МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»

Высшая школа электроники и компьютерных наук
Кафедра «Электронные вычислительные машины»

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой ЭВМ
_____ Д.В. Топольский
«___»_____ 2025 г.

Динамически изменяющий визуализатор процесса работы двигателя внутреннего сгорания

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУРГУ-090301.2025.405 ПЗ ВКР

Руководитель работы,
к.т.н., доцент каф. ЭВМ
_____ Д.В. Топольский
«___»_____2025г.

Автор работы,
студент группы КЭ-405
_____ А.А. Пимонов
«___»_____2025 г.

Нормоконтролёр,
ст. преподаватель каф. ЭВМ
_____ С.В. Сяськов
«___»_____2025 г.

Челябинск-2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Высшая школа электроники и компьютерных наук
Кафедра «Электронные вычислительные машины»

УТВЕРЖДАЮ
Заведующий кафедрой ЭВМ
_____ Д.В. Топольский
«___»_____2025 г.

ЗАДАНИЕ

на выпускную квалификационную работу бакалавра

студенту группы КЭ-405
Пимонову Андрею Александровичу
обучающемуся по направлению
09.03.01 «Информатика и вычислительная техника»

1. **Тема работы: «**Динамически изменяющий визуализатор процесса работы двигателя внутреннего сгорания» утверждена приказом по университету от «21» апреля 2025 г. № 648-13/12.

2. **Срок сдачи студентом законченной работы:** 01 июня 2025 г.

3. **Исходные данные к работе:**

3.1. Платформа приложения: персональный компьютер.

3.2. Операционная система: Windows 7+ или GNU/Linux.

3.3. Используемые языки программирования: C, C++.

4. Перечень подлежащих разработке вопросов:

1. Аналитический обзор современной научно-технической, нормативной, методической литературы, затрагивающей исследуемую научно-техническую проблему.
2. Разработка архитектуры прораммы.
3. Создание инструментов для реализации основного функционала программы.
4. Проведение тестирования.

5. Дата выдачи задания: 2 декабря 2024 г.

Руководитель работы _____/ Д.В. Топольский/

Студент _____/ А.А. Пимонов/

КАЛЕНДАРНЫЙ ПЛАН

Этап	Срок сдачи	Подпись руководителя
Аналитический обзор современной научно-технической, нормативной, методической литературы, затрагивающей исследуемую научно-техническую проблему	03.03.2025	
Разработка архитектуры программы	22.03.2025	
Создание инструментов для реализации основного функционала программы	15.04.2025	
Проведение тестирования	29.04.2025	
Компоновка текста работы и сдача на нормоконтроль	25.05.2025	
Подготовка презентации и доклада	30.05.2025	

Руководитель работы _____/ *Д.В. Топольский* /

Студент _____/ *А.А. Пимонов* /

Аннотация

Пимонов А.А. Динамически изменяющий визуализатор процесса работы двигателя внутреннего сгорания. – Челябинск: ФГАОУ ВО «ЮУрГУ (НИУ)», ВШ ЭКН; 2025, 86 с., 13 ил., библиогр. список – 18 наим.

Тема выпускной квалификационной работы – Динамически изменяющий визуализатор процесса работы двигателя внутреннего сгорания.

В ходе работ был проведён аналитический обзор аналогов и основных технологический решений для разработки решения. Определены основные требования к проекту, как функциональные, так и нефункциональные.

Программа была спроектирована, разработана и протестирована на соответствие запланированным требованиям.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Двигатели внутреннего сгорания (ДВС) остаются основным источником механической энергии во многих отраслях, включая автомобильную промышленность, энергетическое оборудование и транспортные системы. Сложность их конструкции и необходимость оптимизации работы требуют использования современных инструментов для анализа и визуализации внутренних процессов.

Динамически изменяющий визуализатор процесса работы ДВС представляет собой инструмент, позволяющий в реальном времени отображать ключевые этапы и параметры работы двигателя. Такой подход обеспечивает более глубокое понимание его механики, процессов горения, термодинамики и взаимодействия элементов. Это особенно важно для обучения специалистов, проведения научных исследований и диагностики неисправностей.

Целью разработки таких визуализаторов является не только представление данных в наглядной форме, но и возможность анализа взаимосвязей между параметрами, что способствует улучшению рабочих характеристик двигателя, сокращению выбросов и повышению топливной эффективности.

Данная работа посвящена созданию и изучению системы визуализации, способной адаптироваться к изменениям параметров в реальном времени, что обеспечивает динамическую демонстрацию происходящих в ДВС процессов.

1. АНАЛИТИЧЕСКИЙ ОБЗОР НАУЧНО-ТЕХНИЧЕСКОЙ, НОРМАТИВНОЙ И МЕТОДИЧЕСКОЙ ЛИТЕРАТУРЫ ПО ТЕМАТИКЕ РАБОТЫ

1.1. КОМПЬЮТЕРНАЯ ВИЗУАЛИЗАЦИЯ ДАННЫХ

Компьютерная графика – это раздел информатики, изучающий методы и алгоритмы визуального представления данных. Её прогресс зависит от двух ключевых аспектов: запросов пользователей и уровня развития аппаратных и программных технологий. По мере роста этих факторов компьютерная графика находит всё более широкое применение в разных областях [1].

Основные сферы её использования включают:

– визуализация данных;

– моделирование процессов и явлений;

– проектирование технических объектов;

– организация пользовательского интерфейса.

Визуализация данных – это представление информации в графическом формате (графики, диаграммы, карты и др.), которое упрощает её восприятие и анализ. В отличие от таблиц с числовыми значениями, графическое отображение делает закономерности, тренды и аномалии более очевидными.

Виды представления информации:

– 1D–визуализация (гистограмма, секторная диаграмма);

– 2D–визуализация (точечная диаграмма, векторная диаграмма, топографическая карта);

– 3D–визуализация (предметная визуализация, техническая визуализация.

Визуализация данных – инструмент, который экономит время, улучшает аналитику и делает информацию доступной даже для неподготовленной аудитории. Грамотный выбор формата представления данных повышает эффективность исследований, отчётов и бизнес–решений [2].

Компьютерное моделирование – это исследование сложных систем через их цифровые аналоги. Современные графические технологии позволяют создавать сложные динамические и анимационные изображения, что особенно востребовано в системах моделирования (симуляторах). Эти системы воспроизводят процессы и явления, которые существуют или потенциально возможны в реальном мире. Оно позволяет получать количественные и качественные данные о системе, изучать её неочевидные свойства до реализации в реальности, оптимизировать проектирование, сокращая затраты на физические эксперименты [3].

Практическое применение компьютерного моделирования охватывает практически все области современной науки и техники. В аэрокосмической отрасли методы вычислительной гидродинамики позволяют моделировать аэродинамические характеристики и тепловые режимы конструкций. Биомедицинские приложения включают моделирование физиологических процессов и виртуальные испытания лекарственных препаратов. Концепция "умных городов" опирается на транспортное моделирование и оптимизацию энергопотребления, демонстрируя междисциплинарный потенциал методов компьютерного моделирования.

Современное компьютерное моделирование представляет собой динамично развивающуюся междисциплинарную область, объединяющую фундаментальные математические методы, передовые вычислительные технологии и предметные знания. Перспективные направления развития области включают технологию цифровых двойников, реализующую концепцию сквозного моделирования в промышленности [4].

Проектирование технических объектов. Проектирование представляет собой ключевой этап в процессе создания технических изделий, определяющий их функциональность, надежность и технологичность. В современных условиях этот процесс претерпел значительные изменения благодаря внедрению компьютерных технологий, которые не только ускорили разработку, но и повысили ее качество.

Современные системы автоматизированного проектирования (CAD) обеспечивают детальную визуализацию объектов, позволяя конструкторам работать с трехмерными моделями высокой точности. Такой подход существенно облегчает анализ проектных решений, поскольку разработчик получает возможность не только оперировать числовыми параметрами, но и визуально оценивать форму, компоновку и взаимодействие деталей. Кроме того, современные CAD–системы предоставляют инструменты для интерактивного взаимодействия с моделью.

Пользовательский интерфейс (User Interface, UI) – это функционал, через который пользователь взаимодействует с системой. UI содержит такие элементы управления, как кнопки, меню и поля ввода, чтобы облегчить использование продукта и сделать его интуитивно понятным. За последнее десятилетие произошла значительная трансформация в подходах к организации взаимодействия между человеком и компьютером. Визуальная парадигма интерфейсов стала не просто преобладающей, но и фундаментальной основой цифрового взаимодействия. Современные операционные системы, такие как Microsoft Windows, Apple macOS и различные дистрибутивы GNU/Linux, в обязательном порядке включают графическую подсистему с оконным интерфейсом, ставшим фактическим стандартом взаимодействия с компьютером [5].

Стандартизация элементов управления – кнопок, меню, полей ввода и других компонентов интерфейса – привела к формированию устойчивых пользовательских ожиданий и паттернов поведения. Это, с одной стороны, упрощает процесс освоения новых приложений, а с другой – накладывает определенные ограничения на разработчиков, вынужденных балансировать между инновациями и соблюдением принципов узнаваемости интерфейса.

Для реализации интерфейса, поддерживающего работу с большими объемами данных, существуют специализированные графические подсистемы и библиотеки, такие как OpenGL, DirectX или Vulkan, которые предоставляют аппаратное ускорение графических операций, поддержку сложных трехмерных сцен, гибкие механизмы работы с текстурами и шейдерами.

1.2. АНАЛИЗ ОСНОВНЫХ ТЕХНОЛОГИЧЕСКИХ РЕШЕНИЙ

В данном разделе рассматриваются возможные технологии и инструменты, применимые в рамках проекта. Такой анализ позволяет оценить преимущества и недостатки каждого варианта, а также подобрать наиболее подходящий комплект программных решений для выполнения поставленных целей. От выбора используемых технологий во многом зависит успех проекта. Грамотно подобранные инструменты не только ускоряют и облегчают процесс разработки, но и обеспечивают стабильную работу и возможность дальнейшего расширения создаваемого приложения.

C++ – это универсальный язык программирования, который появился в начале 1980–х годов как расширение языка C. Разработанный Бьёрном Страуструпом, он добавил к классическому C объектно–ориентированные возможности, такие как классы, наследование и полиморфизм, сохранив при этом эффективность и низкоуровневый контроль, характерные для своего предшественника. Одной из ключевых особенностей C++ является его многопарадигменность. Он поддерживает процедурное, объектно–ориентированное, обобщённое (шаблоны) и даже функциональное программирование, что делает его гибким инструментом для решения самых разных задач. Благодаря этому на C++ пишут операционные системы, игры, высокопроизводительные вычисления, встраиваемые системы и многое другое [6].

Синтаксис C++ унаследован от C, но значительно расширен. Классы позволяют инкапсулировать данные и методы, работающие с ними, а механизмы наследования и виртуальных функций дают возможность строить сложные иерархии объектов. Шаблоны (template) вводят обобщённое программирование, позволяя писать универсальные алгоритмы, работающие с разными типами данных без потери производительности.

Управление памятью может быть как ручным, так и автоматизированным. В отличие от языков с полной сборкой мусора, программист может сам решать, когда выделять и освобождать память, что критично для задач, требующих максимальной эффективности. Однако современные стандарты (начиная с C++11) предлагают умные указатели (smart pointers), которые минимизируют риски утечек памяти, сохраняя контроль над ресурсами.

