого Андроид. Также это язык дает возможность легко взаимодействовать с памятью устройств, что в играх тоже немаловажно.

*C*++, например, позволяет писать игры кроссплатформенными, а *PHP* с *JavaScript* идеально подходят для разработки браузерных игр. Если вы хотите писать игры на каком-либо движке, то необходимо знать *C*# – его используют для того, чтобы прописывать скрипты. Главное – правильно выбрать язык для данной поставленной задачи. *Unity* дружит не только с Си Шарп, а также и с *JavaScript*, а *Minecraft* вообще был написан на *Java*.

## Структура игрового кода

В зависимости от того, что мы выберем (разрабатывать игру на движке или с нуля), будет влиять на то, как будет строиться игровой код.

Если делать всё своими руками, то придётся прописывать физику, механику, графику, искусственный интеллект и баланс. Если мы выбираем движок, то о выше перечисленным сильно заморачиваться не придётся.

Физика – это то, как мир игры реагирует на действия игрока или объектов внутри мира. Вот какие могут быть физические действия:

* ходьба;
* езда;
* прыжки;
* удары;
* выстрелы;
* падение предметов и так далее.

Если писать физику самим, то для обычного прыжка придется:

* проверить, находится ли игрок на земле;
* изменять координату *Y*, чтобы игрок поднимался вверх;
* со временем закончить подъём;
* продолжать падение до тех пор, пока под ногами игрока не окажется земля.

Также необходимо будет поработать над анимацией всего этого.

Все эти действия уже прописаны в движках, и нужно всего лишь подогнать её под свои нужды. Для примера:

* объект был помещён в рабочую область *Unity*;
* указано, что для него нужно использовать физику;
* при запуске игры объект упадёт.

И для этого не придётся писать код вообще – всё уже предусмотрено.

## Возможности графической библиотеки *OpenGL*

*OpenGL* – это программный интерфейс для взаимодействия с графическим процессором. В этот интерфейс входит приблизительно 300 отдельных команд, которые используются для визуализации двухмерной или трёхмерной графики в реальном времени. Например, в играх.

Библиотека *OpenGL* создана как платформонезависимый, он может использоваться для любого аппаратного обеспечения. Поэтому в саму *OpenGL* не входят функции для создания окон или для захвата пользовательского ввода; для этих задач вам необходимо использовать средства той операционной системы, в которой работаете. Также в *OpenGL* нет высокоуровневых функций для описания моделей трехмерных или двухмерных объектов [1].

Для вывода изображения на экран используются основные графические операции *OpenGL*.

* Строит фигуры из геометрических примитивов, создавая математическое описание объектов (примитивами в *OpenGL* считаются точки, линии, полигоны, битовые карты и изображения).
* Позиционирует объекты в трехмерном пространстве и выбирает точку наблюдения для осмотра полученной композиции.
* Вычисляет цвета для всех объектов. Цвета могут быть определены приложением, получены из расчета условий освещенности, вычислены при помощи текстур, наложенных на объекты или из любой комбинации этих факторов.
* Преобразует математическое описание объектов и ассоциированной с ними цветовой информации в пиксели на экране. Этот процесс называется растеризацией (или растровой разверткой).

Также на протяжении этих операций *OpenGL* может выполнять и другие операции, например, удаление частей фигуры, спрятанных за другой фигурой. Если необходимо вы можете производить некоторые операции с пиксельными данными, после того, как сцена растеризована, но до того, как она выводится на экран.

В *OpenGL* почти все команды похожи порядком этап обработки, называемый конвейером визуализации *OpenGL*. Этот порядок показан на рисунке 1.2 и, хотя он не является жестким правилом для всех реализаций, тем не менее, дает представление о том, что делает *OpenGL*.

Диаграмма показывает, что *OpenGL* обрабатывает данные как конвейер. Геометрические данные (вершины, линии и полигоны) проходят путь, включающий оценку и повершинные операции, в то время как пиксельные данные (пиксели, изображения и битовые карты) в части общего процесса обрабатываются иначе. Оба типа данных проходят через одинаковые финальные шаги (растеризация и операции над фрагментами) до того, как результирующие пиксельные данные записываются в буфер кадра.

Все данные, геометрические или пиксельные могут быть сохранены в списках *display* *lists* для текущего или последующего использования. (Альтернативой занесению данных в списки является их немедленная обработка, известная как непосредственный режим.) Когда исполняется список, сохраненные в нем данные обрабатываются также, как если бы они поступали в непосредственном режиме.

Все геометрические примитивы описываются своими вершинами. Параметрические кривые и поверхности могут быть изначально описаны с помощью контрольных точек и полиномиальных функций, называемых базисными функциями. Вычислители предоставляют метод для определения реальных вершин, представляющих поверхность, по ее контрольным точкам. Этот метод полиномиальная аппроксимация, он позволяет получить нормаль поверхности, координаты текстуры, цвета и значения координат в пространстве [5, c.69].

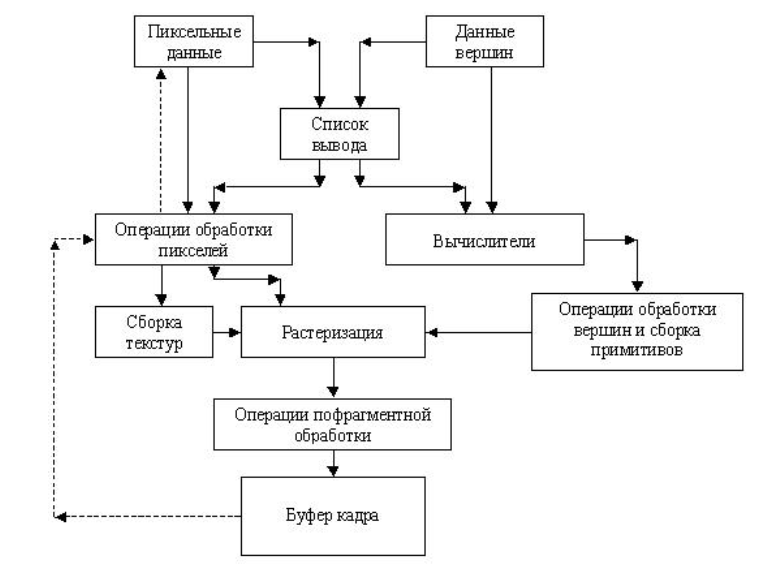


Рисунок 1.2 – Графический конвейер *OpenGL*

Для геометрических данных следующим этапом является выполнение повершинных операций (*per*-*vertex* *operations*). В течение этого этапа вершины преобразуются в примитивы. Некоторые типы вершинных данных трансформируются матрицами чисел с плавающей точкой размерности 4х4. Пространственные координаты проецируются с позиции в 3*D* мире в позицию на вашем экране. Если активизированы дополнительные возможности библиотеки *OpenGL*, то этот этап становится еще более сложным. Если используется текстурирование, координаты текстуры могут быть сгенерированы и изменены на этом шаге. Если используется освещение, вычисления, связанные с ним, производятся с использованием трансформированных вершин, нормалей поверхностей, позиций источников света, свойств материала, а также другой информации, позволяющей вычислить цветовую величину.

Отсечение – большая часть сборки примитивов – это уничтожение частей геометрии, выпадающих за полупространство, определенное плоскостью. Отсечение точек просто отвергает или не отвергает вершину; отсечение линий или полигонов может добавить дополнительные вершины в зависимости от ситуации (того, как именно линия или полигон отсекаются). В любом случае, после этого процесса производится перспективное разделение, которое заставляет более дальние объекты выглядеть меньше, чем ближние. Затем выполняются операции с портом просмотра (*viewport*) и глубиной (координатой *z*). Если включено распознавание лицевых граней, и примитив является полигоном, то на этом шаге грань может быть отвергнута в зависимости от теста на лицевые грани. В зависимости от режима рисования полигонов, они могут быть нарисованы в виде точек или линий. Результатом этого этапа являются завершенные примитивы, то есть трансформированные и отсеченные вершины со связанными цветом, глубиной и, иногда, координатами текстуры.

В то время как геометрические данные движутся по конвейеру своим путем, пиксельные данные двигаются иным маршрутом. Первым делом массивы данных из системной памяти распаковываются, то есть преобразуются из какого-либо формата, в формат с необходимым числом компонент. Далее данные масштабируются, базируются и обрабатываются пиксельными картами. Результат сжимается и записывается в текстурную память или отправляется на этап растеризации. Если пиксельные данные читаются из буфера кадра, над ними выполняются пиксельные операции (*pixel*-*transfer* *operations*). Затем результаты упаковываются в соответствующий формат и возвращаются в массив в системной памяти. Существуют специальные операции копирования пикселей (*pixel* *copy* *operations*) для копирования данных из одной части буфера кадра в другие или из буфера кадра в текстурную память

Приложения *OpenGL* могут накладывать текстурные изображения на геометрические объекты, чтобы заставить их выглядеть более реалистично. Если используется несколько изображений текстур, разумно поместить их в объекты текстуры, чтобы можно было легко переключаться между ними. Некоторые реализации *OpenGL* могут иметь дополнительные ресурсы для ускорения операций с текстурами. Например, может существовать специализированная быстрая текстурная память. Если такая память присутствует, текстурным объектам могут быть назначены приорит

Растеризация – это процесс преобразования геометрических и пиксельных данных во фрагменты. Каждый фрагмент соответствует пикселю в буфере кадра. Шаблоны линий и полигонов, толщина линии, размер точек, модель заливки, вычисления, связанные с наложением для поддержки сглаживания, принимаются в расчет при развертке двух вершин в линию или вычислении внутренних пикселей полигона. Каждый фрагмент имеет ассоциированные с ним значения цвета и глубины.

До того, как величины будут сохранены в буфере кадра, над ними производится серия операций, которые могут изменить или даже выбросить некоторые фрагменты. Все эти операции могут быть включены или выключены. Первая операция, которая может быть произведена – это текстурирование, когда из текстурной памяти для каждого фрагмента генерируется и накладывается на него тексел (элемент текстуры). Также могут производиться (в порядке выполнения) вычисления тумана, тест отреза (*scissor* *test*), альфа-тест, тест трафарета (*stencil* *test*) и тест буфера глубины (для удаления невидимых поверхностей). Если фрагмент не проходит один из включенных тестов, это может закончить его путь по конвейеру. Далее могут быть произведены наложение, смешивание цветов (*dithering*), логические операции и маскирование с помощью битовой маски. Наконец, фрагмент заносится в соответствующий буфер, где становится пикселем.

К преимуществам *OpenGl* можно отнести его независимость от платформ и производителей. Программа с использованием этой библиотеки сможет работать на операционных системах *Windows*, *MacOS*, *Linux*, *Solaris* и других. *OpenGl* использовался даже в написании первых игр для игровых консолей *Xbox* 360 и *Playstation* 2. Широкий выбор дополнений и спецификаций, постоянные обновления библиотеки предоставляют возможность написания программ на таких языках программирования, как: С, С++, *Fortran*, *Java*, а также *C*#, после подключения специальной надстройки *SharpGL* или *OpenTK*.

## Понятие сетевого взаимодействия

Коммуникация в сети представляет собой следующее: вся сеть состоит из отдельных элементов – хостов, которые представляют собой компьютеры и другие подключенные устройства. Между собой они соединены каналами связи (кабели *Ethernet*, *Wi*-*Fi* и т. д.) и маршрутизаторами. Маршрутизаторы объединяют компьютеры в подсети и контролируют передачу данных между ними.

Но компьютеры-хосты не взаимодействуют абы как между собой. Они применяют протоколы. Протокол представляет собой соглашения о том, как пакеты данных будут передаваться по каналам коммуникации. Таким образом, протокол упорядочивает взаимодействие.

Существует множество различных протоколов. Протоколы, которые используются для передачи данных по сети, составляют семейство протоколов *TCP*/*IP*. Основные из них: *Internet* *Protocol* (*IP*), *Transmission* *Control* *Protocol* (*TCP*) и *User* *Datagram* *Protocol* (*UDP*). Причем эти протоколы организованы в уровневую систему.

На рисунке 1.3 представлена уровневая система сетевых протоколов.

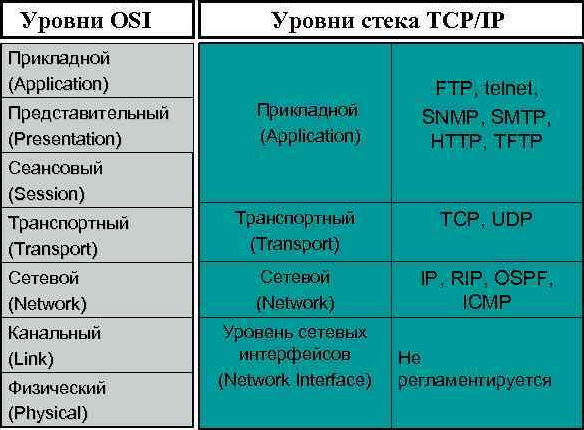


Рисунок 1.3 – Уровневая система сетевых протоколов

*IP* представляет сетевой уровень. Он использует нижележащие уровни, которые представляют физические каналы коммуникации – кабели *Ethernet* и т. д., для передачи пакетов с данными другому хосту.

Выше *IP* располагается транспортный уровень, который образуют протоколы *TCP* и *UDP*. Эти протоколы используют определенные порты для передачи данных. *TCP* позволяет отследить потерю пакетов и их дублирование при передаче. *UDP* подобного не позволяет сделать и нацелен на простую передачу данных. Однако приложение взаимодействует с уровнем *TCP* / *UDP* не напрямую, а через специальный *API*, который предоставляют сокеты. Сокеты – это не какой-либо протокол, это просто интерфейс для создания сетевых приложений, который опирается на встроенные возможности операционной системы.

В зависимости от используемого протокола различают два вида сокетов: потоковые сокеты (*stream* *socket*) и дейтаграммные сокеты (*datagram* *socket*). Потоковые сокеты используют протокол *TCP*, дейтаграммные – протокол *UDP*.

В итоге, когда приложение посылает сообщение приложению, запущенному на другом хосте, то приложение обращается к сокетам для передачи данных на уровень *TCP* / *UDP*. Далее с этого транспортного уровня данные передаются сетевому уровню – уровню протокола *IP*. И этот протокол передает данные далее физическим уровням, и после этого данные уходят по сети.

Чтобы уникально определять хосты в сети каждый хост имеет адрес. Существует несколько различных протоколов адресов. В настоящее время наиболее распространен протокол *IPv*4, который предполагает представление адреса в виде 32-битного числа, например, 37.120.16.63. Такой адрес содержит четыре числа, разделенных точками, и каждое число находится в диапазоне от 0 до 255. Однако также в последнее время набирает оборот использование адресов протокола *IPv*6, которые представляют собой 128-битное значение.

