МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Кафедра «Прикладная математика и информатика»
(наименование)

09.03.03 Прикладная информатика
(код и наименование направления подготовки, специальности)

Разработка программного обеспечения
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему «Создание и исследование кроссплатформенной библиотеки для интерактивных приложений»

Обучающийся М. Шидер
(Инициалы Фамилия) (личная подпись)

Руководитель Доктор социологических наук, доцент, Е. В. Желнина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант кандидат филол. наук, доцент, М. В. Дайнеко
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2025

Аннотация

Тема выпускной квалификационной работы: «Создание и исследование кроссплатформенной библиотеки для интерактивных приложений». Работа была выполнена обучаемым Тольяттинского Государственного Университета, кафедры прикладной математики и информатики, группы ПИб-2106а, Шидер Мирой. Выпускная квалификационная работа посвящена реализации современной кроссплатформенной библиотеки для создания интерактивных окон.

Цель выпускной квалификационной работы является создание кроссплатформенной графической библиотеки на основе Vulkan API для разработки интерактивных приложений, обеспечивающей высокую производительность, гибкость и удобство интеграции в различные проекты компании. Объект исследования – кроссплатформенная библиотека. Предмет исследования – графическое desktop-приложение как использование кроссплатформенной библиотеки

Задачи работы:

- Провести анализ существующих графических библиотек и фреймворков, выявить их преимущества и недостатки.
- Изучить особенности Vulkan API и его отличия от других графических API
- Разработать архитектуру кроссплатформенной графической библиотеки, обеспечивающую поддержку операционных систем.
- Реализовать основные компоненты библиотеки.
- Протестировать библиотеку, оценить её стабильность и эффективность.
- Разработать демонстрационное приложение, использующее созданную библиотеку, для наглядной оценки её возможностей.

Бакалаврская работа содержит пояснительную записку объемом 48 страниц, включая 43 иллюстраций, 2 таблицы, список литературы из 26 наименований, включая 22 зарубежных.

Abstract

The title of the graduation work is "Development and Research of a Cross-Platform Library for Interactive Applications".

The graduation work consists of an introduction, three chapters, 53 figures, 4 tables, a conclusion, and a list of 25 references including foreign sources.

The aim of this graduation work is to create a cross-platform graphical library based on Vulkan API for developing interactive applications, ensuring high performance and ease of integration into various company projects.

The object of the graduation work is the cross-platform library

The subject of the graduation work is a graphical desktop application as an example of using the cross-platform library.

The key issue of the graduation work is to study the features of Vulkan API and develop an efficient architecture for the cross-platform graphical library.

The graduation work may be divided into several logically connected parts which are: theoretical research and analysis, library development and implementation, testing and practical application.

The first part describes in detail the theoretical foundations, including analysis of existing graphical libraries and study of Vulkan API capabilities for cross-platform development.

The second part outlines the development process of the library's architecture and implementation of its core components, including the window management system.

The third part consists of comprehensive testing of the library and demonstration of its practical application through a developed sample application.

n conclusion we'd like to stress that the developed library successfully combines high performance with cross-platform compatibility, as confirmed by testing results and practical implementation.

The work is of interest for a wide circle of readers interested in cross-platform development, graphical libraries, and interactive applications.

Оглавление

Введение

Актуальность темы исследования обусловлена парадоксальной ситуацией современного технологического развития: несмотря на кажущийся прогресс, человечество сталкивается с системной проблемой утраты критически важных знаний и навыков. Этот феномен, который можно обозначить как «технологический регресс», проявляется как в исторической ретроспективе, так и в современных реалиях цифровой эпохи.

В 2019 году на Московской конференции DevGAMM известный разработчик игр Джонатан Блоу в своем докладе «Предотвращая коллапс цивилизации» (Preventing the Collapse of Civilization) [5] сформулировал ключевую проблему современности: технологии, включая программное обеспечение, деградируют из-за разрыва преемственности знаний и отсутствия механизмов их сохранения.

Кубок Ликурга (IV век н.э.) демонстрирует утрату римлянами технологии создания нано материалов. Византийский «греческий огонь», бывший грозным оружием своего времени, не сохранился до наших дней. Антикитерский механизм (II-I века до н.э.) оставался непревзойденным образцом точной механики на протяжении многих столетий.

В современную эпоху мы наблюдаем аналогичные процессы: космическая программа «Аполлон-11» (1969 г.) сегодня не может быть воспроизведена NASA без колоссальных усилий, а российская космонавтика, несмотря на исторический прорыв Гагарина (1961 г.), сталкивается с проблемами надежности современных ракет-носителей.

Особую тревогу вызывает состояние современного программного обеспечения. Как отмечает Блоу, сложность систем достигла критического уровня, когда даже профессионалы (как Боб Колвелл из Intel) сталкиваются с невозможностью воспроизвести или модернизировать существующие решения. Кейси Муратори в своем анализе «проблемы 30 миллионов строк кода» показывает, что современные программы опираются на неподъемный

пласт зависимостей, что делает их уязвимыми и трудно поддерживаемыми [19].

Современные технологические тренды лишь усугубляют ситуацию. Массовый переход на унифицированные платформы (такие как Unreal Engine) приводит к утрате специализированных решений и кастомных движков. Геополитическая ситуация с санкциями и технологическими ограничениями делает вопрос технологического суверенитета особенно острым. Примеры катастроф, вызванных ошибками в ПО (авиакатастрофы, сбои в критической инфраструктуре), демонстрируют системные риски современного подхода к разработке.

Исходя из вышесказанного, целью данной работы стало создание собственного, независимого программного продукта. Это особенного выгодно, так как данный продукт не только ускорит процесс разработки программного обеспечения для клиентов компании, но и будет нелицензионным и не попадет под санкции, что в современных реалиях не мало важный аспект.

Задачи:

- Провести анализ существующих графических библиотек и фреймворков, выявить их преимущества и недостатки.
- Изучить особенности Vulkan API и его отличия от других графических API
- Разработать архитектуру кроссплатформенной графической библиотеки, обеспечивающую поддержку различных операционных систем.
- Реализовать основные компоненты библиотеки
- Протестировать библиотеку, оценить её стабильность и эффективность.
- Разработать демонстрационное приложение, использующее созданную библиотеку, для наглядной оценки её возможностей.

Глава 1 Постановка задачи на разработку программного обеспечения для информационной системы предприятия

1.1 Значение программного обеспечения для информационной системы предприятия

Компания «ВорлдИнтерТех РУС» занимается разработкой веб-приложений, облачных сервисов, а также цифровизацией бизнеса и интернет маркетингом. Сфера деятельности организации в основном сосредоточена на разработке программного обеспечения различного типа и информационных технологий для последующей интеграции разработанных приложений или программных модулей в коммерческие предприятия.

Разрабатываемые программные продукты являются своеобразным мостом, связующим звеном, между коммерческими компаниями и конечными потребителями. Из выше сказанного следует, что сервисы, разрабатываемые компанией, очень разнообразны. Например, компания разрабатывает приложения в сфере игровой разработки, оптовых продаж, предоставления услуг и маркетинга.

Для формирования более точного представления о компании на рисунке 1 представлена организационная структура предприятия:

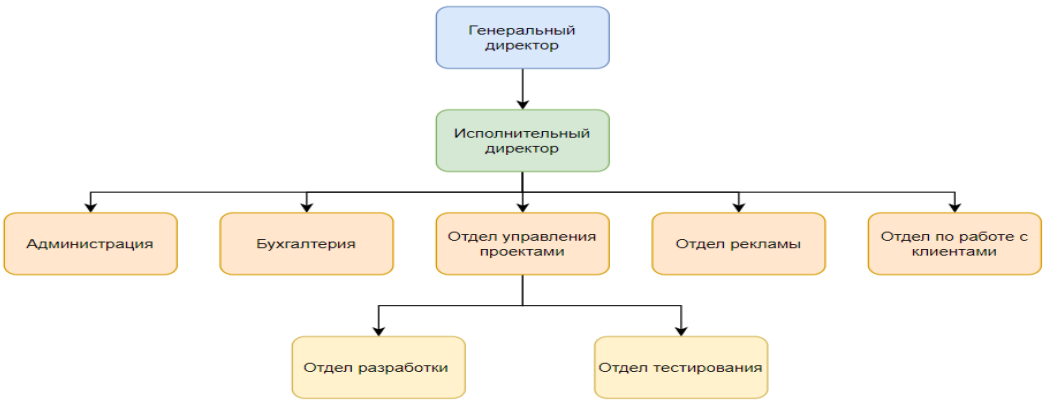

Рисунок 1 – Схема подразделений предприятия

Так же на рисунке 2 представлена схема иерархии распределения обязанностей:

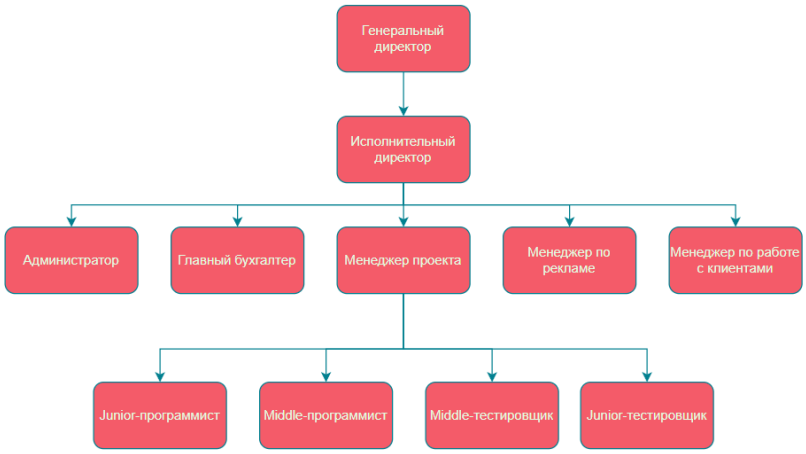

Рисунок 2 – Иерархическая схема предприятия

Рассмотрим так же функциональную модель организации.

