МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технологический университет «СТАНКИН»
(ФГБОУ ВО «МГТУ «СТАНКИН»)

Институт
информационных технологий

Кафедра
информационных систем

Основная образовательная программа 09.03.02
«Информационные системы и технологии»

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентированное программирование»

Тема: «Разработка игры с использованием Vulkan»

Руководитель
к.т.н., доцент
_____ **Разумовский А.И.**
подпись

Выполнил
студент группы ИДБ-22-07
_____ **Макурина Д.С.**
подпись

Москва, 2024

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В современном мире игровой индустрии технологии игровых движков и графических API становятся все более мощными и гибкими, предоставляя разработчикам широкие возможности для создания увлекательных и креативных игровых проектов. В этом контексте использование современных графических API, таких как Vulkan, становится неотъемлемой частью процесса разработки игр, обеспечивая высокую производительность и гибкость в создании визуального контента.

Цель данной курсовой работы заключается в разработке игры с использованием графического API Vulkan с применением ключевых концепций объектно-ориентированного программирования, ресурсов приобретения инициализации (RAII) и паттерна проектирования системы компонентов сущности (ECS). ООП позволяет организовать код игры в виде объектов и классов, что способствует повышению его читаемости, модульности и масштабируемости. RAII обеспечивает автоматическое управление ресурсами, обеспечивая безопасное и эффективное их использование. Паттерн проектирования ECS позволяет структурировать игровой мир в виде сущностей, компонентов и систем, обеспечивая гибкость и производительность в разработке и управлении игровыми объектами.

ГЛАВА 1. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

1.1. ИСТОРИЯ СОЗДАНИЯ ООП

Основа ООП была заложена в начале 1960-х годов. Прорыв в использовании экземпляров и объектов был достигнут в MIT с PDP-1, и первым языком программирования для работы с объектами стал Simula 67. Он включала в себя большую часть концепций объектно-ориентированного программирования: классы и объекты, подклассы, виртуальные функции, безопасные ссылки и механизмы, позволяющие внести в программу коллекцию программных структур, описанных общим заголовком класса.

Термин "объектно-ориентированное программирование" был впервые использован Xerox PARC в языке программирования Smalltalk. Понятие ООП использовалось для обозначения процесса использования объектов в качестве основы для расчетов. Команда разработчиков была вдохновлена проектом Simula 67, но они спроектировали свой язык так, чтобы он был динамичным. В Smalltalk объекты могут быть изменены, созданы или удалены, что отличает его от статических систем, которые обычно используются. Этот язык программирования также был первым, использовавшим концепцию наследования. Именно эта особенность позволила Smalltalk превзойти как Simula 67, так и аналоговые системы программирования.

В 1970-х годах язык программирования Smalltalk стал стимулом для сообщества Lisp к внедрению объектно-ориентированных техник, которые были представлены разработчикам с помощью Lisp-машин. Различные эксперименты с расширениями Lisp в конечном итоге привели к созданию Common Lisp Object System (CLOS) – первого стандартизованного объектно-ориентированного языка. CLOS был уникален тем, что органично сочетал в себе функциональное и объектно-ориентированное программирование, а также позволял расширять себя с помощью протокола Meta-object protocol.

Объектно-ориентированное программирование стало доминирующей методологией в начале и середине 1990-х годов благодаря широкому

распространению поддерживающих его языков программирования, таких как Visual FoxPro 3.0, C++ и Delphi. Популярность ООП также поддерживалась ростом интереса к графическим пользовательским интерфейсам, которые в значительной мере основывались на принципах объектно-ориентированного программирования. Примером тесной связи между динамическими библиотеками графического интерфейса пользователя и объектно-ориентированными языками программирования является фреймворк Cocoa на Mac OS X, который был написан на Objective-C – объектно-ориентированном расширении языка C, основанном на Smalltalk с поддержкой динамических сообщений. Некоторые даже считают, что именно связь с графическими интерфейсами пользователя привела объектно-ориентированное программирование на передний план в мире технологий.

1.2. ОПРЕДЕЛЕНИЕ ООП И ЕГО ОСНОВНЫЕ КОНЦЕПЦИИ

Объектно-ориентированное программирование – это метод программирования, основанный на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определенного класса, а классы являются членами определенной иерархии наследования. В соответствии с этим определением одни языки программирования являются объектно-ориентированными, а другие – нет. Страуструп полагает: «Если термин объектно-ориентированный язык вообще имеет смысл, то он должен относиться к языку, хорошо поддерживающему объектно-ориентированный стиль программирования… поддержка такого стиля программирования считается хорошей, если средства языка обеспечивают удобное использование этого стиля. Язык не поддерживает объектно-ориентированное программирование, если написание программ в этом стиле требует особых усилий или опыта; в этом случае говорят, что язык просто позволяет программистам использовать объектно-ориентированный подход» [1]. С теоретической точки зрения существует возможность имитации объектно-ориентированного программирования с помощью обычных языков,

таких как Pascal или ассемблер, но это чрезвычайно трудно. Карделли и Вегнер утверждают: «Язык программирования является объектно-ориентированным тогда и только тогда, когда он удовлетворяет следующие условия:

1. Он поддерживает объекты, представляющие собой абстракции данных с интерфейсом в виде именованных операций и сокрытым локальным состоянием.
2. Объекты имеют ассоциированный с ними тип.
3. Типы могут наследовать атрибуты супертипов» [2].

Поддержка наследования в объектно-ориентированных языках означает возможность выражения отношения «is a» среди типов. Если язык не поддерживает механизм наследования, то его нельзя считать объектно-ориентированным. Согласно этому определению, языки Smalltalk, Object Pascal, C++, Eiffel, CLOS, C# и Java являются объектно-ориентированными.

Основой объектно-ориентированного стиля является объектная модель. Эта модель состоит из следующих четырех главных элементов:

1. Абстракция
2. Инкапсуляция
3. Модульность
4. Иерархия

Абстракция – один из основных методов, позволяющих справиться со сложностью. Она выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко описывает его концептуальные границы с точки зрения наблюдателя.

Абстракция и инкапсуляция дополняют друг друга. В центре внимания абстракции находится наблюдаемое поведение объекта, а инкапсуляция сосредоточена на реализации, обеспечивающей заданное поведение. Как правило, инкапсуляция осуществляется с помощью сокрытия информации, то есть утаивания всех несущественных деталей объекта. Обычно скрываются как структуры объекта, так и реализация его методов.

Модульность – это разделение программы на фрагменты, которые компилируются по отдельности, но связаны между собой. В соответствии с определением Парнаса: «Связи между модулями – это их предположения о работе друг друга» [3]. В большинстве языков, поддерживающих принцип модульности как самостоятельную концепцию, интерфейс модуля отделен от его реализации. Таким образом, можно утверждать, что модульность и инкапсуляция тесно связаны между собой.

Иерархия – это ранжирование, или упорядочение абстракций. Наиболее важными видами иерархии в сложных системах являются структура классов (иерархия «общее/частное») и структура объектов (иерархия «целое/часть»).

ГЛАВА 2. РАЗРАБОТКА ИГРЫ

2.1. НАСТРОЙКА ПРОЕКТА

Vulkan – это графический вычислительный API нового поколения, который обеспечивает высокоэффективный кроссплатформенный доступ к современным графическим процессорам, используемым в самых разных устройствах: от ПК и консолей до мобильных телефонов и встроенных платформ. В отличие от OpenGL он обеспечивает приложениям большую производительность от снижения нагрузки на ЦП, уменьшения объема памяти и более высокой степени стабильности.

Подключение динамической библиотеки Vulkan производится путем импорта библиотеки Volk. Volk – это набор утилит для генерации кода для доступа к функциям Vulkan API на различных языках программирования.

Для создания окна, в котором отображается графическое содержимое приложения используется библиотека GLFW. GLFW предоставляет удобный кроссплатформенный интерфейс для создания окон и управления пользовательским вводом в графических приложениях. Хотя GLFW изначально создавалась для работы с OpenGL, она также хорошо поддерживает Vulkan и может быть использована для создания окна и контекста Vulkan, а также для обработки пользовательского ввода. Таким образом, использование GLFW в совокупности с Vulkan упрощает создание графических приложений, обеспечивая унифицированный и удобный интерфейс для работы с окнами и пользовательским вводом на различных платформах.

2.2. СОЗДАНИЕ ОКОННОГО И ОСНОВНОГО РИЛОЖЕНИЯ

Класс Window, представленный в приложении 1, предназначен для управления окном приложения с использованием библиотеки GLFW и интеграции этого окна с Vulkan. Этот класс инкапсулирует функциональность

для создания окна, обработки событий изменения размеров, а также создания Vulkan поверхности для рендеринга. Рассмотрим детали и функционал класса.

В классе Window объявлена и определена переменная, хранящая указатель на окно GLFW. Как было упомянуто раньше, GLFW – это библиотека для управления окнами, не зависящая от платформы. Это означает, что мы можем использовать эту переменную для открытия окна на разных операционных системах, таких как Mac, Windows или Linux.

Конструктор класса принимает ширину, высоту и имя окна. Он инициализирует члены класса и вызывает функцию initWindow для создания окна. GLFW изначально был разработан для создания контекста OpenGL, но поскольку мы используем Vulkan, эта функции отключена с помощью команды glfwWindowHint с параметром GLFW_NO_API.

Конструктор копирования и оператор присваивания копированием в классе Window удалены, поскольку переменная window хранит указатель на окно GLFW. В программе соблюдается принцип RAII (Resource Acquisition Is Initialization), который заключается в том, что получение ресурса совмещается с инициализацией, а освобождение – с уничтожением объекта. Если случайно создать копию объекта окна, вызовется один из деструкторов, что приведет к завершению работы общего окна GLFW и оставит висячий указатель (рис. 2.1).

