Оглавление

[ВВЕДЕНИЕ 3](#_Toc192199851)

[ПОСТАНОВКА ЗАДАЧИ 5](#_Toc192199852)

# ВВЕДЕНИЕ

Игровая индустрия — постоянно развивающаяся и растущая область развлечений. На текущий момент объем рынка видеоигр и мультимедиа превзошли индустрии музыки и даже кино. Более того, видеоигры официально признаны произведениями искусства. Согласно данным цифровых сервисов дистрибуции, каждый день выходит минимум 40 проектов, и эта цифра постоянно растет.

Инструменты для создания игр претерпели множество изменений с момента образования индустрии. В момент зарождения индустрии игры разрабатывались с нуля под конкретное аппаратное обеспечение что заметно ограничивало скорость разработки, позже появились игровые движки, инкапсулирующие и разделяющие в себе множество подсистем позволяющих быстро разрабатывать под разные платформы и устройства. Так же с появлением графических API возможности игровых движков заметно выросли.

За счёт графического API можно получить доступ к ресурсам видеокарты для решения графических задач, и задач требующих параллельных вычислений. Такими задачами является создание визуальных эффектов, управление шейдерами, текстурирование и т.д.

До 2016 года на рынке графических API существовали два основных игрока: DirectX (для устройств Microsoft) и OpenGL (кроссплатформенный API до недавнего времени считавшийся стандартом индустрии). Но на данный момент тенденция в разработке графических API сводиться к более низкому уровню. Таким образом появился низкоуровневый кроссплатформенный графический API Vulkan, сместивший в своей универсальности OpenGL, выпущенный консорциумом Khronos Group. Использование в игровом движке графического API Vulkan приведет к большему контролю над ресурсами видеопроцессора и, следовательно, позволяет повысит производительность классических игровых сценариев.

Таким образом, тема выпускной квалификационной работы:

Объектом выпускной квалификационной работы является разработка игрового движка.

Предмет выпускной квалификационной работы – разработка игрового движка с использованием графических API.

Цель выпускной квалификационной работы – разработка эффективного и расширяемого инструмента для разработки видеоигр.

В соответствии с поставленной целью необходимо решить следующие задачи:

1. Анализ современного состояния проблемы разработки игровых движков.
2. Осуществить постановку задачи на разработку игрового движка «LampyEngine» с использованием API Vulkan.
3. Осуществить выбор и обоснования методов и средств разработки игрового движка «LampyEngine».
4. Разработать проектные решения для игрового движка «LampyEngine».
5. Реализовать и протестировать игровой движок «LampyEngine».
6. Рассчитать экономические затраты на разработки игрового движка «LampyEngine».

Методы и средства разработки

Практическая значимость заключается в разработанном игровом движке, отвечающем всем базовым требованиям к подобным инструментам.

На защиту выносятся:

1. Техническое задание на разработку игрового движка «LampyEngine».
2. Игровой движок «LampyEngine».
3. Тест-кейсы для проверки корректности работы игрового движка «LampyEngine».
4. Расчет экономических затрат на разработку игрового движка «LampyEngine».

# ПОСТАНОВКА ЗАДАЧИ

Разрабатываемый игровой движок будет:

1. Кроссплатформенным (Поддерживать как работу на Linux, так и на Windows системах)
2. Поддерживать написание скриптов на языке Lua
3. Иметь несколько графических API (OpenGL и Vulkan) для совместимости с большинством устройств
4. Базироваться на базе ECS системы для достижения максимальной производительности
5. Иметь модуль физики с интеграцией через ECS.
6. Поддерживать расширения с помощью встраивания сторонних модулей
7. Иметь открытый исходный код.

Разрабатываемый игровой движок не будет:

1. Поддерживать сборку проекта на консоли или мобильные устройства.
2. Иметь технологии для удаленной разработки.
3. Поддерживать разработку многопользовательских игр.
4. Поддерживать редактирование BIM моделей.
5. Иметь сборку под мобильные приложения.
6. Иметь поддержку проприетарных графических API таких как Direct3D(Windows и XBox) и Metal(MacOS).

Бизнес-цели:

1. Создать высокопроизводительный движок для разработки мультимедийных приложений который поддерживает разработку и сборку проектов на системах с Windows или Linux.
2. Реализовать поддержку движком Vulkan, OpenGL позволяя разработчикам выбирать наиболее подходящее API.
3. Предоставить инструменты для упрощенной интеграции скриптов, редактора сцен и физических симуляций.

Критерии успеха:

1. Расширение аудитории потенциальных разработчиков и пользователей.
2. Увеличение гибкости движка и его инструментов.
3. Сокращение времени разработки игр, что сделает движок привлекательным для инди-разработчиков и небольших студий.

В рамках проекта игрового движка предполагается существование двух функциональных ролей: разработчик игр, разработчик движка.

Разработчик игр использует конечное приложение как инструмент для создания и отладки игровых проектов. Он взаимодействует с разными инструментами и системами движка, включая редактор сцен и систему скриптов.

Возможности разработчика игр:

* Создание сцен через специфичный редактор.
* Написание игровой и системной логики на Lua
* Работа с ECS для организации объектов
* Использование встроенной физической модели.
* Подключение расширений и сторонних библиотек.
* Сборка и тестирование игрового проекта на Windows и Linux.
* Импорт ресурсов в виде 3D-моделей текстур, звуков в поддерживаемых форматах.

Разработчик движка занимается улучшением функционала, оптимизацией работы рендеринга физики, ECS, системы ресурсов и других внутренних механизмов движка.

Возможности разработчика движка:

* Разработка и оптимизация графического API (OpenGL/Vulkan)
* Расширение и поддержка ECS-архитектуры.
* Интеграция и поддержка Lua-скриптинга включая добавления новых взаимодействий с ядром или модулей движка.
* Разработка новых редакторов и инструментов.
* Поддержка кроссплатформенности (Windows/Linux).
* Ведение документации и примеров использования.

Основные функции представлены в виде диаграммы прецедентов на рисунке 1.