Однако такие адреса очень сложно запомнить, поэтому в реальности чаще оперируют доменами. Домены представляют специальные названия, используемые для интернет-адресов. Например, есть доменное имя "*www*.*microsoft*.*com*", ему соответствует адрес в формате *IPv*4 2.23.143.150. Но для протокола *IP*, через который идет взаимодействие, доменные адреса не существуют. Поэтому при отправке или передаче данных по доменному имени, компьютер еще обращается к службам *Domain* *Name* *System* (*DNS*), который выполняют сопоставление между интернет-адресами в формате *IPv*4 или *IPv*6 и доменными названиями.

Кроме адреса при сетевых взаимодействиях используются порты. Порт представляет 16-битное число в диапазоне от единицы до 65535. Использование портов позволяет разграничить несколько запущенных приложений на одном хосте.

## Протокол *UDP* и сравнение с *TCP*

*UDP* (англ. *User* *Datagram* *Protocol* – протокол пользовательских датаграмм) – один из ключевых элементов набора сетевых протоколов для Интернета. С *UDP* компьютерные приложения могут посылать сообщения (в данном случае называемые датаграммами) другим хостам по *IP*-сети без необходимости предварительного сообщения для установки специальных каналов передачи или путей данных. Протокол был разработан Дэвидом П. Ридом в 1980 году и официально определён в *RFC* 768.

*UDP* использует простую модель передачи, без явных «рукопожатий» для обеспечения надёжности, упорядочивания или целостности данных. Датаграммы могут прийти не по порядку, дублироваться или вовсе исчезнуть без следа, но гарантируется, что если они придут, то в целостном состоянии. *UDP* подразумевает, что проверка ошибок и исправление либо не нужны, либо должны исполняться в приложении. Чувствительные ко времени приложения часто используют *UDP*, так как предпочтительнее сбросить пакеты, чем ждать задержавшиеся пакеты, что может оказаться невозможным в системах реального времени. При необходимости исправления ошибок на сетевом уровне интерфейса приложение может задействовать *TCP* или *SCTP*, разработанные для этой цели.

Природа *UDP* как протокола без сохранения состояния также полезна для серверов, отвечающих на небольшие запросы от огромного числа клиентов, например *DNS* и потоковые мультимедийные приложения вроде *IPTV*, *Voice* *over* *IP*, протоколы туннелирования *IP* и многие онлайн-игры.

Сравнение с *TCP*. *TCP* – ориентированный на соединение протокол, что означает необходимость «рукопожатия» для установки соединения между двумя хостами. Как только соединение установлено, пользователи могут отправлять данные в обоих направлениях.

Надёжность – *TCP* управляет подтверждением, повторной передачей и тайм-аутом сообщений. Производятся многочисленные попытки доставить сообщение. Если оно потеряется на пути, сервер вновь запросит потерянную часть. В *TCP* нет ни пропавших данных, ни (в случае многочисленных тайм-аутов) разорванных соединений.

Упорядоченность – если два сообщения последовательно отправлены, первое сообщение достигнет приложения-получателя первым. Если участки данных прибывают в неверном порядке, *TCP* отправляет неупорядоченные данные в буфер до тех пор, пока все данные не могут быть упорядочены и переданы приложению.

Тяжеловесность – *TCP* необходимо три пакета для установки сокет-соединения перед тем, как отправить данные. *TCP* следит за надёжностью и перегрузками.

Потоковость – данные читаются как поток байтов, не передается никаких особых обозначений для границ сообщения или сегментов.

*UDP* – более простой, основанный на сообщениях протокол без установления соединения. Протоколы такого типа не устанавливают выделенного соединения между двумя хостами. Связь достигается путём передачи информации в одном направлении от источника к получателю без проверки готовности или состояния получателя. В приложениях для голосовой связи через интернет-протокол (*Voice* *over* *IP*, *TCP*/*IP*) *UDP* имеет преимущество над *TCP*, в котором любое «рукопожатие» помешало бы хорошей голосовой связи. В *VoIP* считается, что конечные пользователи в реальном времени предоставят любое необходимое подтверждение о получении сообщения.

Ненадёжный – когда сообщение посылается, неизвестно, достигнет ли оно своего назначения – оно может потеряться по пути. Нет таких понятий, как подтверждение, повторная передача, тайм-аут.

Неупорядоченность – если два сообщения отправлены одному получателю, то порядок их достижения цели не может быть предугадан.

Легковесность – никакого упорядочивания сообщений, никакого отслеживания соединений и т. д. Это небольшой транспортный уровень, разработанный на *IP*.

Датаграммы – пакеты посылаются по отдельности и проверяются на целостность только если они прибыли. Пакеты имеют определенные границы, которые соблюдаются после получения, то есть операция чтения на сокете-получателе выдаст сообщение целиком, каким оно было изначально послано.

Нет контроля перегрузок – *UDP* сам по себе не избегает перегрузок. Для приложений с большой пропускной способностью возможно вызвать коллапс перегрузок, если только они не реализуют меры контроля на прикладном уровне.

# ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ «КОЛЬЦЕВЫЕ ГОНКИ»

## Метод декомпозиции в игровом приложении «Кольцевые гонки»

Для разработки приложения используется собственный игровой движок, который должен уметь работать со спрайтовой графикой. Именно в нем должна присутствовать возможность вывода спрайтов и двумерных анимаций. Так же для реализации игры разрабатываемый движок должен уметь просчитывать физику движения тел в двумерном пространстве. Также используются такие паттерны программирования, как декоратор и фабрика. Декоратор будет применяться для изменений характеристик автомобиля игрока, а фабрика – для генерации призов, которые будут появляться на карте.

В игре должны присутствовать призы такие, как топливо, патроны, колёса. Игрокам разрешается выезжать за пределы трассы, но, в случае, если игрок находиться за границей трассы, то его скорость замедляется.

Расход топлива должен зависеть от скорости движения, а также, в начале игры, топлива должно быть недостаточно для прохождения всей трассы.

При проектировании архитектуры игрового движка применяется метод декомпозиции. Декомпозиция – это научный метод, использующий структуру задачи и позволяющий заменить решение одной большой задачи решением серии меньших задач, пусть и взаимосвязанных, но более простых. На рисунке 2.1 проиллюстрирован принцип декомпозиции [6].

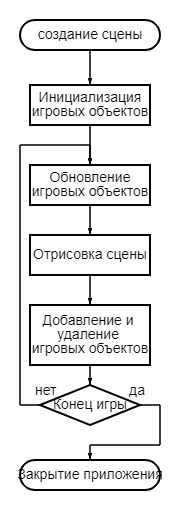


Рисунок 2.1 – Принцип декомпозиции

Парадигма декомпозиции в компьютерном программировании – это стратегия организации программы в виде нескольких частей, обычно подразумевающая особый способ организации текста программы. Обычно целью использования парадигмы декомпозиции является оптимизация некоторой метрики, связанной со сложностью программы, например, модульностью программы или ее ремонтопригодностью.

Большинство парадигм декомпозиции предлагают разбить программу на части, чтобы минимизировать статические зависимости между этими частями и максимизировать связность каждой части.

Процесс декомпозиции позволил разработать обобщённый алгоритм работы проекта игровой логики нашего приложения. Ключевым элементом в алгоритме является сцена, которая создает, отображает, динамически добавляет и удаляет элементы в игре. В алгоритме каждый блок действий представляет собой этап разработки, который разбивается на подзадачи.

Каждый этап разработки позволяет получить информацию о необходимых ресурсах и действиях и, при необходимости, сведения о реализации данных действий. После сбора информации можно чётко обозначить цель и последовательность небольших задач для её достижения. Также, оценивая каждый этап разработки, легче сформировать структуру приложения.

Главным преимуществом использования декомпозиции является решение задачи с минимальными затратами и максимальной гибкостью. Каждый этап можно протестировать и выявить ошибки, а после заняться рефакторингом кода. Это позволяет избежать возникновения ошибок при объединении этапов.

Данная схема классов движка предоставляет всё необходимое для последующего написания игрового приложения, а также большую гибкость для его последующих модификаций и доработок.

## Описание паттернов проектирования для игрового приложения «Кольцевые гонки»

Далее рассматриваются паттерны программирования, которые будут использоваться в игровом приложении, а именно: фабричный метод и декоратор.

Фабричный метод (*Factory* *method*) – это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Данный паттерн применяется в следующих случаях:

* когда вы хотите дать возможность пользователям расширять части вашего фреймворка или библиотеки;
* когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код;
* когда вы хотите экономить системные ресурсы, повторно используя уже созданные объекты, вместо порождения новых.

«Фабричный метод» работает по следующему алгоритму:

* продукт определяет общий интерфейс объектов, которые может произвести создатель и его подклассы;
* конкретные продукты содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий;
* создатель объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов. Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт. Несмотря на название, важно понимать, что создание продуктов не является единственной функцией создателя. Обычно он содержит и другой полезный код работы с продуктом. Аналогия: большая софтверная компания может иметь центр подготовки программистов, но основная задача компании – создавать программные продукты, а не готовить программистов [7];
* конкретные создатели по-своему реализуют фабричный метод, производя те или иные конкретные продукты. Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы возвращать существующие объекты из какого-то хранилища или кэша.

Из преимуществ этого паттерна стоит выделить то, что он позволяет сделать код создания объектов более универсальным, не привязываясь к конкретным классам (*ConcreteProduct*), а оперируя лишь общим интерфейсом (*Product*). Также, этот паттерн позволяет установить связь между параллельными иерархиями классов.

Единственным существенным недостатком является то, что использование данного паттерна может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

Декоратор (*Decorator*) – это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Для определения нового функционала в классах нередко используется наследование. Декораторы же предоставляет наследованию более гибкую альтернативу, поскольку позволяют динамически в процессе выполнения определять новые возможности у объектов.

Данный паттерн применяется в следующих случаях:

* когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует;
* когда нельзя расширить обязанности объекта с помощью наследования.

«Декоратор» работает по следующему алгоритму:

* компонент задаёт общий интерфейс обёрток и оборачиваемых объектов;
* конкретный компонент определяет класс оборачиваемых объектов. Он содержит какое-то базовое поведение, которое потом изменяют декораторы;
* базовый декоратор хранит ссылку на вложенный объект-компонент. Им может быть, как конкретный компонент, так и один из конкретных декораторов. Базовый декоратор делегирует все свои операции вложенному объекту. Дополнительное поведение будет жить в конкретных декораторах;
* конкретные декораторы – это различные вариации декораторов, которые содержат добавочное поведение. Оно выполняется до или после вызова аналогичного поведения обёрнутого объекта;
* клиент может оборачивать простые компоненты и декораторы в другие декораторы, работая со всеми объектами через общий интерфейс компонентов [8].

Из преимуществ этого паттерна стоит выделить тот факт, что с ним нет необходимости создавать подклассы для расширения функциональности объекта, а также есть возможность динамически подключать новую функциональность до или после основной функциональности объекта *ConcreteComponent*.

Двумя существенными недостатками данного паттерна являются: трудность конфигурировать многократно обёрнутые структуры и обилие крошечных классов.

## Принципы взаимодействия классов в игровом приложении «Кольцевые гонки»

Игровое приложение должно предоставлять следующий функционал:

* перемещение игроков по экрану;
* отображение игровых объектов;
* регистрация завершения игры;
* декорирование характеристик автомобилей игроков;
* генерация призов для игроков;
* обработка ввода пользователя.

Перед тем как приступить к написанию программного кода, требуется спроектировать архитектуру графического приложения и иерархию всех игровых интерфейсов и классов. Но для начала нужно выделить и описать основные сущности гоночного приложения, без которых реализовать игру было бы невозможно. К таким сущностям относятся следующие игровые объекты:

* гоночная машина, для которой требуется реализовать методы перемещения по игровой трассе и использовать все различные бонусы. Для данной сущности будет характерно иметь такие характеристики как скорость передвижения и направление движения, чтобы корректно реализовать основные методы класса;
* приз, главной задачей которого является изменение характеристик гоночного автомобиля при столкновении;
* гоночная трасса. Её основные задачи: генерировать на себе со временем все разновидности игровых призов, разрешать коллизии при перемещении игровых объектов и содержать в себе всё необходимое для реализации игрового процесса;
* стена. Требуется для того, чтобы гонщик не мог выехать за пределы автотрассы или же срезать круг.

В игровом приложении реализуется шаблон проектирования «Декоратор», который обеспечивает добавление игрокам нового функционала, такого как уменьшение/увеличение скорости игрока. Схема декоратора в игровом приложении показана на рисунке 2.2.

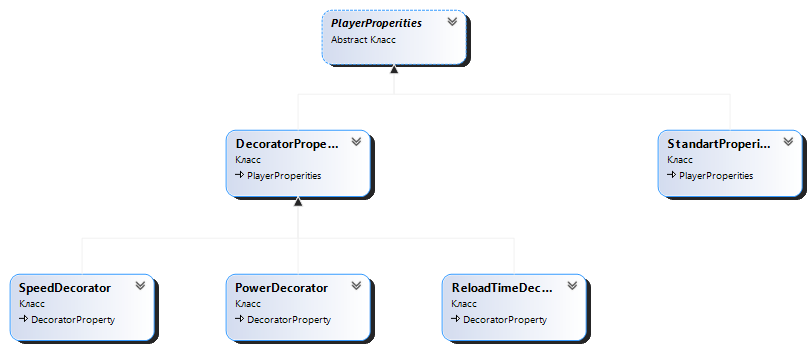


Рисунок 2.2 – Схема декоратора игрового приложения

Класс *PlayerProperities* является абстрактным и содержит в себе информацию присущую игроку, такую как запас здоровья, боезапас, скорость, время перезарядки и т.д.

Класс *StandartProperities* наследуется от *PlayerProperities* и определяет характеристики игрока, а именно: количество патронов, скорости, время перезарядки оружия, топлива, количества колёс и т.п.

Класс *DecoratorProperty* наследуется от *PlayerProperities* и содержит в себе конструктор объектов, позволяющий настраивать характеристики игрока. От него наследуются три класса – *ReloadTimeDecorator*, *SpeedDecorator* и *PowerDecorator*. *ReloadTimeDecorator* влияет на перезарядку оружия, *SpeedDecorator* – на скорость игрока, а *PowerDecorator* – на силу игрока в столкновениях.

Присутствует так же реализация паттерна «Фабричный метод», который применяется для генерации различных призов на игровом поле. В начале был создан абстрактный класс *PrizeFactory*, который содержит в себе данные присущие любому призу. От него наследуются конкретные призы, такие как топливо (*FuelPrizeFactory*), патроны (*AmmoPrizeFactory*), колёса (*SpeedPrizeFactory*). Сами призы генерируются в определенных точках трассы случайным образом. В течение гонки на трассе постепенно может находится больше десятка призов одновременно, чтобы сохранить баланс игрового процесса и не дать явного преимущества какому-то игроку. Схема фабричного метода показано на рисунке 2.3.