В данном отделе разработки работают следующие сотрудники:

– Junior- программисты;

– Middle – программисты.

Рассмотрим потоки входных/выходных данных и механизмов отдела разработки в контексте организации (см. рис. 3).

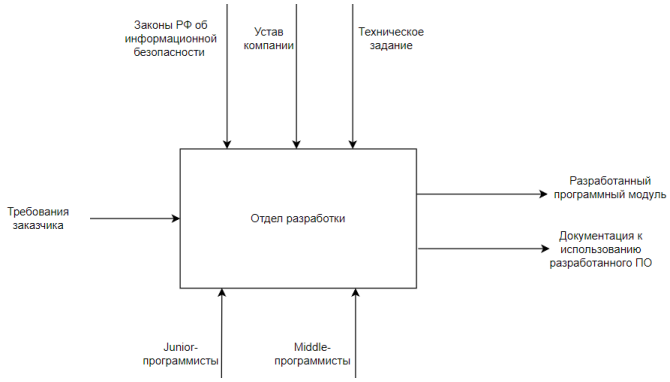

Рисунок 3 – Схема информационных потоков отдела разработки

Исходя из того, что компания разрабатывает большое количество программного обеспечения, ей потребовался программный продукт, который позволил бы ускорить процесс разработки и соответственно увеличил бы экономическую выгоду.

На рынке уже представлены программные решения данной проблемы (их мы рассмотрим позже), однако компания приняла решение разработки собственного программного продукта.

Рассмотрим выгодность такого решения.

На рисунке 4 показана диаграмма «Причина-следствие», показывающая взаимосвязь между проблемой и факторами, способствующими ей:

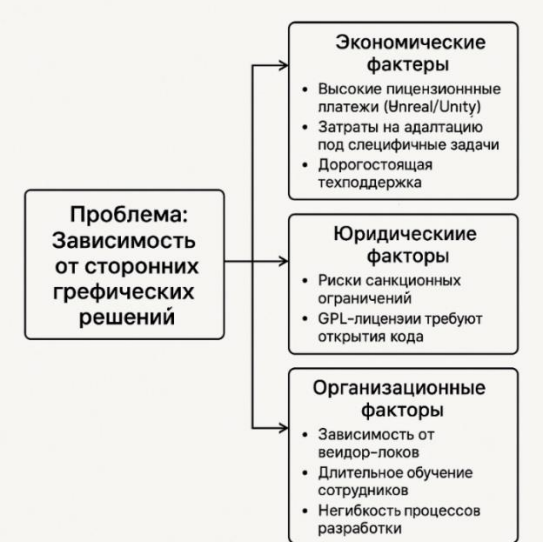

Рисунок 4 – Диаграмма «Причина следствие»

Разработка собственной графической библиотеки позволит компании:

– снизить зависимость от сторонних решений (Unreal, Unity), уменьшив лицензионные затраты;
– повысить производительность за счет глубокой оптимизации под Vulkan API;
– обеспечить безопасность за счет полного контроля над кодовой базой и отказа от стороннего ПО с закрытым исходным кодом;

– увеличить гибкость разработки, адаптируя библиотеку под специфические нужды компании;
– снизить риски санкционных ограничений, создавая программное обеспечение, независимое от иностранных поставщиков.

В качестве примера успешного применения можно представить следующий кейс. Компании, разрабатывающие игровые движки и специализированное ПО, нередко переходят на собственные графические движки. Например, id Software разработала id Tech Engine, позволивший достичь высоких FPS и качественного рендеринга в играх серии DOOM. Внедрение собственной технологии позволило компании не зависеть от сторонних решений и адаптировать движок под свои нужды.

Таким образом разрабатываемое ПО может способствовать экономическому развитию за счет:

– снижения затрат на лицензирование сторонних решений (Unreal Engine, Unity);
– оптимизации расходов на адаптацию существующих движков под специфические задачи;
– создания конкурентного преимущества, предлагая заказчикам гибкое и независимое ПО.

Возрастание конкурентоспособности происходит за счет:

– полный контроль над кодовой базой позволяет адаптировать библиотеку под конкретные проекты;
– улучшение кроссплатформенности обеспечивает поддержку Windows, Linux, macOS;
– оптимизация под Vulkan API повышает производительность приложений.

Далее приведена таблица 1, показывающая пример успешных проектов в сфере графических библиотек [11], [25].

Таблица 1 – Примеры успешных проектов в сфере графических библиотек

Проект / Компания	Достигнутые результаты и выгоды
id Tech Engine (id Software)	Высокая FPS, полная кастомизация, отсутствие лицензионных платежей.
Frostbite Engine (EA)	Оптимизированный рендеринг, внутренняя разработка без зависимости от Unity/Unreal.
Godot Engine	Открытый исходный код, низкий порог входа, отсутствие лицензионных сборов.
Source Engine (Valve)	Гибкость, высокая производительность, независимость от сторонних решений.

Стоит отметить, что разработка собственной графической библиотеки- это стратегически важное решение, обеспечивающее независимость от сторонних решений, оптимизацию производительности и гибкость в реализации проектов.

В ближайшие годы на эту сферу существенно повлияют:

– искусственный интеллект - ускорит генерацию графики и автоматизирует рутинные задачи;

– облачные технологии и WebAssembly - повысят доступность и удобство разработки;

– кроссплатформенность и DevOps - позволят быстро выпускать стабильные версии под разные ОС;

– Ray Tracing + Vulkan - выведут графику на новый уровень реалистичности и производительности;

– ИИ-моделирование сцен - откроет возможности генерации 3D-контента по описанию.

На рисунке 5 приведена диаграмма описывающая данные тенденции:

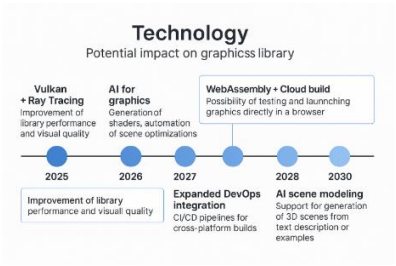

Рисунок 5 – Диаграмма влияния новых технологий на наше ПО

Эти тенденции помогут снизить издержки, увеличить производительность и усилить конкурентные преимущества компании в сфере разработки графического ПО.

1.2 Основные функции программного обеспечения для информационной системы предприятия

Основной функцией разрабатываемого ПО является автоматизация и упрощение процесса разработки программных продуктов.

Разрабатываемое ПО может:

– автоматически инициализировать и управлять окнами, интерфейсами и событиями;
– использовать слоистую архитектуру, где каждый слой выполняет свои задачи без вмешательства пользователя;
– профилировать и логировать происходящее без ручного мониторинга;
– подгружать ресурсы (шрифты, изображения, звуки) автоматически с помощью библиотек STB и Miniaudio.

На рисунке 6 показано, как автоматизация позволяет существенно сократить время выполнения задач:

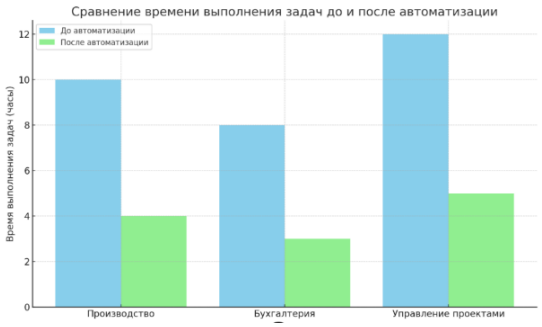

Рисунок 6 – График сравнения времени выполнения задач до и после автоматизации

На графике показано, как автоматизация позволяет существенно сократить время выполнения задач:

В производстве – с 10 до 4 часов.

В бухгалтерии – с 8 до 3 часов.

В управлении проектами – с 12 до 5 часов.

1.3 Обзор существующих решений и их анализ

Проведем обзор в виде сравнительной таблицы 2:

Талица 2 – Сравнение готовых решений

Движок	Открытый код	Поддержка Vulkan	Лицензия	Контроль над кодом	Подходит для адаптации
Unreal	Нет	Да	Условно-бесплатно	Низкий	Сложно
Unity	Частично	Частично	Коммерческая	Ограниченный	Сложно
Godot	Да	Частично	MIT	Высокий	Хорошо
Собственное ПО	Да	Полная	Внутренняя	Полный	Максимальная гибкость

Несмотря на наличие мощных движков [22], [23] на рынке, они не отвечают всем требованиям компании:

Зависимость от сторонних решений:

- риски по лицензиям и санкциям;
- недостаточная гибкость для узкоспециализированных задач;
- часть исходного кода закрыта, что снижает безопасность и контроль.

Разработка собственного программного продукта:

- повышает независимость;
- увеличивает производительность за счёт глубокой оптимизации;
- обеспечивает полную адаптацию под нужды компании.

1.4 Требования к функционалу и интерфейсу приложения

Наш программный продукт имеет следующие функциональные требования:

Создание графического интерфейса и рендеринг:

- реализация собственного графического движка с использованием Vulkan API;
- рендеринг графики с высокой производительностью и контролем ресурсов;
- поддержка кроссплатформенной работы (Windows, Linux, macOS) через абстрактные интерфейсы Xlib и Win64 API.