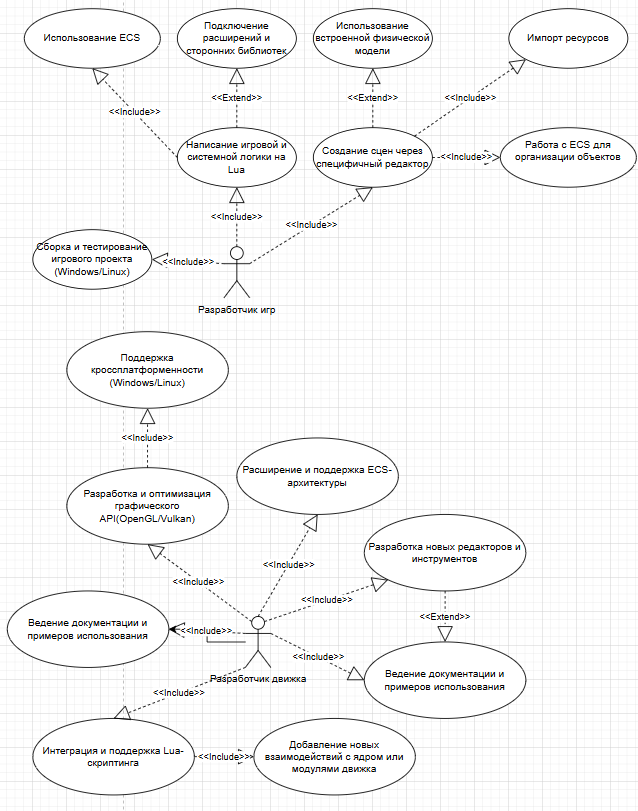
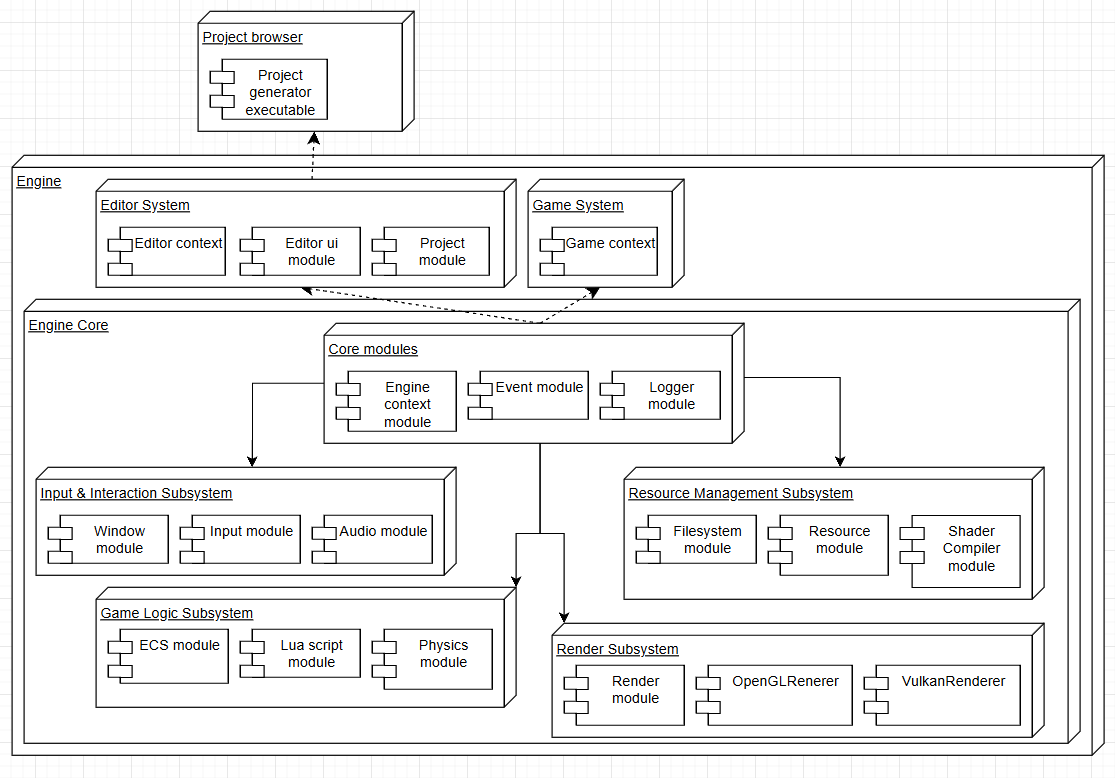


Рисунок 1 – Диаграмма прецендентов игрового движка.

**Проектные решения**

**Архитектура движка**

Исходя из функциональных требований к движку была организована модульная структура, разделяющая обязанности между различными системами.  
 Рассмотрение архитектуры проекта стоит начать от общего к частному ради достижения полной ясности и прозрачности его внутреннего устройства. Диаграмма компонентов архитектуры приложения представлена на рисунке 2.



*Рисунок 2 – Диаграмма компонентов игрового движка*

Верхний срез архитектуры состоит из нескольких ключевых блоков:

1. Engine – главный исполняемый файл движка использующий все основные системы и модули.
2. Engine context – абстракция для создания определенных конфигурируемых режимов запуска движка. Нужный контекст создается при запуске приложения и конструируется на основе параметров запуска.
3. Editor system – система-надстройка над основным решением расширяющая основной функционал движка до полноценного редактора.
4. Project browser – отдельный исполняемый файл выполняющий задачу создания и конфигурирования проектов.
5. Game system – аналогичная Editor system настройка которая в свою очередь модифицирует основное решение до запуска определенного проекта в режиме конечной сборки.

Порядок запуска объектов верхнего уровня изображен на диаграмме последовательности.

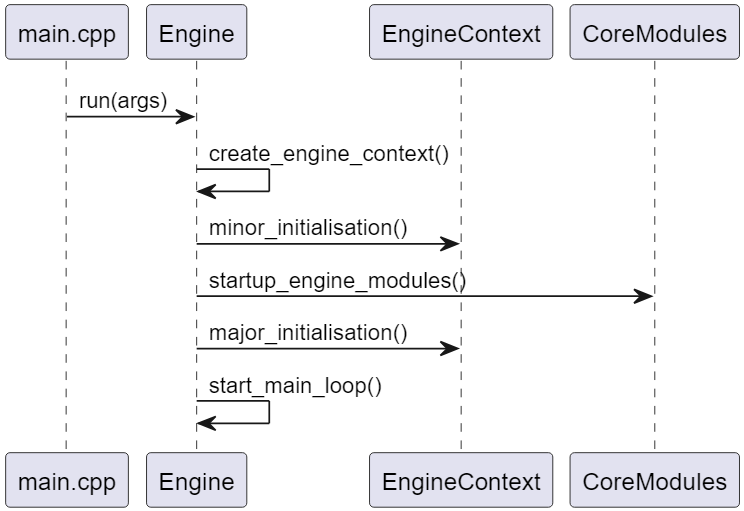


Рисунок 3 – Диаграмма последовательности запуска движка.

На рисунке 3 явно виден процесс запуска движка, где от входной точки исполнения программы поток поднимается к основному классу Engine который инициализирует контекст движка и модули ядра (Core Modules – на диаграмме последовательности запуска это обобщение над всеми основными системами движка).

Для полного понимания архитектуры движка следует рассмотреть блоки по отдельности.

**Рассмотрение ядра движка**

Ядро движка (Engine core) служит связующим звеном для всех систем и модулей которое организует корректную последовательность инициализации подсистем приложения.

Корневые модули ядра (Core modules), организующие внутреннюю логику движка:

* Модуль контекста (Engine context module) является абстракцией для создания конфигурируемых режимов запуска. В зависимости от параметров, заданных при запуске, движок создает нужный контекст и благодаря этому способен функционировать как редактор или исполнятся как игра. Это позволяет системе сохранять модульность и модифицируемость.
* Модуль событий (Event module) основной инструмент для передачи данных между системами и модулями, он содержит в себе логику для создания различных событий, изменения передаваемых параметров и соблюдения потокобезопасности. Необходимость создания собственной реализации системы событий обусловлена контекстом языка C++ и в том числе требованием регистрации собственных событий в Lua-скриптах.
* Модуль логирования (Logger module) модуль, без которого не может существовать ни одна комплексная программа. Этот модуль предоставляет интерфейс для регистрации и сохранения сообщений во внутреннюю консоль и до записи в отдельный log файл. Модуль так же предоставляет различные категории логирования такие как verbose, debug, warning, error и fatal.

Теперь можем перейти к рассмотрению частных систем, включенных в ядро движка (Engine core), реализующих различный функционал.

Для обработки ввода и инициализации контекста взаимодействия с операционной системой существует система «Input & Interaction system» которая объединяет в себе модули окна, ввода и аудио. Эти модули напрямую взаимодействуют с целевой платформой для получения доступа к созданию окон, считывания ввода с клавиатуры, мыши или других пользовательских устройств. В искомую систему так же вынесен модуль для работы со звуками, хотя такая группировка не совсем очевидна, но в подробном рассмотрении обнаруживается, что популярные аудиобиблиотеки диктуют необходимость получения устройств воспроизведения, и в некоторых частных случаях управления параметрами конкретного аудиоустройства.

Поскольку в движке подразумевается поддержка множества API рендеринга и взаимодействия с видеокартой, подсистема рендеринга вынесена в обособленный модуль, в котором OpenGL и Vulkan реализуют и расширяют интерфейс базового класса. Так же вынесение в отдельную подсистему (Render Subsystem) было продиктовано современными требованиями к организации подобных решений.

Для создания логики работы с ресурсами движка организована подсистема управления ресурсами (Resource Management Subsystem). В данную систему включены модули для работы с файловой системой (Filesystem module), ресурсами движка (Resource module) и компилятор шейдеров (Shader compiler module). Собственная реализация методов для работы с файловой системой требуется для обеспечения кроссплатформенности и ясности. Модуль ресурсов служит публичным интерфейсом для управления ресурсами. Модуль компиляции шейдеров использует программы-компиляторы для преобразования шейдеров с языка GLSL в промежуточный язык для параллельных вычислений SPIR-V.