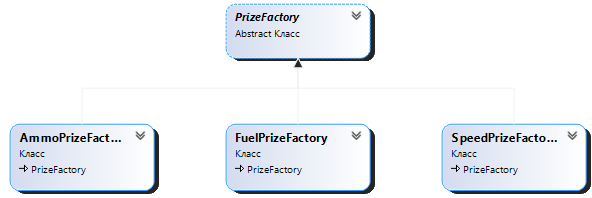


Рисунок 2.3 – Схема фабричного метода в игровом приложении

Новые объекты игры можно добавить на игровое поле с помощью метода *Start*() в классе *SpawnManager*. При добавлении объекта определяется его тип и в зависимости от него объект добавляется по ссылке в соответствующую коллекцию. Помимо добавления имеющиеся на игровом поле объекты можно удалять с помощью *RemoveGameObject*(), передавая в параметр метода *ID* игрового объекта. Так же имеются методы *Update*() для просчета всех изменившихся координат объектов, и отрисовки *Draw*() всех объектов в цикле в соответствии с их реализацией метода *Draw*(). Метод обработки клавиатуры *GetButtonDown*() проверяет все нажатия клавиш для каждого игрока, путем вызова у них функции *KeyboardMovement*(). Метод *PlayerInteraction*() определяет был ли совершен выстрел каким-либо игроком и, в зависимости от расположения и направления движения этого игрока, создает новый объект пули *Bullet* с соответствующими характеристиками движения.

Библиотека *EngineLibrary* включает в себя основной абстрактный класс *GameObject*, в котором реализована каждая игровая сущность и ее атрибуты как «сценарий», позиция на игровом поле, тэг, активна ли она и ее текстура, в который входят такие компоненты, как:

* *colliderComponent* – базовый класс, описывающий твёрдое тело игрового объекта, хранит его размер и вершины, обновляемые методом *Update* каждый кадр. Также имеет перегруженный метод *CheckGameObjectIntersection* для проверки пересечения с другими объектами, обладающими твёрдыми телами;
* *systemCollider* – базовый класс, описывающий систему твёрдых тел, обеспечивающую возможность каждой части игрового объекта иметь своё твёрдое тело, для этого класс имеет массив классов *ColliderComponent* и методы, позволяющие их добавлять и удалять. Также имеется метод *GetNumberColliderIntersection*, который выводит номер пересекающегося коллайдера из имеющихся у игрового объекта;
* *objectScript* – базовый абстрактный класс, характеризующий «сценарий» объекта на сцене. Имеет абстрактные методы *Start*, инициализирующий игровой объект при создании, и *Update*, описывающий поведение этого объекта в каждом кадре;
* *transform* – базовый класс, хранит габариты и местоположение объекта. Метод *SetMovement* смещает объект на заданную величину по горизонтали и по вертикали;
* *texture*2*D* – базовый класс, описывающий двухмерную текстуру. Хранит размеры тектуры и его *id*;
* *textureBox* – класс-наследник интерфейса *IDisposable*, предназначенный для работы с текстурами. Содержит методы, позволяющие добавлять и удалять текстуры, а также, в зависимости от ситуации, перегруженным методом *Set* заменяет текущую текстуру на необходимую из имеющихся на определённое время.

Некоторые игровые процессы зависят от времени, и для этого необходим статический класс *Time*. Этот класс содержит свойство, описывающее время *CurrentTime* с момента запуска приложения, и свойство *DeltaTime* с подсчитанным временем между кадрами, обновление необходимо производить каждый кадр, вызывая метод *UpdateTime*.

Класс *ContentPipe* работает с тектурами, имеет методы *LoadTexture* и *DeletTexture*, обеспечивающие загрузку и удаление текстур с оперативной памяти.

Отрисовка текстур игровых объектов производится в классе *SpriteBatch* статическим методом *Draw*, в нём считывается размер, положение и вершины текстур.

Для управления игроками используется класс *Input*, позволяющий взаимодействовать с вводом с клавиатуры. Методом *GetButtonDown* возращается реакция на нажатие клавиши ввода. Методом *GetAxis* можно узнать ось направления движения, например, для прицела игрока.

Библиотека *GameLibrary* включает в себя следующие классы:

* *bullet* – класс-наследник *ObjectScript*, реализующий основную логику пули, хранит в себе поле скорости и методы, реализующие взаимодействие с игроком;
* *player* – класс-наследник *ObjectScript*, который реализует логику игрока, хранит в себе управление игроком, скорости игрока, изменение характеристик;
* *spawnManager* – класс-наследник *ObjectScript*, описывающий поведение игрового объекта в игре;
* *gameEvents* – статический класс событий игры, который хранит в себе делегаты и события игры;
* *maps* – базовый класс, хранит габариты и местоположение объекта. Метод *SetMovement* смещает объект на заданную величину по горизонтали и по вертикали; *ElementsFactory* – Класс создания элементов местности;
* *elementsFactory* – Класс создания элементов местности;
* *maps* – базовый класс, хранит габариты и местоположение объекта. Метод *SetMovement* смещает объект на заданную величину по горизонтали и по вертикали;
* *prizeSpawn* – класс-наследник *ObjectScript*, отвечающий за подбирание оружия во время игры.

# ВЕРИФИКАЦИЯ РАЗРАБОТАННОГО ИГРОВОГО ПРИЛОЖЕНИЯ «КОЛЬЦЕВЫЕ ГОНКИ»

## Принцип работы игрового приложения «Кольцевые гонки»

Для того чтобы приступить к визуализации игровых объектов в окне *WPF* требуется подключить графическую библиотеку *OpenGL*, после чего разместить компонент покадрового рендера на форму. Таким образом будет создан графический контекст, в котором циклически 60 раз в секунду будут отображаться все игровые геометрические примитивы объектов. Но так как *OpenGL* не предусмотрена для работы на языке *C*#, то вместо его используем *OpenTK* – его аналог на языке, в котором будем работать. Далее, потребуется установить и настроить камеру (точку зрения, с которой игроки будут наблюдать за игровым миром). Следующим шагом после того как будет создан контекст *OpenTK* (холст для рисования) и настроена камера, начнётся этап структурирования, создания и программирования игровых классов [2, с. 23].

Метод *MainWindow*() является пользовательским интерфейсом игрового приложения. В этом методе происходит инициализация основных компонентов игры, таких как игровое поле, его основные объекты, игроки, потоки параллельных вычислений. Потоки необходимы для того, чтобы обеспечить параллельное вычисление координатных компонент, не нагружая при этом поток отрисовки этими вычислениями. В данном случае это потоки просчета столкновений между объектами: игрока с игроком, игрока с ближайшими линиями дороги, определение ближайших линий дороги, столкновений игроков с призами и пулями, генерация призов через определенное время.

По заданию, генерация призов должна происходить случайным образом. Для этого при разработке игровых классов используется шаблон «фабричный метод», который было решено добавить для реализации гибкого добавления однородных разновидностей призовых объектов. Фабричный метод выпускает несколько видов призов таких, как: ускорение, топливо, замедление. Если же гоночный автомобиль подбирает приз «топливо», то его бензобак пополняется. Расход топлива гоночного автомобиля происходит не только во время движения, по игровой трассе, но и просто не двигаясь, и он прямо пропорционален текущей скорости передвижения. Чем больше текущая скорость передвижения, тем больше топлива за единицу времени расходуется у гоночного автомобиля. Для того чтобы визуально отличать и понимать смысл каждого игрового объекта необходимо наложить на них текстуры, которые своим внешним видом будут максимально хорошо ассоциироваться с описанными ранее классами.

## Результаты тестирования игрового приложения «Кольцевые гонки»

Одним из важнейших этапов разработки игровых приложений является тестирование разрабатываемого программного обеспечения. Необходимо проверить правильность взаимодействия всех элементов системы, их поведение и работоспособность.

Для тестирования основного функционала приложения был создан проект модульных тестов *GameTesting*, в котором тестируются главные функции класса *GameField*. Это функции проверки столкновения игровых объектов между собой и как следствие изменение характеристик этих объектов.

Ниже приведено описание тестовых методов приложения:

* *collisionPlayers*() – тестовый метод, проверяющий корректность метода столкновений *CollisionPlayerPlayer*() двух игроков;
* *collisionPlayerFuelPrize*() – тестовый метод, который проверяет происходит ли столкновение игрока с объектом приза;
* *collisionPlayerFuelPrize*() – тестовый метод, проверяющий правильность реализацию приза «Топливо»;
* *collisionPlayerCartridgePrize*() – тестовый метод, проверяющий правильность реализацию приза «Патроны»;
* *collisionPlayerHealthPrize*() – тестовый метод, проверяющий правильность реализацию приза «Здоровье»;
* *collisionPlayerBullets*() – тестовый метод, проверяющий корректность метода столкновений *CollisionBulletPlayer*() игрока с пулей.
* *collisionPlayerRoad*() – тестовый метод, проверяющий корректность метода столкновений *CollisionPlayerRoadLine*() игрока с линией дороги.
* *collisionPlayerCheckPoints*() – тестовый метод, проверяющий корректность метода столкновений *CollisionCheckLinePlayer*() игрока с линией чекпоинта.
* *checkLapsChange*() – тестовый метод, который проверяет проехал ли автомобиль круг, если он пересек подряд все линии чекпоинтов.
* *checkPlayerDeath*() – тестовый метод, проверяющий победителя при смерти одного их игроков.

Все модульные тесты прошли успешно, что свидетельствует о правильной реализации соответствующих методов определения столкновений, начисления очков призами, а также фиксации проезда одного круга трассы.

## Верификация игрового приложения «Кольцевые гонки»

После загрузки всех основных компонентов игрового поля, начинается игровой процесс. При старте приложения оба игрока имеют под своим управлением гоночные автомобили с полностью заправленным топливным баком, которые находятся на стартовой позиции игровой трассы. Игрокам необходимо управлять своими автомобилями с помощью клавиатуры, выполнять точные повороты, подбирать различные призы на трассе, чтобы приехать к финишу первым. Вначале оба игрока имеют автомобили с полными топливными баками и полным здоровьем. Старт игры показан на рисунке 3.1.

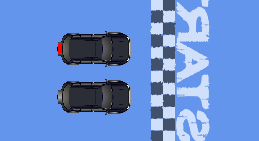


Рисунок 3.1 – Игроки на старте

В таблице 3.1 представлено игровое управление.

Таблица 3.1 – Игровое управление

|  |  |  |
| --- | --- | --- |
| Игроки | Игрок 1 | Игрок 2 |
| Передвижение вперёд/назад | W/S | Up/Down |
| Поворот налево/направо | A/D | Left/Right |
| Стрельба | Space | Numpad-0 |
| Подбор призов | C | Numpad-Enter |

Цель каждого игрока – проехать пять кругов и прийти к финишу первее соперника, не допустить уничтожения своего автомобиля пулями и контролировать запас топлива, чтобы предотвратить его полный расход до окончания гонки [3]. При нулевом запасе автомобиль заглохнет и не сможет продолжать движение. Чтобы проехать круг, игрокам необходимо проехать сначала половину круга и коснуться дорожной разметки внизу, после чего доехать до начала круга. Тогда игра засчитает круг, и при достижении пятого круга игра заканчивается. Игровой процесс игры изображён на рисунке 3.2.



Рисунок 3.2 – Игровой процесс

Характеристики каждого из игроков указываются на боковых углах экрана: слева для первого игрока, справа для второго игрока. На них отображается наличие колёс автомобился у игрока, количество кругов, пройденных игроком, количество топлива и патронов. Характеристики автомобиля игрока показана на рисунке 3.3.



Рисунок 3.3 – Характеристики игрока

В пределах трассы могут генерироваться 3 типа «призов»:

* топливо;
* патроны;
* здоровье.

Призы создаются случайным образом и располагаются в различных точках трассы, координаты которых были определены заранее, чтобы они находились в пределах трассы. Одновременно на трассе, в одном и том же месте не может находится более двух призов.

При попадании пуль в игрока его здоровье уменьшается на 1 единицу, а также снижается скорость его движения. Если игрок промазал, то при вылете пули за пределы экрана, она уничтожается. На рисунке 3.4 показан спрайт пули.

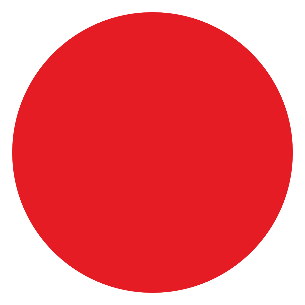


Рисунок 3.4 – Пуля

При подборе приза «топливо» игрок пополняет свой топливный бак на 5 единиц топлива. Оно является одним из важнейших призов, т.к. способствует проезду всех пяти кругов гонки. На рисунке 3.5 показан спрайт приза «Топливо».

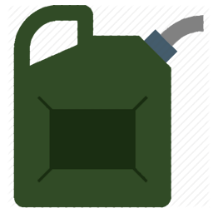


Рисунок 3.5 – Приз «Топливо»

При подборе автомобилем приз «патроны» игрок получает магазин на 10 патронов. Особенностью патронов является возможность их пролета сквозь всю трассу, что делает их опасными для игроков. Поэтому игроку необходимо уворачиваться от пуль, перемещением и поворотами вокруг оси центра автомобиля. На рисунке 3.6 показан спрайт приза «Патроны».



Рисунок 3.6 – Приз «Патроны»

При подборе приза «здоровье» игрок получает дополнительные единицы здоровья в количестве 1 единицы. Если же до этого игрок получал различные повреждения от выстрелов, тем самым уменьшив скорость своего движения, то с помощью этого приза он может увеличить свою скорость на величину уменьшения при попадании пули. На рисунке 3.7 показан спрайт приза «Колесо».



Рисунок 3.7 – Приз «Колесо»

Для того чтобы одержать победу в гонке необходимо проехать пять кругов трассы. Надписи побед для первого и второго игрока изображены на рисунках 3.8 и 3.9 соответственно.



Рисунок 3.9 – Надпись победы для первого игрока

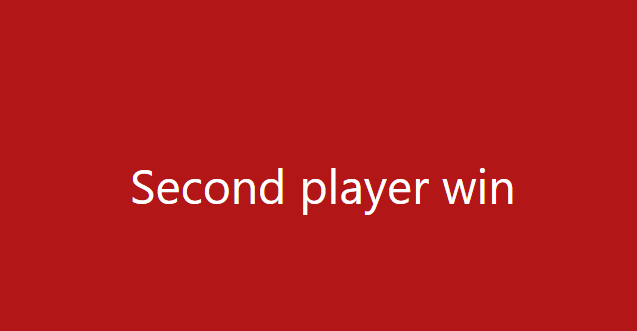


Рисунок 3.10 – Надпись победы для второго игрока

Проезжать круги необходимо только в правильном направлении, в противном случае они не будут начисляться. Если игрок приехал к финишу первым, сохранив достаточно топлива и здоровья, он увидит соответствующее сообщение о победе с номером победителя.