Работа с окнами и пользовательским интерфейсом:

- использование собственной библиотеки SWL (Simple Window Library);
- создание и удаление окон;
- управление курсором, разрешением и событиями ввода;
- реализация модуля std_ui для взаимодействия с пользователем и интерфейсными элементами.

Звуковое сопровождение:

- воспроизведение звука с помощью Miniaudio – лёгкой и кроссплатформенной библиотеки;
- поддержка различных аудиобэкендов.

Оптимизация и производительность:

- использование архитектуры, вдохновлённой движком Frostbite (слои и события);
- применение Data-Oriented Design (DoD) для оптимизации загрузки CPU и кэша;
- использование SOLID-принципов, с фокусом на производительность.

Профилирование и отладка:

– встроенные средства анализа производительности (логгирование, профилирование);
– интеграция с внешними отладчиками (CodeLite, Visual Studio).

Установлены следующие требования к интерфейсу:

а) удобство:

1) интуитивный, простой интерфейс;

2) доступ к основным функциям в 2–3 клика.

б) быстродействие:

1) мгновенный отклик на действия;

2) минимальные задержки даже при высокой нагрузке.

в) минимализм:

1) лёгкий и ненагруженный визуально UI;

2) использование лёгких библиотек (STB, Miniaudio).

г) гибкость:

1) возможность расширения без полной переработки;

2) поддержка пользовательских настроек и тем.

Глава 2 Процесс разработки программного обеспечения

2.1 Проектирование программного обеспечения

Начнем с проектирования архитектуры нашего ПО.

Архитектура приложения – это структурная организация программной системы, определяющая:

- компоненты и их взаимодействие (модули, сервисы, слои);
- принципы работы (паттерны проектирования, потоки данных);
- масштабируемость и поддержку (микросервисы, монолит);
- интеграцию с внешними системами (API, базы данных).

Популярные подходы формируют архитектуру исходя из OOP, Clean Code, – SOLID [8].

Дизайн ориентированный на данные (Data Oriented Design, DoD) – это набор правил по написанию программ эффективных с точки зрения производительности на уровне обращения к основной памяти и работы процессора. В отличие от других популярных принципов (основные из которых были разобраны в предыдущей главе), данный подход ориентируется в своей сути на производительность компьютера, а не на производительность отдельно взятого программиста. Данный подход отличается тем, что как минимум имеет доказательной базой своей эффективности, так как он вводит всем понятную метрику – время.

Однако альтернативные решения на подобие Data Oriented Design не имеют полноценного технического руководства в виде десятков и сотен книг, поэтому в проектировании архитектуры мы будем опираться на произвольные решение и подходы других программистов, не получившие какого-либо названия или классификации [1], [3], [18].

Собственное архитектурное решение, частично основанное на некоторых архитектурных подходах игрового движка Electronic Arts –

Frostbite [6]. Презентован бывшим разработчиком австралийского филиала Electronic Arts – Яном Черниковым (YouTube TheCherno). В основе лежит Event Layer система [7].

На рисунке 7 приведено схематическое изображение архитектуры проекта:

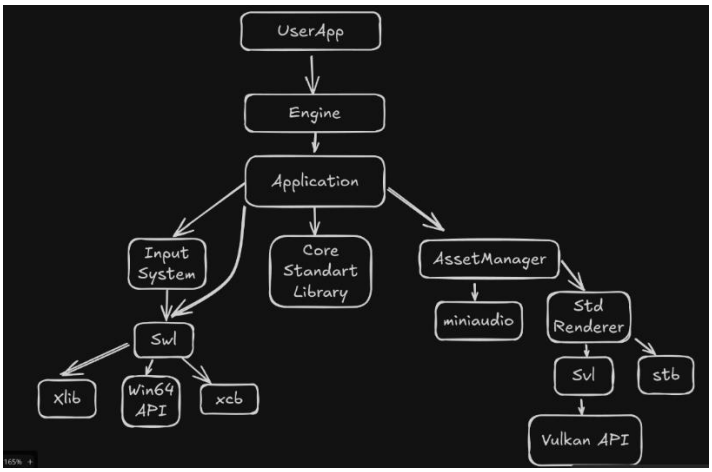

Рисунок 7 – Архитектура проекта

Теперь рассмотрим взаимодействие компонентов систему друг с другом. Для этого была построена Use-case диаграмма (см. рис. 8), демонстрирующая взаимодействия между пользователем, то есть программистами, операционной системой и системой сборки или непосредственно программными модулями нашего продукта.

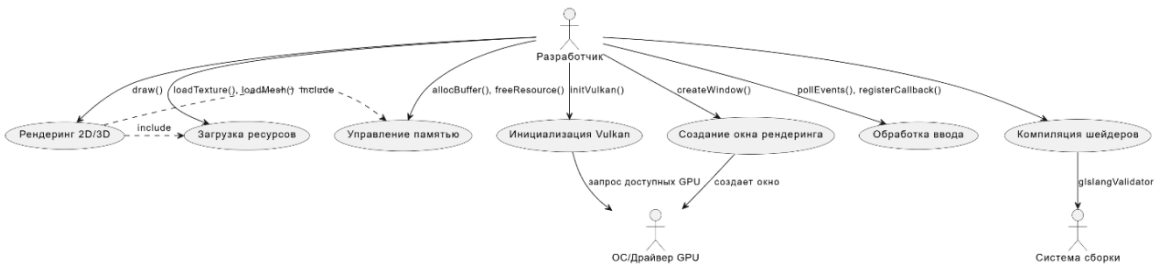

Рисунок 8 – Use-case диаграмма

Также была построена диаграмма классов, представленная на рисунке 9.

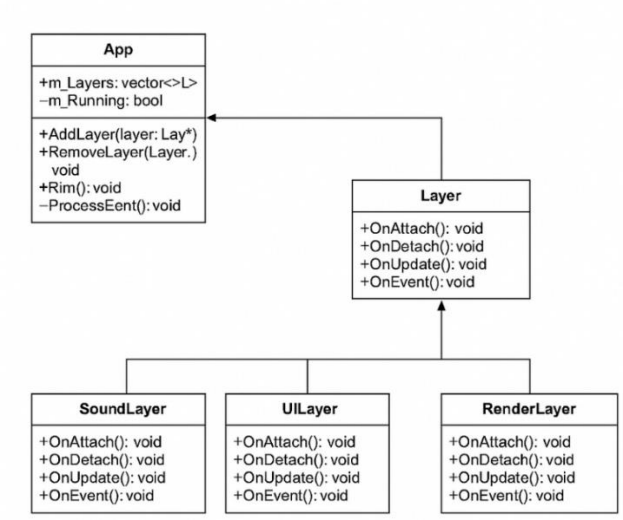

Рисунок 9 – Диаграмма классов.

На диаграмме классов изображена структура ключевых компонентов системы:

– Application – основной класс, управляющий запуском, обновлением и завершением приложения. Взаимодействует со слоями;

– Layer – абстрактный базовый класс для всех слоёв (например, UI, аудио, рендер). Каждый слой реализует методы onAttach(), onUpdate() и onEvent().

Подклассы:

– UILayer, AudioLayer, RenderLayer – конкретные реализации слоя интерфейса, звука и графики;

– Event – класс, описывающий события. Обрабатываются слоями через onEvent().

Это отражает архитектуру, основанную на Event Layer System: приложение обрабатывает события и обновляет слои по циклу, обеспечивая модульность и расширяемость.

2.2 Выбор технологий и инструментов для разработки

Разрабатывая всесторонне комплексный программный продукт, на подобие текущей работы нужно с умом подходить к выбору технологического стека. Далее приводится описание и обоснования выбора выбранных технологий.

2.2.2 Язык программирования Си

Язык программирования Си – это компилируемый высокоуровневый язык. Первая версия была создана в 1972 году гениальнейшим программистом Деннисом Макалистэйр Ритчи. Си создавался как язык системного уровня для программирования операционной системы UNIX [2]. По задумке автора данный язык был основой для написания утилит и программ внутри операционной системы. Все программы и утилиты, запускаемые в UNIX, были написаны на данном языке. Частично и модули операционной системы были также написаны на Си.

Первые версии Си не имели стандарта, на сегодняшний день данный язык претерпел много изменений [12]. Многие аспекты были модифицированы и улучшены относительно изначальной идеи [9]. Для реализации своего продукта был использован стандарт 1999 года – C99.

Данный язык был создан как кроссплатформенная альтернатива языков ассемблера [24]. Он предназначался для создания системных вещей на подобии: операционных систем, компиляторов, дебаггеров, текстовых редакторов, утилит командной оболочки и прочих приложений, используемых программистами и пользователями.

Язык был создан очень давно, за эти годы было создано множество альтернатив и замен языку Си, но несмотря на это всё ещё не существует ни одного языка, который бы справлялся с задачей системного программирования лучше. На Си написаны все операционные системы, используемые сегодня на персональных компьютерах, все API данных ОС, все

драйвера в этих системах. Си используется в качестве основного языка для OpenGL и Vulkan API.

Даже, если вы используете другой язык, где-то глубоко внутри данный язык имеет обертку над Си, поэтому имеет смысл избавиться от посредников и разговаривать с операционными система и графическими API на их родном языке – СИ.

Все операционные системы написаны на Си, все драйверы для этих операционных систем написаны на СИ, API для этих систем представлен на языке СИ. Win32/64 api, linux api, posix api, android: Native Development Kit.