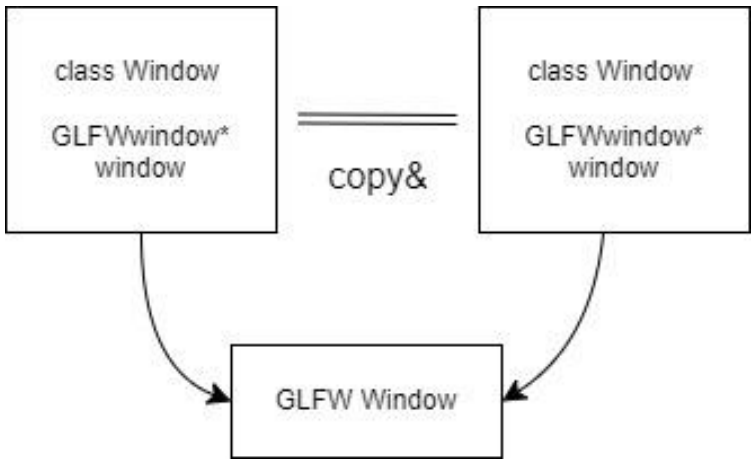

Рис. 2.1. Копия указателя на окно GLFW

Класс App, представленный в приложении 2, – основной компонент приложения, управляющего логикой и взаимодействием Vulkan с GLFW. Этот класс является центральным элементом игры, обеспечивая инициализацию, управление игры, обработку событий и рендеринг. В нем создается объект класса Window, который предоставляет интерфейс для создания поверхности, необходимой для рендеринга. Он также управляет обработкой событий, таких как изменение размеров окна и его закрытие.

Запуск игры осуществляется в функции main, представленной в приложении 3. В ней создается экземпляр класса App, инициализируются все необходимые ресурсы, и запускается основной игровой цикл. Функция также обрабатывает возможные исключения, которые могут возникнуть во время выполнения приложения, обеспечивая корректное завершение работы в случае ошибки.

2.3. СОЗДАНИЕ ГРАФИЧЕСКОГО КОНВЕЙЕРА

Для начала разберемся с тем, как компьютер рисует изображения на примере простого треугольника. Треугольник может быть определен тремя вершинами, по одной на каждый угол. В простой двумерной системе координат положение каждой вершины можно представить двумя числами: координатой x и y. Таким образом, для описания треугольника нам нужно шесть значений. Итак, есть шесть значений на входе, а на выходе необходимо

изображение, представляющее треугольник. Поэтому все, что нужно сделать, это определить, какие пиксели больше всего содержатся в треугольнике, и задать каждому из них цвет (рис. 2.2). Именно эту задачу выполняет графический конвейер.

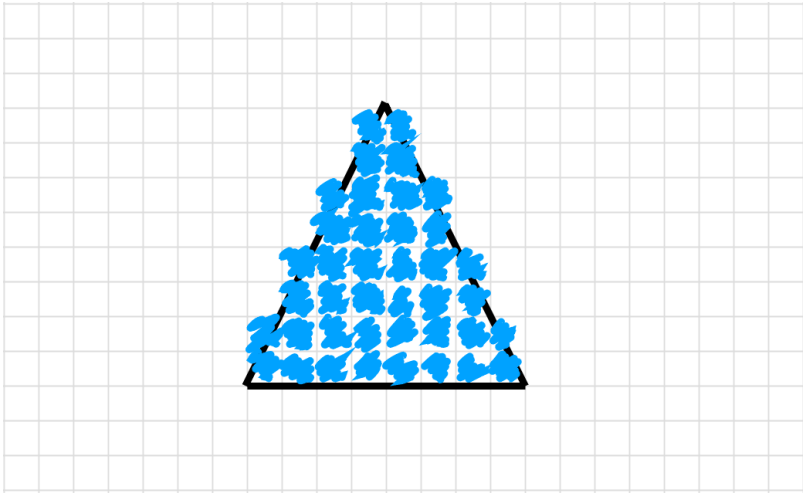

Рис. 2.2. Пиксели, содержащиеся в треугольнике

Графический конвейер представляет собой линейную последовательность этапов, где каждый этап принимает на вход некоторые данные, выполняет над ними некоторые операции и выводит преобразованные данные в качестве входных данных на следующий этап. Можно воспринимать графический конвейер как сборочную линию, на каждом этапе которой необработанные данные преобразуются во что-то более похожее на окончательное изображение.

Первый этап – это входной сборщик, который принимает на вход список чисел и группирует их в геометрию. В примере с треугольником первые шесть чисел можно использовать для формирования первого треугольника, следующие шесть – для второго треугольника и так далее.

Затем вершинный шейдер обрабатывает каждую вершину индивидуально и выполняет такие преобразования, как вращение и перемещение.

Этап растеризации разбивает геометрию на фрагменты для каждого пикселя, который принадлежит треугольнику. Это важный шаг, так как

именно здесь вычисляется, какие пиксели должны быть закрашены, чтобы сформировать изображение.

Фрагментный шейдер обрабатывает каждый фрагмент индивидуально. и выводит такие значения, как цвет, используя интерполированные данные из таких объектов, как текстуры, нормали и освещение.

Последний этап – это этап смешивания цветов. На этом этапе цветов применяются операции для смешивания значений из нескольких фрагментов, которые соответствуют одному и тому же пикселю в конечном изображении.

Существует два типа этапов: фиксированные функции и программируемые. Входной сборщик, растеризация и смешивание цветов – это фиксированные функции. Как программисты, мы имеем меньше контроля над тем, какие операции выполняют эти этапы. Мы можем лишь настроить каждый этап, установив переменные, которые меняют его поведение. Для программируемых этапов у нас есть возможность загрузить собственный код для выполнения графическим процессором. Эти небольшие программы, работающие на графическом процессоре, называются шейдерами и могут быть написаны на C-подобном языке под названием GLSL. Перейдем к описанию реализации конвейера.

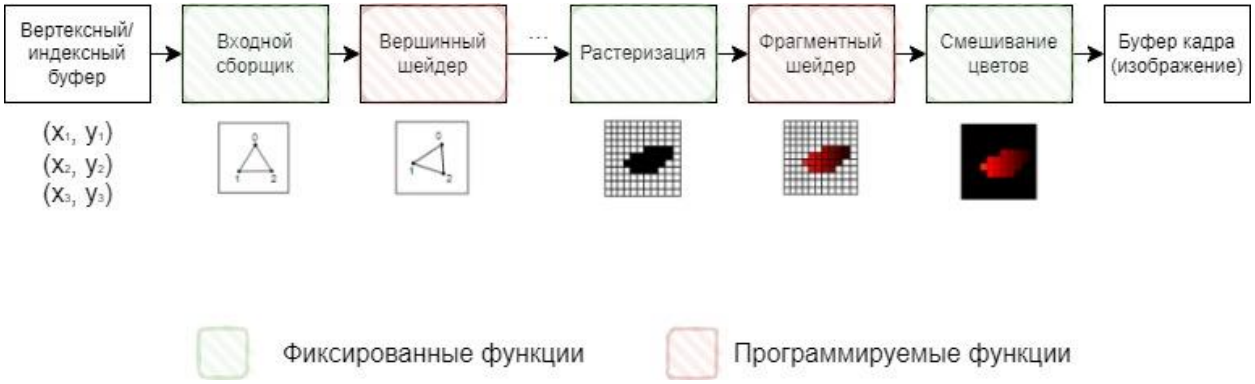

Рис. 2.3. Графический конвейер

Работа конвейера начинается с вершинного и фрагментного шейдеров. В каждом из них прописаны main функции, которые выполняются один раз для каждой имеющейся вершины в вершинном буфере и один раз для каждого имеющегося фрагмента в фрагментном буфере.

В качестве входных данных вершинный шейдер получает каждую вершину с входного сборщика, а затем он выводит ее позицию. Вместо того, чтобы возвращать это значение в функции main, есть специальная переменная под названием gl_Position. Это четырехмерный вектор, который отображается в изображении буфера кадра. Левый верхний угол соответствует координатам (-1, -1), а правый нижний угол – (1, 1). Это означает, что центр находится в точке с координатами (0, 0) (рис. 2.4).

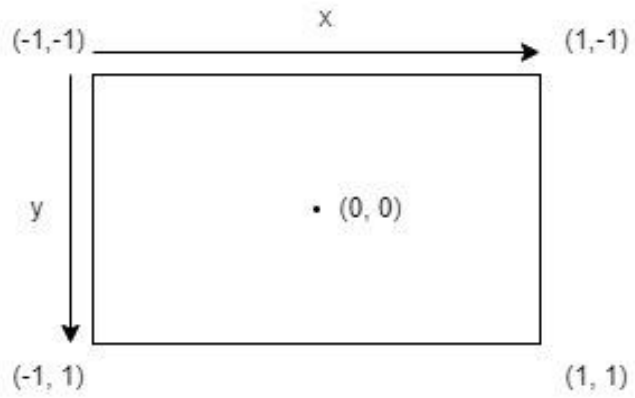

Рис. 2.4. Координатная система Vulkan

В отличие от вершинного шейдера, в фрагментном нет встроенной выходной переменной, поэтому необходимо объявить ее самостоятельно. Для ее объявления прописывается квалификатор, который принимает значение местоположения. В зависимости от того, как был настроен графический конвейер, фрагментный шейдер способен выводить данные в несколько разных мест. Далее указывается, что эта переменная будет использоваться как выходная с квалификатором out. И, наконец, в основной функции этой переменной присваивается значение, равное вектору из четырех элементов: красный, зеленый синий и альфа-канал. Таким образом, фрагментный шейдер запускается для каждого фрагмента, что определяется тем, какие пиксели в основном содержит геометрия на этапе растеризации. Подобно тому, как код C++ необходимо скомпилировать перед выполнением, необходимо компилировать код шейдера в промежуточный двоичный формат, известный как SPIR_V, каждый раз, когда в него вносятся изменения. Скомпилированные

файлы добавляются в программу с помощью метода readFile класса Pipeline, конструктор которого принимает константные строки, указывающие пути к файлам с шейдерами.