После завершения игры остаётся лишь два варианта развития событий. Первый вариант – это начать новую игру и посоревноваться соперником. Второй вариант – это выйти из игрового приложения.

# ЗАКЛЮЧЕНИЕ

Разработана игра жанра «кольцевые гонки» для двух игроков на одном экране, в которой каждый игрок управляет гоночным автомобилем.

В ходе разработки графического игрового приложения решены следующие задачи:

* проведён аналитический обзор игр жанра «гонки» и средств для их разработки;
* разработано игровое поле и объекты, которые на нём позиционируются;
* разработана система генерации призов и изменения характеристик гоночных автомобилей;
* применены шаблоны проектирования для придачи гибкости архитектуре игрового приложения;
* разработан программный код игры;
* проведено тестирование разработанного приложения.

При реализации игрового проекта были использованы средства языка программирования *C*# *WPF* и шаблоны проектирования: «декоратор» – для изменения характеристик гоночных автомобилей и «фабричный метод» – для реализации генерации игровых призов. При разработке графического интерфейса была использована общедоступная библиотека *OpenGL* для визуализации сцены и игровых объектов, с которыми следует взаимодействовать пользователям, а также спрайтовая графика, для придания игровым объектам реалистичного вида.

Разработанное игровое приложение состоит из кольцевой трассы, на которой со временем генерируются призы, и двух гоночных автомобилей, управляемых пользователями. Игровая трасса состоит из четырёх контрольных точек, последовательно преодолев которые гонщику засчитывается пройденный круг, и пятнадцати позиций для генерации различных типов призов случайным образом. Целью игры стала задача преодоления пяти кругов кольцевой трасы, последовательно пройдя через все контрольные точки. Также игрокам следует следить за уровнем топлива. Тот игрок, чья машина заглохнет первой, автоматически считается проигравшим.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Отличия *DirectX* и *OpenGL* [Электронный ресурс] – Режим доступа: *http*://*komiwiki*.*syktsu*.*ru*/*index*.*php*/Каковы*\_*отличия*\_DirectX\_*и*\_OpenGL* – Дата доступа: 27.11.2022.

2. Натан, Адам *WPF*4. Подробное руководство / Адам Натан. – Символ Плюс, 2011 – 880 с.

3. Эволюция жанра гонки [Электронный ресурс] – Режим доступа: *https*://*habr*.*com*/*ru*/*post*/398353/ – Дата доступа: 27.11.2022.

4. Рихтер, Джеффри *CLR* *via* *C*#. Программирование на платформе *Microsoft* .*NET* *Framework* 4.5 на языке *C*#. 4-е издание. / Джеффри Рихтер. – СПб: Питер, 2019. – 896 с.

5. *John* *Kessenich*. *OpenGL* *Programming* *Guide*. 9-е издание / *John* *Kessenich*, *Graham* *Sellers*, *Dave* *Shreiner*. – *New* *Jersey*, 2017. – 986 с.

6. Декомпозиция [Электронный ресурс] – Режим доступа: *https*://*ru*.*wikipedia*.*org*/*wiki*/Декомпозиция – Дата доступа: 27.11.2022.

7. Фабричный метод [Электронный ресурс] – Режим доступа: *https*://*metanit*.*com*/*sharp*/*patterns*/2.1.*php* – Дата доступа: 27.11.2022.

8. Декоратор [Электронный ресурс] – Режим доступа: *https*://*metanit*.*com*/*sharp*/*patterns*/4.1.*php* – Дата доступAа: 27.11.2022.

# ПРИЛОЖЕНИЕ А

(обязательное)

**Листинг игрового приложения «Кольцевые гонки»**

**Код программы для *GameObject*.*cs*:**

using OpenTK;

using OpenTK.Graphics.OpenGL;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace EngineLibrary

{

/// <summary>

/// Класс компонента, описывающий твердое тело

/// </summary>

public class ColliderComponent

{

/// <summary>

/// Лист игровых объектов имеющих твёрдое тело

/// </summary>

private static List<GameObject> collidersOfGameObjects;

/// <summary>

/// игровой объект имеющее твёрдое тело

/// </summary>

private GameObject gameObject;

/// <summary>

/// Масштаб твёрдого тела

/// </summary>

private float colliderScale;

/// <summary>

/// Смещение твёрдого тела

/// </summary>

private Vector2 offsetCollider;

/// <summary>

/// Вершины твёрдого тела

/// </summary>

public Vector2[] BoundCorners { get; private set; }

/// <summary>

/// Неактивность элемента

/// </summary>

public bool IsInactive { get; private set; }

/// <summary>

/// Конструктор класса

/// </summary>

/// <param name="gameObject">Игровой объект, которому принадлежит компонент</param>

/// <param name="scale">Размер коллайдера</param>

/// <param name="offset">Cмещения коллайдера от центра</param>

public ColliderComponent(GameObject gameObject, float scale = 1, Vector2 offset = new Vector2())

{

if (collidersOfGameObjects == null)

collidersOfGameObjects = new List<GameObject>(50);

this.gameObject = gameObject;

collidersOfGameObjects.Add(gameObject);

colliderScale = scale;

offsetCollider = offset;

IsInactive = false;

BoundCorners = new Vector2[4];

}

/// <summary>

/// Изменение активности твёрдого тела

/// </summary>

/// <param name="active"></param>

public virtual void SetIsInactive(bool active)

{

IsInactive = active;

}

/// <summary>

/// Изменение игрового объекта

/// </summary>

/// <param name="gameObject"></param>

public virtual void SetGameObject(GameObject gameObject)

{

this.gameObject = gameObject;

collidersOfGameObjects.Add(gameObject);

}

/// <summary>

/// Удаление игрового объекта из листа

/// </summary>

public virtual void DelGameObject(GameObject gameObject)

{

collidersOfGameObjects.Remove(gameObject);

}

/// <summary>

/// Проверка на пересечение со всеми объектами

/// </summary>

/// <returns>true - если есть пересечение</returns>

public virtual bool CheckGameObjectIntersection()

{

foreach (var otherGameObject in collidersOfGameObjects)

{

if (otherGameObject == gameObject || otherGameObject.Collider.IsInactive) continue;

float otherGameObjectX = otherGameObject.Transform.Position.X + (otherGameObject.Texture.Texture.Width \* otherGameObject.Transform.Scale.X);

float otherGameObjectY = otherGameObject.Transform.Position.Y + (otherGameObject.Texture.Texture.Height \* otherGameObject.Transform.Scale.Y) / 2;

if (otherGameObjectX <= gameObject.Transform.Position.X + (gameObject.Texture?.Texture.Width ?? 1) \* gameObject.Transform.Scale.X && otherGameObjectX >= gameObject.Transform.Position.X)

{

if (otherGameObjectY <= gameObject.Transform.Position.Y + (gameObject.Texture?.Texture.Height ?? 1) \* gameObject.Transform.Scale.Y && otherGameObjectY >= gameObject.Transform.Position.Y)

{

return true;

}

}

}

return false;

}

/// <summary>

/// Проверка на перечесение с данным объектом

/// </summary>

/// <param name="otherGameObject"></param>

/// <returns></returns>

public virtual bool CheckGameObjectIntersection(GameObject otherGameObject)

{

if (otherGameObject.Collider == null || otherGameObject == gameObject || otherGameObject.Collider.IsInactive) return false;

UpdateBounds();

otherGameObject.Collider.UpdateBounds();

//foreach (var otherBoundCorners in otherBoundCornersList)

//{

if (BoundCorners[1].X >= otherGameObject.Collider.BoundCorners[0].X && BoundCorners[0].X <= otherGameObject.Collider.BoundCorners[1].X

&& BoundCorners[0].Y <= otherGameObject.Collider.BoundCorners[1].Y && BoundCorners[1].Y >= otherGameObject.Collider.BoundCorners[0].Y)

return true;

//}

return false;

}

/// <summary>

/// Проверка на пересечение с данными объектами

/// </summary>

/// <param name="intersecredGameObject"></param>

/// <returns></returns>

public virtual bool CheckGameObjectIntersection(out GameObject intersecredGameObject)

{

foreach (var otherGameObject in collidersOfGameObjects)

{

if (otherGameObject.Collider == null || otherGameObject == gameObject || otherGameObject.Collider.IsInactive) continue;

UpdateBounds();

otherGameObject.Collider.UpdateBounds();

//foreach (var otherBoundCorners in otherBoundCornersList)

//{

if (BoundCorners[1].X >= otherGameObject.Collider.BoundCorners[0].X && BoundCorners[0].X <= otherGameObject.Collider.BoundCorners[1].X

&& BoundCorners[0].Y <= otherGameObject.Collider.BoundCorners[1].Y && BoundCorners[1].Y >= otherGameObject.Collider.BoundCorners[0].Y)

{

intersecredGameObject = otherGameObject;

return true;

}

//}

}

intersecredGameObject = null;

return false;

}

/// <summary>

/// Проверка на пересечение с данными объектами

/// </summary>

/// <param name="intersecredGameObject"></param>

/// <returns></returns>

public virtual bool CheckGameObjectIntersection(out GameObject intersecredGameObject, params string[] tagNames)

{

foreach (var otherGameObject in collidersOfGameObjects)

{

if (otherGameObject.Collider == null || otherGameObject == gameObject || otherGameObject.Collider.IsInactive) continue;

bool hasTag = false;

for (int i = 0; i < tagNames.Length && !hasTag; i++)

{

hasTag = otherGameObject.GameObjectTag == tagNames[i];

}

if (hasTag)

{

UpdateBounds();

otherGameObject.Collider.UpdateBounds();

if (BoundCorners[1].X >= otherGameObject.Collider.BoundCorners[0].X && BoundCorners[0].X <= otherGameObject.Collider.BoundCorners[1].X

&& BoundCorners[0].Y <= otherGameObject.Collider.BoundCorners[1].Y && BoundCorners[1].Y >= otherGameObject.Collider.BoundCorners[0].Y)

{

intersecredGameObject = otherGameObject;

return true;

}

}

}

intersecredGameObject = null;

return false;

}

/// <summary>

/// Обновление вершин твёрдого тела

/// </summary>

public virtual void UpdateBounds()

{

Vector2 position = gameObject.Transform.Position;

var angle = gameObject.Transform.Angle;

var centre = gameObject.Transform.Centre;

Vector2 buff;

Vector2[] vertices = new Vector2[2]

{

new Vector2(0, 0),

new Vector2(1, 1)

};

for (int i = 0; i < 2; i++)

{

//buff = vertices[i] - centre;

//vertices[i].X = buff.X \* (float)Math.Cos(angle) - buff.Y \* (float)Math.Sin(angle);

//vertices[i].Y = buff.X \* (float)Math.Sin(angle) + buff.Y \* (float)Math.Cos(angle);

//vertices[i] += centre;

vertices[i].X \*= gameObject.Texture?.Texture.Width ?? 1;

vertices[i].Y \*= gameObject.Texture?.Texture.Height ?? 1;

vertices[i] \*= gameObject.Transform.Scale \* colliderScale;

vertices[i] += offsetCollider + position;

}

BoundCorners = vertices;

GL.Begin(PrimitiveType.Lines);

for (int i = 0; i < 2; i++)

{

GL.Vertex2(BoundCorners[i]);

}

GL.End();

}

}

}

**Код программы для *ContentPipe*.*cs*:**

using OpenTK.Graphics.OpenGL;

using System;

using System.Collections.Generic;

using System.IO;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Drawing.Imaging;

namespace EngineLibrary

{

public class ContentPipe

{

public static Texture2D LoadTexture(string path)

{

if (!File.Exists(@"Resources\" + path))

throw new FileNotFoundException(@"File not found at `Resources\" + path + "`");

int id = GL.GenTexture();

GL.BindTexture(TextureTarget.Texture2D, id);

Bitmap bmp = new Bitmap(@"Resources\" + path);

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

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

bmp.UnlockBits(data);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapS, (int)TextureWrapMode.Clamp);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapT, (int)TextureWrapMode.Clamp);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Linear);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Linear);

int width = bmp.Width;

int height = bmp.Height;

bmp.Dispose();

return new Texture2D(id, width, height);

}

public static void DeletTexture(List<int> texturesId)

{

foreach (var id in texturesId)

GL.DeleteTexture(id);

}

}

}

**Код программы для *GameObject*.*cs*:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace EngineLibrary

{

public class GameObject

{

/// <summary>

/// Свойство, хранящее текстуру объекта

/// </summary>

public virtual TexturesBox Texture { get; protected set; }

/// <summary>

/// Класс, хранящее позицию объекта

/// </summary>

public virtual TransformComponent Transform { get; protected set; }

/// <summary>

/// Сценарий выполения

/// </summary>

public virtual ObjectScript Script { get; protected set; }

public virtual ColliderComponent Collider { get; protected set; }

/// <summary>

/// Тэг игрового объекта

/// </summary>

public string GameObjectTag { get; set; }

/// <summary>

/// Активность игрового объекта

/// </summary>

public bool IsActive { get; set; } = true;

/// <summary>

/// Конструктор игрового объекта.