Софт написанный на Си:

- Nginx – один из самых популярных серверов;
- PostgreSQL – СУБД «The World's Most Advanced Open Source Relational Database»;
- Redis;
- Vulkan, OpenGL – графические API;
- Win64 API, Posix Api, Linux Api – системные API;
- Unix, Linux, Windows, Free BSD, Open BSD, Net BSD, Minix;
- Операционные системы для всех игровых приставок: Playstation, XBOX;
- CPython.

Многие библиотеки используются в других языка через биндинги для данного языка с использованием механизма FFI [4].

Так же в проекте используется компилятор clang. В качестве IDE emacs с настроенным конфигом. Codelite/Visual Studio в качестве дебаггера. Premake5 в качестве билд системы.

2.2.3 Vulkan Api

Самый новый из имеющихся графический API, спецификация создана Khronos Group, которые ранее создали и утвердили спецификацию OpenGL.

Данный API используется на всех современных десктопных (Desktop) платформах, а также на мобильных устройствах [13]. Vulkan можно назвать низкоуровневым графическим API, он предполагает более точное описание графического конвейера и действий, которые от него требуется [10]. API изначально разрабатывался с учётом многопоточности, поэтому имеет большое количество примитивов синхронизации под разные задачи, а действия внутри графических конвейеров записываются в команды и выполняются асинхронно от основного потока, задачи синхронизации лежат в зоне ответственности приложения [14], [20]. Также Vulkan хранит всё состояние на стороне приложения, а не на стороне драйвера, это необходимо не только для многопоточности, но и для того, чтобы Vulkan был легковесной обёрткой над драйвером. Легковесность – это отличительная черта данного API, так как в отличие от OpenGL, который представляет из себя набор команд для конфигурации драйвера, здесь программисту даётся полная свобода действий над тем, как и какие команды пишутся в буфер, а также когда эти команды будут выполнены.

Резюмируя плюсы Vulkan можно выделить:

– Многопоточность;
– Кроссплатформенность (Linux, Windows, Android);
– Легковесность;
– Контроль.

Vulkan требует более тщательного подхода к описанию программы, требует сильно больше времени для написания кода, но при этом даёт намного больший контроль над происходящим в приложении, в отличии от других API [17].

2.2.4 Xlib и Win64 API

Xlib - Это библиотека, написанная на Си для работы с реализацией графического сервера X (Xorg) [15]. Сервер X выбран исходя из своей

распространённости в среде Linux, подавляющее большинство современных дистрибутивов реализуют оконный менеджмент через него, тогда как более новый Wayland ещё не получил такую распространённость.

Win64 API – это набор функций, структур и констант, предоставляемых операционной системой Windows для взаимодействия с её компонентами [16]. Написан преимущественно на C/C++, является основным интерфейсом для разработки нативных приложений под Windows.

2.2.5 Библиотеки, используемые в проекте

При выборе библиотек использовалось несколько простых принципов:

- Single Header;
- OpenSource;
- Cross Platform.

Данный проект зависит от двух библиотека: stb и miniaudio.

Говоря о библиотеке STB мы говорим не об одной библиотеке, а о наборе библиотек с открытым исходным кодом, написанных американским программистом Шоном Барреттом (Sean Barrett).

В проекте используются: stb_image (загрузчик изображений), stb_image_writer (программа для сохранения изображений в популярных форматах по типу png, jpg, gif в память или на диск), stb_truetype (растеризатор шрифта).

Загрузчик изображений поддерживает форматы:

- PNG;
- JPEG;
- GIF;
- BMP;
- HDR;
- PIC;
- PNM;

– TGA.

Данная библиотека поддерживает все современные платформы:

– Linux;
– FreeBSD, OpenBSD, NetBSD;
– Windows;
– Android;
– IOS;
– Web (через Emscripten).

Библиотека Miniaudio – это библиотека с открытым исходным кодом, написанная австралийским программистом Девидом Рейдом (David Reid) [21].

Данная библиотека поддерживает все современные платформы, что и предыдущая, а также поддерживает все существующие бекенды на данных платформах:

– Linux: ALSA, PulseAudio, JACK;
– FreeBSD / OpenBSD / NetBSD: OSS, sndio, audio(4);
– Windows WASAPI, DirectSound, WinMM;
– Web: Emscripten, WebAudio;
– MacOS: Core Audio;
– Android: Aaudio, OpenSL | ES;
– IOS:Core Audio,

2.3 Реализация функциональных требований

Сначала опишем включаемые файлы, которые присутствуют на скриншотах, используемых в данной главе.

Файлы включения:

– Core/Types.h – типы данных, утилиты, данные, которые нужны практически в каждом файле программы;

– Core/SimpleApplication.h – файл, содержащий интерфейс для работы с приложением;
– Core/SimpleMath.h – вся математика для работы с векторами, матрицами, кватернионами;
– Core/SystemInfo.h – системная информация;
– TopDownBattle/TopDownBattleLayer.h – слой игры.

Точка входа в программу достаточна стандартна для программы на Си. Сначала включаются файлы, необходимые для старта приложения (см. рис. 10).

```
#include <Core/Types.h>
#include <Core/SimpleApplication.h>
#include <Core/SimpleMath.h>
#include <Core/SystemInfo.h>
#include <TopDownBattle/TopDownBattleLayer.h>
```

Рисунок 10 – Включение заголовочных файлов

В точке входа мы выводим в терминал информацию о системе. Данный шаг требуется для сборки на этапе разработки. Системная информация содержит название операционной системы, порядок группировки бит (на пользовательских машинах это всегда Little Endian), битность вещественного числа real объявленного в Core/Types, тип билда (Debug – разработческая сборка, Release – релизная сборка, Dist – билд дистрибьюции/готовый продукт) (см. рис. 11).

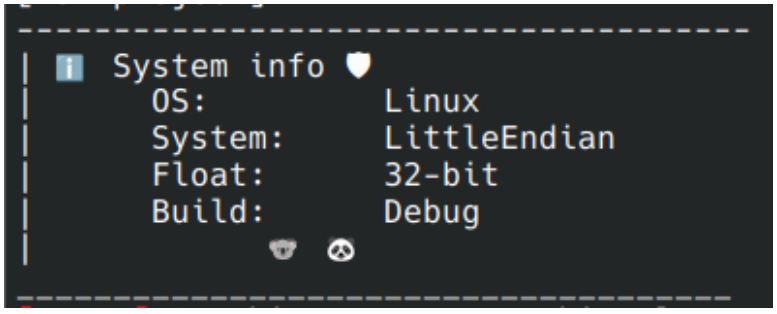

Рисунок 11 – Системная информация

Далее формируется структура настройки приложения, где прописывается имя, аргументы терминала, количество аргументов, флаг дебага, флаг вертикальной синхронизации, режим экрана, размер окна, минимальный размер окна и позиция окна на экране.

Далее эта структура передаётся в функцию создания, после этого регистрируется слой игры, далее приложение запускается. Функция запуска имеет внутри «бесконечный цикл», который будет выполняться до тех пор пока приложение не будет закрыто пользователем (см. рис. 12). Финальной функцией является уничтожение приложения, в этой функции удаляются все выделенные ресурсы.

```
i32
main()
{
    system_info_print();

    SimpleApplicationSettings appSet = (SimpleApplicationSettings) {
        .pName = "Mira Schieder's Diplom",

        .ArgsCount = 0,
        .aArgs = NULL,

        .IsDebug = 1,
        .IsVsync = 1,
        .ScreenMode = SimpleScreenMode_Custom,

        .Size = v2_new(2560, 1349),
        .MinSize = v2_new(2560, 1349),
        .Position = v2_new(0, 0)
    };
    simple_application_create(&appSet);

    td_layer_add_to_application();

    simple_application_run();
    simple_application_destroy();

    return 0;
}
```

Рисунок 12 – Главная функция

Первым делом в создании приложения мы глобально устанавливаем кодировку UTF-8 для стандартной библиотеки Си. Функция «_init_slog» занимается инициализацией логирования.

Приложение используется паттерн Singleton для хранения состояния. Предполагается, что процесс содержит всего одно приложение. Нельзя

создавать несколько приложений в одном процессе.

В качестве Singleton инстанса используется переменная «g». Подобное имя входит в конвенцию используемую в проекте, подобные переменные создаются глобально для .c файла и имеют модификатор «static». Переменная имеет сокрытие на уровне object файла.

Также в функции создания приложения инициализируется библиотека работы с окнами simple window library (swl), а после создаётся само окно.начала опишем включаемые файлы, которые присутствуют на скриншотах, используемых в данной главе.