Класс Pipeline, представленный в приложении 4, содержит в себе ссылку на объект класса Device, который инкапсулирует логическое и физическое устройства. Физическое устройство – это графический процессор, способный работать с Vulkan. Чтобы связать его с приложением необходимо создать логическое устройство. При его создании указываются очереди, которые будут использоваться для выполнения команд, а также любые дополнительные возможности, которые должны быть включены, например, слои валидации. Оно также управляет различными буферами и выполняет команды рендеринга. Поэтому класс Device крайне необходим в создании конвейера.

PipelineConfigInfo – структура, представленная в приложении 4, которая, которая помогает организовать некоторые данные, использующиеся для настройки графического конвейера. Она заполняется значениями по умолчанию в функции defaultPipelineConfigInfo. Это позволяет легко создавать конвейер и делить конфигурацию между несколькими. Рассмотрим каждый член структуры:

1. ViewportInfo: параметр, определяющие область вывода, через которую происходит отображение изображения.
2. InputAssemblyInfo: настройки сборки входных данных, определяющие, как примитивы собираются из вершин.
3. RasterizationInfo: параметры растеризации, такие как режим заполнения полигонов, режим отсечения и смещение глубины.
4. MultisampleInfo: параметры мультисэмплинга, используемые для сглаживания изображения путем усреднения значений пикселей.
5. ColorBlendAttachment: параметры смешивания цветов для прикрепления цвета буфера кадра.

6. ColorBlendInfo: настройки смешивания цветов для всех прикреплений цвета в буфере кадра.
7. DepthStencilInfo: параметры тестирования глубины и трафарета для определения видимости и порядка рисования пикселей.
8. DynamicStateEnables: список состояний, которые могут динамически изменяться во время выполнения команд.
9. DynamicStateInfo: информация о динамических состояниях, включая их текущее состояние и способ обработки.
10. PipelineLayout: макет конвейера, который определяет формат данных, доступных в шейдерах конвейера.
11. RenderPass: проход рендеринга, определяющий прикрепления и подпроходы, используемые для рендеринга сцены.
12. Subpass: индекс подпрохода в рамках прохода рендеринга, используемый для связывания с конвейером.

Итак, используя вершинный и фрагментный шейдеры, физическое и логическое устройства, а также данные структуры PipelineConfigInfo метод createGraphicsPipeline создает графический конвейер.

2.4. СОЗДАНИЕ ЦЕПОЧКИ ОБМЕНА

Цепочка обмена – это очередь из изображений, ожидающих вывода на экран. Графический конвейер выводит данные в буфер кадра, который может иметь различные вложения, такие как буфер цвета или буфер глубины. Любая современная операционная система обычно использует как минимум два буфера одновременно: передний буфер и задний буфер.

Передний буфер (front buffer) – это буфер, который непосредственно отображается на экране монитора. Он содержит изображение, которое пользователь видит в реальном времени. Все изменения, происходящие с изображением (например, при отрисовке нового кадра или обновлении содержимого), сначала происходят в переднем буфере.

Задний буфер (back buffer) – это буфер, который содержит изображение, готовое к отображению на экране. Все новые изображения отрисовываются и сохраняются в заднем буфере.

Вертикальная синхронизация (V-Sync) – это момент, в который монитор начинает отображать новое изображение в зависимости от частоты обновления (рис. 2.5). Например, если частота обновления дисплея составляет 60 Гц, то есть 60 раз в секунду, происходит вертикальная синхронизация и передний буфер меняется с задним буфером (рис. 2.6).

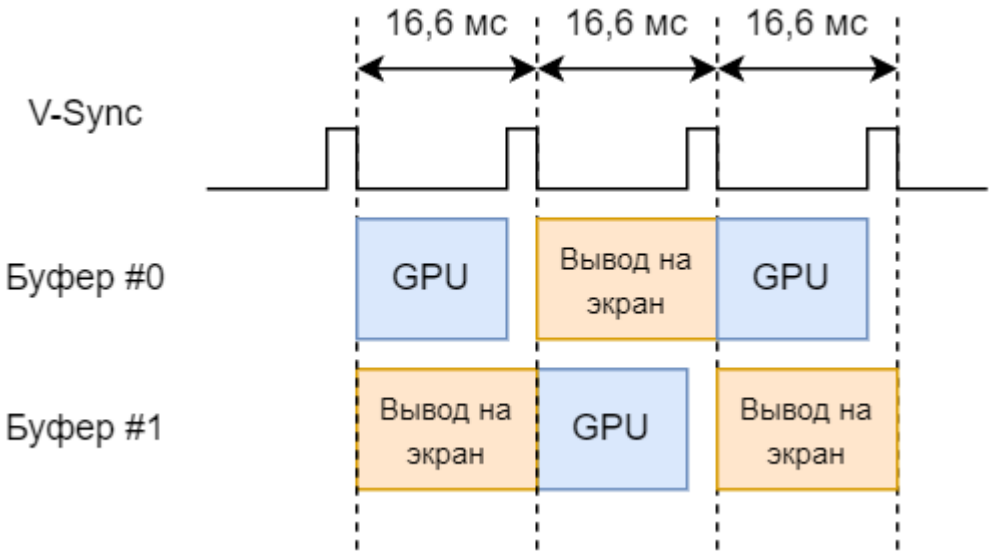

Рис. 2.5. Вертикальная синхронизация с двумя буферами

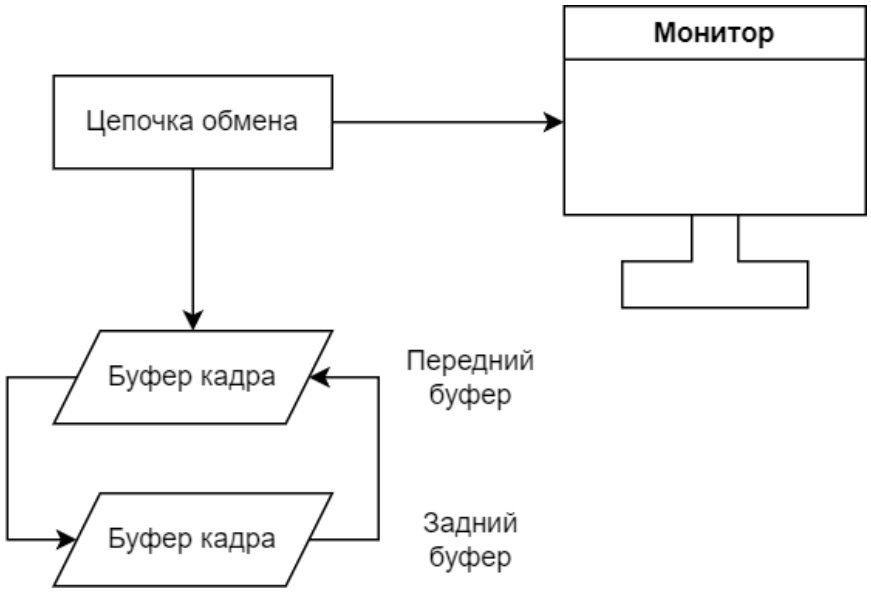

Рис. 2.6. Двойная буферизация

Синхронизацию можно игнорировать. Во многих играх это делается путем отключения V-Sync в настройках графики, в этом случае можно наблюдать разрывы изображения, когда монитор отображает несколько кадров одновременно. Стоит обратить внимание на то, что каждый кадровый буфер хранит в себе множество уникальных вложений, например, цветовые буферы, буферы глубины и другие данные, необходимые для отображения изображения. Но иногда допустимо совместное использование определенных вложений между буферами. Это позволяет сэкономить память и ресурсы. Однако, следует учитывать возможный недостаток такого подхода. После завершения рендеринга кадра графическому устройству придется дожидаться следующей синхронизации и замены буфера, прежде чем начать работу над следующим кадром. Это может привести к некоторым задержкам в обработке следующего кадра и, в конечном итоге, к снижению общей производительности или к возникновению ощутимых прерываний.

Если бы у нас был второй задний буфер, мы могли бы немедленно начать работу над следующим кадром. Это называется тройная буферизация (рис. 2.7). Ее основным недостатком является то, что для хранения дополнительного изображения требуется дополнительная память. Основной плюс возникает в тех случаях, когда рендеринг кадра не успевает за частотой обновления монитора. При двойной буферизации, даже если кадр просто не попадает в V-Sync, графическому устройству приходится ждать до следующей синхронизации, прежде чем оно сможет снова начать работать. Это приводит к внезапному падению частоты кадров. Тройная буферизация позволяет избежать этой проблемы, всегда предоставляя для работы буфер кадра (рис. 2.8). Работая с OpenGL или другими графическими API не приходится сталкиваться с настройкой тройной буферизацией. Обычно она обрабатывается автоматически, но, как и в случае с большинством объектов Vulkan, необходимо ее настроить. Рассмотрим реализацию класса цепочки обмена.