C++ – один из основных языков для разработки графических приложений, где важна высокая производительность и точный контроль над ресурсами. Благодаря своей скорости и эффективности он широко используется в создании игр, профессиональных графических редакторов, систем компьютерного зрения. Графические приложения часто работают с низкоуровневыми API, такими как OpenGL, Vulkan или DirectX, которые требуют ручного управления памятью и оптимизации вызовов. В отличие от языков с автоматическим управлением ресурсами, C++ позволяет минимизировать накладные расходы, что критично для плавного рендеринга сложных сцен. Например, игровые движки вроде Unreal Engine построены на этом языке именно потому, что он обеспечивает предсказуемую производительность и возможность тонкой настройки каждого аспекта работы с графикой [7].

C++ даёт больше возможностей для сложных абстракций, но C остаётся незаменимым там, где важны минимализм, полный контроль и предсказуемость. Если задача не требует ООП или обобщённого программирования, C может оказаться более эффективным выбором.

C – это небольшой язык с минимальным набором ключевых слов и простой структурой. В нём нет классов, шаблонов, исключений, перегрузки операторов и других сложных механизмов C++, что делает его легче для изучения и анализа. Компиляторы C обычно работают быстрее, а код компилируется с меньшими накладными расходами. Программы на C часто компилируются в более компактный машинный код, так как в них нет накладных расходов на RTTI (динамическую идентификацию типов), исключения или виртуальные таблицы [8]. Поэтому части приложения, не требующие ООП и имеющие потребность в быстроте вычислений целесообразней реализовывать на языке C.

OpenGL – это кроссплатформенный графический API, предназначенный для рендеринга 2D и 3D графики с аппаратным ускорением. Разработанный в 1992 году компанией Silicon Graphics, он стал одним из основных инструментов в компьютерной графике, используемым в играх, научной визуализации, CAD–системах и мультимедийных приложениях [9].

Данная технология позволяет задавать геометрию объектов, их координаты в пространстве, а также дополнительные параметры (ориентацию, размер и т. д.), определяет визуальные свойства (цвет, текстуру, материал и пр.) и положение камеры, берёт на себя преобразование этих данных в изображение на экране.

Основные возможности OpenGL:

– геометрические и растровые примитивы;
– видовые и модельные преобразования;
– работа с цветом;
– наложение текстуры;
– сглаживание;
– работа с тенями;
– освещение.

Программы, написанные с помощью OpenGL можно успешно перенести на такие платформы как Unix, Linux, SunOS, IRIX, Microsoft Windows, Apple MacOS и многие другие [10].

ImGui (Dear ImGui) – это популярная библиотека для создания графических интерфейсов в реальном времени, разработанная специально для интеграции в приложения на C++. В отличие от традиционных GUI-фреймворков, таких как Qt или wxWidgets, ImGui следует парадигме "immediate mode" (немедленного режима), где интерфейс не хранит состояние элементов, а перерисовывается каждый кадр. Такой подход делает его идеальным инструментом для встроенных редакторов, инструментов отладки и визуализации данных в играх, графических приложениях и симуляторах.

Библиотека задумывалась как минималистичная, но мощная альтернатива для разработчиков, которым нужен лёгкий в интеграции интерфейс без сложных зависимостей. ImGui рисует элементы управления напрямую через низкоуровневые графические API, такие как OpenGL, DirectX или Vulkan, что позволяет достичь высокой производительности даже при частых обновлениях интерфейса.

Одной из ключевых особенностей ImGui является его простота в использовании. Для создания кнопки, ползунка или окна достаточно всего нескольких строк кода. Например, типичный цикл рендеринга интерфейса выглядит так: сначала объявляется новое окно, затем добавляются элементы управления, а библиотека автоматически обрабатывает ввод пользователя (мышь, клавиатура) и возвращает состояние элементов (нажата ли кнопка, значение слайдера и т. д.). Всё это происходит без необходимости явно создавать обработчики событий или описывать сложные иерархии [11].

Несмотря на минималистичный дизайн, библиотека предоставляет богатый набор элементов: от базовых кнопок и текстовых полей до сложных деревьев узлов, таблиц и даже встроенных графиков. Стиль интерфейса легко настраивается – можно изменять цвета, шрифты и размеры элементов, чтобы адаптировать внешний вид под конкретный проект.

Библиотека широко используется в индустрии: от инструментов разработки игр (например, редакторы в Frostbite или Capcom RE Engine) до научных симуляторов и даже встроенных систем управления.

SDL2 (Simple DirectMedia Layer 2) – это кроссплатформенная библиотека, написанная на C, которая предоставляет разработчикам низкоуровневый доступ к аудио, клавиатуре, мыши, джойстику и графике через OpenGL или Direct3D. Созданная в 1998 году Сэмом Лантинга (Sam Lantinga), SDL2 стала стандартом для разработки игр, мультимедийных приложений и эмуляторов, сочетая простоту использования с высокой производительностью. Библиотека абстрагирует платформо–зависимые функции, позволяя писать код, который работает на Windows, Linux, macOS, Android и iOS без изменений.

Одно из ключевых преимуществ SDL2 – её модульная архитектура. Она разбита на несколько подсистем, каждая из которых отвечает за определённую функциональность. Например, SDL_Renderer предоставляет простой 2D–рендеринг с аппаратным ускорением, в то время как SDL_Vulkan и SDL_OpenGL позволяют интегрировать современные графические API. Для работы со звуком есть SDL_Audio, а для обработки событий ввода – SDL_Events.

Графика в SDL2 может выводиться как через программный рендеринг (SDL_Surface), так и через GPU (SDL_Texture + SDL_Renderer). Это делает библиотеку гибкой: она подходит и для простых 2D–игр, и для сложных проектов, где рендеринг выполняется через шейдеры OpenGL/Vulkan [12]. Например, SDL2 часто используют вместе с OpenGL, создавая окно через SDL_CreateWindow, а затем настраивая контекст для рендеринга.

Обработка ввода в SDL2 унифицирована для всех платформ. События клавиатуры, мыши, сенсорного экрана и геймпадов поступают в единую очередь (SDL_Event), которую можно опрашивать в цикле приложения.

Eigen – это библиотека линейной алгебры для C++, предоставляющая удобные и оптимизированные матричные и векторные операции. Она широко используется в машинном обучении, компьютерной графике, физических симуляциях и робототехнике. Eigen позволяет работать с матрицами, векторами, кватернионами и другими математическими структурами, поддерживает различные числовые типы и операции, такие как умножение матриц, разложение (SVD, LU, QR) и решение систем линейных уравнений. Библиотека известна своей высокой производительностью.

Boost.Units – часть библиотеки Boost, добавляющая в C++ систему единиц измерения на уровне типов. Она позволяет предотвратить ошибки, связанные с несоответствием единиц (например, сложение метров и секунд), ещё на этапе компиляции. С Boost.Units можно определять собственные физические величины (скорость, ускорение, энергию) и автоматически проверять их корректность в расчётах.

Bullet Physics – библиотека физического движка для симуляции динамики твёрдых тел, столкновений и мягких тканей. Используется в играх, виртуальной реальности, кинопроизводстве (для спецэффектов) и робототехнике. Bullet поддерживает обнаружение столкновений, расчёт физики твёрдых и деформируемых объектов, а также взаимодействие с системами частиц. Она интегрируется с OpenGL для визуализации.

Библиотека Cantera – это открытая программная библиотека, написанная на C++ и Python, которая предназначена для моделирования химической кинетики, термодинамики и транспортных процессов. Она широко используется в научных и инженерных расчетах, связанных с горением, химическими реакциями, теплообменом и течениями реагирующих газов. Cantera предоставляет мощные инструменты для работы с химическими механизмами, позволяя рассчитывать равновесные и неравновесные состояния, скорости реакций, коэффициенты переноса и другие параметры сложных химических систем.

Одним из ключевых преимуществ Cantera является ее способность интегрироваться с другими вычислительными инструментами, такими как CFD–программы (например, OpenFOAM), что позволяет проводить многомерное моделирование процессов горения и химического взаимодействия. Библиотека поддерживает различные форматы описания химических механизмов, включая CHEMKIN и XML, что упрощает использование готовых кинетических моделей.

Для работы с Cantera пользователь обычно создает сценарий на C++ или Python, в котором задает начальные условия (температуру, давление, состав смеси), выбирает модель (например, идеальный газовый реактор или пламя) и запускает расчет. Библиотека предоставляет удобные методы для доступа к результатам, которые затем можно визуализировать или использовать в дальнейших вычислениях.

Cantera особенно полезна в исследованиях горения, проектировании двигателей, каталитических процессов, а также в энергетике и экологии для анализа выбросов и оптимизации процессов сгорания. Ее открытый исходный код и активное сообщество разработчиков делают ее популярным инструментом в области computational fluid dynamics (CFD) и химического моделирования.

CMake – это кроссплатформенная система автоматизации сборки программного обеспечения, которая широко используется в проектах на C++ для управления процессом компиляции, линковки и развертывания приложений. Ее основное назначение – упростить создание переносимых и масштабируемых проектов, которые могут быть собраны на разных операционных системах (Windows, Linux, macOS) с различными компиляторами (GCC, Clang, MSVC и др.). CMake не занимается непосредственной сборкой, а генерирует файлы для систем сборки, таких как Make, Ninja, Visual Studio или Xcode, что делает его универсальным инструментом в разработке.

Одним из ключевых преимуществ CMake является его способность абстрагироваться от особенностей конкретных платформ и компиляторов. Вместо того чтобы писать отдельные Makefile’ы или файлы проектов для каждой системы, разработчик описывает структуру проекта в едином CMakeLists.txt, а CMake автоматически адаптирует сборку под нужную среду. Это особенно полезно в крупных проектах с множеством зависимостей, где ручная настройка сборки становится сложной и error–prone.

С помощью CMake можно эффективно управлять зависимостями, подключать внешние библиотеки (как системные, так и сторонние) и настраивать условную компиляцию (например, для разных версий программы или под разные архитектуры процессоров). Он поддерживает модульное построение проектов, позволяя разбивать код на логические компоненты с отдельными настройками сборки.

OpenAL (Open Audio Library) – это кроссплатформенный программный интерфейс для работы с трехмерным позиционным звуком, который часто используется в C++ проектах, связанных с игровой разработкой, виртуальной реальностью, аудиоприложениями и симуляторами. Библиотека предоставляет разработчикам удобные инструменты для пространственного аудиовоспроизведения, позволяя создавать эффекты удаленности, направления и движения звуковых источников, что значительно усиливает реалистичность и погружение в интерактивных приложениях. OpenAL абстрагирует низкоуровневые операции работы с аудиоустройствами, предоставляя единый API для разных операционных систем, что упрощает портирование проектов между платформами. OpenAL расширяем: программисты, либо компании, не входящие в число разработчиков OpenAL, могут добавлять в него свои расширения. Успешные расширения могут быть войти в спецификации OpenAL в её новой версии.

1.3. ОБЗОР АНАЛОГОВ

Программное обеспечение для численного моделирования (например, ANSYS, AVL FIRE, GT–SUITE) может быть использовано для целей визуализации работы ДВС и позволяет детально изучать процессы внутри него. Рассмотрим наиболее популярные и мощные программные решения для визуализации работы ДВС.

ANSYS – это мощный программный пакет для инженерного моделирования (CAE), который используется для численного анализа (FEA, CFD) в различных отраслях: машиностроение, аэрокосмическая промышленность, электроника.