Файлы включения (см. рис. 13):

– Core/Types.h – типы данных, утилиты, данные, которые нужны практически в каждом файле программы;
– Core/SimpleApplication.h – файл, содержащий интерфейс для работы с приложением;
– Core/SimpleMath.h – вся математика для работы с векторами, матрицами, кватернионами;
– Core/SystemInfo.h – системная информация;
– TopDownBattle/TopDownBattleLayer.h

```
void
simple_application_create(SimpleApplicationSettings* pSettings)
{
    // DOCS: Set locale to utf-8
    setlocale(LC_ALL, ".UTF8");
    _init_slog();

    g = (SimpleApplication) {
        .pName = simple_string_new(pSettings->pName),
        .IsDebug = pSettings->IsDebug,
        .IsVsync = pSettings->IsVsync,
    };

    SimpleString* pHdr = simple_string_header(g.pName);

    // DOCS: Creating window
    swl_init((SwlInit) { .MemoryAllocate = malloc, .MemoryFree =
free, });
    SwlWindowSettings set = {
        .pName = pHdr->pData,
        .NameLength = pHdr->Size,
        .Position = { .X = pSettings->Position.X, .Y =
pSettings->Position.Y },
        .Size = {
            .Width = IfFalseThen(pSettings->Size.Width, 100),//968,
            .Height = IfFalseThen(pSettings->Size.Height, 100),
            .MinHeight = IfFalseThen(pSettings->MinSize.X, 100),
            .MinWidth = IfFalseThen(pSettings->MinSize.Y, 100)
        },
    };
    swl_window_create(&g.Window, set);
```

Рисунок 13 – Функция создания приложения

Далее идёт установка callback’а обработки событий для swl. Callback-функция служит мостом соединяющим цепочку событий окна и прилоежние. После установки callback’а инициализируется asset manager, который нужен для загрузки внешних ресурсов (см. рис. 14).

```
107     swl_window_set_event_call(&g.Window,
    simple_application_on_event);
108
109     std_asset_manager_create();
```

Рисунок 14 – Продолжение функции создания приложения

Архитектура данного программного продукта предполагает наличие разных слоёв приложения, которые вызываются в управляющих методах приложения.

Звуковой слой отвечает за инициализацию и деинициализацию звуковой библиотеки miniaudio. Miniaudio имеет свой рабочий конвеер, который запускается в отдельном потоке, поэтому данный слой не содержит функции обновления (update), так как звуки запускаются пользователем (в данном случае пользователем выступает код приложения, а не человек см. рис. 15).

```
111     SimpleLayer soundLayer = {
112         .pName = simple_string_new("Звуковой слой"),
113         .OnAttach = sound_layer_on_attach,
114         .OnDestroy = sound_layer_on_destroy,
115     };
116     simple_application_add_layer(soundLayer);
```

Рисунок 15 – Регистрация звукового слоя

Интерфейсный слой отвечает за работу библиотеки пользовательского интерфейса – std ui. Данная библиотека полностью вписана в архитектуру

«приложения» (simple_application_*), поэтому обширно использует функции слоя (см. рис. 16).

```
118     SimpleLayer uiLayer = {
119         .pName = simple_string_new("Интерфейсный слой"),
120         .OnAttach = ui_layer_attach,
121         .OnUpdate = ui_layer_update,
122         .OnBeforeUpdate = ui_layer_before_update,
123         .OnAfterUpdate = ui_layer_after_update,
124         .OnEvent = ui_layer_event,
125         .OnDestroy = ui_layer_destroy,
126     };
127     simple_application_add_layer(uiLayer);
```

Рисунок 16 – Регистрация интерфейсного слоя

Графический слой отвечает за работу с std_renderer. Он создает и уничтожает рендерер, обновляет его и обрабатывает события (см. рис. 17).

```
129     SimpleLayer graphLayer = {
130         .pName = simple_string_new("Графический слой"),
131         .OnAttach = graphics_layer_attach,
132         .OnAfterUpdate = graphics_layer_after_update,
133         .OnEvent = graphics_layer_event,
134         .OnDestroy = graphics_layer_destroy,
135     };
136     simple_application_add_layer(graphLayer);
137 }
```

Рисунок 17 – Регистрация графического слоя

Все слои записываются в массив, хранящийся в глобальной переменной, после чего эти слои используются в цикле обновления.

Выполнение цикла зависит от поля IsRunning в переменной «g». Внутри цикла инициализируется SimpleProfiler – профилировщик, который используется для подсчёта deltaTime и времени работы приложения. Запускается профилирование функцией simple_profiler_start и останавливается после работы всех слоёв.

Обработка callback-функций зависит от поля IsMinimized, когда приложение свёрнуто мы не запускаем обработку и экономим ресурсы (см.

рис. 18).

Рисунок 18 – Цикл обновления слоёв

Уничтожение приложения выглядит схожим образом на цикл обновления тем, что мы уничтожением слоёв, вызывая соответствующую функцию слоя.

Далее удаляется окно, вызывается хелпер функция рендера для уничтожения рендер ресурсов, уничтожение asset manager, уничтожение рендера и деинициализация логирования (см. рис. 19).

Рисунок 19 – Функция уничтожения приложения

Обработка событий происходит по тому же сценарию, что и предыдущие функции, событие сначала обрабатывается приложением, а после прокидывается остальным слоям. Помимо этого, делается проверка на то, что событие было уже обработано, если проверка успешна, то мы заканчиваем обработку, иначе продолжаем передавать его дальше по циклу (см. рис. 20).

```
139 void
140 simple_application_destroy()
141 {
142     i64 count = sva_count(g.aLayers);
143     for (i64 i = 0; i < count; ++i)
144     {
145         SimpleLayer layer = g.aLayers[i];
146         layer.OnDestroy();
147     }
148
149     swl_window_destroy(&g.Window);
150
151     // DOCS: Need to call vkDeviceWaitIdle
152     std_renderer_prepare_for_destruction();
153
154     std_asset_manager_destroy();
155
156     std_renderer_destroy();
157
158     slog_deinit();
159 }
```

Рисунок 20 – Функция обработки событий

Разберем интерфейс библиотеки работы с окнами. Функция инициализация библиотеки передаёт структуру SwlInit, которая содержит функции работы с памятью, тем самым делая библиотеку независимой от алокаторов из стандартной библиотеки (см. рис. 21).

```
void swl_init(SwlInit swlInit);
```

Рисунок 21 – Функция инициализаций библиотеки swl

Функции создания и уничтожения окна, принимает указатель на вход вместе с настройками для окна (см. рис. 22).

```
562 void swl_window_create(SwlWindow* pSwlWindow, SwlWindowSettings
    set);
563 void swl_window_destroy(SwlWindow* pWindow);
```

Рисунок 22 – Функции контроля экземпляра окна

Функция обновления окна представлена на рисунке 23.

Рисунок 23 – Функция обновления окна

Функция swl_window_get_resolutions получает все возможные расширения экрана с помощью нативного api. Код swl_window_get_resolution получает текущее расширение окна, а swl_window_get_cursor_position получает положение курсора внутри окна (см. рис. 24).

```
566 SwlResolutions swl_window_get_resolutions(SwlWindow* pWindow);
567 SwlResolution swl_window_get_resolution(SwlWindow* pWindow);
568 /* DOCS: Get current cursor position */
569 swl_v2 swl_window_get_cursor_position(SwlWindow* pWindow);
```

Рисунок 24 – Функции getter’ы

Код swl_window_set_event_call устанавливает callback для обработки события, который будет вызываться в случае получения его из API операционной системы (см. рис. 25).

```
570 /* DOCS: Set event callback (user-space), that handles events of
    the window. */
571 void swl_window_set_event_call(SwlWindow* pWindow, void
    (*onEvent)(SwlEvent*));
```

Рисунок 25 – Функция установки callback’а

Наборы setter’ов для окна. swl_window_set_above выводит окно поверх остальных. swl-window-set_maximized делает размер окна максимальным для

текущего экрана, swl_window_set_fullscreen устанавливает окно в полный экран, swl_window_set_wo_header убирает заголовок у окна, swl_window_set_background устанавливает цвет фона (см. рис. 26).

```
572 /* DOCS: Set window above other, work on linux, windows wm is
    stupid */
573 void swl_window_set_above(SwlWindow* pWindow);
574 /* DOCS: Like F11 in browser */
575 void swl_window_set_maximized(SwlWindow* pWindow);
576 /* DOCS: As it said windows goes fullscreen */
577 void swl_window_set_fullscreen(SwlWindow* pWindow);
578 /* DOCS: Remove header from window */
579 void swl_window_set_wo_header(SwlWindow* pWindow);
580 /* DOCS: Set background color using r,g,b as value from 0 to 255. */
581 void swl_window_set_background(SwlWindow* pWindow, swl_u8 r255,
    swl_u8 g255, swl_u8 b255);
```

Рисунок 26 – Функции setter’ы

swl_key_is_printable указывает может ли кнопка быть отображена, например, в текстовом поле, когда пользователь вводит логин или имя персонажа. swl_key_to_string конвертирует кнопку в текстовый формат (см. рис. 27).

```
584      //////                                    //
585      //       DOCS: Utils & Helpers            //
586      //                                  //////
587
588 swl_i32 swl_key_is_printable(SwlKey key);
589 const char* swl_key_to_string(SwlKey swlKey);
```

Рисунок 27 – Утилитарные функции для кнопок.

Библиотека работы с Vulkan API абстрагирует Vulkan API и упрощает работу с ней.

Объекты Vulkan:

– VkInstance – объект инициализирующий библиотеку Vulkan, он также хранит всю информацию приложения внутри себя – это

необходимо потому, что Vulkan не имеет глобального состояния на стороне драйвера, как, например, OpenGL;

- vkSurface - объект для связи/общения вулкан с WindowManager;
- VkPhysicalDevice - объект абстракция над физической видеокартой. Используется во многих местах для вызова вулкан функций и конфигурирования;
- VkDevice - виртуальное устройство используется в большинстве функций вулкана, нужно для выделения специфичного функционала, требуемого от физической видеокарты в отдельный объект;
- VkSwapchainKHR – объект, позволяющий отображать результат работы прохода рендеринга (render pass) на поверхность экрана (vkSurface), производит ассоциацию между поверхностью и массивом отображаемых изображений;
- VkCommandPool – это контейнер команд, функионально напоминающий «кучу» (heap) для выделения памяти под командные буферы. Он используется для оптимизации расходов выделения памяти под отдельный буфер команд и в качестве группирующего объекта, внутри которого находятся эти самые буферы. Командные буферы хранятся внутри данного объекта;
- VkCommandBuffer – буфер, в который записываются команды для выполнения на GPU, после выполнения команд буфер можно переиспользовать. Он служит для оптимизации выполнения команд, так как все команды выполняются пачками, а также для дополнительного контроля их очередности их выполнения;
- VkBuffer – это хранилище данных, прямой аналог кучи только для данных передаваемых на/из GPU. Эти данные GPU использует для рендеринга или произведения вычислений (compute pipeline);

– VkWriteDescriptorSet – объект конфигурации передачи данных между CPU и GPU.