За игровую логику, описание объектной модели движка, а также поддержку рефлексивного поведения отвечает подсистема игровой логики (Game Logic Subsystem), объединяющая в себе модуль ECS (Entity component system), скриптов (Lua script module) и физики (Physics module). Объектная модель (ECS) регламентирует использование компонентов и сущностей мира, следствием чего является соответствующая компоновка модулей физики и скриптов.

Таким образом, ядро движка (Engine Core) собирает в себе весь функционал, где каждая подсистема предоставляет публичный интерфейс для использования в разных контекстах и формах, таких как редактор или игра.

**Архитектура систем редактора и игры**

Сам по себе движок не являет собой полезную программу или конечное решение, а лишь предоставляет инструменты для взаимодействия с миром игры, ресурсами, вводом и т.д. Основную полезную работу с помощью интерфейсов движка выполняют контексты-расширения, которые включаются в инициализацию и процесс исполнения программы как отдельные абстрактные объекты. Рассмотрим функциональные расширения в виде системы редактора (Editor System) и игры (Game System).

Система редактора (Editor System) – контекст реализующий интерактивный инструмент, использующий корневой функционал движка. Редактор содержит два ключевых модуля:

1. Проектный модуль (Project module) – реализует функционал хранения основной информации о проекте и его структуре. Для создания или открытия проектов используется отдельное приложение – проектный браузер (Project browser).
2. Модуль редакторского интерфейса (Editor UI module) предлагает набор инструментов для взаимодействия с функциями движка. Среди основных инструментов можно выделить: инспектор сцены, настройки сущностей и файловый менеджер.

Система игры (Game system) – включает в себя исключительно процесс запуска конечного проекта созданного в редакторе с учетом расположения ресурсов и файлов проекта в упакованном виде.

**Структура проектов на игровом движка**

Проекты движка и их структуру описывает система редактора и проектный модуль. При активации проектного модуля запрашивается приложение браузера проектов (рисунок 4).

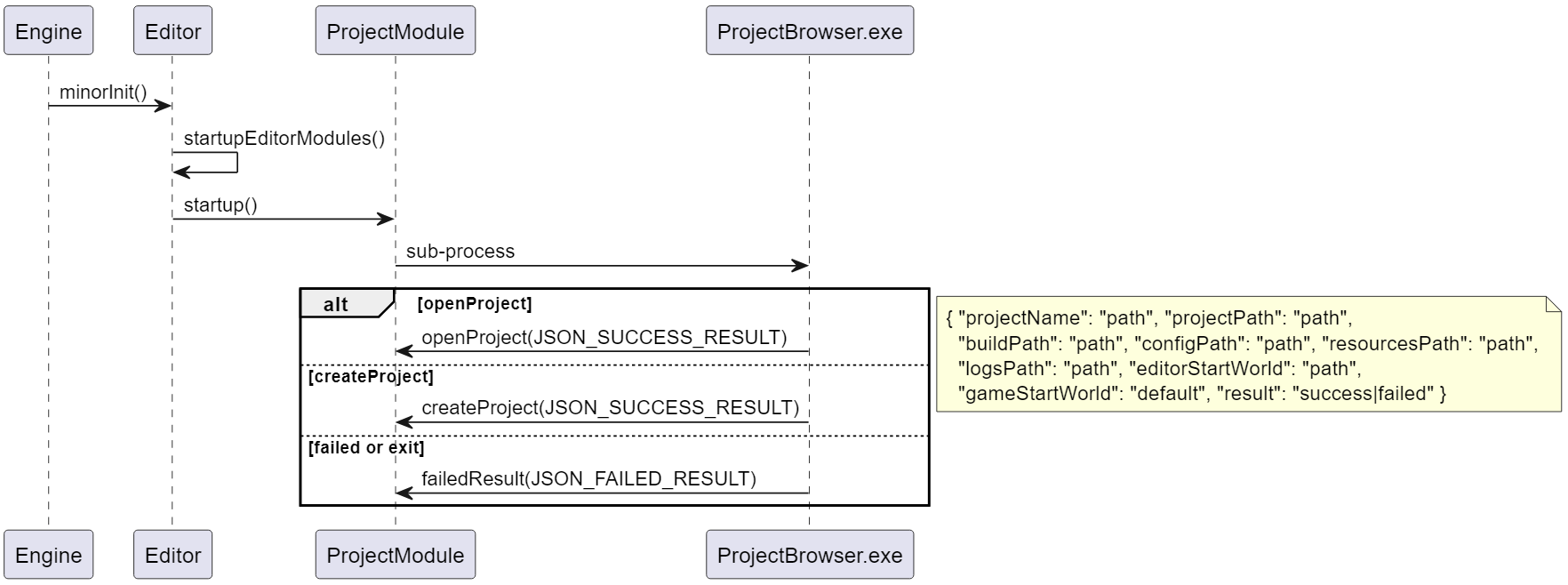


Рисунок 4 – Диаграмма последовательности выбора и создания проекта.

На диаграмме видно, что запуск браузера проекта происходит во время «минорной» инициализации контекста-редактора, и как результат он возвращает JSON структуру проекта. В JSON описаны служебные значения, относительные пути до ключевых директорий проекта таких как папки ресурсов, сборки, конфигурации и логов. Так же стоит отметить, что информация о текущих мирах движка и игры упраздняет необходимость вручную реализовывать их поиск.

**Объектная модель движка**

В процессе проектирования игрового движка одним из самых щепетильных вопросов для разработчиков является выбор объектной модели движка, от которой будет зависеть структура будущих проектов и архитектуры в целом. Существует несколько популярных объектных моделей:

1. Классическая объектно-ориентированная модель диктует жесткую иерархию объектов, исполняющих как задачу хранения данных, так и исполнения внутренней логики. Этот подход интуитивен и прост в освоении позволяя программистам быстрее вникать в архитектуру того или иного проекта, но и накладывает ограничения по масштабированию и общей гибкости.
2. Современная компонентная модель, завоевавшая свою популярность вместе с движками Unity и Godot, дает возможность собирать объекты, используя набор независимых компонентов. Компоненты модели отвечают за отдельный аспект поведения или состояния. Несмотря на возможное возникновения путаницы в связях между компонентами, этот подход дает больше контроля над структурой проекта.
3. ECS (Entity Component System) – сравнительно молодая объектная система, задачи которой разделены между тремя понятиями, где сущность (Entity) удерживает компоненты-данные (Components), обрабатываемые системами (Systems). Такой подход позволяет легко отделить логику от параметров, облегчая процесс компоновки объектов и сериализацию.
4. Прототипно-ориентированные модели дают возможность собирать варианты объектов, так как сущности создаются путем копирования и расширения объекта-прототипа. Так можно легко создавать вариации, но это может ухудшать читаемость архитектуры.

C учетом всех преимуществ по производительности, гибкости и масштабируемости была выбрана модель ECS. Однако выбор конкретной реализации ECS играет немаловажную роль, так как требуется не только эффективность, но и удобство разработки. Среди существующих решений, таких как EnTT, specs, Bevy ECS и других, особенно удачным представляется Flecs. Flecs имеет лаконичную структуру и демонстрирует высокую производительность. Стоит отметить, что Flecs написана на C и предоставляет высокоуровневый интерфейс для C++, что делает ее оптимальной для проекта.

Теперь следует рассмотреть, каким образом Flecs ECS интегрирована в движок. Обратим внимание на модуль ECS из подсистемы игровой логики (Рисунок 4).