/// </summary>

/// <param name="texture"></param>

/// <param name="scale"></param>

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

public GameObject()

{

//Texture = null;

//Script = null;

//Collider = null;

//Transform = null;

}

public virtual void SetComponent(object component)

{

switch (component)

{

case TexturesBox textureBox:

Texture = textureBox;

break;

case ObjectScript objectScript:

Script = objectScript;

break;

case ColliderComponent systemCollider:

Collider = systemCollider;

break;

case TransformComponent transform:

Transform = transform;

break;

}

}

public void Update()

{

if (Collider != null)

Collider.SetIsInactive(!IsActive);

if (Script != null)

Script.Update(this);

//if (!IsActive || (ParentGameObject != null && !ParentGameObject.IsActive)) return;

}

}

}

**Код программы для *Input*.*cs*:**

using OpenTK;

using OpenTK.Input;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace EngineLibrary

{

/// <summary>

/// Класс, позволяющий управлять вводом с клавиатуры

/// </summary>

public static class Input

{

/// <summary>

/// Метод, возращающий значение ввода основных осей направления

/// </summary>

/// <param name="axis">Ось направления</param>

/// <returns>Положительное или отрицательное значение оси</returns>

public static int GetAxis(AxisOfInput axis)

{

KeyboardState keystate = Keyboard.GetState();

int move = 0;

switch (axis)

{

case AxisOfInput.Horizontal:

if (keystate.IsKeyDown(Key.D)) move++;

if (keystate.IsKeyDown(Key.A)) move--;

break;

case AxisOfInput.Vertical:

if (keystate.IsKeyDown(Key.W)) move--;

if (keystate.IsKeyDown(Key.S)) move++;

break;

case AxisOfInput.AlternativeHorizontal:

if (keystate.IsKeyDown(Key.Right)) move++;

if (keystate.IsKeyDown(Key.Left)) move--;

break;

case AxisOfInput.AlternativeVertical:

if (keystate.IsKeyDown(Key.Up)) move--;

if (keystate.IsKeyDown(Key.Down)) move++;

break;

}

return move;

}

/// <summary>

/// Метод, возращающий реакцию на нажатие клавиши ввода

/// </summary>

/// <param name="key">Клавиша ввода</param>

/// <returns>Реакция true или false</returns>

public static bool GetButtonDawn(Key key)

{

KeyboardState keystate = Keyboard.GetState();

return keystate.IsKeyDown(key);

}

/// <summary>

/// Ось направления ввода

/// </summary>

public enum AxisOfInput

{

/// <summary>

/// Горизонтальная ось

/// </summary>

Horizontal = 0,

/// <summary>

/// Вертикальная ось

/// </summary>

Vertical = 1,

/// <summary>

/// Альтернативная горизонтальная ось

/// </summary>

AlternativeHorizontal = 2,

/// <summary>

/// Альтернативная вертикальная ось

/// </summary>

AlternativeVertical = 3,

}

}

}

**Код программы для *ObjectScript*.*cs*:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace EngineLibrary

{

/// <summary>

/// Абстрактный класс сценария поведения игрового объекта

/// </summary>

public abstract class ObjectScript

{

/// <summary>

/// Поведение на момент создание игрового объекта

/// </summary>

public abstract void Start(GameObject gameObject = null);

/// <summary>

/// Обновление игрового объекта

/// </summary>

public abstract void Update(GameObject gameObject);

}

}

**Код программы для *SpriteBatch*.*cs*:**

using OpenTK;

using OpenTK.Graphics.OpenGL;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace EngineLibrary

{

/// <summary>

/// Класс для отрисовки тексуры

/// </summary>

public class SpriteBatch

{

public static void Begin(int screenWidth, int screenHeight)

{

GL.MatrixMode(MatrixMode.Projection);

GL.LoadIdentity();

GL.Ortho(-screenWidth / 2f, screenWidth / 2f, screenHeight / 2f, -screenHeight / 2f, 0f, 1f);

}

/// <summary>

/// Метод для отрисовки текстуры

/// </summary>

/// <param name="gameObject"></param>

public static void Draw(GameObject gameObject)

{

var texture = gameObject.Texture.Texture;

var scale = gameObject.Transform.Scale;

var position = gameObject.Transform.Position;

var angle = gameObject.Transform.Angle;

var centre = gameObject.Transform.Centre;

Vector2 buff;

Vector2[] vertices = new Vector2[4]

{

new Vector2(0, 0),

new Vector2(1, 0),

new Vector2(1, 1),

new Vector2(0, 1)

};

GL.BindTexture(TextureTarget.Texture2D, gameObject.Texture.Texture.ID);

GL.Begin(PrimitiveType.Quads);

for (int i = 0; i < 4; i++)

{

GL.TexCoord2(vertices[i]);

buff = vertices[i] - centre;

vertices[i].X = buff.X \* (float)Math.Cos(angle) - buff.Y \* (float)Math.Sin(angle);

vertices[i].Y = buff.X \* (float)Math.Sin(angle) + buff.Y \* (float)Math.Cos(angle);

vertices[i] += centre;

vertices[i].X \*= texture.Width;

vertices[i].Y \*= texture.Height;

vertices[i] \*= scale;

vertices[i] += position;

GL.Vertex2(vertices[i]);

}

GL.End();

GL.Begin(PrimitiveType.Lines);

for (int i = 0; i < 2; i++)

{

GL.Vertex2(gameObject.Collider.BoundCorners[i]);

}

GL.End();

}

}

}

**Код программы для *Texture*2*D*.*cs*:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace EngineLibrary

{

/// <summary>

/// Класс для хранения текстуры

/// </summary>

public class Texture2D

{

private int id;

private int width, height;

/// <summary>

/// Свойство, хранящее id текстуры

/// </summary>

public int ID

{

get

{

return id;

}

}

/// <summary>

/// Ширина текстуры

/// </summary>

public int Width

{

get { return width; }

}

/// <summary>

/// Высота текстуры

/// </summary>

public int Height

{

get { return height; }

}

/// <summary>

/// Конструктор создания объекта класса

/// </summary>

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

/// <param name="width"></param>

/// <param name="height"></param>

public Texture2D(int id, int width, int height)

{

this.id = id;

this.width = width;

this.height = height;

}

}

}

**Код программы для *TextureBox*.*cs*:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace EngineLibrary

{

public class TexturesBox : IDisposable

{

public Texture2D Texture { get; private set; }

private Dictionary<string, Texture2D> TextureDictionary;

float delta = 0;

float currTime = 0;

public TexturesBox(Texture2D texture)

{

TextureDictionary = new Dictionary<string, Texture2D>();

TextureDictionary.Add("default", texture);

Texture = texture;

}

public TexturesBox()

{

TextureDictionary = new Dictionary<string, Texture2D>();

}

public void Add(string name, Texture2D texture2D)

{

TextureDictionary.Add(name, texture2D);

}

public void Del(string name)

{

TextureDictionary.Remove(name);

}

public void Edit(Texture2D texture2D, string name)

{

TextureDictionary[name] = texture2D;

}

public void Set(string name)

{

if (Time.CurrentTime - currTime >= delta)

{

Texture = TextureDictionary[name];

currTime = Time.CurrentTime;

delta = 0;

}

//changeTime -= 1;

}

public void Set(string name, float delta)

{

if (Time.CurrentTime - currTime >= this.delta)

{

Texture = TextureDictionary[name];

this.delta = delta;

currTime = Time.CurrentTime;

}

//changeTime -= 1;

}

public List<int> GetIdTextures()

{

List<int> result = new List<int>(10);

foreach (var textur in TextureDictionary)

result.Add(textur.Value.ID);

return result;

}

public void PlayAnimation()

{

//changeTime -= 1;

//if (changeTime == 0) return;

//Texture = animations[currentAnimation].GetSpriteFromAnimation();

}

public void Dispose()

{

ContentPipe.DeletTexture(GetIdTextures());

TextureDictionary.Clear();

}

}

}

**Код программы для *Time*.*cs*:**

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace EngineLibrary

{

/// <summary>

/// Статический класс управления временем

/// </summary>

public static class Time

{

private readonly static Stopwatch watch;

private static long previousTicks;

/// <summary>

/// Текущее время с запуска приложения

/// </summary>

public static float CurrentTime { get; private set; }

/// <summary>

/// Разница во времени между кадрами

/// </summary>

public static float DeltaTime { get; private set; }

/// <summary>

/// Конструктори статического класса

/// </summary>

static Time()

{

watch = new Stopwatch();

Reset();

}

/// <summary>

/// Обновление подсчитанных значений

/// </summary>

public static void UpdateTime()

{

long ticks = watch.Elapsed.Ticks;

CurrentTime = (float)ticks / TimeSpan.TicksPerSecond;

DeltaTime = (float)(ticks - previousTicks) / TimeSpan.TicksPerSecond;

previousTicks = ticks;

}

/// <summary>

/// Сброс таймера и счетчика

/// </summary>

public static void Reset()

{

watch.Reset();

watch.Start();

previousTicks = watch.Elapsed.Ticks;

}

}

}

**Код программы для *TransformComponent*.*cs*:**

using OpenTK;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace EngineLibrary

{

public class TransformComponent

{

/// <summary>

/// Позиция игрового объекта

/// </summary>

public Vector2 Position { get; set; }

/// <summary>

/// Позиция игрового объекта

/// </summary>

public Vector2 Centre { get; set; }

/// <summary>

/// Позиция игрового объекта

/// </summary>

public float Angle { get; private set; }

/// <summary>

/// Размер игрового объекта

/// </summary>

public Vector2 Scale { get; set; }

private Vector2 movementInCurrentFrame;

/// <summary>

/// Конструктор компонента

/// </summary>

/// <param name="position">начальная позиция</param>

/// <param name="scale">Начальный размер</param>

public TransformComponent(Vector2 position, Vector2 scale, Vector2 centre, float angle)

{

Position = position;

Scale = scale;

Centre = centre;

Angle = angle;

}

/// <summary>

/// Перемещение объкта

/// </summary>

/// <param name="movement">Вектор перемещения</param>

public void SetMovement(Vector2 movement)

{

movementInCurrentFrame = movement;

Position += movement;

}

/// <summary>

///

/// </summary>

/// <param name="movement">Вектор перемещения</param>

public void SetAngle(float angle)

{

Angle += angle;

if (Angle > Math.PI \* 2)

Angle -= (float) Math.PI \* 2;

else if (Angle < 0)

Angle += (float) Math.PI \* 2;

}

/// <summary>

/// Возврат позиция в этом кадре

/// </summary>

public void ResetMovement()

{

Position -= movementInCurrentFrame;

}

}

}

**Код программы для *AmmoPrizeFactory*.*cs*:**

using EngineLibrary;

using GameLibrary.GameObjects;

using GameLibrary.Prize;

using OpenTK;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.Factory

{

public class AmmoPrizeFactory : PrizeFactory

{

/// <summary>

///

/// </summary>

/// <param name="position">Позиция появления</param>

/// <returns>Игровой объект</returns>

public override GameObject CreatePrize(Vector2 position)

{

GameObject gameObject = new GameObject();

gameObject.SetComponent(new TransformComponent(position, new Vector2(0.15f, 0.15f), new Vector2(1f, 1f), 0));

gameObject.SetComponent(new TexturesBox(ContentPipe.LoadTexture("MazeElements/Prize/ammo.png")));

gameObject.SetComponent(new ColliderComponent(gameObject, 0.5f, new Vector2(10, 5)));

gameObject.GameObjectTag = "Spawn";

PrizeSpawn speedPrize = new PrizeSpawn();

speedPrize.InitializePrizeSpawn(TypeProperty.Ammo, 10, 5f);

speedPrize.Start(gameObject);

gameObject.SetComponent(speedPrize);

return gameObject;

}

}

}

**Код программы для *BulletFactory*.*cs*:**

using EngineLibrary;

using GameLibrary.GameObjects;

using OpenTK;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.Factory

{

public class BulletFactory

{

/// <summary>

/// Создание игрового объекта пули, которая убивает

/// </summary>

/// <param name="position">Позиция появления пули</param>

/// <param name="direction">Направление пули</param>

/// <param name="tag">Тег игрового объекта, создающий пулю</param>

/// <returns>Игровой объект</returns>

public GameObject CreateBullet(Vector2 position, Vector2 direction, string tag, float power = 1)

{

GameObject gameObject = new GameObject();

gameObject.SetComponent(new TransformComponent(position, new Vector2(1f, 1f), new Vector2(0, 0), 0));

TexturesBox texture;

texture = new TexturesBox(ContentPipe.LoadTexture("Bullets/b\_right.png"));

texture.Add("Down", ContentPipe.LoadTexture("Bullets/b\_down.png"));

texture.Add("Up", ContentPipe.LoadTexture("Bullets/b\_up.png"));

texture.Add("Left", ContentPipe.LoadTexture("Bullets/b\_left.png"));

texture.Add("Right", ContentPipe.LoadTexture("Bullets/b\_right.png"));

gameObject.SetComponent(texture);

gameObject.SetComponent(new ColliderComponent(gameObject, 1, new Vector2(10, 10)));

gameObject.GameObjectTag = "Bullet";

Bullet bullet = new Bullet();

bullet.SetSettings(direction, tag, power);

bullet.Start(gameObject);

gameObject.SetComponent(bullet);

return gameObject;

}

}

}

**Код программы для *FuelPrizeFactory*.*cs*:**

using EngineLibrary;

using GameLibrary.GameObjects;

using GameLibrary.Prize;

using OpenTK;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.Factory

{

public class FuelPrizeFactory : PrizeFactory

{

/// <summary>

///

/// </summary>

/// <param name="position">Позиция появления</param>

/// <returns>Игровой объект</returns>

public override GameObject CreatePrize(Vector2 position)

{

GameObject gameObject = new GameObject();

gameObject.SetComponent(new TransformComponent(position, new Vector2(0.15f, 0.15f), new Vector2(1f, 1f), 0));

gameObject.SetComponent(new TexturesBox(ContentPipe.LoadTexture("MazeElements/Prize/fuel.png")));

gameObject.SetComponent(new ColliderComponent(gameObject, 0.5f, new Vector2(10, 5)));

gameObject.GameObjectTag = "Spawn";

PrizeSpawn fuelPrize = new PrizeSpawn();

fuelPrize.InitializePrizeSpawn(TypeProperty.Fuel, 5, 5f);

fuelPrize.Start(gameObject);

gameObject.SetComponent(fuelPrize);

return gameObject;

}

}

}

**Код программы для *PrizeFactory*.*cs*:**

using EngineLibrary;

using OpenTK;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.Factory

{

public abstract class PrizeFactory

{

/// <summary>

///

/// </summary>

/// <param name="position">Позиция появления</param>

/// <returns>Игровой объект</returns>

public abstract GameObject CreatePrize(Vector2 position);

}

}

**Код программы для *SpeedPrizeFactory*.*cs*:**

using EngineLibrary;

using GameLibrary.GameObjects;

using GameLibrary.Prize;

using OpenTK;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.Factory

{

public class SpeedPrizeFactory : PrizeFactory

{

/// <summary>

///

/// </summary>

/// <param name="position">Позиция появления</param>

/// <returns>Игровой объект</returns>

public override GameObject CreatePrize(Vector2 position)

{

GameObject gameObject = new GameObject();

gameObject.SetComponent(new TransformComponent(position, new Vector2(0.15f, 0.15f), new Vector2(1f, 1f), 0));

gameObject.SetComponent(new TexturesBox(ContentPipe.LoadTexture("MazeElements/Prize/tire.png")));

gameObject.SetComponent(new ColliderComponent(gameObject, 0.5f, new Vector2(10, 5)));

gameObject.GameObjectTag = "Spawn";

PrizeSpawn speedPrize = new PrizeSpawn();

speedPrize.InitializePrizeSpawn(TypeProperty.Tires, true, 5);

speedPrize.Start(gameObject);

gameObject.SetComponent(speedPrize);

return gameObject;

}

}

}

**Код программы для *Bullet*.*cs*:**

using EngineLibrary;

using GameLibrary.Factory;

using OpenTK;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.GameObjects

{

public class Bullet : ObjectScript

{

/// <summary>

/// Скорость пули

/// </summary>

public float Speed { get; } = 300;

/// <summary>

/// Экземпляр сцены игры

/// </summary>

protected Game game;

private Vector2 flyDirection;

private string[] interactionTag = new string[2];

protected string tag;

protected float power;

/// <summary>

/// Установление направления полета пули

/// </summary>

/// <param name="direction">Вектор направления</param>

/// <param name="tag">Тег игрового объекта, создающий пулю</param>

public void SetSettings(Vector2 direction, string tag, float power)