CFD–симуляции: ANSYS Fluent и ANSYS CFX являются популярными инструментами для моделирования процессов течения жидкости и газа, теплообмена и сгорания в ДВС. С помощью этих инструментов можно моделировать поток воздуха и топлива в камере сгорания, а также взаимодействие этих веществ с теплотой и давлением, что помогает оптимизировать процессы сгорания и повысить эффективность работы двигателя [13].

ANSYS Thermal позволяет моделировать распределение температуры в различных частях двигателя, что важно для оценки эффективности системы охлаждения и предотвращения перегрева.

Моделирование выбросов: ANSYS также предлагает решения для анализа выбросов, включая создание моделей, которые могут предсказать уровень загрязняющих веществ, таких как оксиды азота (NOx), углеводороды (HC) и углекислый газ (CO2).

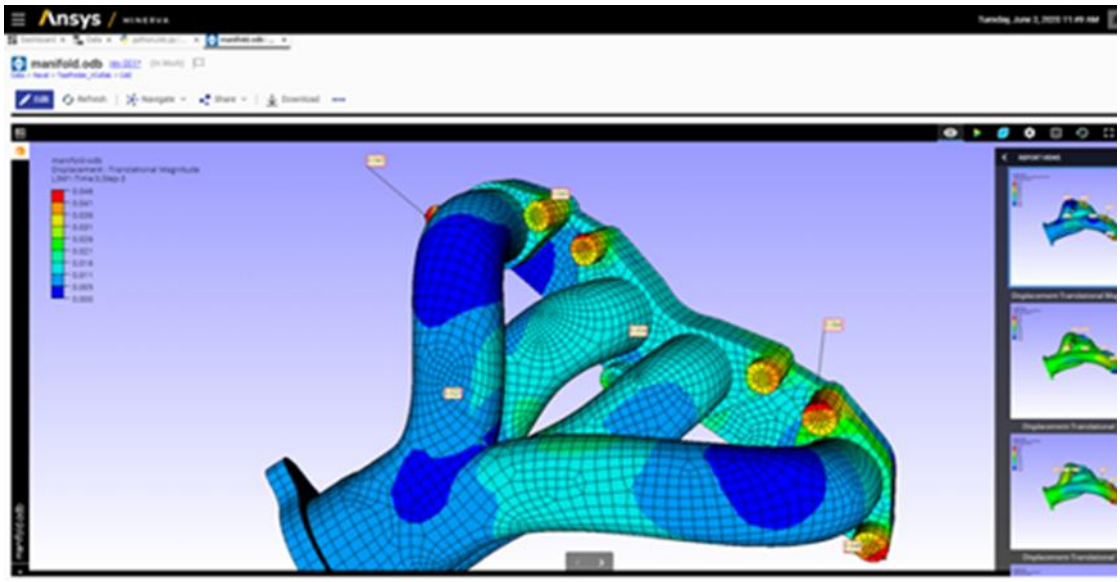

Рисунок 1 – Интерфейс программы ANSYS

Преимущества:

– высокая точность;
– широкий спектр функционала;
– интеграция с другими системами.

Недостатки:

– высокая стоимость;
– требует огромных вычислительных мощностей для работы;
– высокая сложность.

AVL FIRE – это профессиональный CFD–пакет, разработанный компанией AVL (ведущий мировой поставщик решений для двигателестроения). Он оптимизирован именно для задач, связанных с двигателями внутреннего сгорания (ДВС), трансмиссией и выхлопными системами, и широко используется автопроизводителями.

AVL FIRE позволяет моделировать сложные процессы сгорания, включая динамику пламени, распыление топлива и его испарение. Это помогает разработать более эффективные системы впрыска и камеры сгорания.

Программа поддерживает моделирование многофазных потоков, что позволяет детально изучить взаимодействие топлива и воздуха в различных режимах работы двигателя, таких как холодный пуск, ускорение и низкие обороты.

AVL FIRE позволяет моделировать теплообмен в двигателе, включая анализ работы системы охлаждения и прогрева. Это помогает выявить горячие зоны, где возможен перегрев, и оптимизировать конструкцию системы охлаждения [14].

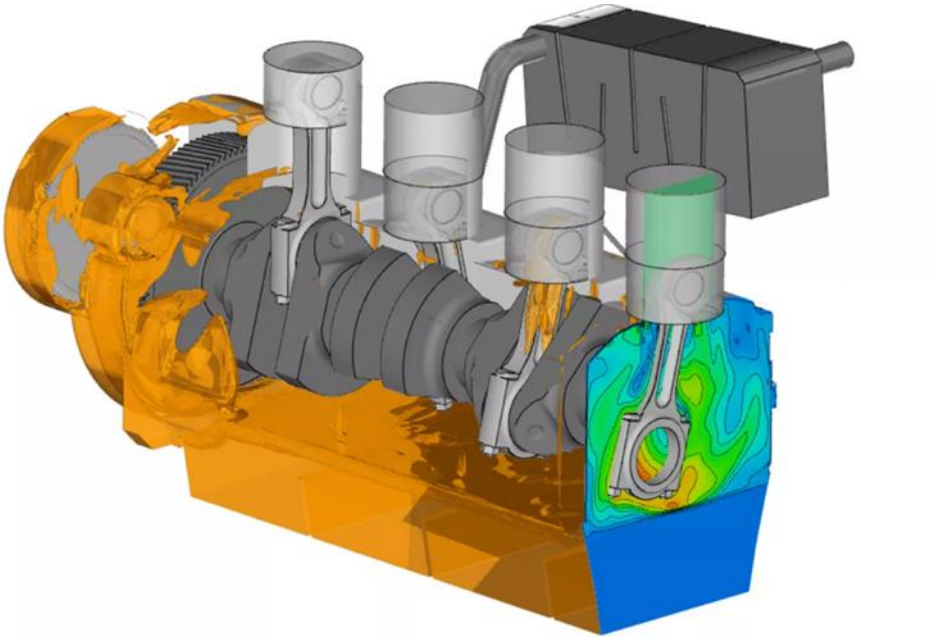

Рисунок 2 – Пример модели ДВС в программе AVL FIRE

Преимущества:

– специализация под ДВС;

– высокая точность;

– интеграция с другими инструментами AVL.

Недостатки:

– сложность настройки;

– дороговизна.

Automation: The Car Company Tycoon – это симулятор разработки автомобилей, где игрок выступает в роли инженера автопроизводителя. Хотя игра не является профессиональным инженерным ПО, её можно использовать для упрощённой визуализации работы ДВС и обучения базовым принципам двигателестроения.

Главная ценность игры – её интерактивный конструктор двигателей. Можно собирать моторы по деталям, меняя всё – от типа блока цилиндров до профиля кулачков распредвала, при этом сразу видеть, как эти изменения влияют на характер двигателя. Например, можно увеличить степень сжатия и моментально получить новую кривую мощности. В отличие от сложных профессиональных пакетов вроде ANSYS или AVL FIRE, здесь всё представлено в доступной форме [15].

Рисунок 3 – Интерфейс игры Automation: The Car Company Tycoon

Преимущества:

– интерактивность;
– простота использования;
– низкие системные требования;
– доступность.

Недостатки:

– низкая вычислительная точность;
– упрощенная физика.

1.4. ВЫВОД ПО ГЛАВЕ 1

Приведенные аналоги, такие как ANSYS и AVL FIRE слишком сложны и дороги для простой визуализации ДВС, их не получится использовать в образовательных целях и его избыточно использовать для простых оценочных расчётов или когда требуется быстро протестировать идею – сложность настройки и время вычислений здесь неоправданно высоки.

Automation – это игра симулятор для понимания основ двигателестроения. Но она совершенно не подходит для реальных расчётов – все физические процессы здесь существенно упрощены, а результаты нельзя считать достоверными для практического применения.

В результате анализа основных технологических решений были выбраны следующие инструменты для разработки приложения:

- языки программирования C и C++;
- библиотека OpenGL для реализации графики;
- библиотека ImGui для реализации интерфейса;
- библиотека SDL2 для обработки ввода;
- библиотека Eigen для работы с вычислениями;
- библиотека Boost для работы с физическими величинами;
- библиотека OpenAL для работы со звуком;
- CMake для сборки проекта;
- библиотека Cantera для симуляции физических процессов;
- библиотека Bullet Physics для сумуляции физических процессов.

2. РАЗРАБОТКА АРХИТЕКТУРЫ ПРОГРАММЫ

2.1. ОПРЕДЕЛЕНИЕ ФУНКЦИОНАЛЬНЫХ ТРЕБОВАНИЙ

Функциональные требования определяют конкретные действия и возможности, которые должна предоставлять система. Они описывают, что именно система должна делать, какие функции выполнять и как реагировать на различные входные данные или действия пользователя.

Программа должна содержать реализацию следующего функционала:

– визуализация работы основных компонентов ДВС (поршни, клапаны, коленвал);

– интерактивное управление (запуск и остоновка двигателя, регулировка оборотов);

– отображение графика расхода выхлопных газов, количества оборотов двигателя.

2.2. ОПРЕДЕЛЕНИЕ НЕФУНКЦИОНАЛЬНЫХ ТРЕБОВАНИЙ:

Нефункциональное требование определяет качественные характеристики системы и устанавливает границы ее работы. Оно описывает не конкретные действия системы, а то, как она должна их выполнять – с какими показателями производительности, степенью надежности, уровнем безопасности и другими атрибутами качества.

Должны обеспечиваться следующие нефункциональные требования:

– плавные анимации визуализации процесса работы ДВС;

– поддерживаемые ОС: Windows 7 и выше, Linux Debian 12 и выше;

– зыки разработки: C, C++;

– программа должна работать на компьютерах с процессором Intel i5 / AMD Ryzen 5 1500X и 8 ГБ оперативной памяти.

2.3. ВЫБОР АРХИТЕКТУРЫ

Оптимально выстроенная архитектура программного обеспечения является критически важным фактором, определяющим эффективность всего жизненного цикла разработки. Рациональная организация компонентов системы позволяет минимизировать как временные, так и финансовые затраты на этапах реализации, последующего сопровождения и модернизации программного продукта.

Архитектурные паттерны представляют собой формализованные решения распространённых проблем проектирования, накопленные в результате многолетней практики разработки сложных программных систем. Их применение способствует созданию более структурированного, предсказуемого и адаптируемого кода, что в конечном итоге приводит к повышению надёжности и снижению стоимости владения программным решением [16].

Особую ценность данные подходы приобретают при работе над крупномасштабными проектами, где важнейшую роль играют такие характеристики как модульность, масштабируемость и простота интеграции новых функциональных возможностей. Следует отметить, что эффективность применения архитектурных паттернов напрямую зависит от квалификации разработчиков и их способности правильно интерпретировать и адаптировать общие принципы к конкретным бизнес–задачам [17].

В рамках разработки визуализатора процесса работы двигателя внутреннего сгорания выгодно применить сразу несколько подходов к организации архитектуры. Гибридная архитектурная модель, сочетающая элементы компонентно–ориентированного и событийно–ориентированного подходов с элементами слоистой структуры. Выбор данной архитектурной парадигмы обусловлен необходимостью обеспечения высокой производительности в режиме реального времени, поддержки модульности системы и возможности последующего расширения функциональности.

Основу архитектуры составляет компонентно–ориентированный подход (Component-Based Architecture), который был выбран благодаря его естественному соответствию предметной области. Компонентно–ориентированная архитектура в проектировании систем относится к методологии, в которой программное обеспечение создается путем сборки предопределенных, повторно используемых компонентов [18]. Каждый компонент инкапсулирует определенную часть функциональности или поведения с четко определенными интерфейсами, которые управляют тем, как компоненты взаимодействуют друг с другом. Этот подход способствует модульности, гибкости и возможности повторного использования в разработке программного обеспечения.