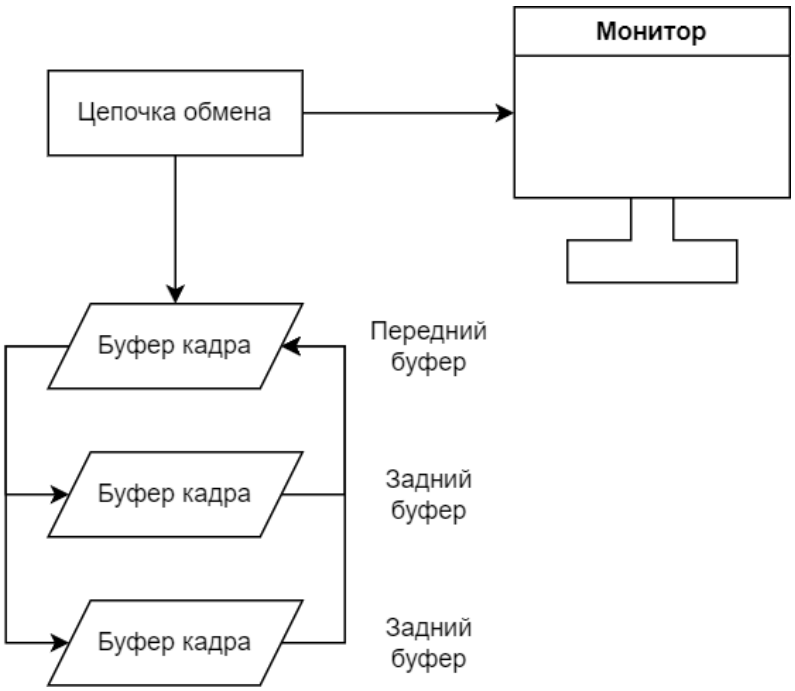

Рис. 2.7. Тройная буферизация

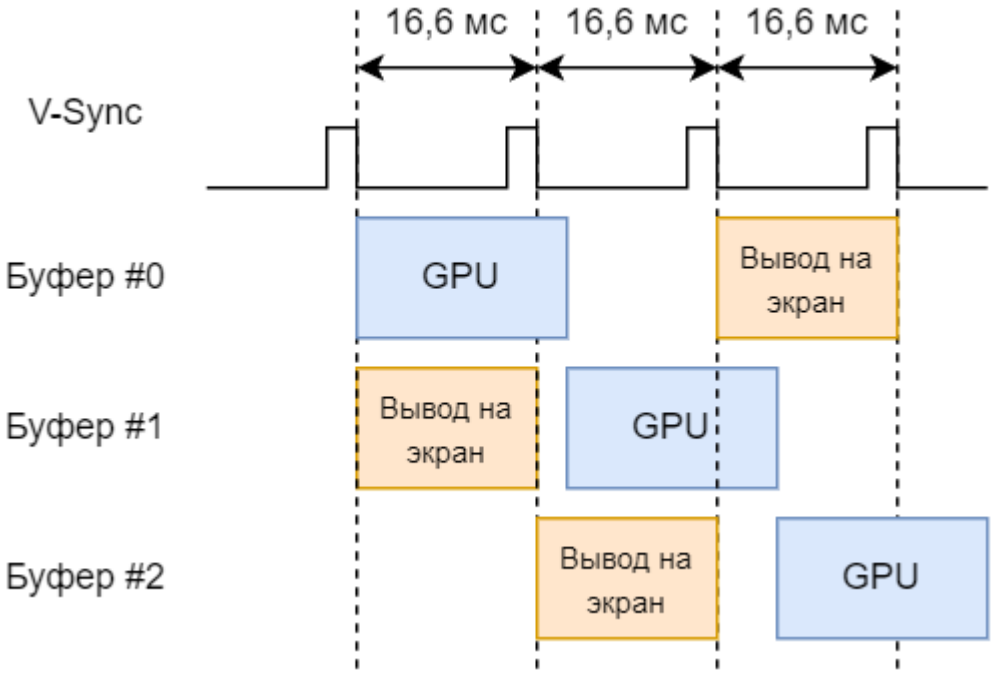

Рис. 2.8. Вертикальная синхронизация с тремя буферами

Класс SwapChain, представленный в приложении 5, отвечает за синхронизацию и настройку двойной или тройной буферизации в зависимости от возможностей графического устройства. Он также создает объекты буфера

кадра и связанные с ним вложения цвета и глубины, которые графический конвейер может использовать при рендеринге.

В классе SwapChain есть функция ChooseSwapPresentMode, которая выбирает режим отображения кадров в зависимости от графического устройства. В документации Vulkan представлено описание всех доступных режимов отображения [4]. Единственный режим, который гарантированно поддерживается – это FIFO. В режиме FIFO графический процессор может настолько быстро обрабатывать кадры, что даже при использовании тройной буферизации все возвращаемые буферы могут быть заняты, и графический процессор будет простаивать до следующего обновления. В режиме Mailbox графический процессор никогда не простаивает, он отбрасывает и перезаписывает старые кадры в обратных буферах. Поэтому по умолчанию был настроен выбор режима Mailbox, а в качестве резервного варианта – V-Sync.

В Vulkan команды не могут быть выполнены напрямую через вызовы функций. Вместо этого они записываются в буфер команд, который затем отправляется в очередь устройства для выполнения (рис. 2.9). Использование буферов команд позволяет записать последовательность команд один раз и многократно использовать их для нескольких кадров. Это отличается от подхода в OpenGL, где команды отрисовки должны повторятся для каждого кадра. Обычно буферы команд записываются один раз при инициализации программы и повторно используются для каждого кадра. Жизненный цикл буфера команд описывается диаграммой состояний. Когда буфер команд отправляется в очередь устройства, он переходит в состояние ожидания. Нельзя использовать буфер команд, который уже находится в состоянии ожидания, и перед повторной отправкой его необходимо завершить.

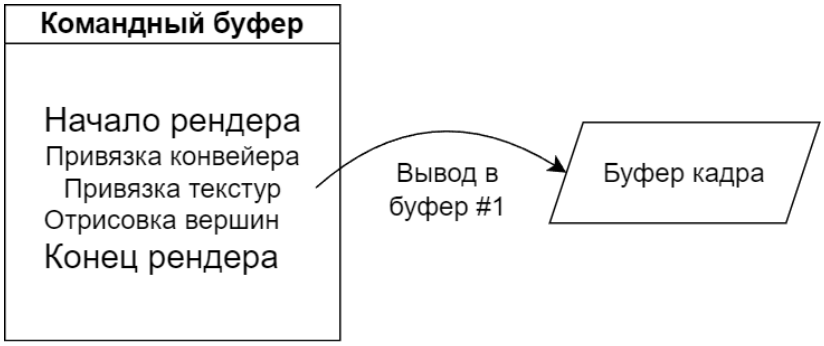

Рис. 2.9. Схема работы командного буфера

Класс SwapChain ограничивает количество буферов команд, которые можно отправить одновременно в очередь устройства. После отправки двух буферов ЦП заблокирует следующий вызов функции AcquireNextImage. Когда графический процессор завершит выполнение одного из командных буферов, он подаст сигнал процессору продолжить выполнение. Таким образом, можно обойтись использованием только двух буферов, даже если буферов кадра больше двух.

Класс Renderer, представленный в приложении 6, настраивает правильный процесс рендеринга с использованием буферов команд и цепочки обмена. Сначала при инициализации объекта Renderer вызывается метод createCommandBuffers, который создает командные буферы. Эти командные буферы предназначены для записи команд рендеринга. В методе beginFrame объекта Renderer вызывается метод acquireNextImage, который получает изображение из цепочки обмена. Полученное изображение будет использоваться для рендеринга на текущем кадре. После получения изображения из цепочки обмена в методе beginFrame, Renderer возвращает соответствующий командный буфер, который будет использоваться для записи команд рендеринга на этом кадре. После записи всех команд рендеринга для текущего кадра в командный буфер, метод endFrame вызывает метод endSwapChainRenderPass с передачей текущего командного буфера. Это завершает проход рендеринга для изображения цепочки обмена в переданном командном буфере. После завершения прохода рендеринга, объект Renderer

вызывает метод presentImage для представления полученного изображения на экране.

2.5. СОЗДАНИЕ ИГРОВЫХ ОБЪЕКТОВ

Для отрисовки сложных фигур используются буферы вершин, представляющие собой области памяти, передаваемые в вершинный шейдер. В этой памяти можно хранить любые данные, если графическому конвейеру сообщается, как эти данные структурированы.

Один из способов структурирования данных – группировка по атрибутам (рис. 2.10). Атрибут вершины – это входная переменная для каждой вершины. Например, можно создать атрибут вершины для двумерной позиции. Если это будет единственный атрибут, буфер вершин будет выглядеть так: каждые два числа будут представлять позиции x и y, объединяясь в двумерный вектор. Теперь предположим, что мы хотим добавить цвет RGB для каждой вершины. Это можно сделать, добавив второй атрибут вершины, состоящий из трех значений. В результате буфер вершин будет содержать сначала два значения для атрибута положения первой вершины, затем три значения для красного, зеленого и синего цветов. Следующие пять значений будут соответствовать второй вершине и так далее.

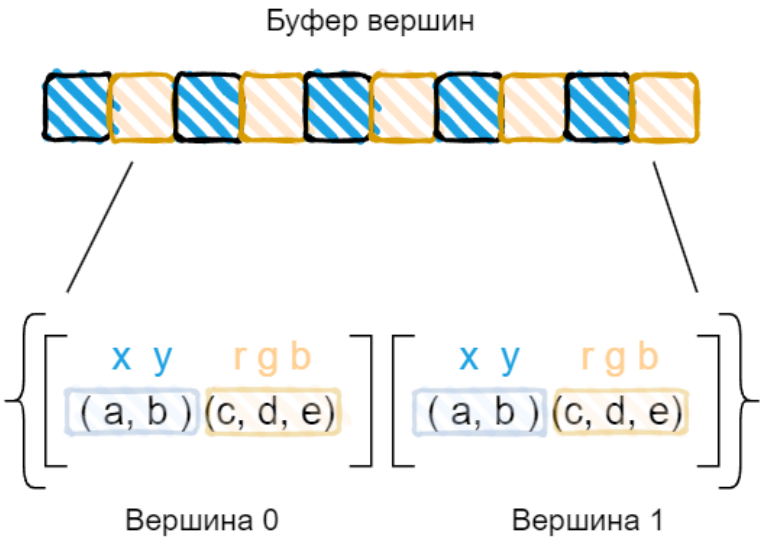

Рис. 2.10. Группировка вершин по атрибутам

Другой способ структурирования данных – разделение атрибутов на отдельные привязки. Например, при добавлении третьего атрибута можно чередовать все три атрибута в одной привязке или разделить их на отдельные привязки для каждого атрибута. Возможны различные комбинации.