{

this.tag = tag;

flyDirection = direction;

this.power = power;

if(tag == "PlayerOne")

interactionTag[0] = "PlayerTwo";

else

interactionTag[0] = "PlayerOne";

}

/// <summary>

/// Поведение на момент создание игрового объекта

/// </summary>

public override void Start(GameObject gameObject = null)

{

game = Game.instance;

}

/// <summary>

/// Обновление игрового объекта

/// </summary>

public override void Update(GameObject gameObject)

{

Vector2 movement = flyDirection \* Speed \* Time.DeltaTime;

gameObject.Transform.SetMovement((global::OpenTK.Vector2)movement);

if (flyDirection.X > 0)

gameObject.Texture.Set("Right");

else if (flyDirection.X < 0)

gameObject.Texture.Set("Left");

else if (flyDirection.Y > 0)

gameObject.Texture.Set("Down");

else if (flyDirection.Y < 0)

gameObject.Texture.Set("Up");

//if (gameObject.Collider.CheckGameObjectIntersection("Wall"))

//{

// game.AddObjectsToRemove(gameObject);

//}

if (gameObject.Collider.CheckGameObjectIntersection(out GameObject interGameObject, interactionTag))

{

PlayerInteraction(interGameObject);

game.AddObjectsToRemove(gameObject);

}

}

/// <summary>

/// Взаимодействие с игроком

/// </summary>

public void PlayerInteraction(GameObject playerGameObject)

{

(playerGameObject.Script as Player).Property.Tires = false;

(playerGameObject.Script as Player).Property = new SpeedDecorator((playerGameObject.Script as Player).Property);

(playerGameObject.Script as Player).ChangeStatsValue(playerGameObject, false);

}

}

}

**Код программы для *Player*.*cs*:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using EngineLibrary;

using GameLibrary;

using GameLibrary.Factory;

using GameLibrary.Maze;

using OpenTK;

using OpenTK.Graphics.OpenGL;

using OpenTK.Input;

using static EngineLibrary.Input;

namespace GameLibrary.GameObjects

{

public enum Direction

{

Right,

Left,

Up,

Down

}

public class Player : ObjectScript

{

/// <summary>

/// Управление игроком

/// </summary>

public PlayerControl Control { get; private set; }

/// <summary>

/// Возможность двигаться у игрока

/// </summary>

public bool IsCanMove { get; set; } = true;

/// <summary>

/// Проеханные круги

/// </summary>

public static int RPLaps { get; private set; } = 0;

public static int BPLaps { get; private set; } = 0;

public PlayerProperities Property { get; set; }

float currSpeed = 0;

float maxSpeed;

Vector2 direction = new Vector2();

public bool IsCheckPoint { get; set; } = false;

private float currentReloadTime;

Vector2 view = new Vector2(0,1);

public override void Start(GameObject gameObject = null)

{

Property = new StandartProperities();

Property.SetProperty(TypeProperty.Fuel, 10);

Property.SetProperty(TypeProperty.Consumption, 0.1f);

//Property.SetProperty(TypeProperty.Ammo, 10f);

if (gameObject.GameObjectTag == "PlayerOne")

Control = new PlayerControl(AxisOfInput.Horizontal, AxisOfInput.Vertical, Key.Space, Key.C);

else if (gameObject.GameObjectTag == "PlayerTwo")

Control = new PlayerControl(AxisOfInput.AlternativeHorizontal, AxisOfInput.AlternativeVertical, Key.KeypadEnter, Key.Keypad0);

maxSpeed = Property.Speed;

}

public override void Update(GameObject gameObject)

{

if (gameObject.IsActive && IsCanMove)

Move(gameObject);

if (Property != null)

Property.UpdateTime(gameObject);

if (IsCanMove && Input.GetButtonDawn(Control.ShootKey) && currentReloadTime < Time.CurrentTime && Property.Ammo > 0)

Shoot(gameObject);

if (Property.Ammo >= 0)

GameEvents.ChangeCount?.Invoke(gameObject.GameObjectTag, Property.Ammo);

//GameEvents.ChangeHealth?.Invoke(gameObject.GameObjectTag, Property.Health);

}

private void Shoot(GameObject gameObject)

{

Vector2 bulletSpawnPosition = gameObject.Transform.Position;

Vector2 bulletDirection = view;

SpawnBullet(bulletSpawnPosition, bulletDirection, gameObject.GameObjectTag , 1);

Property.SetProperty(TypeProperty.Ammo, Property.Ammo - 1);

currentReloadTime = Time.CurrentTime + Property.ReloadTime;

}

/// <summary>

/// Создание пули из фабрики

/// </summary>

/// <param name="position">Позиция создания</param>

/// <param name="direction">Направление пули</param>

void SpawnBullet(Vector2 position, Vector2 direction,string gameObjectTag, float power = 1)

{

BulletFactory factory = new BulletFactory();

Game.instance.AddObjectOnScene(factory.CreateBullet(position, direction, gameObjectTag, power));

}

/// <summary>

/// Метод движения игрока

/// </summary>

private void Move(GameObject gameObject)

{

int directionX, directionY;

directionX = Input.GetAxis(Control.HorizontalAxis);

directionY = Input.GetAxis(Control.VerticalAxis);

if(directionY == 0)

{

SpeedDown(100);

ChangeStatsValue(gameObject, Property.Consumption \* Time.DeltaTime);

}

else

ChangeStatsValue(gameObject, Property.Consumption \* Time.DeltaTime \* 4);

if(Property.Fuel > 0)

{

if (directionX > 0 && currSpeed > 0)

{

//gameObject.Texture.Set("Right");

gameObject.Transform.SetAngle(0.01f \* currSpeed \* Time.DeltaTime);

}

else if (directionX < 0 && currSpeed > 0)

{

gameObject.Transform.SetAngle(-0.01f \* currSpeed \* Time.DeltaTime);

//gameObject.Texture.Set("Left");

}

if (directionY > 0)

{

//gameObject.Texture.Set("Down");

if (direction == new Vector2(1, 1) && currSpeed > 0)

SpeedDown(300);

else

{

direction = new Vector2(-1, -1);

SpeedUp(100);

}

}

else if (directionY < 0)

{

//gameObject.Texture.Set("Up");

if (direction == new Vector2(-1, -1) && currSpeed > 0)

SpeedDown(200);

else

{

direction = new Vector2(1, 1);

SpeedUp(100);

}

}

}

view = direction;

view.X \*= (float)Math.Cos(gameObject.Transform.Angle);

view.Y \*= (float)Math.Sin(gameObject.Transform.Angle);

gameObject.Transform.SetMovement(view \* currSpeed \* Time.DeltaTime);

DetectCollision(gameObject);

}

private void SpeedUp(float value)

{

if (currSpeed < maxSpeed)

{

currSpeed += value \* Time.DeltaTime;

}

else if (currSpeed > maxSpeed)

SpeedDown(value);

}

private void SpeedDown(float value)

{

if (currSpeed > 0)

{

currSpeed -= value \* Time.DeltaTime;

}

else if (currSpeed < 0)

{

currSpeed = 0;

}

}

/// <summary>

/// Распознавание столкновений и реакция на них

/// </summary>

private void DetectCollision(GameObject gameObject)

{

if (gameObject.Collider.CheckGameObjectIntersection(out GameObject player, "PlayerOne", "PlayerTwo"))

{

player.Transform.SetMovement(view);

gameObject.Transform.ResetMovement();

}

if (gameObject.Collider.CheckGameObjectIntersection(out GameObject wall, "Wall"))

{

maxSpeed = Property.Speed / 4;

}

else

maxSpeed = Property.Speed;

if (gameObject.Collider.CheckGameObjectIntersection(out GameObject checkPoint, "CheckPoint"))

{

IsCheckPoint = true;

}

if (gameObject.Collider.CheckGameObjectIntersection(out GameObject start, "Start") && IsCheckPoint)

{

SetLaps(gameObject.GameObjectTag, 1);

IsCheckPoint = false;

}

}

/// <summary>

/// Изменение значения характеристик игрока

/// </summary>

/// <param name="value">Значение, которое прибавляется к текущему значению монет</param>

public void ChangeStatsValue(GameObject gameObject, float value)

{

//if (gameObject.Collider.CheckIntersection("Bullet") || Property.Fuel <= 10)

//{

Property.SetProperty(TypeProperty.Fuel, Property.Fuel - value);

GameEvents.ChangeFuel?.Invoke(gameObject.GameObjectTag, Property.Fuel);

// }

}

/// <summary>

/// Изменение значения характеристик игрока

/// </summary>

/// <param name="value">Значение, которое прибавляется к текущему значению монет</param>

public void ChangeStatsValue(GameObject gameObject, bool value)

{

GameEvents.ChangeTires?.Invoke(gameObject.GameObjectTag, value);

}

public void ChangeStatsValue(float value, string gameObjectTag ,string gametag)

{

if (gametag == "Death")

{

//Property.SetProperty(TypeProperty.Fuel, 10);

////Coins += value;

//GameEvents.ChangeHealth?.Invoke(gameObjectTag, Property.Fuel);

////GameEvents.ChangeCoins?.Invoke(gameObject.GameObjectTag, Coins);

}

}

public static void SetLaps(string tag, int value)

{

if (tag == "PlayerOne")

{

RPLaps += value;

GameEvents.ChangeLaps?.Invoke(tag, value);

}

else

{

BPLaps += value;

GameEvents.ChangeLaps?.Invoke(tag, value);

}

}

/// <summary>

/// Структура игрового управления персонажа

/// </summary>

public struct PlayerControl

{

/// <summary>

/// Горизонтальная ось передвижения

/// </summary>

public AxisOfInput HorizontalAxis { get; private set; }

/// <summary>

/// Вертикальная ось передвижения

/// </summary>

public AxisOfInput VerticalAxis { get; private set; }

/// <summary>

/// Кнопка стрельбы

/// </summary>

public Key ShootKey { get; private set; }

/// <summary>

/// Кнопка стрельбы

/// </summary>

public Key GetKey { get; private set; }

/// <summary>

/// Конструктор структуры

/// </summary>

/// <param name="horizontalAxis">Горизонтальная ось передвижения</param>

/// <param name="verticalAxis"> Вертикальная ось передвижения</param>

/// <param name="shootKey">Кнопка стрельбы</param>

public PlayerControl(AxisOfInput horizontalAxis, AxisOfInput verticalAxis, Key shootKey, Key getKey)

{

HorizontalAxis = horizontalAxis;

VerticalAxis = verticalAxis;

ShootKey = shootKey;

GetKey = getKey;

}

}

}

}

**Код программы для *PlayerProperities*.*cs*:**

using EngineLibrary;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.GameObjects

{

public abstract class PlayerProperities

{

protected Player player;

protected float timeDeactivate = 2;

protected float timer = 0;

/// <summary>

/// Запас здоровья игрока

/// </summary>

public abstract float Fuel { get; protected set; }

/// <summary>

/// Запас здоровья игрока

/// </summary>

public abstract bool Tires { get; set; }

/// <summary>

/// Боезапас

/// </summary>

public abstract int Ammo { get; protected set; }

/// <summary>

/// Скорость

/// </summary>

public abstract float Speed { get; protected set; }

/// <summary>

/// Время перезарядки оружия

/// </summary>

public abstract float ReloadTime { get; protected set; }

/// <summary>

///

/// </summary>

public abstract float Consumption { get; protected set; }

//public virtual void SetPlayer(Player player)

//{

// this.player = player;

//}

public virtual void SetProperty(TypeProperty type, float value)

{

switch (type)

{

case TypeProperty.Fuel:

Fuel = value;

break;

case TypeProperty.Ammo:

Ammo = (int)value;

break;

case TypeProperty.Consumption:

Consumption = value;

break;

}

}

public virtual void UpdateTime(GameObject player)

{

timer += Time.DeltaTime;

if (timer >= timeDeactivate)

{

timer = 0;

DeactivateProperities(player);

}

}

protected abstract void DeactivateProperities(GameObject player);

}

}

**Код программы для *SpawnManager*.*cs*:**

using EngineLibrary;

using GameLibrary.Factory;

using OpenTK;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.GameObjects

{

/// <summary>

/// Класс игрового менеджера

/// </summary>

public class SpawnManager : ObjectScript

{

private const float timeToSpawn = 0.5f;

private float currentTimeToSpawn;

private Game maze;

private PrizeFactory spawnFactory;

/// <summary>

/// Поведение на момент создание игрового объекта

/// </summary>

public override void Start(GameObject gameObject)

{

maze = Game.instance;

currentTimeToSpawn = Time.CurrentTime;

}

/// <summary>

/// Обновление игрового объекта

/// </summary>

public override void Update(GameObject gameObject)

{

if (currentTimeToSpawn < Time.CurrentTime)

{

Random random = new Random();

int chance = random.Next(0, 100);

if (maze.EmptyBlocks.Count == 0) return;

Vector2 position = maze.GetRandomPosition();

if (chance < 33)

{

spawnFactory = new SpeedPrizeFactory();

}

else if (chance > 33 && chance <= 66)

{

spawnFactory = new FuelPrizeFactory();

}

else if(chance > 66 && chance <= 99)

{

spawnFactory = new AmmoPrizeFactory();

}

maze.AddObjectOnScene(spawnFactory.CreatePrize(position));

currentTimeToSpawn += timeToSpawn;

}

}

}

}

**Код программы для *TypeProperty*.*cs*:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.GameObjects

{

public enum TypeProperty

{

Fuel,

Tires,

Ammo,

Speed,

ReloadTime,

Consumption

}

}

**Код программы для *GameEvents*.*cs*:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.Maze

{

/// <summary>

/// Статический класс событий игры

/// </summary>

public static class GameEvents

{

/// <summary>

/// Делегат события изменения количества здоровья

/// </summary>

/// <param name="tagPlayer">Тег игрового объекта игрока</param>

/// <param name="value">Значение собранных монет</param>

public delegate void FuelDelegate(string tagPlayer, float value);

/// <summary>

/// Событие изменения количества здоровья

/// </summary>

public static FuelDelegate ChangeFuel { get; set; }

/// <summary>

/// Делегат события изменения количества здоровья

/// </summary>

/// <param name="tagPlayer">Тег игрового объекта игрока</param>

/// <param name="value">Значение собранных монет</param>

public delegate void TiresDelegate(string tagPlayer, bool value);

/// <summary>

/// Событие изменения количества здоровья

/// </summary>

public static TiresDelegate ChangeTires{ get; set; }

/// <summary>

/// Делегат события изменения количества монет

/// </summary>

/// <param name="tagPlayer">Тег игрового объекта игрока</param>

/// <param name="value">Значение собранных монет</param>

public delegate void LapsDelegate(string tagPlayer, int value);

/// <summary>

/// Событие изменения количества монет

/// </summary>

///

public static LapsDelegate ChangeLaps { get; set; }

/// <summary>

/// Делегат события окончания игры

/// </summary>

public delegate void EndGameDelegate(string winPlayer);