Физические сущности, такие как поршни, цилиндры и клапаны реализованы в виде отдельных компонентов, инкапсулирующих как данные, так и поведение. Это позволяет достичь высокой степени связности внутри компонентов при минимальной зависимости между ними. Каждый физический компонент (например, поршень или клапан) содержит исключительно логику, относящуюся к его функциональному назначению, что соответствует принципу единой ответственности.

Дополнением к компонентной модели стал событийно–ориентированный подход (Event–Driven Architecture, EDA), который был применен для организации взаимодействия между компонентами.

Этот подход был выбран для решения следующих задач:

– уменьшение связанности между модулями;
– синхронизация физики, графики и звука;
– гибкость при добавлении нового функционала.

Архитектура, основанная на событиях, представляет собой концепцию построения программных систем, в которой взаимодействие между отдельными модулями осуществляется посредством асинхронного обмена сообщениями-событиями. Данная парадигма предполагает декомпозицию системы на автономные компоненты, каждый из которых функционирует независимо и реагирует исключительно на релевантные для его функциональности события.

Структурно система, реализующая событийно-ориентированный подход, включает три ключевых элемента. Первый элемент – источник событий, ответственный за инициирование и генерацию сообщений. Второй элемент представляет собой подсистему обработки, осуществляющую идентификацию входящих событий и выполнение соответствующих бизнес–процессов. Третий критически важный компонент – механизм маршрутизации событий, обеспечивающий надежную доставку сообщений между производителями и потребителями.

Использование такого подхода в проекте позволяет реализовать механизм слабой связанности, когда компоненты взаимодействуют через генерацию и обработку событий, не имея прямых ссылок друг на друга. Например, событие зажигания (IgnitionEvent) автоматически уведомляет как звуковую систему о необходимости воспроизведения соответствующего звука, так и графическую подсистему для визуализации вспышки в цилиндре.

Слоистая архитектура предлагает деление программного обеспечения на отдельные уровни (слои), каждый из которых выполняет строго определенный набор функций. Главная цель такого подхода – обеспечить независимость и модульность компонентов, а также четкую организацию кода для оптимизации разработки, масштабирования и поддержки приложений.

В рамках данной архитектурной модели система структурируется в виде набора горизонтальных уровней, где каждый уровень выполняет строго определенную функциональную роль в рамках общей архитектуры приложения. Такое разделение предполагает четкое распределение обязанностей между уровнями – от уровня представления данных до уровня реализации бизнес–логики и уровня доступа к данным.

Компоненты организованы в горизонтальные слои, каждый из которых выполняет определённую роль в приложении, что дополняет концепцию компонентно–ориентированного подхода.

Ключевые слои программы:

– слой физической сиимуляции;
– слой графической системы;
– слой пользовательского интерфейса;
– слой звуковой системы.

Спроектированная UML-диаграмма классов архитектуры программы представленна на рисунке 4.

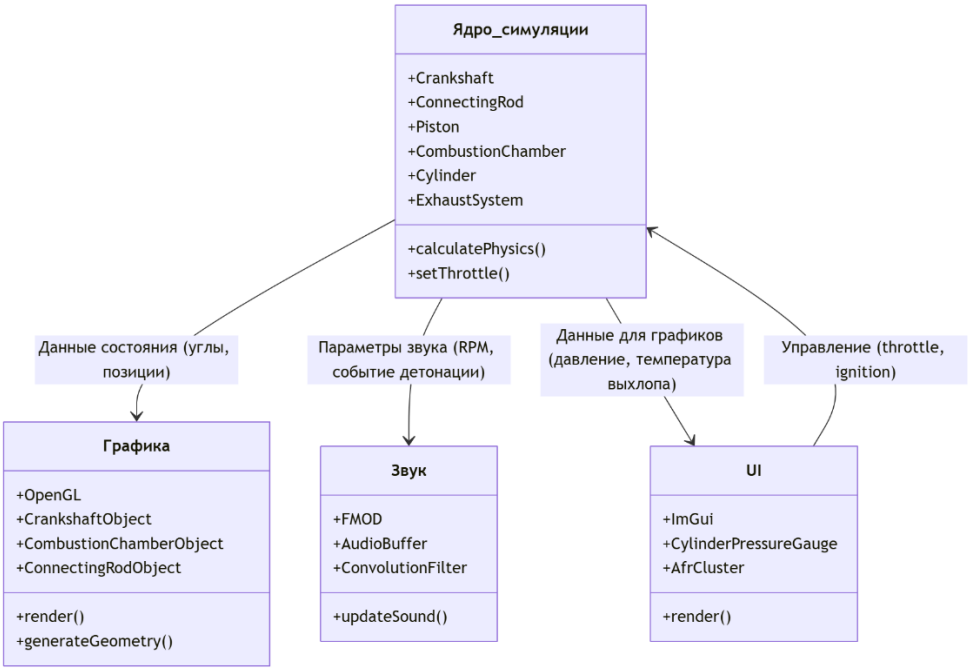

Рисунок 4 – UML-диаграмма классов архитектуры программы

Ядро симуляции представляет собой фундаментальный слой системы, содержащий физическую модель двигателя. В его состав будут входить такие ключевые компоненты, как Crankshaft (коленчатый вал), Piston (поршень), ConnectingRod (шатун), CombustionChamber (камера сгорания) и другие. Каждый из этих классов инкапсулирует строго определенную физическую логику работы соответствующего узла двигателя. Ядро предоставляет данные другим слоям системы через два основных механизма: непосредственный доступ к полям классов и специализированные методы получения состояния (getAngle(), getPressure() и аналогичные). При этом важно отметить, что ядро полностью независимо от других слоев системы и не содержит никаких ссылок на графику, звук или пользовательский интерфейс.

Графическая система отвечает за визуализацию работы двигателя в реальном времени. Основу этого слоя будут составлять специализированные *Object-классы (CrankshaftObject, PistonObject и другие), которые преобразуют физические данные, получаемые из ядра, в трехмерные модели. Каждый такой класс будет реализовывать методы generateGeometry() для создания геометрического представления соответствующего компонента двигателя и render() для его отрисовки. Данный слой строго зависит только от ядра симуляции, получая от него все необходимые данные о состоянии системы, но при этом не оказывая никакого обратного влияния на физическую модель и не взаимодействуя напрямую с другими слоями приложения.

Звуковая система реализует генерацию звуков работы двигателя на основе физических параметров. Этот слой активно использует данные о текущих оборотах двигателя и давлении в цилиндрах, получаемые из ядра симуляции.

Пользовательский интерфейс обеспечивает взаимодействие с пользователем и визуализацию диагностической информации. В отличие от других слоев, UI выполняет двунаправленное взаимодействие с ядром системы: с одной стороны, он будет получает данные через методы доступа, а с другой – передавать управляющие команды.

2.4. ВЫВОД ПО ГЛАВЕ 2

В ходе проектирования системы были сформулированы функциональные и нефункциональные требования, определяющие ключевые характеристики программы.

Принятые архитектурные решения обеспечивают четкое разделение ответственности между компонентами, высокую производительность вычислений и гибкость для дальнейшего расширения функционала.

3. СОЗДАНИЕ ИНСТРУМЕНТОВ ДЛЯ РЕАЛИЗАЦИИ ОСНОВНОГО ФУНКЦИОНАЛА ПРОГРАММЫ

3.1. РЕАЛИЗАЦИЯ ЯДРА СИМУЛЯЦИИ

Используя выбранные архитектурные паттерны реализуем ключевые классы и компоненты программы, отвечающие за симуляцию работы двигателя внутреннего сгорания.

Физическая модель двигателя построена на комбинации нескольких специализированных библиотек. Механика взаимодействия деталей (поршней, шатунов, коленчатого вала) реализована с использованием библиотеки Bullet Physics для моделирования твердых тел и механических систем, что позволило корректно воспроизвести кинематику работы кривошипно–шатунного механизма.

Термодинамические процессы в цилиндрах, включая сжатие топливно-воздушной смеси, воспламенение и расширение газов, рассчитываются при помощи библиотеки химико–термодинамического моделирования Cantera, которая учитывает реальные свойства рабочих тел и химические реакции при сгорании.

Интеграция этих библиотек в единую систему потребовала разработки промежуточного слоя, обеспечивающего обмен данными между подсистемами. Например, давление, рассчитанное в Cantera, передаётся в Bullet Physics для определения силы, действующей на поршень, а полученная скорость коленвала влияет на частоту звука.

Сначала термодинамическая модель в Cantera рассчитывает параметры газа после сгорания топливной смеси, включая давление и температуру. Эти данные преобразуются в механическую силу, которая воздействует на поршень в физическом движке Bullet Physics. Здесь учитывается площадь поршня для корректного перевода давления в ньютоны.

После шага симуляции в Bullet Physics, который определяет новое положение и скорость поршня, эти данные используются для управления звуковым сопровождением через OpenAL. Частота звука двигателя напрямую зависит от скорости движения поршня, что создаёт эффект реалистичного нарастания оборотов. Визуализация, представленная условной функцией updatePistonPosition, синхронизирована с физической симуляцией через получение актуальных координат из Bullet Physics.

Особенностью данной реализации является использование дельта-времени (deltaTime) для синхронизации расчётов между подсистемами. DeltaTime – это фундаментальное понятие в физическом моделировании и игровых движках, обозначающее временной шаг между двумя последовательными кадрами симуляции. Если представить работу двигателя в реальном времени, то его состояние непрерывно меняется, но компьютер может обрабатывать эти изменения только дискретно (пошагово). DeltaTime как раз и определяет, сколько реального времени прошло между текущим и предыдущим шагом вычислений. Это критически важно для стабильной работы симулятора, особенно при изменяющейся частоте кадров. Термодинамические расчёты выполняются реже механических (например, только при положении поршня в верхней мёртвой точке), что оптимизирует производительность без потери точности.

Код, реализующий вышеупомянутую логику представлен в листинге А.1 приложения А.

3.2. РЕАЛИЗАЦИЯ ГРАФИЧЕСКОЙ СИСТЕМЫ

Графическая реализация программы основана на комплексном подходе, сочетающем использование предварительно подготовленных 3D моделей основных компонентов двигателя и их программную анимацию в реальном времени. Исходные модели создаются в 3D редакторе Blender, где разрабатывается их геометрическая структура. Эти модели проходят оптимизацию, направленную на обеспечение эффективного рендеринга без потери визуальной достоверности.

Реализованы модели основных компонентов двигателя внутреннего сгорания, задействованных в визуализации.

На рисунке 5 представлена реализации моделей поршня и шатуна.