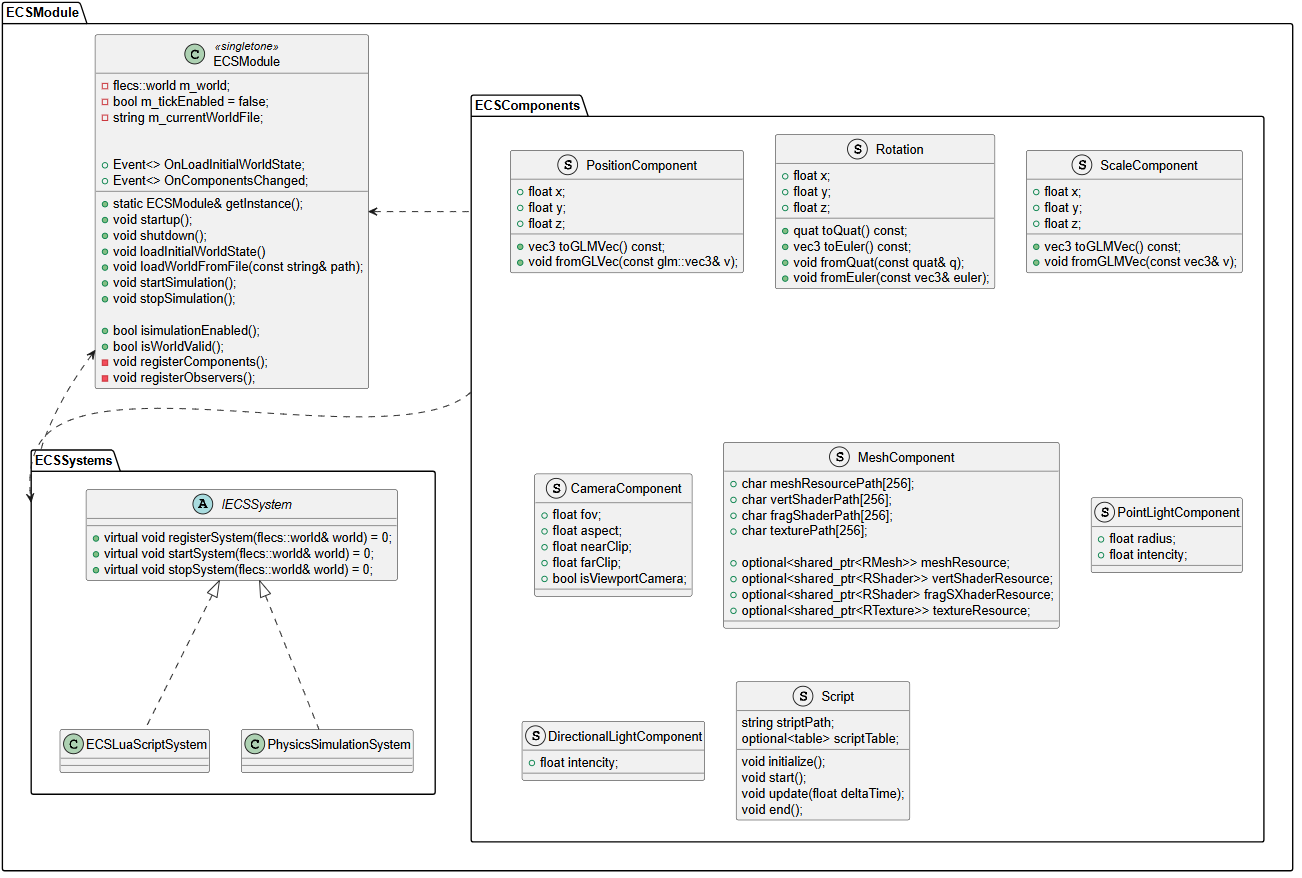


Рисунок 5 – Диаграмма классов модуля ECS

Модуль инкапсулирует в себе работу с системой ECS. Мир содержится в ECSModule классе. Компоненты регистрируются как структуры содержащие простые типы (POD – Plain old data). Для организации логики компонентов, в базовой реализации, основные системы обобщены интерфейсом IECSSystem:

* ECSLuaScriptSystem – обрабатывает логику Lua-скриптов (Script – структура-компонент, содержащая путь до файла скрипта), вызывает основные события старта, обновления и завершения симуляции. Эта система напрямую работает с модулем Lua-скриптов.
* PhysicsSimulationSystem – глубоко интегрированная в модуль физики (Bullet) система. Она регистрирует компоненты статичных или динамичных тел в физическом мире и напрямую взаимодействует с соответствующими объектами.

В отдельном блоке диаграммы определен пакет ECSComponents, в нем расположены базовые компоненты трансформации, света, сеток и скриптов.

В процессе разработки игрового проекта инструментарий редактора (функции движка и Lua-расширения), позволяет расширять ECS системы и компоненты.

**Интеграция Lua**

При разработке подобных инструментов, важную роль выполняет система рефлексии – механизм, позволяющий программе анализировать и изменять собственную структуру и поведение во время выполнения. Это полезно для тяжеловесных приложений, требующих перекомпиляции. В игровых движках рефлексия дает возможность управлять сущностями, параметрами компонентов, а также расширять логику игровых сценариев.

Исходя из существующих решений для организации рефлексивного поведения, каждое из которых имеет свои преимущества, представим наиболее распространённые из них в сравнительной таблице 1.

|  |  |  |  |
| --- | --- | --- | --- |
| Подход | Примеры | Преимущества | Ограничения |
| 1. Статическая генерация кода | EnTT meta, macros, codegen | Высокая производительность, прямой контроль | Требует генерации и пересборки, сложно масштабировать |
| 2. RTTI и typeid | typeid, std::any, кастомные реестры | Использует стандартные возможности языка C++ | Ограничено типами, плохо интегрируется с внешними системами |
| 3. Встроенная мета-система | Unreal Engine (UCLASS, UFUNCTION) | Гибкая, мощная, поддерживает инструменты редактора | Высокий порог входа, сложно реализовать с нуля |
| 4. Интеграция скриптового языка | Lua, Python, Squirrel | Изменение поведения на лету, максимальная гибкость | Дополнительный интерпретатор, требуется интеграция API |

Основываясь на масштабах проекта и характеристик этих решений, выбор пал на использование скриптового языка Lua (библиотека sol2). Обусловлено это простотой интеграции с C/C++ через стандартный API и компактности ядра Lua.

Перейдем к рассмотрению интеграции Lua в движок. Представление модуля Lua (рисунок 6) состоит всего из двух компонентов: интерпретатора в RAII обертке и публичного singleton класса.

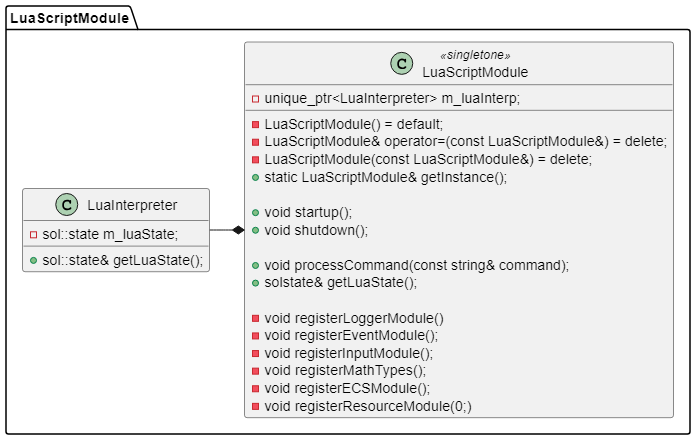
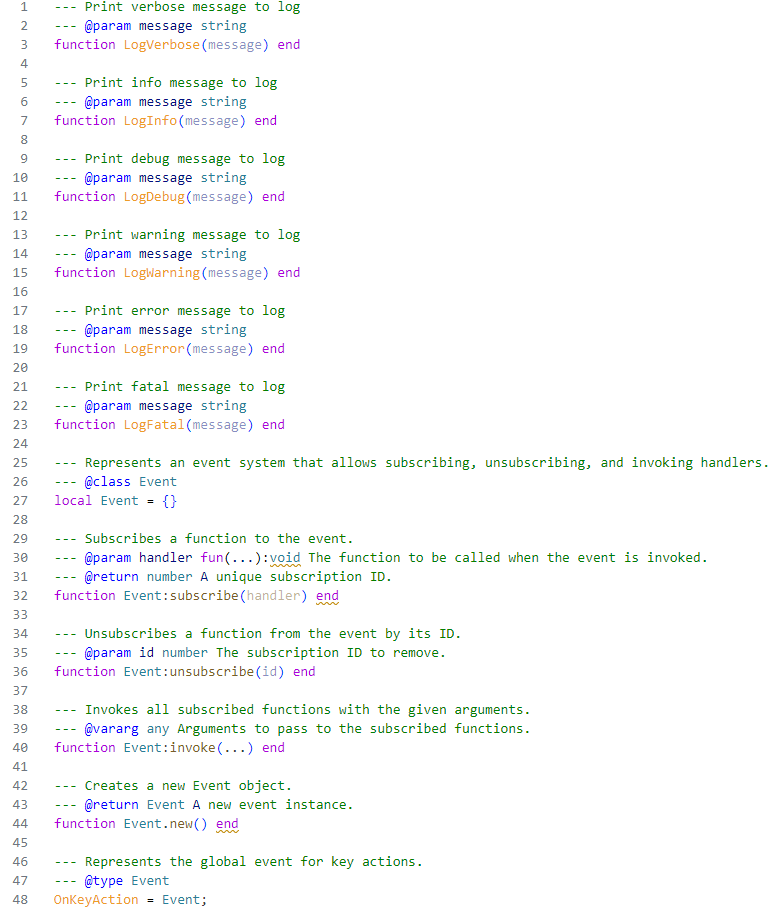
****