Подробнее про SVL (Simple Vulkan Library) API. Функция svl_create создаёт внутри объекты вулкан исходя из настроек, переданных в функцию.

Кратко перечислим, что происходит внутри:

– Создание инстанса/экземпляра/сущность сущности вулкана;
– Создание поверхности – vkSurface;
– Создание физического устройства (vkPhysicalDevice). На данном этапе происходит выбор видеокарты для работы;
– Поиск индексов очередей (queue). Вулкан может использоваться не только для рендеринга, но и для compute/вычислений;
– Создание виртуального устройства.;
– Создание очередей (queue);
– Создание примитивов синхронизации – они нужны для синхронизации разных потоков;
– Создание swapchain. Можно выделить следующие обязанности swapchain: управляет кадрами, говорит как эти кадры показывать на мониторе (Vsync On/Off – вертикальная синхронизация, ограничение по кадрам связанное с частотой экрана), формат экрана (например, sRGB) – цветовая палитра экрана, расширение экрана, создание FrameBuffer, ImageViews. Framebuffer объекток.;
– Создание CmdPool CmdBuffer.

Уничтожение svl-объекта ведёт за собой уничтожение всех основных ресурсов аллоцированных для работы svl:

– уничтожение представлений изображений для swapchain;
– уничтожение самого swapchain;
– уничтожение render pass’ов (информация о всех зарегистрированных render pass хранится внутри svl);
– уничтожение примитивов синхронизации;

– vkCommandPool (в результате чего автоматически уничтожатся все vkCommandBuffer привязанные к нему);
– уничтожается vkSurface и vkInstance.

Создание и уничтожение инстанса svl представлены двумя простыми функциями – svl_create и svl_destroy. Функция создания возвращает сущность svl в виде структуры SvlInstance, а на вход принимает указатель на структуру SvlSettings. Функция уничтожения инстанса svl принимает только указатель на структуру SvlInstance, минимальное кол-во аргументов должно упростить будущий рефакторинг и модификацию сигнатуры функции, за счёт использования структур в будущем сигнатура останется прежней и любые изменения не поломают API.

```
576 /*#######################
577   DOCS(typedef): Svl Api
578   #######################*/
579
580 SvlInstance svl_create(SvlSettings* pSettings);
581 void svl_destroy(SvlInstance* pInstance);
```

Рисунок 28. Функции контроля экземпляра svl

SvlBuffer – это абстракция над VkBuffer. Код представленный ниже призван упростить работу с памятью Vulkan. Вмето большого количества опций доступных в оригинальном API, здесь используется более урезанная версия, заточенная исключительно под двумерный отрисовщик, используемый в данном приложении. Уменьшение кол-ва опций в данном случае ведёт к упрощению программного интерфейса, а также минимизации ошибок. Ниже представлены три функции для работы с буферами данных: создание, уничтожение и установка значений буфера, исходя из данных, отступа и размера (см. рис. 29).

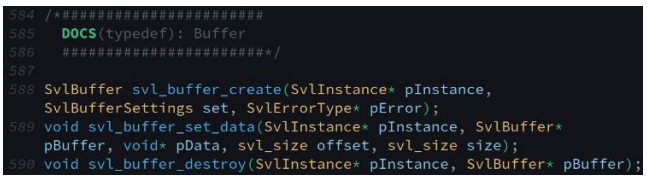

Рисунок 29 – Функции работы с буфером

SvlUniform – абстрагирует работу с VkDescriptorSet и специальным буферам выделяемым для переменных, передаваемых шейдеру.

Функции работы с юниформами представлена на рисунке 30.

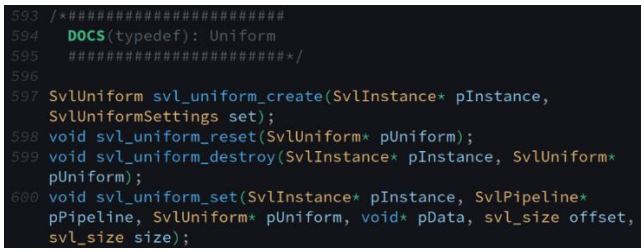

Рисунок 30 – Функции работы с юниформ

Функции работы с рендер пасом представлена на рисунке 31.

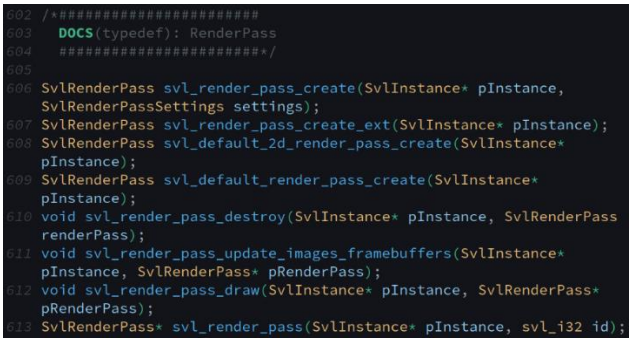

Рисунок 31 –. Функции работы с рендер пасом

Функции работы с пайплайном представлена на рисунке 32.

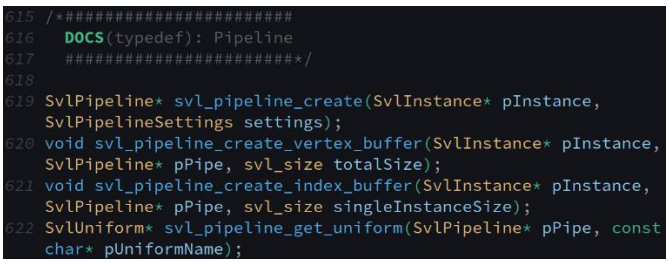

Рисунок 32 – Функции работы с пайплайном

Функции работы с setter’ом для svl представлена на рисунке 33.

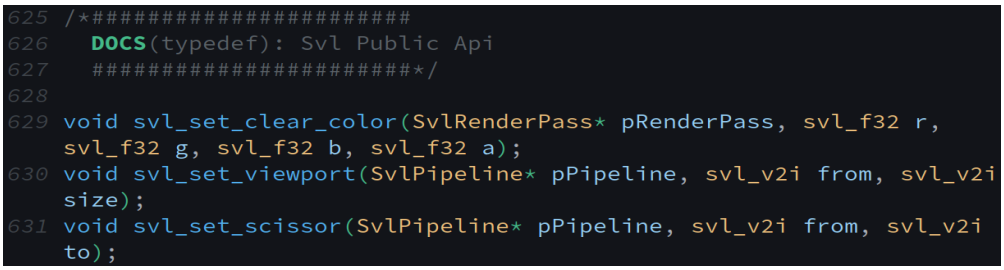

Рисунок 33 – Функции setter’ы для svl

Функции работы с cmd представлена на рисунке 34.

Рисунок 34 – Функции работы с cmd

Функции работы с frame представлена на рисунке 35.

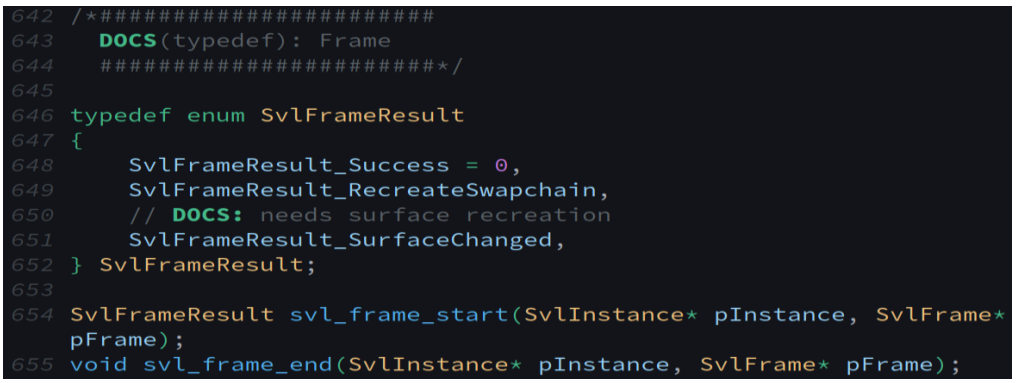

Рисунок 35 – Функции работы с frame

Функции работы с result представлена на рисунке 36.

Рисунок 36 – Функции работы с result

Функции работы с текстурами представлена на рисунке 37.

Рисунок 37 – Функции работы с текстурами

В приложении используется sRGB потому что он детальнее передаёт цветовую гамму сцены, чем остальные варианты - меньше светлых оттенков, больше темных, слудовательно, больше цветов может отобразить - больший

диапозон.

Std Asset Manager – это объект, отвечающий за загрузку и выгрузку ресурсов пользовательского приложения и ядра библиотеки.

Функции create и destroy в данном случае создают и уничтожают внутренние хранилища, которые разделены по типам ресурсов, которые нужны для работы приложения:

– Текстуры,
– Шрифты,
– Текстурные атласы,
– Звуки.