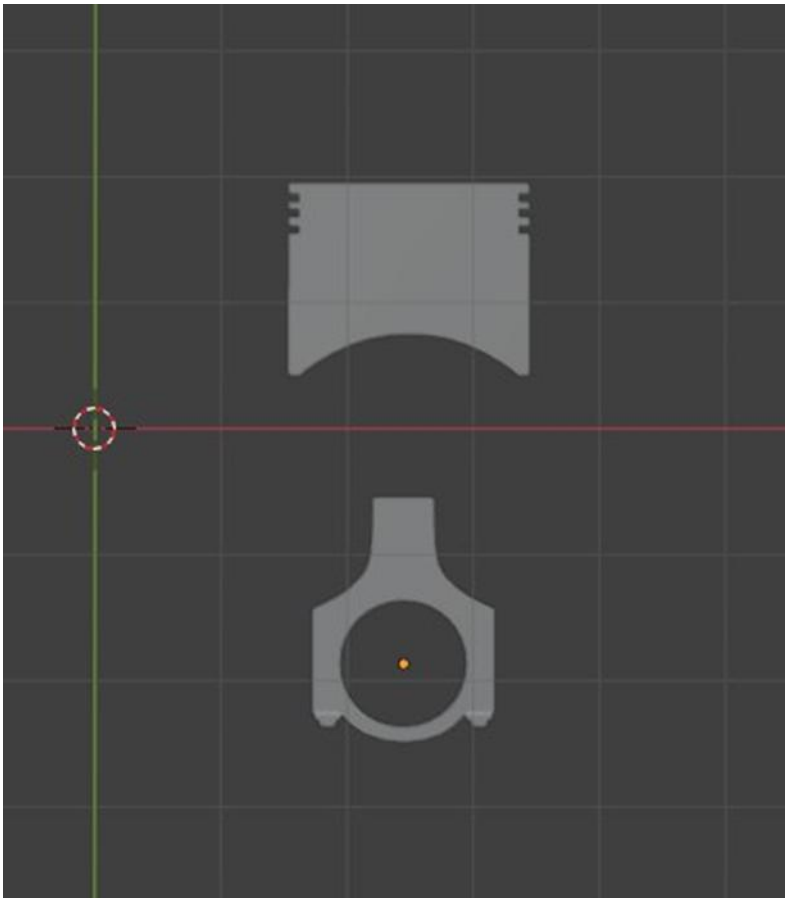

Рисунок 5 – Модели поршня и шатуна в программе Blender

На рисунке 6 представлена реализация моделей головки блока цилиндров и впускного и выпускного валов.

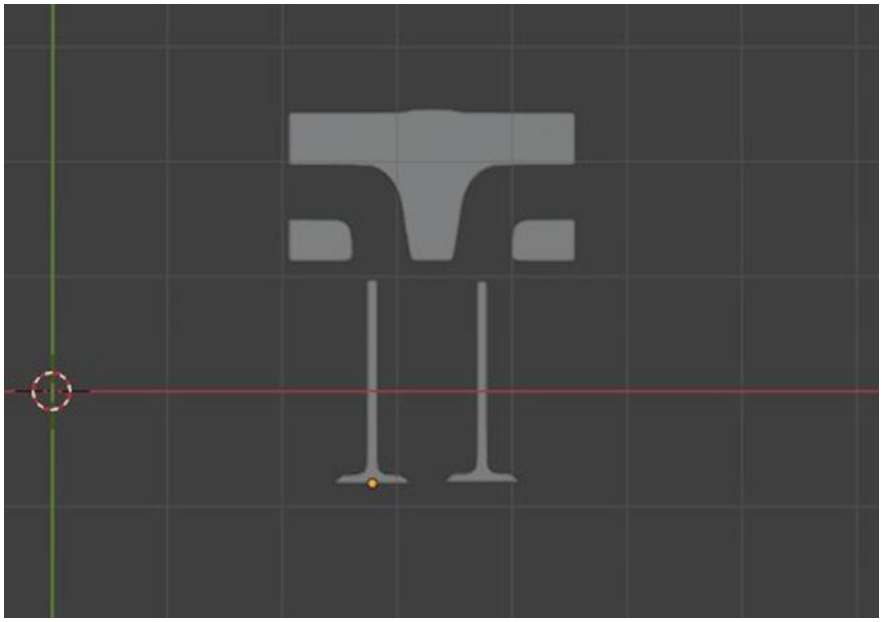

Рисунок 6 – Модели головки блока цилиндров и валов в программе Blender

На рисунке 7 представлена реализация модели коленчатого вала.

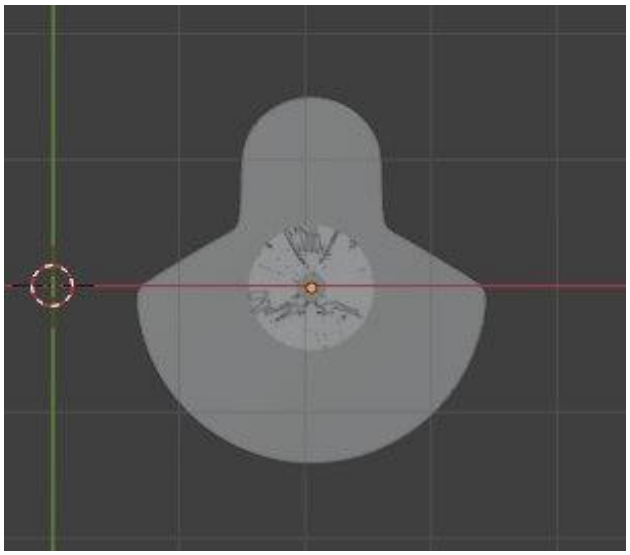

Рисунок 7 – Модель коленчатого вала в программе Blender

Реализация графического слоя главным образом осуществляется с помощью графического API OpenGL. Экспортированные из Blender модели загружаются в программу через специализированную систему управления ассетами, реализованную в классе AssetManager, что позволяет гибко работать с графическими ресурсами. Каждый компонент двигателя, такой как коленчатый вал, поршневая группа, представлен отдельным графическим объектом, который динамически связывается с соответствующим элементом физической модели. Позиционирование и анимация этих объектов в реальном времени осуществляется на основе математических расчетов, получаемых из ядра физической симуляции.

На рисунке 8 представлена модель двигателя, собранная классом AssetManager из заготовленных в программе Blender моделей.

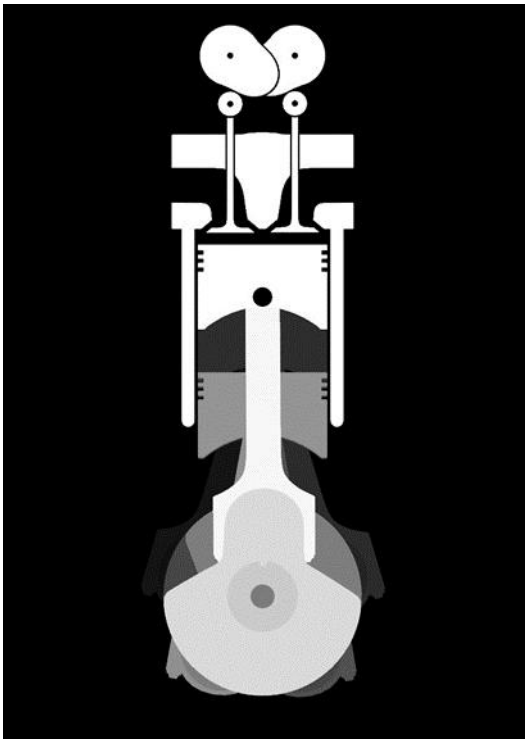

Рисунок 8 – Реализация модели ДВС

В программе реализована иерархическая система анимации деталей двигателя. Каждая деталь двигателя (коленвал, поршень, шатун) – отдельный SceneObjectAsset класса AssetManager. Трансформации родительских объектов автоматически влияют на дочерние. Например, при вращении коленвала (родитель) шатуны (дети) получают соответствующее движение.

Все графические компоненты спроектированы с учетом принципов модульности и масштабируемости, что обеспечивает возможность легкого расширения функционала и добавления новых элементов двигателя без необходимости существенной переработки базовой системы визуализации.

Реализация графической системы представлена в листинге А.2 Приложения А.

3.3. РЕАЛИЗАЦИЯ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

Реализация пользовательского интерфейса выполнена с использованием библиотеки ImGui, которая обеспечивает интерактивное управление параметрами системы в реальном времени. Этот выбор обусловлен простотой интеграции и возможностью создания гибкого интерфейса, который не требует сложной системы обработки событий. Интерфейс построен по принципу "оконного менеджера", где каждая панель отвечает за определённый аспект симуляции – от базовых настроек двигателя до визуализации рабочих характеристик.

Важной частью являются графики и окна интерфейса предоставляющие техническую информацию и отображающие ключевые параметры двигателя в реальном времени.

На рисунке 9 представлено окно пользовательского интерфейса, реализующее функционал тахометра.

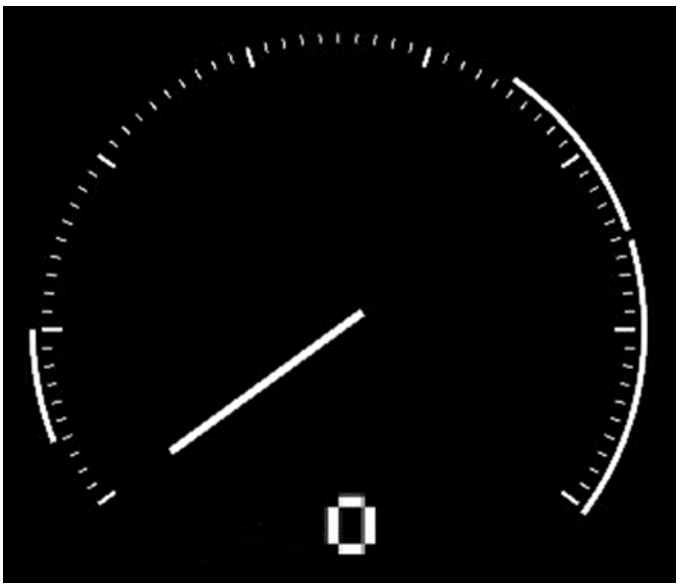

Рисунок 9 – Реализация тахометра

График расхода выхлопных газов – это графическое представление зависимости массы выхлопных газов от угла поворота коленчатого вала (или, в альтернативном представлении, от времени в рамках одного рабочего цикла). Кривая на графике формируется на основе расчёта массы газов, покидающих цилиндр через выпускные клапаны за единицу времени. В начале такта выпуска, когда поршень начинает движение от нижней мёртвой точки к верхней, поток резко возрастает из-за высокого давления в цилиндре – это проявляется как крутой подъём кривой. Максимальный расход достигается в момент, когда разница давлений между цилиндром и выхлопной системой наибольшая. Затем, по мере опустошения цилиндра, кривая плавно снижается.

При изменении оборотов двигателя характер графика существенно меняется. На низких оборотах кривая имеет один выраженный пик с относительно плавным спадом, так как у газов есть больше времени для выхода. На высоких оборотах график становится более острым.

На рисунке 10 приведен пример графика расхода выхлопных газов.

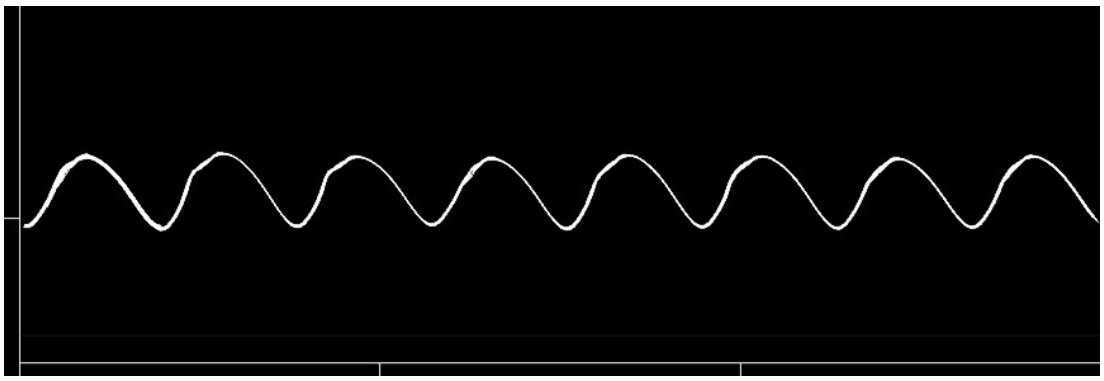

Рисунок 1 – Реализация графика расхода выхлопных газов

Также реализован интерфейс, отображающий степень открытия дроссельной заслонки. Степень открытия в свою очередь меняется при нажатии на газ (кнопка R на клавиатуре).