Рисунок 6 – Диаграмма классов модуля Lua

В момент инициализации LuaScriptModule создает объект интерпретатора и регистрирует методы основных подсистем и модулей. Под конец инициализации сущность интерпретатора будет иметь все основные функции и методы с привязками к их полной реализации в c++.

Но просто пробросить компоненты движка в среду Lua недостаточно, поскольку в реальной разработке программист игровой логики должен легко ориентироваться в функциях и методах предоставляемых движком. Для решения подобной проблемы необходимо грамотно подойти к написанию документации и что не мало важно предоставить автоматически генерируемый API в виде большего файла lua подключаемого к игровому проекту.

После регистрации типов и методов движка LuaScriptModule создает отдельный объект engine\_api.lua представляющий собой список объявлений с документацией. Фрагмент кода генерируемого API предоставлен на рисунке 6



*Рисунок 7 – Диаграмма классов верхнего уровня рендеринга*

Из фрагмента видны задокументированные сущности корневой подсистемы движка (модуль логирования и модуль событий). Приведена документация в виде комментариев к каждому элементу кода. Таким образом разработчику достаточно подключить файл API движка, чтобы начать комфортно работать над игровыми проектами.

**Рендеринг и используемые API**

В контексте нашего проекта подсистема рендеринга поддерживает отрисовку в двух API (OpenGL и Vulkan). Такое решение было принято с опорой на структуру игрового движка Godot, который следует использованию разных API для специфичных задач, где OpenGL (режим Compatibility) отвечает за совместимость с большинством устройств, а Vulkan (режим Forward+) за получение максимальной производительности и контроля над рендерингом.

Разделение между несколькими API требует организации грамотной прослойки между движком требующим отрисовку интерфейса или мира игры и конкретной реализацией рендеринга. Основной сложностью разделения OpenGL и Vulkan является разная философия этих графических интерфейсов. OpenGL является высокоуровневой и во многих аспектах абстрактной системой, а Vulkan ближе к низкому уровню и требует практически полного контроля над основными процессами рендеринга.

Стоит ненадолго заострить внимание на работе с шейдерами в OpenGL и Vulkan, поскольку OpenGL использует GLSL формат, проводя шейдеры через этап фронтенда (компиляции на лету), а Vulkan перекомпилированный SPIR-V. Эта проблема решается использованием расширений OpenGL (например, GL\_ARB\_gl\_spirv) позволяет использовать формат SPIR-V без передачи GLSL кода, что позволит компилировать шейдеры перед запуском конечного приложения. Однако такой подход требует поддержки со стороны драйвера и код SPIR\_V должен быть совместим с OpenGL.

В текущей реализации игрового движка было принято решение построить прослойку рендеринга (IRenderer на рисунке 6) на основе абстракции объектов используемых для отрисовки на низком уровне, то есть верхний слой рендеринга повторяет структуру низкого уровня схожей с Vulkan. В такой структуре с более высокоуровневыми API, такими как OpenGL, достаточно следовать инструкциям верхнего уровня для корректной отрисовки.

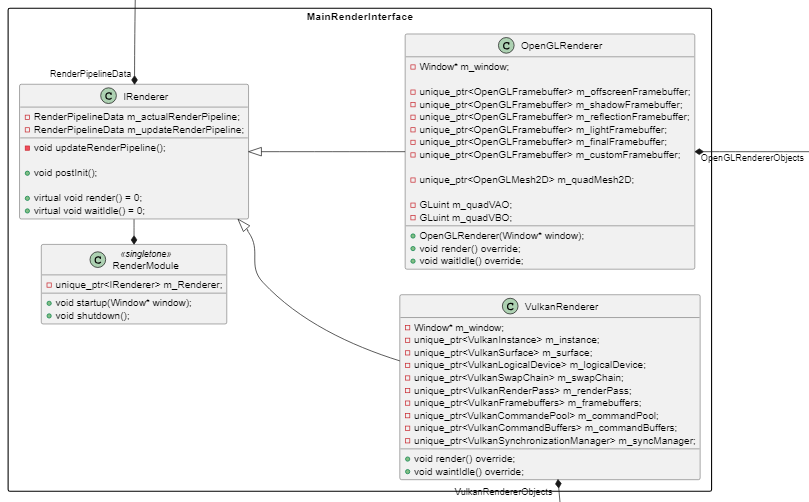


Рисунок 6 – Диаграмма классов верхнего уровня рендеринга.

Диаграмма иллюстрирует главные объекты подсистемы рендеринга, в ней публичным интерфейсом выступает singleton класс RenderModule, который содержит базовый класс-прослойку IRenderer. IRenderer в свою очередь может быть расширен до конкретной реализации OpenGL (OpenGLRenderer) и Vulkan (VulkanRenderer)

Следом стоит рассмотреть процесс передачи визуального состояния из логики движка в подсистему рендеринга для того, чтобы узнать, как актуализируется информация и отслеживаются изменения.

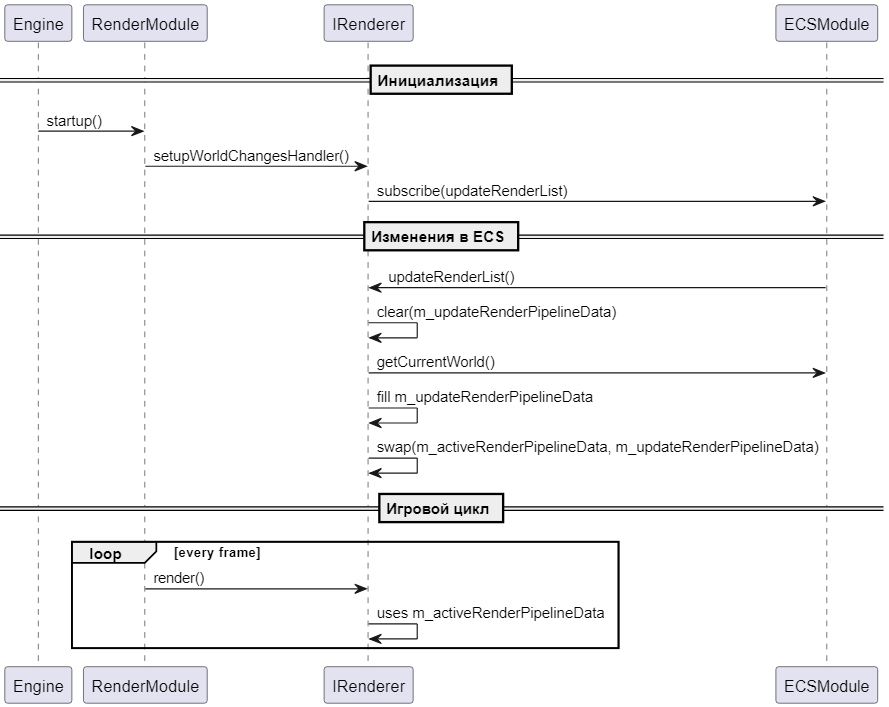


Рисунок 7 – Диаграмма последовательности запуска рендеринга и обработки состояния сцены.