/// <summary>

/// Событие окончания игры

/// </summary>

public static EndGameDelegate EndGame { get; set; }

/// <summary>

/// Делегат события получения оружия игроком

/// </summary>

/// <param name="tagPlayer">Тег игрового объекта игрока</param>

public delegate void CountBullets(string tagPlayer, int count);

/// <summary>

/// Событие получения оружия игроком

/// </summary>

public static CountBullets ChangeCount { get; set; }

}

}

**Код программы для *Maps*.*cs*:**

using EngineLibrary;

using GameLibrary.Map;

using OpenTK;

using System;

using System.Collections.Generic;

using System.Drawing;

using System.IO;

using System.Linq;

using System.Runtime.Serialization.Formatters.Binary;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.Maze

{

public class Maps

{

/// <summary>

/// Фабрика создания элементов лабиринта

/// </summary>

public MazeElementsFactory ElementsFactory { get; private set; }

public List<Vector2> EmptyBlocks { get; private set; } = new List<Vector2>();

/// <summary>

/// Коллекция линий дороги

/// </summary>

public List<RacePoint> roadLines { get; private set; } = new List<RacePoint>();

public void CreateMap()

{

var game = Game.instance;

Random random = new Random();

ElementsFactory = new MazeElementsFactory();

float worldScale = Game.instance.HeightOfApplication / 15;

//var points = new List<Vector2>();

//points.Add(new Vector2(100, 100));

//points.Add(new Vector2(200, 100));

//points.Add(new Vector2(300, 100));

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

//{

// var gameObject = new GameObject();

// gameObject.GameObjectTag = "Road";

// TexturesBox texture1;

// texture1 = new TexturesBox(ContentPipe.LoadTexture("Road.png"));

// var script = new RoadLine();

// script.Start(gameObject);

// gameObject.SetComponent(texture1);

// gameObject.SetComponent(script);

// gameObject.SetComponent(new TransformComponent(points.ElementAt(i), new Vector2(1, 1), new Vector2(0.3f, 0.5f), 0));

// gameObject.SetComponent(new ColliderComponent(gameObject, 1f, new Vector2(0, 0)));

// game.AddObjectOnScene(gameObject);

//}

Bitmap bitmap = new Bitmap(@"Resources\Mazes\Maze\_4.bmp");

for (int i = 0; i < bitmap.Height; i++)

{

for (int j = 0; j < bitmap.Width; j++)

{

System.Drawing.Color color = bitmap.GetPixel(j, i);

GameObject gameObject = null;

if (color.R == 0 && color.G == 0 && color.B == 0)

gameObject = ElementsFactory.CreateMazeElement(new Vector2(j, i) \* worldScale, "Wall");

else if (color.R == 255 && color.G == 0 && color.B == 0)

game.PlayerOneFactory.StartPosition = new Vector2(j, i) \* worldScale;

else if (color.R == 0 && color.G == 0 && color.B == 255)

game.PlayerTwoFactory.StartPosition = new Vector2(j, i) \* worldScale;

else if (color.R == 0 && color.G == 255 && color.B == 0)

gameObject = ElementsFactory.CreateFinish(new Vector2(j, i) \* worldScale, "Start");

else if (color.R == 0 && color.G == 255 && color.B == 255)

gameObject = ElementsFactory.CreateFinish(new Vector2(j, i) \* worldScale, "CheckPoint");

else

EmptyBlocks.Add(new Vector2(j, i));

if (gameObject != null)

game.AddObjectOnScene(gameObject);

}

}

}

}

}

**Код программы для *MazeElementsFactory*.*cs*:**

using EngineLibrary;

using OpenTK;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.Maze

{

/// <summary>

/// Класс фабрики создания элементов лабиринта

/// </summary>

public class MazeElementsFactory

{

/// <summary>

/// Создает элемент лабиринта

/// </summary>

/// <param name="position">Позиция объекта на сцене</param>

/// <param name="TagName">Тег игрового объекта</param>

/// <returns>Созданный игровой объект</returns>

public GameObject CreateMazeElement(Vector2 position, string TagName)

{

float worldScale = Game.instance.HeightOfApplication / 15;

GameObject gameObject = new GameObject();

gameObject.SetComponent(new TexturesBox(ContentPipe.LoadTexture("MazeElements/" + TagName + ".png")));

gameObject.SetComponent(new TransformComponent(new Vector2( position.X, position.Y), new Vector2(1,1) \* worldScale / gameObject.Texture.Texture.Width, new Vector2(0, 0), 0));

gameObject.SetComponent(new ColliderComponent(gameObject, 1));

gameObject.GameObjectTag = TagName;

return gameObject;

}

/// <summary>

/// Создает элемент лабиринта

/// </summary>

/// <param name="position">Позиция объекта на сцене</param>

/// <param name="TagName">Тег игрового объекта</param>

/// <returns>Созданный игровой объект</returns>

public GameObject CreateFinish(Vector2 position, string TagName)

{

float worldScale = Game.instance.HeightOfApplication / 15;

GameObject gameObject = new GameObject();

gameObject.SetComponent(new TexturesBox(ContentPipe.LoadTexture("MazeElements/" + TagName + ".png")));

gameObject.SetComponent(new TransformComponent(new Vector2(position.X, position.Y), new Vector2(1.25f, 1.25f) \* worldScale / gameObject.Texture.Texture.Width, new Vector2(0, 0), 0));

gameObject.SetComponent(new ColliderComponent(gameObject, 1));

gameObject.GameObjectTag = TagName;

return gameObject;

}

}

}

**Код программы для *RacePoint*.*cs*:**

using EngineLibrary;

using OpenTK.Graphics.OpenGL;

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.Map

{

/// <summary>

/// Класс, представляющий линию дороги

/// </summary>

[Serializable]

public class RacePoint : ObjectScript

{

public override void Start(GameObject gameObject = null)

{

}

public override void Update(GameObject gameObject)

{

}

}

}

**Код программы для *DecoratorProperty*.*cs*:**

using EngineLibrary;

using GameLibrary.GameObjects;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary

{

public class DecoratorProperty : PlayerProperities

{

protected PlayerProperities playerProperities;

/// <summary>

/// Конструктор объекта

/// </summary>

/// <param name="playerProperities"></param>

public DecoratorProperty(PlayerProperities playerProperities)

{

this.playerProperities = playerProperities;

}

public override float Fuel { get => playerProperities.Fuel; protected set => playerProperities.SetProperty(TypeProperty.Ammo, value); }

public override bool Tires { get => playerProperities.Tires; set => playerProperities.Tires = value; }

public override int Ammo { get => playerProperities.Ammo; protected set => playerProperities.SetProperty(TypeProperty.Ammo, value); }

public override float Speed { get => playerProperities.Speed; protected set => playerProperities.SetProperty(TypeProperty.Ammo, value); }

public override float ReloadTime { get => playerProperities.ReloadTime; protected set => playerProperities.SetProperty(TypeProperty.Ammo, value); }

public override float Consumption { get => playerProperities.Consumption; protected set => playerProperities.SetProperty(TypeProperty.Ammo, value); }

protected override void DeactivateProperities(GameObject player)

{

(player.Script as Player).Property = new StandartProperities();

}

}

}

**Код программы для *PlayerConstructor*.*cs*:**

using EngineLibrary;

using GameLibrary.GameObjects;

using OpenTK;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary

{

public class PlayerConstructor

{

/// <summary>

/// Тег плеера

/// </summary>

public string PlayerTag { get; private set; }

/// <summary>

/// Начальная позиция игрока в лабиринте

/// </summary>

public Vector2 StartPosition { get; set; }

/// <summary>

/// Создание игрового объекта персонажа

/// </summary>

/// <returns>Игровой объект</returns>

public GameObject CreatePlayer(string path)

{

PlayerTag = path;

var gameObject = new GameObject();

gameObject.GameObjectTag = path;

TexturesBox texture1;

texture1 = new TexturesBox(ContentPipe.LoadTexture(path + "Car.png"));

var script = new Player();

script.Start(gameObject);

gameObject.SetComponent(texture1);

gameObject.SetComponent(script);

gameObject.SetComponent(new TransformComponent(StartPosition, new Vector2(0.1f, 0.1f), new Vector2(0.3f, 0.5f), 0));

gameObject.SetComponent(new ColliderComponent(gameObject, 0.5f, new Vector2(0, 0)));

return gameObject;

}

}

}

**Код программы для *PowerDecorator*.*cs*:**

using GameLibrary.GameObjects;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary

{

public class PowerDecorator : DecoratorProperty

{

public PowerDecorator(PlayerProperities playerProperities) : base(playerProperities)

{

}

/// <summary>

/// Скорость

/// </summary>

//public override float Power { get => playerProperities.Power \* 2; }

}

}

**Код программы для *ReloadTimeDecorator*.*cs*:**

using GameLibrary.GameObjects;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary

{

public class ReloadTimeDecorator : DecoratorProperty

{

public ReloadTimeDecorator(PlayerProperities playerProperities) : base(playerProperities)

{

}

/// <summary>

/// Скорость

/// </summary>

public override float ReloadTime { get => playerProperities.ReloadTime / 2; }

}

}

**Код программы для *SpeedDecorator*.*cs*:**

using EngineLibrary;

using GameLibrary.GameObjects;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary

{

public class SpeedDecorator : DecoratorProperty

{

public SpeedDecorator(PlayerProperities playerProperities) : base(playerProperities)

{

}

/// <summary>

/// Скорость

/// </summary>

public override float Speed { get => playerProperities.Speed / 4f; }

protected override void DeactivateProperities(GameObject player)

{

if ((player.Script as Player).Property.Tires)

{

(player.Script as Player).ChangeStatsValue(player, true);

(player.Script as Player).Property = new StandartProperities();

}

}

}

}

**Код программы для *StandartProperities*.*cs*:**

using EngineLibrary;

using GameLibrary.GameObjects;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary

{

/// <summary>

/// Класс с характеристиками игрока

/// </summary>

public class StandartProperities : PlayerProperities

{

/// <summary>

/// Боезапас

/// </summary>

public override int Ammo { get; protected set; }

/// <summary>

/// Скорость

/// </summary>

public override float Speed { get; protected set; } = 400;

/// <summary>

/// Время перезарядки оружия

/// </summary>

public override float ReloadTime { get; protected set; } = 0.5f;

public override float Fuel { get; protected set; }

public override bool Tires { get; set; } = true;

public override float Consumption { get; protected set; }

protected override void DeactivateProperities(GameObject player)

{

}

}

}

**Код программы для *PrizeSpawn*.*cs*:**

using EngineLibrary;

using GameLibrary.GameObjects;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary.Prize

{

/// <summary>

/// Класс подбираемого оружия в лабиринте

/// </summary>

public class PrizeSpawn : ObjectScript

{

private Game maze;

private PlayerProperities dropOutPrize;

private TypeProperty typeProperty;

private float cuurentTimeToDisappear;

private int value;

private bool valueBool;

/// <summary>

/// Инициализация места подбираемого оружия

/// </summary>

/// <param name="disappearTime">Время исчезнование места</param>

public void InitializePrizeSpawn(PlayerProperities playerProperities, float disappearTime)

{

dropOutPrize = playerProperities;

//nameOfgun = name;

cuurentTimeToDisappear = Time.CurrentTime + disappearTime;

}

// <summary>

/// Инициализация места подбираемого оружия

/// </summary>

/// <param name="name">Название оружия</param>

/// <param name="gun">Подбираемое оружие</param>

/// <param name="disappearTime">Время исчезнование места</param>

public void InitializePrizeSpawn(TypeProperty typeProperty, int value, float disappearTime)

{

this.typeProperty = typeProperty;

this.value = value;

cuurentTimeToDisappear = Time.CurrentTime + disappearTime;

}

// <summary>

/// Инициализация места подбираемого оружия

/// </summary>

/// <param name="name">Название оружия</param>

/// <param name="gun">Подбираемое оружие</param>

/// <param name="disappearTime">Время исчезнование места</param>

public void InitializePrizeSpawn(TypeProperty typeProperty, bool value, float disappearTime)

{

this.typeProperty = typeProperty;

valueBool = value;

cuurentTimeToDisappear = Time.CurrentTime + disappearTime;

}

/// <summary>

/// Поведение на момент создание игрового объекта

/// </summary>

public override void Start(GameObject gameObject = null)

{

maze = Game.instance;

}

/// <summary>

/// Обновление игрового объекта

/// </summary>

public override void Update(GameObject gameObject)

{

if (cuurentTimeToDisappear < Time.CurrentTime)

{

maze.AddObjectsToRemove(gameObject);

}

if (gameObject.Collider.CheckGameObjectIntersection(out GameObject player, "PlayerOne", "PlayerTwo"))

{

if (player.GameObjectTag == "PlayerOne" && Input.GetButtonDawn((player.Script as Player).Control.GetKey))

{

if (dropOutPrize == null)

{

if(typeProperty == TypeProperty.Tires)

(player.Script as Player).Property.Tires = valueBool;

else

(player.Script as Player).Property.SetProperty(typeProperty, value);

}

else

(player.Script as Player).Property = dropOutPrize;

maze.AddObjectsToRemove(gameObject);

}

else if (player.GameObjectTag == "PlayerTwo" && Input.GetButtonDawn((player.Script as Player).Control.GetKey))

{

if (dropOutPrize == null)

{

if (typeProperty == TypeProperty.Tires)

(player.Script as Player).Property.Tires = valueBool;

else

(player.Script as Player).Property.SetProperty(typeProperty, value);

}

else

(player.Script as Player).Property = dropOutPrize;

maze.AddObjectsToRemove(gameObject);

}

}

}

}

}

**Код программы для *MainWindow*.*xaml*.*cs*:**

using OpenTK.Graphics.OpenGL;

using System;

using System.Drawing;

using System.Media;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Forms;

using System.Windows.Input;

using EngineLibrary;

using OpenTK.Graphics;

using GameLibrary;

using System.Windows.Media;

using WpfOpenGlControl;

using GameLibrary.GameObjects;

using GameLibrary.Maze;

using Color = System.Windows.Media.Color;

namespace Maze

{

/// <summary>

/// Логика взаимодействия для MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

Game game = new Game();

public MainWindow()

{

InitializeComponent();

GameEvents.ChangeLaps += ChangeLaps;

GameEvents.ChangeFuel += ChangeFuel;

GameEvents.ChangeTires += ChangeTires;

GameEvents.ChangeCount += ChangeCout;

GameEvents.EndGame += EndGame;

PlayerOnePanel.Visibility = Visibility.Visible;

PlayerTwoPanel.Visibility = Visibility.Visible;

Background = new SolidColorBrush(Color.FromRgb(47, 91, 140));

}

private void WindowsFormsHost\_Initialized(object sender, EventArgs e)