На рисунке 11 приведен пример реализации.

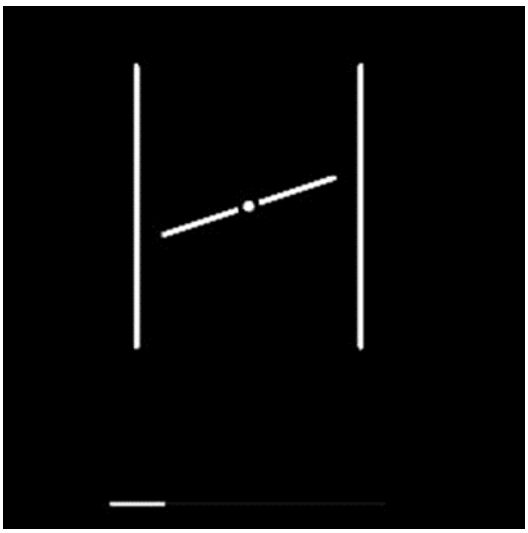

Рисунок 11 – Реализация интрефейса дроссельной заслонки

В окне зажигание визуализируется система зажигания двигателя, где каждый кружок соответствует свече зажигания в конкретном цилиндре. Это интерактивный элемент интерфейса, который отображает текущее состояние зажигания. Кружки меняют цвет в момент искрообразования, имитируя реальные вспышки свечей.

На рисунке 12 представлен интерфейс системы зажигания при заглушенном двигателе.

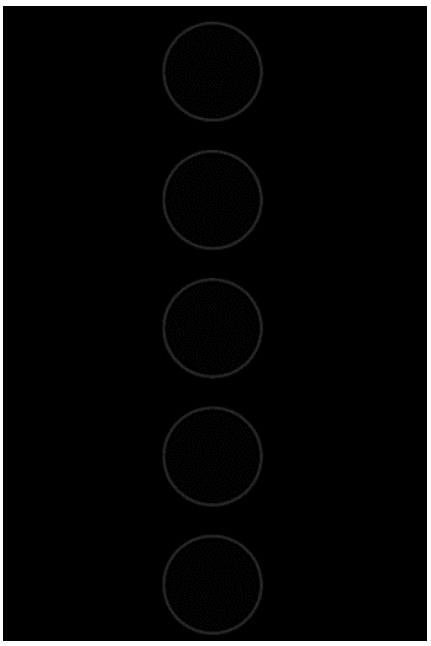

Рисунок 12 – Интерфейс системы зажигания при заглушенном двигаетле

На рисунке 13 представлен интерфейс системы зажигания при работающем двигателе.

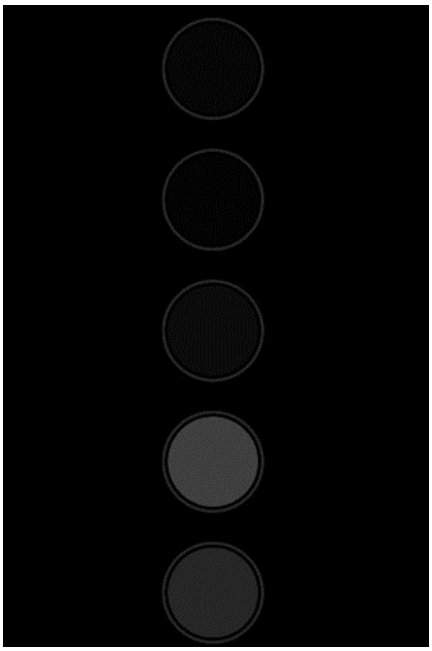

Рисунок 13 – интерфейс системы хажигания при работающем двигателе

Программная реализация пользовательского интерфейса представлена в листинге А.3 приложения А.

3.4. РЕАЛИЗАЦИЯ ЗВУКОВОЙ СИСТЕМЫ

Звуковая система реализована с использованием библиотеки OpenAL, которая обеспечивает пространственное воспроизведение звуков с учётом физических параметров работы двигателя. Основной задачей системы является создание реалистичного звукового сопровождения, синхронизированного с процессами, происходящими в двигателе – вспышками в цилиндрах, движением поршневой группы, работой клапанов и выхлопной системы

Интеграция с физической симуляцией происходит через общий цикл обновления: каждый кадр звуковая система получает актуальные параметры работы двигателя и соответствующим образом корректирует генерируемые звуки.

Программная реализация звуковой системы представлена в листинге А.4 приложения А.

3.5. ВЫВОД ПО ГЛАВЕ 3

Была разработана программа, реализующая функционал визуализатора работы ДВС. Визуализация основывается на физической модели, разработанной с помощью библиотек Bullet Physics и Cantera, что позволило добиться правдоподобности визуализации без необходимости разработки алгоритмов с нуля.

Реализованная архитектура обеспечивает быстродействие программы и хорошую оптимизацию, при этом позволяет легко расширять и дорабатывать функционал в дальнейшем.

Визуализация выполнена с использованием OpenGL, что обеспечивает плавный рендеринг модели двигателя с детализированными компонентами.

4. ПРОВЕДЕНИЕ ТЕСТИРОВАНИЯ

После реализации приложения следующим этапом является проведение тестирования для проверки корректной работы всех аспектов программы.

Составим чек-лист – список проверок, которые нужно протестировать. Он поможет не упустить важные сценарии и структурировать процесс тестирования.

Чек-лист тестирования:

- запуск программы;
- отображение модели двигателя;
- нажатие кнопки запуска двигателя;
- нажатие кнопки газа;
- отображение графика расхода выхлопных газов;
- отображение интрефейса дроссельной заслонки;
- отображение тахометра;
- нажатие кнопки глушения двигателя.

4.1. ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

Функциональное тестирование приложения будет направлено на проверку соответствия его работы заявленным требованиям. Особое внимание уделю основным сценариям использования, чтобы убедиться, что все ключевые функции работают корректно, а логика приложения не содержит ошибок. На первом этапе тестирования, с помощью составленного чек-листа, проверю правильность функционирования программы. Результаты тестирования приведены в таблице 1.

Таблица 1 – Результаты функционального тестирования

Тест	Результат	Тест пройден?
Запуск программы	Программа запускается корректно, ошибок при старте не возникло	Да
Отображение модели двигателя	Модель двигателя отображается без артефактов, все компоненты, загруженные из программы Blender, визуализируются правильно	Да
Запуск двигателя	Анимация движения модели двигателя, обновляющаяся в реальном времени, поднятие оборотов на тахометре до холостых, отображение информации на графике, воспроизведение звука	Да
Нажатие на газ	Анимация открытия дроссельной заслонки, увеличение оборотов двигателя, обновление информации на графике в реальном времен	Да
Отображение графика расхода выхлопных газов	График корректно отображается и обновляется в реальном времени, пики расхода соответствуют тактам выпуска	Да
Отображение интерфейса дроссельной заслонки	Положение заслонки визуализируется синхронно с нажатием кнопки газа, процент открытия отображается шкалой	Да
Отображение тахометра	Показания соответствуют расчётным оборотам двигателя, стрелка движется плавно, без рывков	Да
Остановка двигателя	Плавное падение оборотов на тахометре до 0, остановка всех движущихся частей модели	Да

4.2. НЕФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

В рамках нефункционального тестирования проведена оценка ключевых характеристик системы, таких как производительность, отказоустойчивость и соответствие техническим требованиям. Характеристики компьютера, на котором проводилось тестирование:

– процессор Ryzen 5 5500;

– 16 ГБ оперативной памяти.

Прогрмма стабильно поддерживала 60 кадров в секунду при 50% загрузке ЦП, что свидетельствует об эффективной оптимизации вычислительных алгоритмов. Пиковые нагрузки наблюдались при увиличении числа оборотов двигателя, что требовало сложных расчёте процессов в Cantera и приводило к росту потребления ресурсов.

4.3. ВЫВОД ПО ГЛАВЕ 4

Было проведено функциональное и нефункциональное тестирование программы, которое показало, что визуализотор соответствует выдвенутым к нему требованиям по производительности и функциональности.

ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломной работы была достигнута цель – разработан динамически изменяющий визуализатор работы двигателя внутреннего сгорания.

Были сформулированы функциональные и нефункциональные требования, определяющие ключевые характеристики программы.

Принятые архитектурные решения обеспечивают четкое разделение ответственности между компонентами, высокую производительность вычислений и гибкость для дальнейшего расширения функционала.

Была разработана программа, реализующая функционал визуализатора работы ДВС. Визуализация основывается на физической модели, разработанной с помощью библиотек Bullet Physics и Cantera, что позволило добиться правдоподобности визуализации.

Проведённое тестирование подтвердило, что визуализатор стабильно работает в реальном времени, обеспечивает предсказуемую реакцию на изменение параметров и может использоваться в образовательных и инженерных целях для изучения принципов работы ДВС.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Компьютерная визуализация [Электронный ресурс] – URL: http://aco.ifmo.ru/el_books/computer_visualization/lectures/1.html (дата обращения: 04.03.2025).
2. Компьютерная визуализация данных [Электронный ресурс] – URL: https://scienceforum.ru/2016/article/2016025747 (дата обращения: 30.04.2025).
3. Компьютерная графика. [Электронный ресурс] – URL: https://hist.bsu.by/images/stories/files/uch_materialy/muz/1_kurs/IT_Priborov/L_zima.pdf (дата обращения: 07.03.2025).
4. Компьютерное моделирование [Электронный ресурс] – URL: https://znanierussia.ru/articles (дата обращения: 07.03.2025).
5. User interface. [Электронный ресурс] – URL: https://www.techtarget.com/searchapparchitecture/definition/user–interface–UI (дата обращения: 07.03.2025).
6. Руководство по языку программирования C++ [Электронный ресурс] – URL: https://metanit.com/cpp/tutorial/ (дата обращения: 10.03.2025).
7. C++ Introduction [Электронный ресурс] – URL: https://www.w3schools.com/cpp/cpp_intro.asp (дата обращения 23.05.2025).
8. C programming language tutorial [Электронный ресурс] – URL: https://www.geeksforgeeks.org/c–programming–language/ (дата обращения: 10.03.2025).
9. Документация по OpenGL [Электронный ресурс] – URL: https://www.khronos.org/opengl/wiki/ (дата обращения: 07.05.2024).
10. Что такое OpenGL? [Электронный ресурс] – URL: https://3dnews.ru/169184 (дата обращения: 11.03.2025).