На диаграмме последовательности (рисунок 7) отображен процесс синхронизации логики и рендера. Явно видно, что при инициализации рендеринга обобщенный класс IRenderer подписывается на изменения мира ECS. Когда состояние мира изменяется, происходит вызов метода, который подменяет текущее состояние прослойки (m\_activeRenderPipelineData), используемое в конкретной реализации рендерера, на актуальное (m\_updateRenderPipelineData).   
 Для предметного понимания того, что генерирует прослойка, разберем диаграммы классов представленных на рисунках 8 и 9.

На первой диаграмме отображены объекты, требуемые для рендеринга. Текстуры, шейдеры, сетки и материалы. Они являются абстрактными классами, где под конкретные API есть свои наследники, реализующие специфичную логику для их работы. Так же эти сущности предоставляют публичный интерфейс для работы на верхнем уровне. Каждый из этих объектов отражает структуру и состав проходов рендеринга и может быть создан с помощью фабричных классов генераторов, которые используют объекты-ресурсы и на их основе конструируют соответствующие объекты-рендера в зависимости от выбранного API.

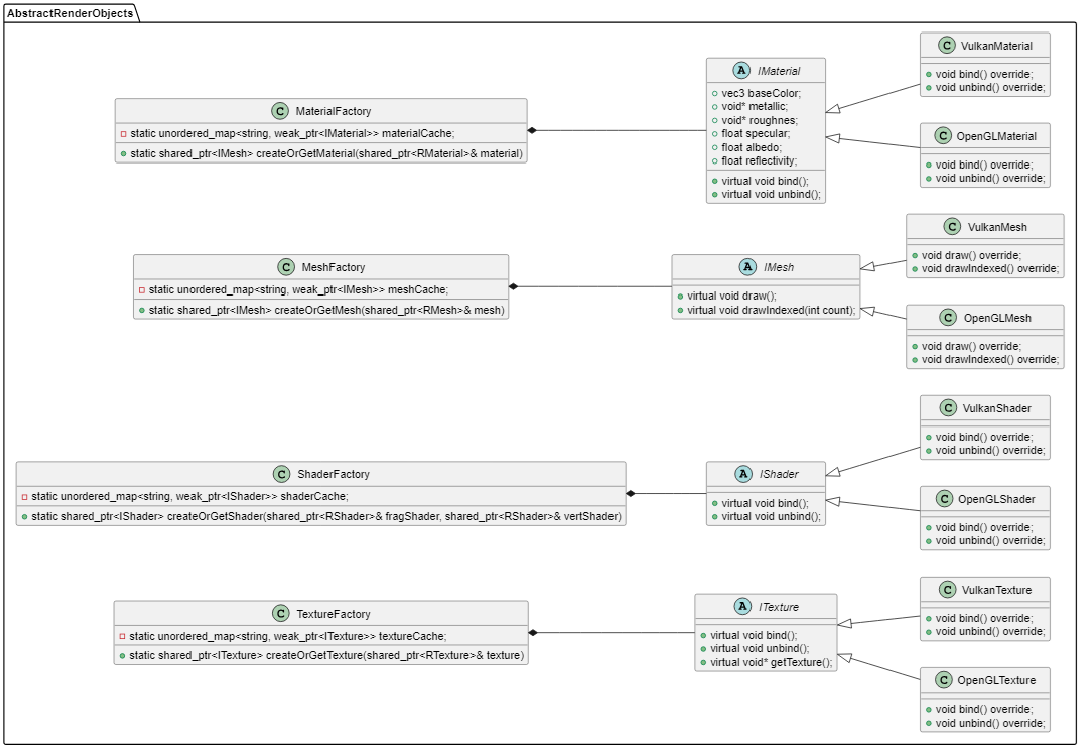


Рисунок 8 – Диаграмма классов обобщенных объектов рендеринга.

На следующей диаграмме отображены объекты прослойки. Видно, что единицей обобщения для рендеринга является структура RenderObject описывающая один вызов отрисовки и содержит в себе модельную матрицу и опциональные настройки отображения. Схожие со структурой Vulkan проходы (RenderPassData) содержат объекты для отрисовки структурированные в виде «батчей» (пакетов, содержащих от 1 до 256 объектов с одинаковой сеткой и разными модельными матрицами) и тип прохода. Конвейер отрисовки (RenderPipelineData) собирает всю информацию для отрисовки одного кадра. В ней содержится 5 рендер проходов: тени, свет, отражения, финальный и пользовательский проходы. В том числе в конвейере содержатся важные для отрисовки по типу структур камеры и объектов освещения.

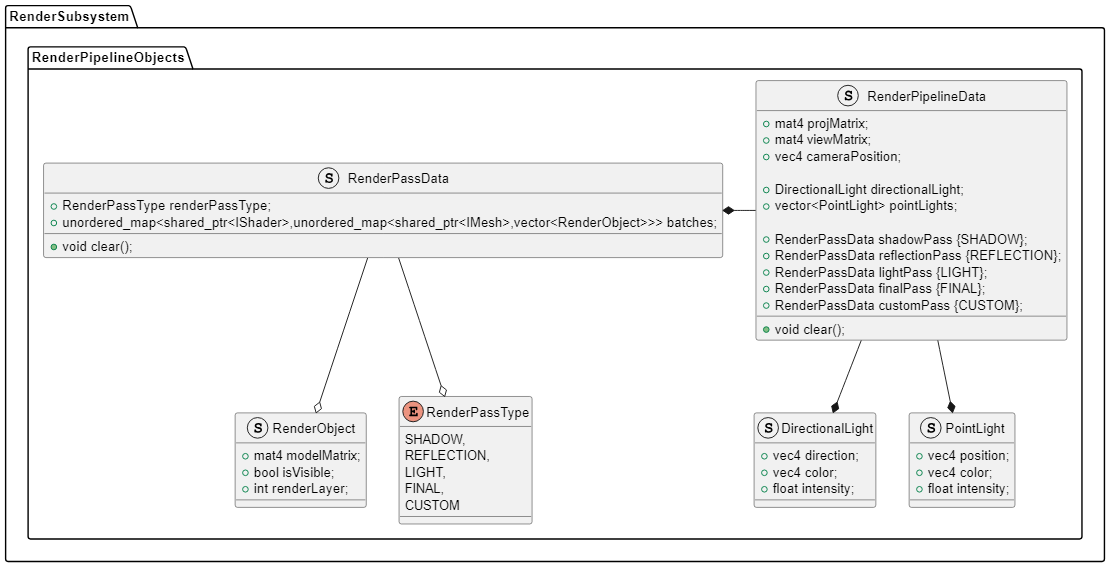


Рисунок 9 – Диаграмма классов данных рендеринга.

Теперь основываясь на информации, генерируемой на уровне прослойки и процессе ее подмена в момент изменения состояния мира, следует перейти к рассмотрению конкретных реализаций OpenGL и Vulkan.

В Vulkan особое внимание стоит уделить управлению объектами и этапам их инициализации. В базовой реализации Vulkan рендеринг в окно не подразумевается, и разработчикам требуются расширения, позволяющие использовать цепочку кадров (swap chain) и поверхность (surface). К тому же в некоторых случаях может потребоваться внеэкранный рендеринг (offscreen renderer), для отображения сцен в отдельных окнах редактора (viewports) или генерации текстур постобработки.