Данные ресурсы хранятся в динамических структурах, которые также требуют высвобождения памяти, что и происходит в функции destroy (см. рис. 38).

```
#ifndef STD_ASSET_MANAGER_H
#define STD_ASSET_MANAGER_H

#include <Std/StdTypes.h>


/*      ####################################
  DOCS ##         Std Asset Manager        ##
       ####################################*/

void std_asset_manager_create();
void std_asset_manager_destroy();
```

Рисунок 38 – Функции asset manager

Функции для работы с текстурами выполняют следующую логику: загружают изображение во внутреннее хранилище исходя из пути, переданного со стороны пользователя, загружают из данных составленных пользователем или полученных из другой программы, возвращают изображение по id в виде void указателя, а также возвращают весь набор загруженных ранее текстур в виде массива (см. рис. 39).

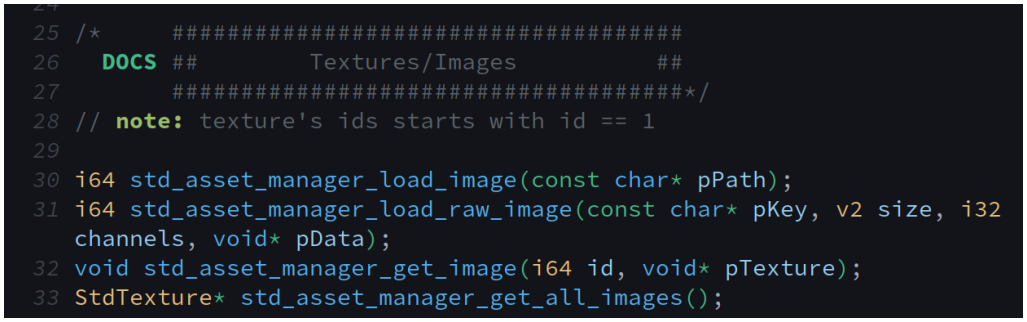

Рисунок 39 – Функции для загрузки и работы с текстурами

Функции для загрузки и взятия текстурных атласов возвращают прозрачные (transparant) структуры, то что они прорачные означает, что структура хранит весь набор данных в себе и ничего не утаивает от пользователя (см. рис. 40).

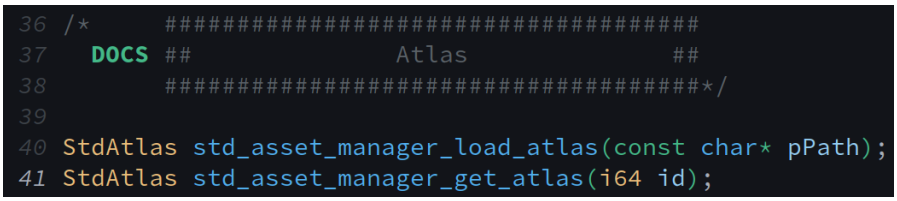

Рисунок 40 – Функции для работы с текстурными атласами

Функции для загрузки шрифта делают следующее:

– Загружает шрифт исходя из пути, размера шрифта, минимального значения диапазона и максимального, в качесвте результата возвращается id шрифта во внутреннем хранилище;
– Получает шрифт по id и возвращает указатель на него;
– Получает весь массив загруженных шрифтов.
– Загружает шрифт по умолчанию для всех заданных размеров.

Загрузка шрифтов предполагает наличие диапозона из unicode – это помогает реализовать загрузку всех возможных глифов самых разных языков.

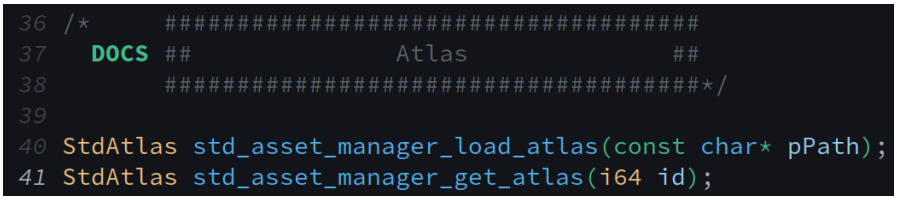

Рисунок 41. Функции для загрузки и работы с шрифтами

Загрузка звука происходит таким же способом, как и предыдущие. Здесь важно отметить, что получение объекта одного звука происходит через void указатель – это необходимо потому, что мы не хотим передавать зависимость на библиотеку miniaudio всем пользователям включающим данную библиот /home/bies/MidData/___DevTools/AI/video-upscaler/models/eky в свой исходный код (в данном случае пользователь – это программный файл).

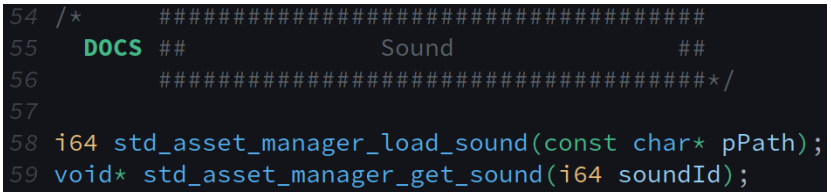

Рисунок 42. Функции для загрузки и работы со звуками

Asset Manager реализует эффективную загрузку и хранение ресурсов. Для идентификации строки, хранящий путь используется хеш-таблица, которая проводит ассоциацию между строкой и уже имеющегося id-значения внутри программы. Все ресурсы хранятся в массиве за счёт чего очень легко получать значения по id, то есть id – это ещё и индекс внутри массива. Следовательно, в лучшем случае получение значения через хеш-таблицу даёт нам эффективность O(1), где 1 - это сравнение двух строк, в среднем O(log N). Получение одного ресурса O(1), где 1 – это изъятие значение по индексу из массива, фактически это одно сложение и одно умножение (проверки и

валидации в данном случае выносятся за скобки, но они не сильно добавляют сложность алгоритму). Подобные значения очень эффективны, поэтому мы можем назвать наш AssetManager высокоэффективным.

Глава 3 Оценка эффективности разработанного программного обеспечения

Разработанное кроссплатформенное программное обеспечение, базирующееся на Vulkan API для создания интерактивных приложений, представляет собой важный шаг в решении проблем оптимизации рабочего процесса разработчиков и улучшения производительности графических приложений. Оценка эффективности включает в себя несколько ключевых аспектов: производительность, экономическая выгода, гибкость и функциональность.

Производительность: Использование Vulkan API, многопоточности и эффективной работы с памятью, а также поддержка различных операционных систем (Windows, Linux, macOS) обеспечивает значительный прирост производительности.

Экономическая эффективность: Разработка собственной библиотеки для обработки окон и графики снижает зависимость от сторонних решений, таких как Unreal Engine и Unity, что напрямую сокращает затраты на лицензии. Также отмечается сокращение расходов за счет глубокой оптимизации, улучшения производительности, и обеспечения безопасности через полный контроль над кодовой базой. Внедрение собственного ПО в долгосрочной перспективе снижает лицензионные затраты и способствует экономии на адаптации сторонних технологий.

Ниже приведен график, показывающий как изменяются затраты компании на разработку, если она введет в производство ваше приложение.

Красная линия (без ПО) отображает неизменные затраты на разработку в течение 6 месяцев, где расходы остаются стабильными, поскольку компания продолжает использовать традиционные методы разработки.

Синяя линия (с ПО) показывает, как с внедрением разработанного ПО затраты постепенно снижаются. Благодаря оптимизации процессов и автоматизации рабочих задач, затраты на разработку начинают уменьшаться с

каждого месяца, что делает проект более экономически эффективным.

График на рисунке 43иллюстрирует, как использование ПО помогает снизить затраты на разработку, что, в свою очередь, повышает общую рентабельность проекта для компании.