11. Документация по ImGui [Электронный ресурс] – URL: https://github.com/ocornut/imgui/wiki/ (дата обращения: 12.03.2025).
12. Документация по SDL2 [Электронный ресурс] – URL: https://wiki.libsdl.org/SDL2/Introduction (дата обращения: 13.03.2025).
13. Описание программного комплекса ANSYS [Электронный ресурс] – URL: https://dispace.edu.nstu.ru/didesk/file/get/376595 (дата обращения: 23.05.2025).
14. Докукментация по AVL FIRE [Электронный ресурс] – URL: https://www.avl.com/en/simulation–solutions/software–offering/simulation–tools–a–z/avl–fire–m (дата обращения: 13.03.2025).
15. Automation wiki [Электронный ресурс] – URL: https://wiki.automationgame.com/index.php?title=Automation_Wiki (Дата обращения 13.03.2025).
16. Martin, R. Clean Architecture: A Craftsman’s Guide to Software Structure and Design / R. Martin // Pearson. – 2017. – P. 35–41.
17. Richards, M. Fundamentals of Software Architecture / M. Richards, N. Ford // O’Reilly Media. – 2020. – P. 23–26.
18. Fowler, M. Patterns of Enterprise Application Architecture / M. Fowler // Addison–Wesley Professional. – 2002. – P. 83–90.

ПРИЛОЖЕНИЕ А

Листинг А.1 – Программный код физического ядра

```
// Инициализация компонентов
Cantera::ThermoPhase* gas = Cantera::newPhase("gri30.yaml"); // Загрузка тер-
модинамической модели
btDefaultCollisionConfiguration* collisionConfig = new btDefaultCollisionCon-
figuration();
btDynamicsWorld* dynamicsWorld = new btDiscreteDynamicsWorld(...); // Созда-
ние мира Bullet Physics
ALuint engineSoundBuffer, engineSoundSource; // Источник звука в OpenAL

void simulateEngineCycle() {
    // 1. Моделирование термодинамики в Cantera
    gas->setState_TPY(cylinderTemperature, cylinderPressure, "C8H18:1,
O2:12.5, N2:47");
    gas->equilibrate("HP"); // Расчёт состояния после сгорания
    double newPressure = gas->pressure();

    // 2. Передача давления в механическую модель
    btRigidBody* pistonBody = dynamicsWorld->getRigidBody(pistonIndex);
    btVector3 force(0, pistonArea * newPressure, 0); // Сила, действующая на
поршень
    pistonBody->applyCentralForce(force);

    // 3. Расчёт механики в Bullet Physics
    dynamicsWorld->stepSimulation(deltaTime, 10); // Шаг симуляции
    double pistonVelocity = pistonBody->getLinearVelocity().y();

    // 4. Генерация звука на основе скорости поршня
    ALfloat soundPitch = 0.5 + abs(pistonVelocity) * 0.1; // Частота звука
зависит от скорости
    alSourcef(engineSoundSource, AL_PITCH, soundPitch);
    alSourcePlay(engineSoundSource);

    // 5. Визуализация
    updatePistonPosition(pistonBody->getWorldTransform().getOrigin());
}

void GasSystem::initialize(double P, double V, double T, const Mix &mix, int
degreesOfFreedom) {
    m_degreesOfFreedom = degreesOfFreedom;
    m_state.n_mol = P * V / (constants::R * T);
    m_state.V = V;
    m_state.E_k = T * (0.5 * degreesOfFreedom * m_state.n_mol * con-
stants::R);
    m_state.mix = mix;
    m_state.momentum[0] = m_state.momentum[1] = 0;

    const double hcr = heatCapacityRatio();
    m_chokedFlowLimit = chokedFlowLimit(degreesOfFreedom);
    m_chokedFlowFactorCached = chokedFlowRate(degreesOfFreedom);
}
    const double new_system_n = system_n + dn;
```

Продолжение листинга А.1

```
    // Обновляем доли компонентов в системе (если общее количество не нуле-
вое)
    if (new_system_n != 0) {
        m_state.mix.p_fuel = new_system_n_fuel / new_system_n;
        m_state.mix.p_inert = new_system_n_inert / new_system_n;
        m_state.mix.p_o2 = new_system_n_o2 / new_system_n;
    }
    else {
        // Если система пуста - обнуляем все доли
        m_state.mix.p_fuel = m_state.mix.p_inert = m_state.mix.p_o2 = 0;
    }

    // Возвращаем количество прореагировавшего топлива
    return a_n_fuel;
}
// Функция вычисляет константу потока для заданных параметров системы
// Возвращает значение, необходимое для достижения целевого расхода
(targetFlowRate)
double GasSystem::flowConstant(
    double targetFlowRate,
    double P,
    double pressureDrop,
    double T,
    double hcr)
{
    // Сохраняем исходные параметры
    const double T_0 = T;
    const double p_0 = P, p_T = P - pressureDrop; // p_0 - давление на входе,
p_T - на выходе

    // Вычисляем критическое отношение давлений для сверхзвукового (чокового)
течения
    const double chokedFlowLimit = std::pow((2.0 / (hcr + 1)), hcr / (hcr -
1));
    const double p_ratio = p_T / p_0; // отношение давлений (выход/вход)

    double flowRate = 0; // Будет содержать расчетный расход

    // Проверяем режим течения (чоковый или дозвуковой)
    if (p_ratio <= chokedFlowLimit) {
        // Чоковый режим течения (сверхзвуковой, скорость на выходе = скоро-
сти звука)
        // Используем уравнения для критического течения
        flowRate = std::sqrt(hcr);
        flowRate *= std::pow(2 / (hcr + 1), (hcr + 1) / (2 * (hcr - 1)));
    }
    else {
        // Дозвуковой режим течения
        // Используем полное уравнение изоэнтропического течения
        flowRate = (2 * hcr) / (hcr - 1);
        flowRate *= (1 - std::pow(p_ratio, (hcr - 1) / hcr));
        flowRate = std::sqrt(flowRate);
        flowRate *= std::pow(p_ratio, 1 / hcr);
    }

    // Корректируем расход с учетом давления и температуры
```

Продолжение листинга А.1

```
    flowRate *= p_0 / std::sqrt(constants::R * T_0); // constants::R – уни-
версальная газовая постоянная

    // Возвращаем константу, на которую нужно умножить flowRate для получения
targetFlowRate
    return targetFlowRate / flowRate;
}
// Обновляет скорость и кинетическую энергию газовой системы с учетом динами-
ческого давления
void GasSystem::updateVelocity(double dt, double beta) {
    // Если в системе нет молекул (n=0), выходим
    if (n() == 0) return;

    // Вычисляем "глубину" системы (объем / площадь поперечного сечения)
    const double depth = volume() / (m_width * m_height);

    // Инициализация изменения импульса по осям x и y
    double d_momentum_x = 0;
    double d_momentum_y = 0;

    // Вычисляем динамическое давление в 4 направлениях:
    // 0: прямое направление (по текущему вектору скорости)
    // 1: обратное направление
    // 2: перпендикулярное направление (поворот на 90°)
    // 3: противоположное перпендикулярное
    const double p0 = dynamicPressure(m_dx, m_dy);  // Основное направление
    const double p1 = dynamicPressure(-m_dx, -m_dy); // Противоположное
    const double p2 = dynamicPressure(m_dy, m_dx);  // Перпендикуляр 1
    const double p3 = dynamicPressure(-m_dy, -m_dx); // Перпендикуляр 2

    // Вычисляем силы давления на грани системы:
    // Умножаем давление на площадь соответствующей поверхности
    const double p_sa_0 = p0 * (m_height * depth); // Сила в основном направ-
лении
    const double p_sa_1 = p1 * (m_height * depth); // Сила в обратном направ-
лении
    const double p_sa_2 = p2 * (m_width * depth);  // Сила в перпендикулярном
    const double p_sa_3 = p3 * (m_width * depth);  // Сила в противоп. пер-
пенд.

    // Суммируем изменения импульса от всех сил:
    // Основное направление добавляет импульс
    d_momentum_x += p_sa_0 * m_dx;
    d_momentum_y += p_sa_0 * m_dy;

    // Обратное направление вычитает импульс
    d_momentum_x -= p_sa_1 * m_dx;
    d_momentum_y -= p_sa_1 * m_dy;

    // Перпендикулярные направления влияют на обе компоненты
    d_momentum_x += p_sa_2 * m_dy;
    d_momentum_y += p_sa_2 * m_dx;

    d_momentum_x -= p_sa_3 * m_dy;
    d_momentum_y -= p_sa_3 * m_dx;
```

Продолжение листинга А.1

```
    // Получаем массу системы и обратную массу (для оптимизации)
    const double m = mass();
    const double inv_m = 1 / m;

    // Вычисляем текущую скорость до изменения
    const double v0_x = m_state.momentum[0] * inv_m;
    const double v0_y = m_state.momentum[1] * inv_m;

    // Обновляем импульс системы с учетом:
    // - вычисленного изменения импульса (d_momentum)
    // - шага времени (dt)
    // - коэффициента сопротивления (beta)
    m_state.momentum[0] -= d_momentum_x * dt * beta;
    m_state.momentum[1] -= d_momentum_y * dt * beta;

    // Вычисляем новую скорость после изменения
    const double v1_x = m_state.momentum[0] * inv_m;
    const double v1_y = m_state.momentum[1] * inv_m;

    // Обновляем кинетическую энергию системы:
    // Вычитаем разницу между новой и старой кинетической энергией
    m_state.E_k -= 0.5 * m * (v1_x * v1_x - v0_x * v0_x);
    m_state.E_k -= 0.5 * m * (v1_y * v1_y - v0_y * v0_y);

    // Защита от отрицательной кинетической энергии
    if (m_state.E_k < 0) m_state.E_k = 0;
}
void GasSystem::dissipateVelocity(double dt, double timeConstant) {
    if (n() == 0) return;

    const double invMass = 1.0 / mass();
    const double velocity_x = m_state.momentum[0] * invMass;
    const double velocity_y = m_state.momentum[1] * invMass;
    const double velocity_squared =
        velocity_x * velocity_x + velocity_y * velocity_y;
    const double s = dt / (dt + timeConstant);
    m_state.momentum[0] = m_state.momentum[0] * (1 - s);
    m_state.momentum[1] = m_state.momentum[1] * (1 - s);

    const double newVelocity_x = m_state.momentum[0] * invMass;
    const double newVelocity_y = m_state.momentum[1] * invMass;
    const double newVelocity_squared =
        newVelocity_x * newVelocity_x + newVelocity_y * newVelocity_y;

    const double dE_k = 0.5 * mass() * (velocity_squared - newVeloc-
ity_squared);
    m_state.E_k += dE_k;
}


// Функция для расчета газообмена между двумя объемами
double GasSystem::computeGasExchange(const ExchangeConditions &cond) {
    // Определение направления потока
    GasVolume *donor = nullptr, *receiver = nullptr;
    double donorP = 0, receiverP = 0;
```