Несмотря на возможность в Vulkan использования объектов сразу и последовательной их инициализации друг за другом, было принято логичное решение, исходя из большего объема кода и актуальных практик разработки, как c использованием Vulkan, так и c++ в целом, сделать RAII обертки для основных объектов. На рисунке 10 отображены основные объекты рендеринга Vulkan, такие как экземпляр API (VulkanInstance), логическое устройство (LogicalDevice), поверхность отрисовки (VulkanSurface) и другие.

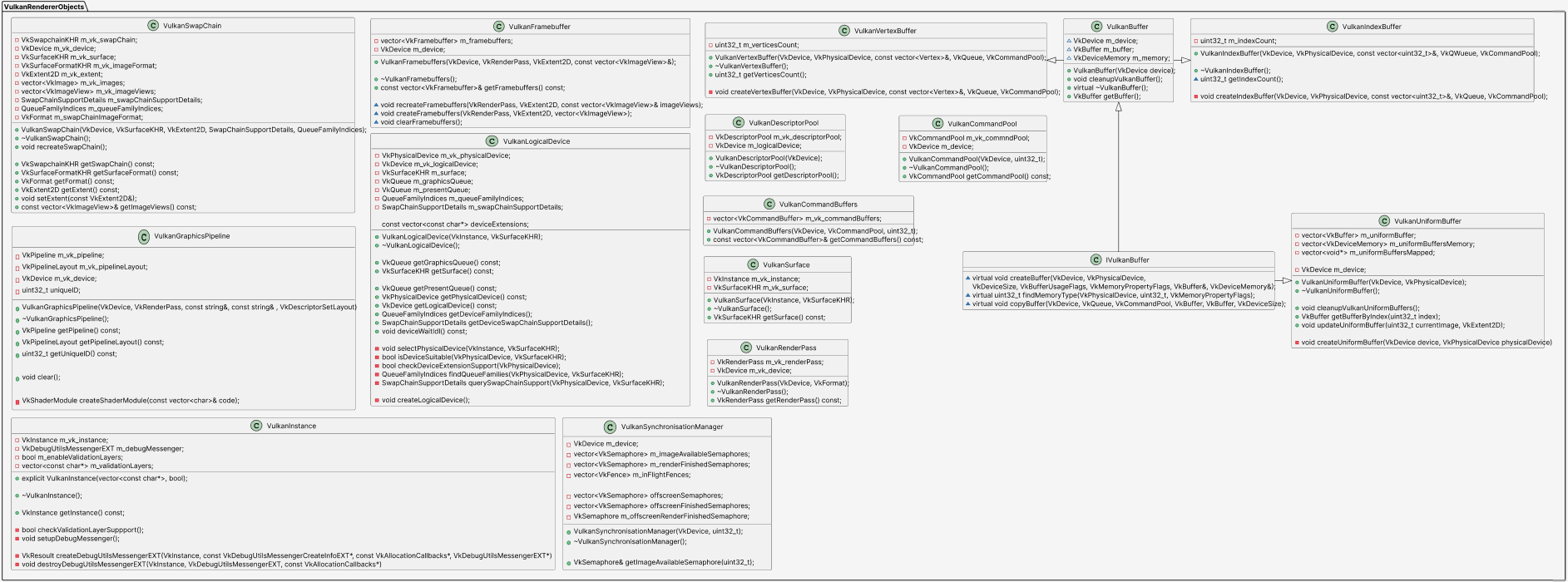


Рисунок 10 – Состав RAII-классов Vulkan.

Теперь рассмотрим последовательность инициализации объектов и их роль в рендеринге Vulkan (Рисунок 11).

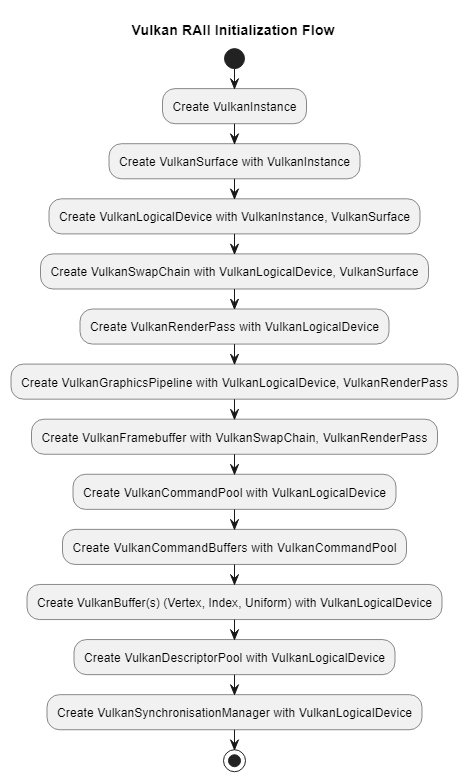


Рисунок 11 – Последовательность инициализации RAII объектов Vulkan.

Пройдем эти этапы по порядку:

1. VulkanInstance – контекст Vulkan API, с помощью которого создаются другие объекты и проверяется поддержка расширений.
2. VulkanSurface – предоставляет платформо-зависимую поверхность вывода (Win32, X11, Wayland) и выступает связующим звеном между рендерингом и оконной системой.
3. VulkanLogicalDevice – отвечает за выбор физического и создание логического устройства, через которое будут проходить команды для рендеринга.
4. VulkanSwapChain – цепочка изображений рендеринга. Количество изображений в цепочке ограничено устройством и может быть изменено через RenderConfig.
5. VulkanRenderPass – Описание структуры отрисовки, определяет какие этапы будут проходить в процессе рендера. Проходы могут быть созданы во множественном экземпляре с разными параметрами в зависимости от организации прослойки.
6. VulkanGraphicsPipline – Содержит шейдеры, входные значения вершин и описание их структуры. Объект может создаваться как на этапе инициализации, так и на этапе создания новых шейдеров.
7. VulkanFramebuffer – объект, описывающий набор целеив изображений (attachments) участвующих в процессе рендеринга, включая буферы цвета и глубины.
8. VulkanCommandPool / VulkanCommandBuffers – Выделяют память для командных буферов, где сами буферы хранят инструкции для исполнения на GPU.
9. VulkanBuffer (VertexBuffer, IndexBuffer, UniformBuffer) – Обертки, управляющие памятью GPU. Используются для передачи вершин, индексов, структур в шейдер, текстур и т.д.
10. VulkanDescriptorPool – Источник, генерирующий связи между шейдерами и uniform-буферами.
11. VulkanSynchronisationManager – Объект синхронизации, который обеспечивает правильную подачу и завершения команд между CPU и GPU.

Во время непосредственного рендеринга Vulkan следует инструкциям, созданным прослойкой, используя отдельные проходы (VulkanRenderPass) для освещения, теней и прочего. В процессе отрисовки обеспечивается передача шейдеров и вершин через VulkanGraphicsPipeline и VulkanVertexBuffer объекты.

Далее переходим к структуре OpenGL. Как было подмечено в начале главы, OpenGL поддерживает высокоуровневую архитектуру, которая легко подстраивается под данные, генерируемые на верхнем уровне. Аналогично организации RAII-оберток Vulkan определенны соответствующие классы, инкапсулирующие работу с буферами кадров, мешами, шейдерами, буферами и текстурами (Рисунок 12).