При использовании вершинного буфера большинство вершин используются несколькими треугольниками, и в этом случае приходится дублировать данные этих вершин. Для большинства моделей дублирование этих данных просто не имеет смысла. И вот здесь на помощь приходят индексные буферы. Индексный буфер – это часть дополнительных данных, которые можно предоставить команде рисования модели, которая сообщает графическому процессору, как объединять вершины в треугольники. Индексный буфер подобен массиву указателей на буфер вершин. Первые три значения индекса говорят, какие три вершины составляют первый треугольник, следующие три вершины – второй треугольник и так далее. Таким образом, вершина может содержать только уникальные значения вершин. А чтобы повторно использовать вершину, мы просто используем ее индекс несколько раз в индексном буфере.

Рассмотрим прямоугольник, использующийся в качестве ракетки игрока. Имея только вершинный буфер, потребовалось бы в общей сложности шесть вершин: три для первого треугольника и еще три для второго, и пришлось бы дублировать эти две вершины (рис. 2.11). Используя индексный буфер, можно указать каждую уникальную вершину только один раз в буфере вершин, и индексный буфер скажет, как соединить эти вершины в треугольники (рис. 2.12). Таким образом, первый треугольник составят индексы 0, 1 и 2, а второй треугольник – индексы 2, 1 и 3.

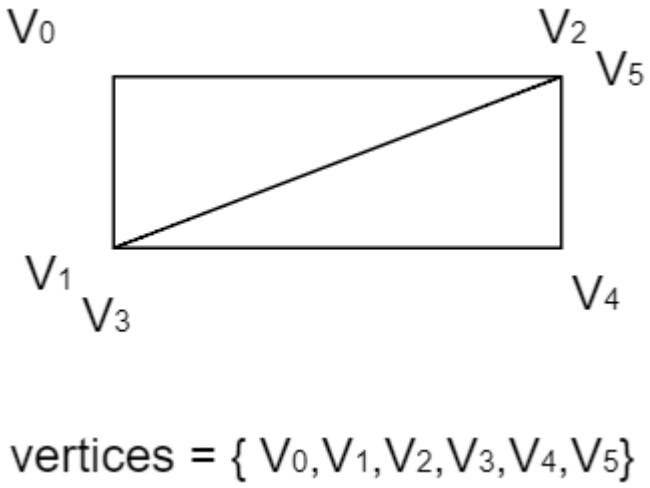

Рис. 2.11. Представление вершин прямоугольника без использования индексов

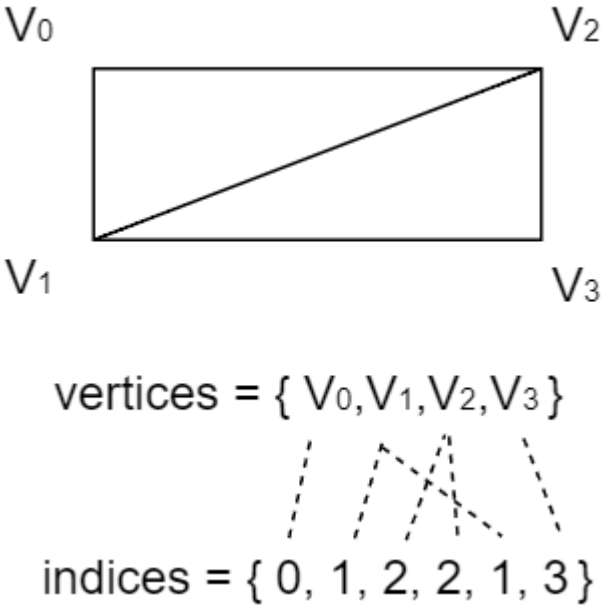

Рис. 2.12. Представление вершин прямоугольника с использованием индексов

Класс Model, представленный в приложении 7, представляет собой объект модели, который используется для хранения и управления данными вершин и индексов. Структура под названием builder содержит два вектора: vertices, хранящий в себе вершины буфера, и indices, хранящий в себе индексы. Метод bind привязывает буферы модели к командному буферу Vulkan. С его

помощью метод draw выполняет рисование модели. Такая реализация класса модели позволяет легко создавать и привязывать фигуры к игровому объекту.

Обычно в игре каждый объект рассматривается как класс GameObject. Он содержит данные и функции, описывающие поведение и характеристики игровых моделей. Хотя не все свойства и методы могут быть необходимы для всех объектов в игре, они все равно присутствуют у всех экземпляров класса. Это делает модель гибкой и легкой для реализации различных типов объектов и их прототипирования.

Итак, класс GameObject, представленный в приложении 8, содержит в себе информацию о модели объекта, его цвете и трансформации, которая включает в себя информацию о положении, повороте и масштабе объекта. Конструкторы и деструкторы GameObject отвечают за инициализацию и освобождение ресурсов, в том числе динамически выделенной памяти.

Классы Number и Primitive наследуются от GameObject и представляют конкретные виды игровых объектов: числовые цифры и примитивы (круги, прямоугольники). Каждый из этих классов имеет методы для создания соответствующих моделей на основе заданных параметров, таких как размер и форма.

За загрузку игровых объектов отвечает функция loadGameObject класса App. Она инициализирует игровые объекты, задает им начальные параметры, такие как положение, масштаб, цвет, и помещает их в соответствующие контейнеры для дальнейшего использования в приложении.

Сначала создаются два игрока с идентификаторами 1 и 2. Каждый игрок представлен объектом класса Player, который создается с использованием примитивной фигуры «Прямоугольник». Этот класс содержит в себе переменную count, представляющую счет игрока, и вектор указателей на игровые объекты, принадлежащих игроку. Для каждого объекта класса Player устанавливаются параметры трансформации (положение и масштаб) и цвет.

Далее создается мяч, который представлен так же объектом класса Primitive, но в этом случае используется примитив «Круг». Устанавливаются

его параметры трансформации и цвет. Затем создается пунктирная линия, состоящая из прямоугольников, которые представлены в векторе border. Каждому из них так же устанавливаются параметры трансформации и цвет.

2.6. СОЗДАНИЕ МЕХАНИКИ ИГРЫ

Функция run, представленная в приложении 9, класса App представляет собой основной игровой цикл приложения. Она начинается с инициализации необходимых объектов и переменных, таких как RenderSystem для отрисовки графики и KeyboardController для управления движением игроков. Затем она входит в главный цикл, который продолжается, пока окно приложения не будет закрыто. В каждой итерации цикла функция:

1. Проверяет события ввода, такие как нажатия клавиш, с помощью функции glfwPollEvents.
2. Вычисляет время, прошедшее с предыдущего кадра.
3. Обрабатывает движение игроков на основе ввода и времени кадра.
4. Проверяет столкновения мяча с игроками и границами поля, корректируя скорость и направление мяча при необходимости.
5. Проверяет условия победы для обоих игроков. В случае победы выводится сообщение о победе и счет сбрасывается.
6. Обновляет позицию мяча в соответствии с его текущим направлением и скоростью.
7. Начинает рендеринг игровых объектов, таких как игроки и мяч, с использованием RenderSystem.
8. Завершает цикл кадра и переходит к следующей итерации.

Класс KeyboardController, представленный в приложении 9, – компонент, отвечающий за управление объектом в игре с помощью клавиатуры. Он разделен на две структуры: Keys1 и Keys2, каждая из которых содержит набор клавиш для каждого из игроков. Класс имеет переменную move_speed, определяющую скорость движения объекта по умолчанию. Это позволяет легко настраивать скорость движения объекта в пределах кода.

Методы move и move2 отвечают за реакцию на нажатия клавиш и соответствующее изменение позиции объекта в зависимости от времени кадра. Оба метода используют текущее окно GLFW, переданное как параметр, чтобы определить состояние нажатия клавиш. По умолчанию, если нажаты клавиши для движения вперед или назад, объект будет двигаться в соответствующем направлении, учитывая текущий угол поворота объекта. Такая структура класса делает его удобным для использования в многопользовательских играх, где разным игрокам могут требоваться разные клавиши управления. Кроме того, наличие переменной скорости позволяет динамически изменять скорость движения объекта в зависимости от событий в игре или пользовательских предпочтений.

Метод findAngle, представленная в приложении 9, класса App используется для определения угла направления для мячика при столкновении с объектом. Сначала вычисляется разница между координатой Y верхней границы прямоугольника и координатой Y мячика. Это дает относительное расстояние между центрами мячика и прямоугольника. Затем относительное расстояние по оси Y делится на половину высоты прямоугольника, чтобы нормализовать его к диапазону от -1 до 1. Это нужно для получения угла направления в пределах $-\pi/2$ до $\pi/2$. Нормализованное относительное расстояние по оси Y умножается на 2, чтобы получить значение угла. Это дает нам угол направления мячика после отскока от объекта. Если мячик сталкивается с верхней частью объекта, угол будет положительным, если с нижней - отрицательным. Вычисленный угол возвращается из метода для использования в основном цикле игры.