{

glControl.MakeCurrent();

}

private void glControl\_Load(object sender, EventArgs e)

{

GL.Enable(EnableCap.Blend);

GL.Enable(EnableCap.Texture2D);

GL.BlendFunc(BlendingFactor.SrcAlpha, BlendingFactor.OneMinusSrcAlpha);

GL.Viewport(0, 0, glControl.Width, glControl.Height);

\_loaded = true;

}

private void glControl\_Paint(object sender, System.Windows.Forms.PaintEventArgs e)

{

Render();

glControl.Invalidate();

}

private bool \_loaded;

private void Render()

{

GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);

GL.ClearColor(System.Drawing.Color.FromArgb(1, 47, 91, 140));

GL.MatrixMode(MatrixMode.Projection);

GL.LoadIdentity();

GL.Ortho(0, glControl.Width, glControl.Height, 0, 0, 1);

game.Rendering();

glControl.SwapBuffers();

}

private void GLControl\_Resize(object sender, EventArgs e)

{

if (!\_loaded)

return;

GL.Viewport(0, 0, glControl.Width, glControl.Height);

glControl.Invalidate();

}

private void ChangeLaps(string player, int value)

{

if (player == "PlayerOne")

{

value += Int32.Parse(PlayerOneLaps.Text);

PlayerOneLaps.Text = value.ToString();

}

else

{

value += Int32.Parse(PlayerTwoCoins.Text);

PlayerTwoCoins.Text = value.ToString();

}

}

private void ChangeFuel(string player, float value)

{

if (player == "PlayerOne")

PlayerOneFuel.Text = value.ToString();

else

PlayerTwoFuel.Text = value.ToString();

}

private void ChangeTires(string player, bool value)

{

if (player == "PlayerOne")

PlayerOneTires.Text = value.ToString();

else

PlayerTwoTires.Text = value.ToString();

}

private void ChangeCout(string player, int count)

{

if (player == "PlayerOne")

POCountBullets.Text = count.ToString();

else

PTCountBullets.Text = count.ToString();

}

private void EndGame(string winPlayer)

{

formHost.Visibility = Visibility.Hidden;

if (winPlayer == "PlayerOne")

{

Background = new SolidColorBrush(Color.FromRgb(9, 46, 168));

WinPanel.Content = "First player win";

}

else

{

Background = new SolidColorBrush(Color.FromRgb(179, 22, 22));

WinPanel.Content = "Second player win";

}

WinPanel.Visibility = Visibility.Visible;

GameEvents.EndGame -= EndGame;

GameEvents.ChangeLaps -= ChangeLaps;

GameEvents.ChangeFuel -= ChangeFuel;

GameEvents.ChangeTires -= ChangeTires;

}

}

}

**Код программы для *Game*.*cs*:**

using EngineLibrary;

using GameLibrary.GameObjects;

using GameLibrary.Maze;

using OpenTK;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GameLibrary

{

public class Game : GameWindow

{

/// <summary>

/// Статическая ссылка на класс

/// </summary>

public static Game instance = null;

public int WidthOfApplication { get; private set; } = 1280;

public int HeightOfApplication { get; private set; } = 720;

public List<Vector2> EmptyBlocks { get; private set; } = new List<Vector2>();

List<GameObject> gameObjects = new List<GameObject>(15);

List<GameObject> gameObjectsToRemove = new List<GameObject>(15);

List<GameObject> gameObjectsToAdd = new List<GameObject>(15);

/// <summary>

/// Конструктор первого игрока

/// </summary>

public PlayerConstructor PlayerOneFactory { get; set; }

/// <summary>

/// Конструктор второго игрока

/// </summary>

public PlayerConstructor PlayerTwoFactory { get; set; }

public Game()

{

if (instance == null)

instance = this;

PlayerOneFactory = new PlayerConstructor();

//PlayerOneFactory.StartPosition = new Vector2(400, 500);

PlayerTwoFactory = new PlayerConstructor();

//PlayerTwoFactory.StartPosition = new Vector2(600, 500);

Maps track = new Maps();

//maze.CreateGameObjectsOnScene();

track.CreateMap();

EmptyBlocks = track.EmptyBlocks;

var gameObject3 = new GameObject();

gameObject3.SetComponent(new TransformComponent(new Vector2(0f, 0f), new Vector2(1, 1), new Vector2(1, 1), 0));

gameObject3.GameObjectTag = "GameManager";

var script3 = new SpawnManager();

script3.Start(gameObject3);

gameObject3.SetComponent(script3);

//GameObject gameObject4 = new GameObject();

//gameObject4.SetComponent(new TransformComponent(new Vector2(0f, 0f), new Vector2(2.1f, 1.7f)));

//gameObject4.SetComponent(new TexturesBox(ContentPipe.LoadTexture("Фон.png")));

//gameObject4.GameObjectTag = "Background";

//gameObjects.Add(gameObject4);

gameObjects.Add(PlayerTwoFactory.CreatePlayer("PlayerOne"));

gameObjects.Add(PlayerOneFactory.CreatePlayer("PlayerTwo"));

gameObjects.Add(gameObject3);

}

/// <summary>

/// Добавление объекта в лист отрисовки

/// </summary>

/// <param name="gameObject">Игровой объект</param>

public void AddObjectOnScene(GameObject gameObject)

{

gameObjectsToAdd.Add(gameObject);

}

private void RemoveRenderGameObjects()

{

foreach (GameObject removeGameObject in gameObjectsToRemove)

{

gameObjects.Remove(removeGameObject);

//removeGameObject.Dispose();

}

gameObjectsToRemove.Clear();

}

private void AddRenderGameObjects()

{

gameObjects.AddRange(gameObjectsToAdd);

gameObjectsToAdd.Clear();

}

public void AddObjectsToRemove(GameObject gameObject)

{

//int count = 0;

if (gameObject.GameObjectTag == "Spawn")

EmptyBlocks.Add(gameObject.Transform.Position / (instance.HeightOfApplication / 15));

gameObjectsToRemove.Add(gameObject);

}

public void Rendering()

{

Time.UpdateTime();

RemoveRenderGameObjects();

AddRenderGameObjects();

foreach (var obj in gameObjects)

{

obj.Update();

if (obj.Texture != null)

SpriteBatch.Draw(obj);

}

if (Player.BPLaps >= 5 || Player.RPLaps >= 5)

{

EndScene();

}

}

/// <summary>

/// Рандомное место в лабиринте

/// </summary>

/// <returns>Позицию</returns>

public Vector2 GetRandomPosition()

{

Random random = new Random();

int index = random.Next(0, EmptyBlocks.Count);

Vector2 position = EmptyBlocks[index];

EmptyBlocks.Remove(position);

return position \* instance.HeightOfApplication / 15;

}

/// <summary>

/// Поведение при завершении сцены

/// </summary>

protected void EndScene()

{

string winPlayer;

if (Player.BPLaps < Player.RPLaps)

winPlayer = PlayerTwoFactory.PlayerTag;

else

winPlayer = PlayerOneFactory.PlayerTag;

GameEvents.EndGame?.Invoke(winPlayer);

}

}

}

# ПРИЛОЖЕНИЕ Б

(обязательное)

**Руководство пользователя**

* 1. Введение.

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

Разработанное программное приложение предназначено для игры двух игроков на одном экране.

Игровое приложение обладает следующим функционалом:

* игровое поле в виде кольцевой трассы;
* перемещение игроков в пределах этой трассы;
* генерация призов.

Для использования программного приложения пользователь должен быть ознакомлен с:

* настоящим руководством пользователя;
* правилами использования ЭВМ.
  1. Назначение и условия применения.

Разработанное игровое приложение предназначено для игры в гонки по кругу с видом сверху между двумя игроками на одном компьютере. Данное приложение позволяет хорошо провести свободное время, улучшает координацию и реакцию игроков. Игра требует концентрации и внимания, применения различных тактик ведения гонки, что способствует развитию различных качеств человека. Спортивный и соревновательный элемент делают эту игру веселой и интересной.

Для корректной работы игрового приложения необходимо соблюдение следующих требований:

* поддерживаемые операционные системы *Windows* *XP* и выше;
* наличие следующих устройств ввода: стандартная клавиатура и сенсорная панель;
* наличие устройства вывода – экран.
  1. Подготовка к работе.

Для установки приложения необходимо загрузить на компьютер готовую программу с игрой. Также требуется, чтобы на компьютере была установлена библиотека *OpenGL* последней версии. После этого в папке с программой запустить файл *game*.*exe*.

Если все инструкции соблюдены и приложение не сообщает об каких-либо ошибках, то программа работает исправно.

* 1. Аварийные ситуации.

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

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

* 1. Правила игры «Кольцевые гонки».

При запуске игрового приложения будет показано игровое поле, то есть кольцевые гонки, на котором двое игроков должны проехать 5 кругов. Оба игрока находятся на одной лини, с которой будут начинать гонку. Игроки выполнены в виде автомобилей, которые в начале игры имеют определённое количество топлива и максимальную скорость. Затем при помощи клавиш *W*, *S*, *A*, *D*, происходит перемещение машины чёрного цвета, стрельба, при наличии поднятого приза «Патроны», происходит при нажатии клавиши *Space*; клавиш *Up*, *Down*, *Right*, *Left* происходит передвижение автомобиля красного цвета, его стрельба осуществляется нажатием клавиши *Numpad*-*Enter*.

На пути к финишу будут появляться случайные призы, которые автомобиль может подобрать. Призы могут быть нескольких типов: топливо, здоровье, патроны и шины.

Топливо тратиться и если игрок не восполнит его, то автомобиль больше не сможет ехать. В верхних углах игрового поля находятся параметры игроков, указывающие на количество топлива, здоровья, патронов и кругов.

# ПРИЛОЖЕНИЕ В

(обязательное)

**Руководство программиста**

* 1. Назначение и условия применения

Разработанное игровое приложение предназначено для игры между двумя игроками на одном компьютере. Данная программа помогает провести время с друзьями, улучшить реакцию, учит рассуждать и логически мыслить, развивает образное и пространственное мышление. Игра позволяет учиться быстрее реагировать на неожиданные повороты событий, а это поможет в учёбе и в жизни. Также развивает концентрацию и внимание и значительно улучшает память, позволяя запоминать всё большие объёмы информации.

Для корректной работы игрового приложения необходимо соблюдение следующих требований:

* поддерживаемые операционные системы *Windows* *XP* и выше;
* наличие следующих устройств ввода: стандартная клавиатура и сенсорная панель;
* наличие устройства вывода – экран.
  1. Характеристики программы

Приложение работает в автоматическом режиме. При запуске открывается главное окно программы, в котором показано игровое поле, то есть кольцевые гонки. На этом игровом поле два игрока, которыми можно управлять с помощью клавиатуры.

* 1. Обращение к программе

Приложение запускается путём открытия файла *game*.*exe* находящегося в папке *Debug*.

* 1. Входные и выходные данные

В данной программе в качестве исходных данных используются язык программирования *C*#, *OpenGL*, *Windows* *Form* и шаблоны проектирования, такие как: «фабричный метод» и «декоратор».

«Фабричный метод» – реализует генерацию «призов», «декоратор» – изменяет характеристики автомобиля.

# ПРИЛОЖЕНИЕ Г

(обязательное)

**Руководство системного программиста**

1. Назначение и условия применения

Разработанное игровое приложение предназначено для игры между двумя игроками на одном компьютере. Данная программа помогает провести время с друзьями, улучшить реакцию, учит рассуждать и логически мыслить, развивает образное и пространственное мышление. Игра позволяет учиться быстрее реагировать на неожиданные повороты событий, а это поможет в учёбе и в жизни. Также развивает концентрацию и внимание и значительно улучшает память, позволяя запоминать всё большие объёмы информации.

Для корректной работы игрового приложения необходимо соблюдение следующих требований:

* поддерживаемые операционные системы *Windows* *XP* и выше;
* наличие следующих устройств ввода: стандартная клавиатура и сенсорная панель;
* наличие устройства вывода – экран.

1. Структура программы

Игровое приложение логически можно разбить на несколько составляющих: графический интерфейс пользователя, средства работы с графикой и непосредственно логику игровых объектов и игрового процесса.

Графический интерфейс включает отображение игрового поля.

Графическая составляющая приложения отвечает за процесс рендеринга сцены и отображения состояния игровой сессии.

Логика игровых объектов включает в себя реализацию игровых механик и логику поведения игровых объектов.

1. Проверка программы

Отсутствие сообщения об ошибках свидетельствует о корректной работе приложения.

# ПРИЛОЖЕНИЕ Д

(обязательное)

**Схема использования паттерна «Декоратор»**

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc120652269)

[1 ИГРОВЫЕ ПРИЛОЖЕНИЯ И СРЕДСТВА ИХ РАЗРАБОТКИ 5](#_Toc120652270)

[1.1 Особенности игрового жанра «Аркада» 5](#_Toc120652271)

[1.2 Особенности спрайтовой графики для игровых приложений 5](#_Toc120652272)

[1.3 Движки для создания игр 7](#_Toc120652273)

[1.4 Языки программирования 7](#_Toc120652274)

[1.5 Структура игрового кода 8](#_Toc120652275)

[1.6 Возможности графической библиотеки *OpenGL* 9](#_Toc120652276)

[1.7 Понятие сетевого взаимодействия 12](#_Toc120652277)

[1.8 Протокол *UDP* и сравнение с *TCP* 14](#_Toc120652278)

[2 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ «КОЛЬЦЕВЫЕ ГОНКИ» 16](#_Toc120652279)

[2.1 Метод декомпозиции в игровом приложении «Кольцевые гонки» 16](#_Toc120652280)

[2.2 Описание паттернов проектирования для игрового приложения «Кольцевые гонки» 17](#_Toc120652281)

[2.3 Принципы взаимодействия классов в игровом приложении «Кольцевые гонки» 19](#_Toc120652282)

[3 ВЕРИФИКАЦИЯ РАЗРАБОТАННОГО ИГРОВОГО ПРИЛОЖЕНИЯ «КОЛЬЦЕВЫЕ ГОНКИ» 23](#_Toc120652283)

[3.1 Принцип работы игрового приложения «Кольцевые гонки» 23](#_Toc120652284)

[3.2 Результаты тестирования игрового приложения «Кольцевые гонки» 23](#_Toc120652285)

[3.3 Верификация игрового приложения «Кольцевые гонки» 24](#_Toc120652286)

[ЗАКЛЮЧЕНИЕ 29](#_Toc120652287)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 30](#_Toc120652288)

[ПРИЛОЖЕНИЕ А 31](#_Toc120652289)

[ПРИЛОЖЕНИЕ Б 66](#_Toc120652290)

[ПРИЛОЖЕНИЕ В 68](#_Toc120652291)

[ПРИЛОЖЕНИЕ Г 69](#_Toc120652292)

[ПРИЛОЖЕНИЕ Д 70](#_Toc120652293)