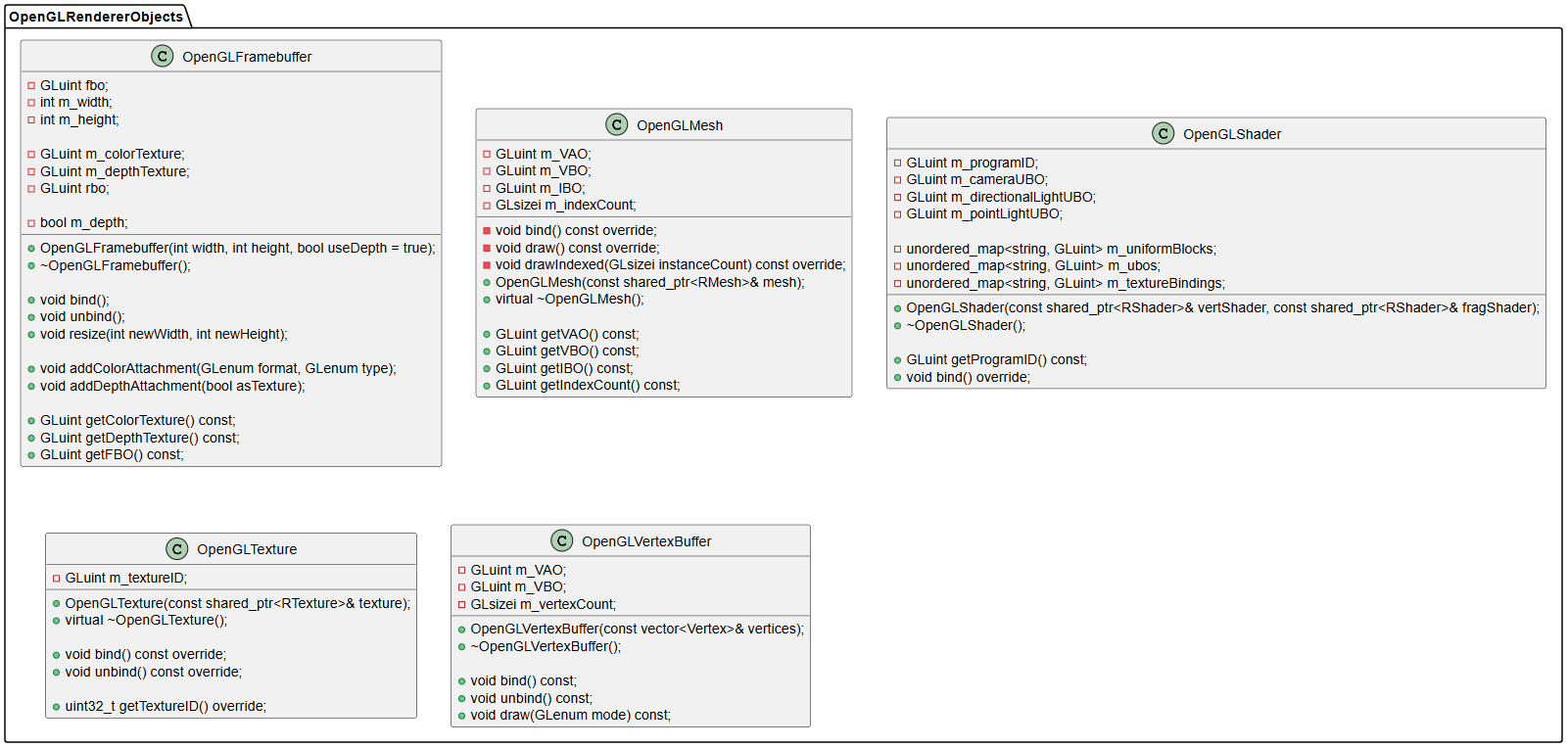


Рисунок 12 – Состав RAII-классов OpenGL.

Поскольку сам процесс отрисовки, порядок исполнения проходов и алгоритм передачи команд на отрисовку в Vulkan-реализации предметно рассмотрены не были, сделаем это в контексте обсуждения OpenGL-реализации.

На диаграмме последовательности (рисунок 13) иллюстрируются этапы обработки проходов рендеринга в OpenGL:

1. Очистка предыдущих результатов отрисовки;
2. Последовательная привязка фрейм-буферов, которые выполняют роль проходов;
   1. Запись прохода теней (ShadowPass);
   2. Запись прохода отражений (ReflectionPass);
   3. Запись прохода света (LightPass);
   4. Запись пользовательского прохода (CustomPass);
3. Отрисовка системных объектов и интерфейса редактора.
4. Завершение отрисовки и смена буферов.

Алгоритм обработки проходов использует передаваемые в метод renderPass пакеты (batches). Обработка структуры команд отрисовки отображена на рисунке 12. В цикле итерируется хеш-таблица с ключом-шейдером и значением в виде меша. Меши в свою очередь относятся к структуре RenderObject содержащей модельную матрицу. Такая организация позволяет рендерить объекты пакетами используя индексированную отрисовку (drawIndexed в OpenGL). В процессе перебора для каждого «батча» активируется шейдерная программа, заполняются uniform-структуры, включая данные трансформации, и вызывается команда отрисовки.

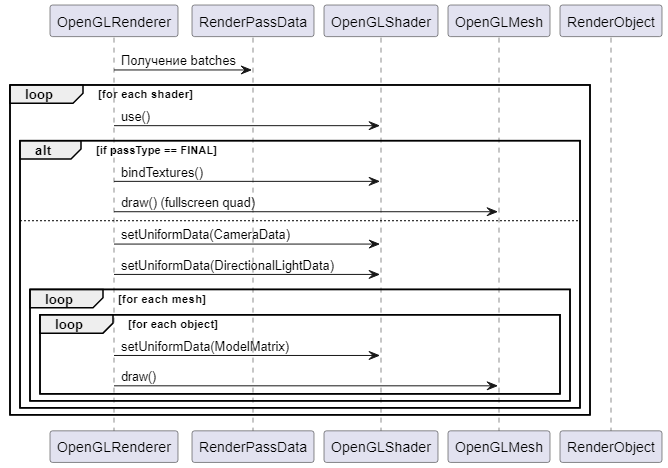


Рисунок 12 – Алгоритм исполнения команд отрисовки.

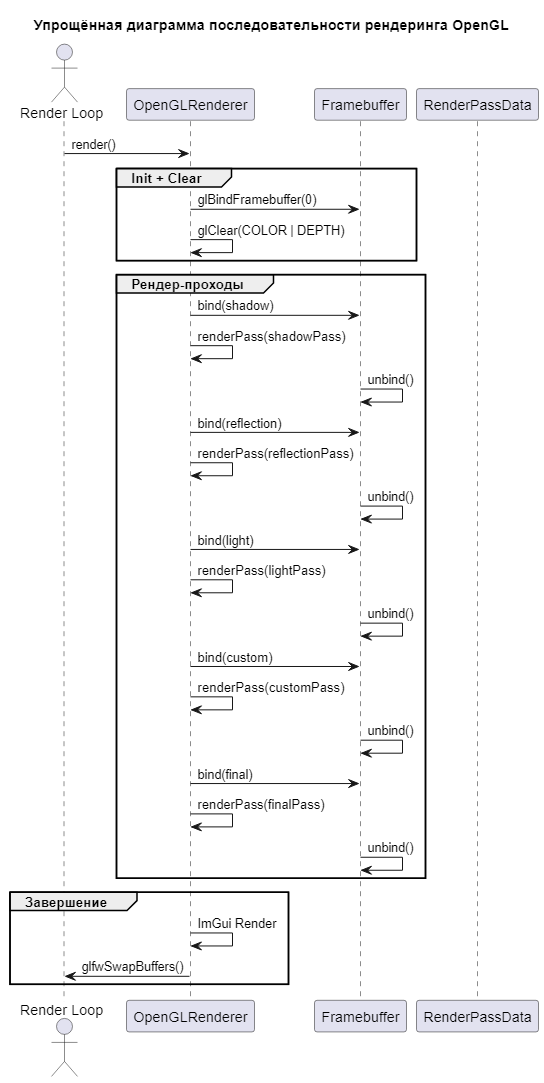


Рисунок 13 – Диаграмма последовательности рендера OpenGL.

Таким образом OpenGLRenderer следует инструкциям, составленным на верхнем уровне.

В заключение раздела можно сказать, что архитектура на основе высокоуровневой абстракции, позволяет поддерживать систему рендеринга актуальной и легко расширяемой, вплоть до добавления новых графических API. Для конфигурирования и модификации процесса рендеринга, например добавления пост-обработки или технологии трассировки лучей достаточно изменить состав проходов прослойки, не внося изменения в конкретную реализацию.

Интерфейсные формы