Чтобы обнаружить столкновение мячика с ракеткой используется метод collisionDetection, представленный в приложении 9. Сначала вычисляются координаты testx и testy, которые представляют собой координаты мячика после возможного столкновения с прямоугольником. Затем проверяется, находится ли мячик слева от прямоугольника. Если да, то testx устанавливается равным X-координате прямоугольника. А если нет, то testx

устанавливается равным X-координате прямоугольника, увеличенной на половину его ширины. Аналогично для координаты Y: если мячик находится выше прямоугольника, testy устанавливается равным Y-координате прямоугольника, если ниже – testy устанавливается равным Y-координате прямоугольника, увеличенной на половину его высоты. Затем расстояние между центрами мячика и прямоугольника вычисляется по формуле Евклидова расстояния: корень из суммы квадратов разностей координат по осям X и Y. Если расстояние между центрами мячика и прямоугольника меньше, чем половина ширины или высоты мячика, то столкновение считается обнаруженным.

После завершения цикла основной работы происходит ожидание завершения работы устройства. Эта функция представляет собой основной поток выполнения приложения, который обрабатывает все аспекты игровой логики, пользовательского ввода и отрисовки графики.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы были достигнуты значительные результаты, подтверждающие актуальность и эффективность применения современных технологий в игровой индустрии. Использование Vulkan в проекте позволило достичь высокой производительности и гибкости в рендеринге графики, что подтвердило его преимущества перед традиционными графическими API. Применение ООП способствовало созданию структурированного и легко поддерживаемого кода, а также обеспечило возможность расширения функциональности игры без значительных изменений в существующей архитектуре. Внедрение концепции RAII улучшило управление ресурсами, минимизируя риски утечек памяти и других ресурсов, что особенно важно для стабильности и производительности игрового приложения. Паттерн проектирования ECS показал свою эффективность в организации игрового мира, предоставляя возможность динамического изменения характеристик объектов и их взаимодействий. Такая структура оказалась удобной для управления большим количеством игровых сущностей, обеспечивая при этом высокую производительность и гибкость.

Результаты данной работы демонстрируют, что использование Vulkan в сочетании с принципами ООП позволяет создавать современные, производительные и легко масштабируемые игровые проекты. В перспективе дальнейшие исследования могут быть направлены на углубленное изучение возможностей Vulkan и оптимизацию разработанных решений для создания более сложных и реалистичных игровых миров. Разработанные методы и подходы могут быть применены не только в игровой индустрии, но и в других областях, требующих высокопроизводительного рендеринга и эффективного управления ресурсами.

СПИСОК ЛИТЕРАТУРЫ

1. Буч, Гради, Максимчук, Роберт А., Энгл, Майкл У., Янг, Бобби Дж., Коналлен, Джим, Хьюстон, Келли А. Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2008. – 720 с.
2. Wirth, N. 1986. Algorithms and Data Structures. Englewood Cliffs, NJ: Prentice-Hall.
3. Liskov, B. 1980. A Design Methodology for Reliable Software Systems. In: Tutorial on Software Design Techniques. Third Edition. New York, NY: IEEE Computer Society, p. 66.
4. Vulkan Documentation: [Электронный ресурс]. URL: https://docs.vulkan.org.

ПРИЛОЖЕНИЕ 1.

КЛАСС ОКНА

```
class  __declspec(dllexport) Window
{
private:
	static void framebufferResizeCallback(GLFWwindow* window, int
width, int height);
	void initWindow();

	int width;
	int height;
	bool framebufferResized = false;

	std::string windowName;
	GLFWwindow* window;

public:
	Window(int w, int h, std::string name);

	Window(const Window&) = delete;
	Window& operator=(const Window&) = delete;

	bool shouldClose() { return glfwWindowShouldClose(window); }
	VkExtent2D getExtent() { return {
static_cast<uint32_t>(width), static_cast<uint32_t>(height) }; }
	bool wasWindowResized() { return framebufferResized; }
	void resetWindowResizedFlag() { framebufferResized = false; }
	GLFWwindow* getWindow() { return window; }
	void createWindowSurface(VkInstance instance, VkSurfaceKHR*
surface);

	~Window();

};
```

ПРИЛОЖЕНИЕ 2.

КЛАСС ОСНОВНОГО ПРИЛОЖЕНИЯ

```
class __declspec(dllexport) App
{
	public:
	static constexpr int WIDTH = 800;
	static constexpr int HEIGHT = 800;

	App();
	~App();

	App(const App &) = delete;
	App &operator=(const App &) = delete;

	void run();

private:
	void loadGameObjects();
	void resetBall();
	bool collisionDetection(GameObject*& circle, GameObject*&
rect);
	bool borderDetectionY(GameObject*& o);
	void pBorderDetectionY(GameObject*& o);
	void changeCounter(int p1_count, int p2_count);
	float findAngle(GameObject*& circle, GameObject*& rect);

	Window _Window { WIDTH, HEIGHT, "Vulkan Ping Pong Game" };
	Device _Device { _Window };
	Renderer _Renderer { _Window, _Device };

	std::map<int, Player*> player;
	std::vector<GameObject*> border;
	std::vector<GameObject*> players;
	std::vector<GameObject*> ball;

};
```

ПРИЛОЖЕНИЕ 3.

КОД ГЛАВНОЙ ФУНКЦИИ

```
struct Leaks
{
    ~Leaks()
    {
        FILE* file = fopen("MemoryLeakReport.txt", "w");
        _CrtSetReportMode(_CRT_WARN, _CRTDBG_MODE_FILE);
        _CrtSetReportFile(_CRT_WARN,
(void*)_get_osfhandle(_fileno(file)));
        _CrtSetReportMode(_CRT_ERROR, _CRTDBG_MODE_FILE);
        _CrtSetReportFile(_CRT_ERROR,
(void*)_get_osfhandle(_fileno(file)));
        _CrtSetReportMode(_CRT_ASSERT, _CRTDBG_MODE_FILE);
        _CrtSetReportFile(_CRT_ASSERT,
(void*)_get_osfhandle(_fileno(file)));
        _CrtDumpMemoryLeaks();

        fclose(file);
    }
}_Leaks;


int main()
{
    VulkanGame::App app{};
    app.run();
    try
    {
        app.run();
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what() << '\n';
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;

}
```

ПРИЛОЖЕНИЕ 4.

КЛАСС ГРАФИЧЕСКОГО КОНВЕЙЕРА

```
struct  __declspec(dllexport) PipelineConfigInfo
{
    PipelineConfigInfo() = default;
    PipelineConfigInfo(const PipelineConfigInfo&) = delete;
    PipelineConfigInfo& operator=(const PipelineConfigInfo&) = delete;

    VkPipelineViewportStateCreateInfo viewportInfo;
    VkPipelineInputAssemblyStateCreateInfo inputAssemblyInfo;
    VkPipelineRasterizationStateCreateInfo rasterizationInfo;
    VkPipelineMultisampleStateCreateInfo multisampleInfo;
    VkPipelineColorBlendAttachmentState colorBlendAttachment;
    VkPipelineColorBlendStateCreateInfo colorBlendInfo;
    VkPipelineDepthStencilStateCreateInfo depthStencilInfo;
    std::vector<VkDynamicState> dynamicStateEnables;
    VkPipelineDynamicStateCreateInfo dynamicStateInfo;
    VkPipelineLayout pipelineLayout = nullptr;
    VkRenderPass renderPass = nullptr;
    uint32_t subpass = 0;
};

class  __declspec(dllexport) Pipeline
{
private:
    static std::vector<char> readFile(const std::string& filepath);

    void createGraphicsPipeline(const std::string& vertFilepath, const
std::string& fragFilepath, const PipelineConfigInfo& configInfo);
    void createShaderModule(const std::vector<char>& code,
VkShaderModule* shaderModule);

    Device& _Device;
    VkPipeline graphicsPipeline;
    VkShaderModule vertShaderModule;
    VkShaderModule fragShaderModule;

public:
    Pipeline(Device& device, const std::string& vertFilepath, const
std::string& fragFilepath, const PipelineConfigInfo& configInfo);
    ~Pipeline();

    Pipeline(const Pipeline&) = delete;
    Pipeline& operator=(const Pipeline&) = delete;

    void bind(VkCommandBuffer commandBuffer);

    static void defaultPipelineConfigInfo(PipelineConfigInfo&
configInfo);};
```

ПРИЛОЖЕНИЕ 5.

КЛАСС ЦЕПОЧКИ ОБМЕНА

```
class  __declspec(dllexport) SwapChain
{
private:
    void init();
    void createSwapChain();
    void createImageViews();
    void createDepthResources();
    void createRenderPass();
    void createFramebuffers();
    void createSyncObjects();

    VkSurfaceFormatKHR chooseSwapSurfaceFormat(
        const std::vector<VkSurfaceFormatKHR>& availableFormats);
    VkPresentModeKHR chooseSwapPresentMode(
        const std::vector<VkPresentModeKHR>& availablePresentModes);
    VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR&
capabilities);

    VkFormat swapChainImageFormat;
    VkFormat swapChainDepthFormat;
    VkExtent2D swapChainExtent;

    std::vector<VkFramebuffer> swapChainFramebuffers;
    VkRenderPass renderPass;

    std::vector<VkImage> depthImages;
    std::vector<VkDeviceMemory> depthImageMemorys;
    std::vector<VkImageView> depthImageViews;
    std::vector<VkImage> swapChainImages;
    std::vector<VkImageView> swapChainImageViews;

    Device& device;
    VkExtent2D windowExtent;

    VkSwapchainKHR swapChain;
    std::shared_ptr<SwapChain> oldSwapChain;

    std::vector<VkSemaphore> imageAvailableSemaphores;
    std::vector<VkSemaphore> renderFinishedSemaphores;
    std::vector<VkFence> inFlightFences;
    std::vector<VkFence> imagesInFlight;
    size_t currentFrame = 0;

public:
    static constexpr int MAX_FRAMES_IN_FLIGHT = 2;

    SwapChain(Device& deviceRef, VkExtent2D windowExtent);
```

```
    SwapChain(Device& deviceRef, VkExtent2D windowExtent,
std::shared_ptr<SwapChain> previous);

    ~SwapChain();

    SwapChain(const SwapChain&) = delete;
    SwapChain& operator=(const SwapChain&) = delete;

    VkFramebuffer getFrameBuffer(int index) { return
swapChainFramebuffers[index]; }
    VkRenderPass getRenderPass() { return renderPass; }
    VkImageView getImageView(int index) { return
swapChainImageViews[index]; }
    size_t imageCount() { return swapChainImages.size(); }
    VkFormat getSwapChainImageFormat() { return swapChainImageFormat;
}
    VkExtent2D getSwapChainExtent() { return swapChainExtent; }
    uint32_t width() { return swapChainExtent.width; }
    uint32_t height() { return swapChainExtent.height; }

    float extentAspectRatio()
    {
        return static_cast<float>(swapChainExtent.width) /
static_cast<float>(swapChainExtent.height);
    }
    VkFormat findDepthFormat();

    VkResult acquireNextImage(uint32_t* imageIndex);
    VkResult submitCommandBuffers(const VkCommandBuffer* buffers,
uint32_t* imageIndex);

    bool compareSwapFormats(const SwapChain& swapChain) const
    {
        return swapChain.swapChainDepthFormat == swapChainDepthFormat
&&
            swapChain.swapChainImageFormat == swapChainImageFormat;
    }

};
```