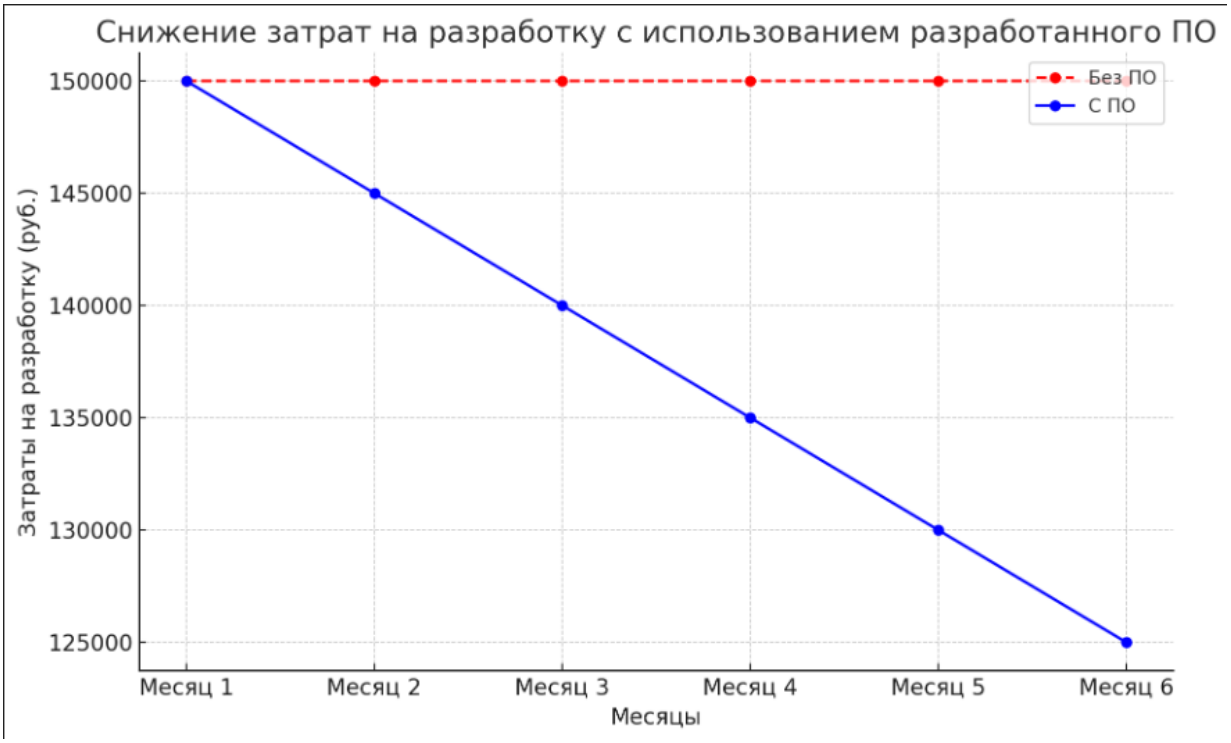

Рисунок 43 – График затрат на разработку

Гибкость и адаптивность: Разработанное ПО обладает высокой гибкостью благодаря поддержке кроссплатформенных технологий и возможности адаптации под специфические нужды компании. Это особенно важно в условиях быстро меняющихся технологических и бизнес-реалий, когда требуется быстрое внедрение новых функциональностей и решение нестандартных задач.

Функциональность и требования: Программное обеспечение соответствует заявленным требованиям: поддержка высокопроизводительных графических приложений, создание и управление окнами, работа с различными типами ресурсов (изображения, шрифты, звуки). Кроме того,

система профилирования и отладки позволяет эффективно мониторить производительность и выявлять узкие места в работе программы.

Проведя анализ функциональности и экономической выгоды разработанного программного продукта, был сделан вывод, что оно полностью удовлетворяет требования заказчика, точнее компании.

Разработка кроссплатформенной библиотеки на базе Vulkan API обеспечивает высокую производительность, снижает расходы на разработку, повышает гибкость и безопасность, а также открывает перспективы для дальнейшего улучшения и масштабирования приложений. Все эти факторы делают разрабатываемое ПО экономически выгодным и эффективным инструментом для решения задач компании, а также для улучшения конкурентоспособности на рынке.

Заключение

Выпускная квалификационная работа была посвящена разработке кроссплатформенной графической библиотеки на основе Vulkan API для создания интерактивных приложений.

Цель работы – создание высокопроизводительного, гибкого и независимого от сторонних решений инструмента – достигнута.

Библиотека обеспечивает:

- кроссплатформенность (поддержка Windows, Linux, macOS);
- высокую производительность благодаря оптимизированному рендерингу через Vulkan и использованию Data-Oriented Design (DoD);
- минимальную зависимость от внешних библиотек (использованы только STB и Miniaudio).

Автоматизацию рутинных задач (управление окнами, ресурсами, событиями).

Результаты соответствуют поставленным задачам:

- анализ существующих решений: выявлены недостатки коммерческих движков (Unreal, Unity) — лицензионные ограничения, сложность адаптации;
- изучение Vulkan API: реализован низкоуровневый контроль графического конвейера с поддержкой многопоточности;
- архитектура библиотеки: применена слоистая модель (по аналогии с Frostbite Engine), что обеспечило модульность и масштабируемость;
- реализация компонентов: созданы ядро (SWL для окон, SVL для Vulkan), система ресурсов (Asset Manager), модули UI и звука;
- тестирование: проверена стабильность на разных ОС, профилирована производительность (FPS, время загрузки ресурсов);

– демонстрационное приложение: игра подтвердила работоспособность библиотеки.

Научная новизна работы заключается в комбинации Vulkan API с минималистичным стеком технологий (C99, STB, Miniaudio), что снижает накладные расходы и реализации слоистой архитектуры, вдохновленной Frostbite Engine, но адаптированной для малых проектов.

Практическая ценность заключается в снижение затрат на лицензии, независимости от санкционных рисков и ускорение разработки за счет готовых модулей (рендеринг, UI, звук).

Работа представляет законченное решение, сочетающее производительность, гибкость и экономическую эффективность, с четкими перспективами развития.

Список используемой литературы и используемых источников

1. Мартин Р. Чистая архитектура: Искусство разработки программного обеспечения. - М.: ДМК Пресс, 2020. - 352 с.
2. Тихомиров А. C99: системное программирование на C. - СПб.: Питер, 2021. - 320 с.
3. Романов В. Архитектура игровых движков. - СПб.: БХВ-Петербург, 2020. - 400 с.
4. Barrett S. stb - Single-file public domain libraries for C/C++ [Электронный ресурс]. - Режим доступа: [https://github.com/nothings/stb] (https://github.com/nothings/stb), свободный. – Дата обращения: 11.05.2025.
5. Blow J. Preventing the Collapse of Civilization [Электронный ресурс]: доклад DevGAMM, 2019. - Режим доступа: [https://www.youtube.com/watch?v=pW-SOdj4Kkk] (https://www.youtube.com/watch?v=pW-SOdj4Kkk), свободный. - Дата обращения: 11.05.2025.
6. Cherno Y. TheCherno Project [Электронный ресурс]. - YouTube. - Режим доступа: [https://youtube.com/@TheCherno] (https://youtube.com/@TheCherno), свободный. - Дата обращения: 11.05.2025.
7. EA DICE. Frostbite Engine Technical Overview [Электронный ресурс]. - Режим доступа: [https://www.ea.com/frostbite] (https://www.ea.com/frostbite), свободный. - Дата обращения: 11.05.2025.
8. GDC Vault. Modern Rendering Architectures: Lessons from Godot, Frostbite, and Unity [Электронный ресурс]: доклад GDC 2023. – Режим доступа: [https://www.gdcvault.com/] (https://www.gdcvault.com/), свободный. – Дата обращения: 11.05.2025.
9. GitHub Blog. Trends in Lightweight C Libraries [Электронный ресурс]. - 2022. - Режим доступа: [https://github.blog/] (https://github.blog/), свободный. - Дата обращения: 11.05.2025.

10. Glatzel S. Vulkan Cookbook. – Birmingham: Packt Publishing, 2021. – 350 с.

11. Godot Engine. Документация [Электронный ресурс]. - Режим доступа: [https://docs.godotengine.org] (https://docs.godotengine.org), свободный. - Дата обращения: 11.05.2025.

12. ISO/IEC. Programming Languages - C (C18): ISO/IEC 9899:2018. - Geneva: ISO, 2018. – 552 с.

13. Khronos Group. Vulkan 1.3 Specification [Электронный ресурс]. – 2022. – Режим доступа: [https://registry.khronos.org/vulkan/] (https://registry.khronos.org/vulkan/), свободный. – Дата обращения: 11.05.2025.

14. Khronos Group. SPIR-V Specification [Электронный ресурс]. - 2023. - Режим доступа: [https://www.khronos.org/spir/] (https://www.khronos.org/spir/), свободный. - Дата обращения: 11.05.2025.

15. Linux Foundation. Xlib Manual Pages [Электронный ресурс]. - 2023. - Режим доступа: [https://www.x.org/releases/current/doc/libX11/] (https://www.x.org/releases/current/doc/libX11/), свободный. - Дата обращения: 11.05.2025.

16. Microsoft. Win32 API Documentation [Электронный ресурс]. - Режим доступа: [https://learn.microsoft.com/en-us/windows/win32/api/] (https://learn.microsoft.com/en-us/windows/win32/api/), свободный. - Дата обращения: 11.05.2025.

17. Miller J., Ginsburg K. Learning Vulkan. - Birmingham: Packt Publishing, 2020. - 310 с.

18. Muratori C. Handmade Hero [Электронный ресурс]. - Режим доступа: [https://handmadehero.org/] (https://handmadehero.org/), свободный. - Дата обращения: 11.05.2025.

19. Muratori C. The 30 Million Line Problem [Электронный ресурс]. - 2020. - Режим доступа: [https://mollyrocket.com/casey/stream_0014.html] (https://mollyrocket.com/casey/stream_0014.html), свободный. - Дата обращения: 11.05.2025.

20. Pignole M. Vulkan Programming Guide: The Official Guide to Learning Vulkan. - Boston: Addison-Wesley, 2020. - 496 с.

21. Reid D. miniaudio: Single-file audio playback and capture library [Электронный ресурс]. - Режим доступа: [https://miniaud.io/] (https://miniaud.io/), свободный. - Дата обращения: 11.05.2025.

22. Unreal Engine. Vulkan Support Documentation [Электронный ресурс]. - Режим доступа: [https://docs.unrealengine.com] (https://docs.unrealengine.com), свободный. - Дата обращения: 11.05.2025.

23. Unity Technologies. Graphics API Support [Электронный ресурс]. - Режим доступа: [https://docs.unity3d.com] (https://docs.unity3d.com), свободный. - Дата обращения: 11.05.2025.

24. Open Source Insights. Analysis of C Cross-Platform Graphics Stacks [Электронный ресурс]. – 2023. – Режим доступа: [https://deps.dev/] (https://deps.dev/), свободный. – Дата обращения: 11.05.2025.

25. Valve. Source Engine Documentation [Электронный ресурс]. - Режим доступа: [https://developer.valvesoftware.com/wiki/Main_Page] (https://developer.valvesoftware.com/wiki/Main_Page), свободный. - Дата обращения: 11.05.2025.