ПРИЛОЖЕНИЕ 6.

КЛАСС РЕНДЕРА

```
class  __declspec(dllexport) Renderer
 {
 private:
     void createCommandBuffers();
     void freeCommandBuffers();
     void recreateSwapChain();

     Window& _Window;
     Device& _Device;
     std::unique_ptr<SwapChain> _SwapChain;
     std::vector<VkCommandBuffer> commandBuffers;

     uint32_t currentImageIndex;
     int currentFrameIndex;
     bool isFrameStarted;

 public:
     Renderer(Window& window, Device& device);

     Renderer(const Renderer&) = delete;
     Renderer& operator=(const Renderer&) = delete;

     VkRenderPass getSwapChainRenderPass() const { return _SwapChain-
>getRenderPass(); }
     bool isFrameInProgress() const { return isFrameStarted; }

     VkCommandBuffer getCurrentCommandBuffer() const
     {
         assert(isFrameStarted && "Cannot get command buffer when
frame not in progress");
         return commandBuffers[currentFrameIndex];
     }

     int getFrameIndex() const
     {
         assert(isFrameStarted && "Cannot get frame index when frame
not in progress");
         return currentFrameIndex;
     }

     VkCommandBuffer beginFrame();
     void endFrame();
     void beginSwapChainRenderPass(VkCommandBuffer commandBuffer);
     void endSwapChainRenderPass(VkCommandBuffer commandBuffer);

     ~Renderer();};
```

ПРИЛОЖЕНИЕ 7.

КЛАСС МОДЕЛИ

```
class  __declspec(dllexport) Model
{
public:
    struct Vertex
    {
        glm::vec2 position;
        glm::vec3 color;
        static std::vector<VkVertexInputBindingDescription>
getBindingDescriptions();
        static std::vector<VkVertexInputAttributeDescription>
getAttributeDescriptions();
    };

    struct Builder
    {
        std::vector<Vertex> vertices{};
        std::vector<uint32_t> indices{};
    };

    Model(Device& device, const Model::Builder &builder);
    ~Model();
    Model(const Model&) = delete;
    Model& operator=(const Model&) = delete;
    void bind(VkCommandBuffer commandBuffer);
    void draw(VkCommandBuffer commandBuffer);

private:
    void createVertexBuffers(const std::vector<Vertex>& vertices);
    void createIndexBuffer(const std::vector<uint32_t>& indices);
    Device& lveDevice;
    VkBuffer vertexBuffer;
    VkDeviceMemory vertexBufferMemory;
    uint32_t vertexCount;

    bool hasIndexBuffer = false;
    VkBuffer indexBuffer;
    VkDeviceMemory indexBufferMemory;
    uint32_t indexCount;

};
```

ПРИЛОЖЕНИЕ 8.

КЛАСС ИГРОВОГО ОБЪЕКТА

```
struct  __declspec(dllexport) TransformComponent
{
    glm::vec2 translation{};
    glm::vec2 scale{ 1.f, 1.f };
    float rotation;

    glm::mat2 mat2()
    {
        const float s = glm::sin(rotation);
        const float c = glm::cos(rotation);
        glm::mat2 rotMatrix{ {c, s}, {-s, c} };

        glm::mat2 scaleMat{ {scale.x, .0f}, {.0f, scale.y} };
        return rotMatrix* scaleMat;
    }
};

class  __declspec(dllexport) GameObject
{
public:
    GameObject() {}

    GameObject(const GameObject&) = delete;
    GameObject& operator=(const GameObject&) = delete;
    GameObject(GameObject&&) = default;
    GameObject& operator=(GameObject&&) = default;

    Model* model{};
    glm::vec3 color{};
    TransformComponent transform{};

    virtual ~GameObject()
    {
        delete model;
    }
};

class  __declspec(dllexport) Number : public GameObject
{
public:
    Number(std::string num, Device& device)
    {
        if (num == "0")
        {
            createNumberZero(device);
        }
        if (num == "1")
```

```
        {
            createNumberOne(device);
        }
        if (num == "2")
        {
            createNumberTwo(device);
        }
        if (num == "3")
        {
            createNumberThree(device);
        }
    }

    void createNumberZero(Device& device)
    {
        Model::Builder modelBuild{};
        modelBuild.vertices =
        {
             {{0.5f, 0.5f}, {0.9f, 0.9f, 0.9f}},
             {{0.5f, -0.5f}, {0.9f, 0.9f, 0.9f}},
             {{-0.5f, -0.5f}, {0.9f, 0.9f, 0.9f}},
             {{-0.5f, 0.5f}, {0.9f, 0.9f, 0.9f}},
             {{0.25f, 0.25f}, {0.9f, 0.9f, 0.9f}},
             {{0.25f, -0.25f}, {0.9f, 0.9f, 0.9f}},
             {{-0.25f, -0.25f}, {0.9f, 0.9f, 0.9f}},
             {{-0.25f, 0.25f}, {0.9f, 0.9f, 0.9f}},
        };

        modelBuild.indices = { 0, 4, 5, 0, 5, 1, 5, 1, 2, 6, 5, 2, 7,
2, 6, 7, 3, 2, 3, 7, 0, 0, 4, 7 };
        model = new Model(device, modelBuild);
        color = { 1.f, 0.f, 0.f };
        transform.scale = { -0.08f, 0.08f };
    }

    void createNumberOne(Device& device)
    {
        Model::Builder modelBuild{};
        modelBuild.vertices =
        {
            {{-0.5f, -0.5f}, {0.9f, 0.9f, 0.9f}},
            {{0.5f, -0.5f}, {0.9f, 0.9f, 0.9f}},
            {{-0.5f, 0.5f}, {0.9f, 0.9f, 0.9f}},
            {{0.5f, 0.5f}, {0.9f, 0.9f, 0.9f}},
        };

        modelBuild.indices = { 0, 1, 2, 2, 1, 3 };
        model = new Model(device, modelBuild);
        color = { 1.f, 0.f, 0.f };
        transform.scale = { 0.03f, 0.08f };
    }
```

```
    void createNumberTwo(Device& device)
    {
        Model::Builder modelBuild{};
        modelBuild.vertices =
        {
             {{0.5f, 0.5f}, {0.9f, 0.9f, 0.9f}},
             {{0.5f, 0.25f}, {0.9f, 0.9f, 0.9f}},
             {{-0.08f, -0.25f}, {0.9f, 0.9f, 0.9f}},
             {{0.5f, -0.25f}, {0.9f, 0.9f, 0.9f}},
             {{0.5f, -0.5f}, {0.9f, 0.9f, 0.9f}},
             {{-0.5f, -0.5f}, {0.9f, 0.9f, 0.9f}},
             {{-0.5f, -0.25f}, {0.9f, 0.9f, 0.9f}},
             {{0.08f, 0.25f}, {0.9f, 0.9f, 0.9f}},
             {{-0.5f, 0.25f}, {0.9f, 0.9f, 0.9f}},
             {{ -0.5f, 0.5f}, {0.9f, 0.9f, 0.9f}}
        };

        modelBuild.indices = { 9, 0, 1, 1, 2, 7, 6, 3, 5, 3, 4, 5, 7,
2, 6, 9, 1, 8 };
        model = new Model(device, modelBuild);
        color = { 1.f, 0.f, 0.f };
        transform.scale = { 0.08f, 0.08f };
    }

    void createNumberThree(Device& device)
    {
        Model::Builder modelBuild{};
        modelBuild.vertices =
        {
            {{0.5f, 0.5f}, {0.9f, 0.9f, 0.9f}},
            {{0.5f, -0.5f}, {0.9f, 0.9f, 0.9f}},
            {{-0.5f, -0.5f}, {0.9f, 0.9f, 0.9f}},
            {{-0.5f, -0.25f}, {0.9f, 0.9f, 0.9f}},
            {{0.0f, -0.25f}, {0.9f, 0.9f, 0.9f}},
            {{0.0f, -0.12f}, {0.9f, 0.9f, 0.9f}},
            {{-0.5f, -0.12f}, {0.9f, 0.9f, 0.9f}},
            {{-0.5f, 0.12f}, {0.9f, 0.9f, 0.9f}},
            {{0.0f, 0.12f}, {0.9f, 0.9f, 0.9f}},
            {{0.0f, 0.25f}, {0.9f, 0.9f, 0.9f}},
            {{-0.5f, 0.25f}, {0.9f, 0.9f, 0.9f}},
            {{-0.5f, 0.5f}, {0.9f, 0.9f, 0.9f}},
        };

        modelBuild.indices = { 11, 0, 10, 10, 9, 0, 0, 9, 8, 0, 8, 1,
7, 8, 6, 8, 5, 6, 8, 5, 1, 5, 1, 4, 1, 4, 2, 4, 2, 3 };
        model = new Model(device, modelBuild);
        transform.scale = { 0.08f, 0.08f };
    }
};
```

```
class  __declspec(dllexport) Primitive : public GameObject
{
public:
    Primitive(std::string prim, Device& device)
    {
        if (prim == "Circle")
        {
            createCircle(device);
        }
        if (prim == "Rectangle")
        {
            createRectangle(device);
        }
    }

    void createRectangle(Device& device)
    {
        Model::Builder modelBuild{};
        modelBuild.vertices =
        {
            {{-0.5f, -0.5f}, {0.9f, 0.9f, 0.9f}},
            {{0.5f, -0.5f}, {0.9f, 0.9f, 0.9f}},
            {{-0.5f, 0.5f}, {0.9f, 0.9f, 0.9f}},
            {{0.5f, 0.5f}, {0.9f, 0.9f, 0.9f}},
        };

        modelBuild.indices = { 0, 1, 2, 2, 1, 3 };
        model = new Model(device, modelBuild);
    }

    void createCircle(Device& device)
    {
        int sides = 50;
        std::vector<Model::Vertex> uniqueVertices{};
        for (float i = 0; i < sides; i++) {
            float angle = i * glm::two_pi<float>() / sides;
            uniqueVertices.push_back({ {glm::cos(angle),
glm::sin(angle)} });
        }
        uniqueVertices.push_back({});

        Model::Builder modelBuild{};
        std::vector<Model::Vertex> vert{};
        for (int i = 0; i < sides; i++)
        {
            vert.push_back(uniqueVertices[i]);
            vert.push_back(uniqueVertices[(i + 1) % sides]);
            vert.push_back(uniqueVertices[sides]);
        }
        modelBuild.vertices = vert;
```

```
            model = new Model(device, modelBuild);
        }
    };
```

ПРИЛОЖЕНИЕ 9.

ФУНКЦИЯ ИГРОВОГО ЦИКЛА

```
void App::run()
 {
     RenderSystem renderSystem{ _Device,
_Renderer.getSwapChainRenderPass() };
     auto currentTime = std::chrono::high_resolution_clock::now();

     KeyboardController playerController;
     KeyboardController player2Controller;

     float yaw = 1.5;
     int speed = 1.5;

     while (!_Window.shouldClose())
     {
         glfwPollEvents();

         auto newTime = std::chrono::high_resolution_clock::now();
         float frameTime = std::chrono::duration<float,
std::chrono::seconds::period>(newTime - currentTime).count();
         currentTime = newTime;

         playerController.move(_Window.getWindow(), frameTime,
player.at(1)->object[0]);
         player2Controller.move2(_Window.getWindow(), frameTime,
player.at(2)->object[0]);

         if (collisionDetection(ball[0], player.at(1)->object[0]))
         {
             speed *= -1;
             yaw = findAngle(ball[0], player.at(1)->object[0]);
         }

         if (collisionDetection(ball[0], player.at(2)->object[0]))
         {
             speed *= -1;
             yaw = findAngle(ball[0], player.at(2)->object[0]);
         }

         pBorderDetectionY(player.at(1)->object[0]);
         pBorderDetectionY(player.at(2)->object[0]);

         if (player.at(1)->count == 3)
         {
             player.at(1)->resetCount();
             std::cout << "Player 1 win!" << std::endl;
         }
```

```
            if (player.at(2)->count == 3)
            {
                player.at(2)->resetCount();
                std::cout << "Player 2 win!" << std::endl;
            }

            if (borderDetectionY(ball[0]))
            {
                speed *= -1;
                yaw *= -1;
            }

            if (ball[0]->transform.translation.x > 1.f)
            {
                player.at(1)->addCount();
                std::cout << "Player 1: ";
                player.at(1)->print_count();
                std::cout << ", Player 2: ";
                player.at(2)->print_count();
                std::cout << std::endl;
                yaw = -1.5;
                resetBall();
            }

            if (ball[0]->transform.translation.x < -1.f)
            {
                player.at(2)->addCount();
                std::cout << "Player 1: ";
                player.at(1)->print_count();
                std::cout << ", Player 2: ";
                player.at(2)->print_count();
                std::cout << std::endl;
                yaw = 1.5;
                resetBall();
            }

            glm::vec2 forward_dir{ sin(yaw), cos(yaw) };
            ball[0]->transform.translation += forward_dir *
glm::vec2(speed) * frameTime;

            if (auto commandBuffer = _Renderer.beginFrame())
            {
                _Renderer.beginSwapChainRenderPass(commandBuffer);
                renderSystem.renderGameObjects(commandBuffer,
player.at(1)->object);
                renderSystem.renderGameObjects(commandBuffer,
player.at(2)->object);
                renderSystem.renderGameObjects(commandBuffer, ball);
                renderSystem.renderGameObjects(commandBuffer, border);
                _Renderer.endSwapChainRenderPass(commandBuffer);
                _Renderer.endFrame();
```

```
        }
    }

    vkDeviceWaitIdle(_Device.device());
}

void App::loadGameObjects()
{
    player.insert({ 1, new Player(new Primitive("Rectangle",
_Device)) });
    player.at(1)->object[0]->transform.translation.x = -.9f;
    player.at(1)->object[0]->transform.scale = { 0.05f, 0.3f };
    player.at(1)->object[0]->color = { 0.f, 0.f, 0.f };


    player.insert({ 2, new Player(new Primitive("Rectangle",
_Device)) });
    player.at(2)->object[0]->transform.translation.x = .9f;
    player.at(2)->object[0]->transform.scale = { 0.05f, 0.3f };
    player.at(2)->object[0]->color = { 0.f, 0.f, 0.f };

    ball.push_back(std::move(new Primitive("Circle", _Device)));
    ball[0]->transform.scale = { 0.03f, 0.03f };
    ball[0]->color = { 0.f, 0.f, 0.f };

    for (int i = 0; i < 21; i++)
    {
        border.push_back(std::move(new Primitive("Rectangle",
_Device)));
    }
    for (int i = 0; i < border.size(); i++)
    {
        border[i]->transform.translation.y = - 1.f;
        border[i]->transform.scale = { 0.01f, 0.04f };
        border[i]->transform.translation.y += 0.1f * i;
        border[i]->color = {0.f, 0.f, 0.f};
    }
}

void App::resetBall()
{
    ball[0]->transform.translation = { 0, 0 };
}

bool App::collisionDetection(GameObject*& circle, GameObject*& rect)
{
    float testx = circle->transform.translation.x;
    float testy = circle->transform.translation.y;

    if (circle->transform.translation.x < rect-
>transform.translation.x)
```

```
     {
         testx = rect->transform.translation.x;
     }
     else if (circle->transform.translation.x > rect-
>transform.translation.x + rect->transform.scale.x/2)
     {
         testx = rect->transform.translation.x + rect-
>transform.scale.x / 2;
     }

     if (circle->transform.translation.y < rect-
>transform.translation.y - rect->transform.scale.y / 2)
     {
         testy = rect->transform.translation.y;
     }
     else if (circle->transform.translation.y > rect-
>transform.translation.y + rect->transform.scale.y/2)
     {
         testy = rect->transform.translation.y + rect-
>transform.scale.y / 2;
     }

     float distancex = circle->transform.translation.x - testx;
     float distancey = circle->transform.translation.y - testy;

     float distance = sqrt(distancex * distancex + distancey *
distancey);

     if (distance - 0.02 < circle->transform.scale.x / 2 || distance -
0.02 < circle->transform.scale.y / 2)
     {
         return true;
     }
     return false;
 }

 bool App::borderDetectionY(GameObject*& o)
 {
     if (o->transform.translation.y + o->transform.scale.y / 2 > 1.f
|| o->transform.translation.y - o->transform.scale.y < -1.f)
     {
         return true;
     }
     return false;
 }

 void App::pBorderDetectionY(GameObject*& o)
 {
     if (o->transform.translation.y + o->transform.scale.y / 2 > 1.f)
     {
         o->transform.translation.y = 1.f - o->transform.scale.y / 2;
```

```
    }
    if (o->transform.translation.y - o->transform.scale.y / 2 < -1.f)
    {
        o->transform.translation.y = -1.f + o->transform.scale.y / 2;
    }
}

float App::findAngle(GameObject*& circle, GameObject*& rect)
{
    float relativeIntersectY = (rect->transform.translation.y +
(rect->transform.scale.y / 2)) - circle->transform.translation.y;
    float normalizedRelativeIntersectionY = (relativeIntersectY /
(rect->transform.scale.y / 2));;
    float bounceAngle = normalizedRelativeIntersectionY * 2;
    return bounceAngle;

}

class  __declspec(dllexport) KeyboardController
{
public:
        struct Keys1
        {
                int move_left = GLFW_KEY_A;
                int move_right = GLFW_KEY_D;
                int move_forward = GLFW_KEY_W;
                int move_back = GLFW_KEY_S;
        };

        struct Keys2
        {
                int move_left = GLFW_KEY_LEFT;
                int move_right = GLFW_KEY_RIGHT;
                int move_forward = GLFW_KEY_UP;
                int move_back = GLFW_KEY_DOWN;
        };

        void move(GLFWwindow* window, float dt, GameObject*
gameObject);
        void move2(GLFWwindow* window, float dt, GameObject*
gameObject);

        Keys1 keys1{};
        Keys2 keys2{};
        float move_speed = 2.0f;

};
```