Продолжение листинга А.1

```
    double flowDirX, flowDirY;
    double donorArea = 0, receiverArea = 0;
    int flowSign = 0;

    // Вычисляем эффективные давления с учетом динамической составляющей
    auto calcEffectivePressure = [](GasVolume* vol, double dirX, double dirY)
{
        return vol->getPressure() + vol->computeDynamicPressureCompo-
nent(dirX, dirY);
    };

    double effP1 = calcEffectivePressure(cond.volumeA, cond.flowDirX,
cond.flowDirY);
    double effP2 = calcEffectivePressure(cond.volumeB, -cond.flowDirX, -
cond.flowDirY);

    if (effP1 > effP2) {
        donor = cond.volumeA;
        receiver = cond.volumeB;
        donorP = effP1;
        receiverP = effP2;
        flowDirX = cond.flowDirX;
        flowDirY = cond.flowDirY;
        donorArea = cond.areaA;
        receiverArea = cond.areaB;
        flowSign = 1;
    } else {
        donor = cond.volumeB;
        receiver = cond.volumeA;
        donorP = effP2;
        receiverP = effP1;
        flowDirX = -cond.flowDirX;
        flowDirY = -cond.flowDirY;
        donorArea = cond.areaB;
        receiverArea = cond.areaA;
        flowSign = -1;
    }

    // Вычисляем объем передаваемого газа
    double transferredAmount = cond.timeStep * computeTransferRate(
        cond.flowCoeff,
        donorP,
        receiverP,
        donor->getTemp(),
        receiver->getTemp(),
        donor->getHeatCapRatio(),
        donor->getChokeLimit(),
        donor->getCachedChokeFactor());

    // Ограничиваем максимальный объем передачи
    double maxTransfer = donor->computeMaxTransfer(receiver);
    transferredAmount = std::min(std::max(transferredAmount, 0.0), 0.9 * do-
nor->getParticleCount());

    // Вычисляем долю передаваемого вещества
    double transferRatio = transferredAmount / donor->getParticleCount();
```

Продолжение листинга А.1

```
    double volumeTransferred = transferRatio * donor->getVolume();
    double massTransferred = transferRatio * donor->getTotalMass();

    if (transferredAmount > 0) {
        // Сохраняем начальные энергетические состояния
        double initialDonorEnergy = donor->computeKineticEnergy();
        double initialReceiverEnergy = receiver->computeKineticEnergy();
        double totalEnergyBefore = donor->computeTotalEnergy() + receiver-
>computeTotalEnergy();

        // Выполняем передачу частиц
        double particleEnergy = donor->getParticleEnergy();
        receiver->acceptParticles(transferredAmount, particleEnergy, donor-
>getComposition());
        donor->releaseParticles(transferredAmount, particleEnergy);

        // Перенос импульса
        double momentumX = donor->getMomentumX() * transferRatio;
        double momentumY = donor->getMomentumY() * transferRatio;
        donor->adjustMomentum(-momentumX, -momentumY);
        receiver->adjustMomentum(momentumX, momentumY);

        // Корректировка энергии после переноса
        double finalDonorEnergy = donor->computeKineticEnergy();
        double finalReceiverEnergy = receiver->computeKineticEnergy();
        receiver->adjustEnergy(-((finalDonorEnergy + finalReceiverEnergy) -
                               (initialDonorEnergy + initialReceiverEnergy)));
    }

    // Обрабатываем динамические эффекты потока
    processFlowDynamics(donor, receiver, volumeTransferred, massTransferred,
                        flowDirX, flowDirY, donorArea, receiverArea,
cond.timeStep);

    return transferredAmount * flowSign;
}

// Вспомогательная функция для обработки динамики потока
void GasSystem::processFlowDynamics(GasVolume* donor, GasVolume* receiver,
                                    double volTrans, double massTrans,
                                    double dirX, double dirY,
                                    double donorArea, double receiverArea,
                                    double timeStep) {
    // Обработка получателя
    if (receiverArea > 0) {
        double receiverVel = std::min((volTrans / receiverArea) / timeStep,
                                      receiver->getSoundSpeed());
        double velX = receiverVel * dirX;
        double velY = receiverVel * dirY;
        receiver->adjustMomentum(velX * massTrans, velY * massTrans);
    }

    // Обработка донора
    if (donorArea > 0 && donor->getTotalMass() > 0) {
        double donorVel = std::min((volTrans / donorArea) / timeStep,
                                   donor->getSoundSpeed());
```

Продолжение листинга А.1

```
        double velX = donorVel * dirX;
        double velY = donorVel * dirY;
        donor->adjustMomentum(velX * massTrans, velY * massTrans);
    }

    // Корректировка энергии
    updateEnergyBalance(donor);
    updateEnergyBalance(receiver);
}

// Функция для обновления энергетического баланса
void GasSystem::updateEnergyBalance(GasVolume* volume) {
    double mass = volume->getTotalMass();
    if (mass <= 0) return;

    double invMass = 1.0 / mass;
    double initialVelX = volume->getInitialMomentumX() * invMass;
    double initialVelY = volume->getInitialMomentumY() * invMass;
    double currentVelX = volume->getMomentumX() * invMass;
    double currentVelY = volume->getMomentumY() * invMass;

    double energyChange = 0.5 * mass * (
        (currentVelX * currentVelX - initialVelX * initialVelX) +
        (currentVelY * currentVelY - initialVelY * initialVelY)
    );
    volume->adjustEnergy(-energyChange);

    // Гарантируем неотрицательность энергии
    if (volume->getKineticEnergy() < 0) {
        volume->setKineticEnergy(0);
    }
}

/**
 * Активирует процесс горения в камере сгорания при выполнении условий
 *
 * Проверяет наличие топлива, соотношение компонентов и другие параметры,
 * инициализирует параметры горения при успешной проверке
 */
void CombustionChamber::startCombustion() {
    // Если горение уже активно - ничего не делаем
    if (m_isCombustionActive) return;

    // Проверка наличия топлива в смеси
    if (m_system.getMixture().fuelPartialPressure == 0) {
        return; // Нет топлива - возгорание невозможно
    }

    // Расчёт соотношения воздух-топливо и коэффициента эквивалентности
    const double airFuelRatio = m_system.getMixture().oxygenPartialPressure /
                                m_system.getMixture().fuelPartialPressure;
    const double equivalenceRatio = airFuelRatio / m_fuelProperties->getIdea-
lAFR();

    // Проверка допустимого диапазона для возгорания
    if (equivalenceRatio < 0.5 || equivalenceRatio > 1.9) {
```

Продолжение листинга А.1

```
            return; // Смесь слишком бедная или богатая для воспламенения
        }

        // Расчёт эффекта разбавления инертными газами
        const double idealInertGas = m_system.getMixture().oxygenPartialPressure
/ 0.7;
        const double dilutionEffect = (m_system.getMixture().inertGasPartialPres-
sure / idealInertGas) - 1;

        // Инициализация параметров горения
        initializeCombustionParameters();

        // Активация флага горения
        m_isCombustionActive = true;
        m_wasCombustionLastCycle = true;

        // Расчёт эффективности горения с учётом различных факторов
        calculateCombustionEfficiency(dilutionEffect);
}

/**
 * Инициализирует начальные параметры процесса горения
 */
void CombustionChamber::initializeCombustionParameters() {
    m_combustionData.initialVolume = getChamberVolume();
    m_combustionData.flamePositionX = 0;
    m_combustionData.flamePositionY = 0;
    m_combustionData.ignitedParticles = 0;
    m_combustionData.totalParticles = m_system.getParticleCount();
    m_combustionData.combustionPercentage = 0;
    m_combustionData.currentMixture = m_system.getMixture();
}

/**
 * Вычисляет эффективность горения с учётом различных факторов
 *
 * @param dilutionEffect Эффект разбавления инертными газами
 */
void CombustionChamber::calculateCombustionEfficiency(double dilutionEffect)
{
    // Получение характеристик топлива
    const double efficiencyVariability = m_fuelProperties->getEfficiencyVari-
ability();
    const double minEfficiency = m_fuelProperties->getMinEfficiencyFactor();
    const double maxEfficiency = m_fuelProperties->getMaxEfficiency();
    const double maxTurbulenceImpact = m_fuelProperties->getMaxTurbulenceIm-
pact();
    const double maxDilutionImpact = m_fuelProperties->getMaxDilutionIm-
pact();

    // Расчёт турбулентности в камере
    const double turbulence = m_turbulenceModel->calculateTurbulenceLevel(
        computeAveragePistonSpeed());

    // Фактор смешивания компонентов
    const double mixingQuality = 1.0 - (
```

Продолжение листинга А.1

```
        normalizeValue(turbulence / maxTurbulenceImpact) *
        normalizeValue(1 - dilutionEffect / maxDilutionImpact)
    );

    // Случайный компонент эффективности
    const double randomFactor = minEfficiency * (
        (1 - efficiencyVariability) +
        efficiencyVariability * (getRandomValue())
    );

    // Итоговая эффективность горения
    m_combustionData.efficiency =
        (mixingQuality * randomFactor + (1 - mixingQuality)) * maxEfficiency;

    // Расчёт скорости распространения пламени
    m_combustionData.flameSpeed = m_fuelProperties->calculateFlameSpeed(
        turbulence,
        m_system.getMixture().oxygenPartialPressure /
        m_system.getMixture().fuelPartialPressure,
        m_system.getTemperature(),
        m_system.getPressure(),
        computeFiringPressure(),
        Pressure(160, PressureUnit::PSI)
    );
}

/**
 * Генерирует случайное значение между 0 и 1
 */
double CombustionChamber::getRandomValue() const {
    return static_cast<double>(rand()) / RAND_MAX;
}

/**
 * Нормализует значение в диапазоне [0, 1]
 */
double CombustionChamber::normalizeValue(double value) const {
    return value < 0 ? 0 : (value > 1 ? 1 : value);
}

/**
 * Обновляет состояние камеры сгорания за временной интервал dt
 *
 * @param dt Временной шаг симуляции в секундах
 */
void CombustionChamber::processChamberState(double dt) {
    // 1. Обновление геометрических параметров камеры
    updateChamberGeometry();

    // 2. Обновление состояний рабочего цикла
    updateEngineCycle();

    // 3. Получение текущих расходов через впуск и выпуск
    updateFlowRates();

    // 4. Обработка теплопередачи и утечек
```

Продолжение листинга А.1

```
    processHeatTransferAndLeakage(dt);

    // 5. Моделирование газообмена
    simulateGasExchange(dt);

    // 6. Обработка процесса горения (если активно)
    if (m_isCombustionActive) {
        processCombustion(dt);
    }
}

/**
 * Обновляет геометрические параметры камеры сгорания
 */
void CombustionChamber::updateChamberGeometry() {
    m_system.updateVolume(getCurrentVolume());
}

/**
 * Обновляет состояния рабочего цикла двигателя
 */
void CombustionChamber::updateEngineCycle() {
    m_cycleState.update();
}

/**
 * Обновляет значения расходов через впускную и выпускную системы
 */
void CombustionChamber::updateFlowRates() {
    const int cylinderIdx = m_piston->getPositionIndex();
    m_currentIntakeRate = m_cylinderHead->getIntakeFlowRate(cylinderIdx);
    m_currentExhaustRate = m_cylinderHead->getExhaustFlowRate(cylinderIdx);
}

/**
 * Обрабатывает теплопередачу и утечки газа
 *
 * @param dt Временной шаг симуляции
 */
void CombustionChamber::processHeatTransferAndLeakage(double dt) {
    // Фиксация пиковой температуры
    if (m_system.getTemperature() > m_maxRecordedTemp) {
        m_maxRecordedTemp = m_system.getTemperature();
    }

    // Расчёт теплопередачи через стенки цилиндра
    const double chamberVolume = getCurrentVolume();
    const double cylinderHeight = chamberVolume / m_cylinderCrossArea;
    const double cylinderWallArea = calculateWallArea(cylinderHeight);

    const double tempDifference = 90.0 - m_system.getTemperature();
    const double heatTransfer = tempDifference * cylinderWallArea * 100 * dt;

    m_system.adjustEnergy(heatTransfer);

    // Моделирование утечек через поршневые кольца
```

Продолжение листинга А.1

```
    m_system.simulateLeakage(
        m_piston->getLeakageCoefficient(),
        dt,
        m_crankcasePressure,
        25.0 // Температура картера
    );
}

/**
 * Вычисляет площадь поверхности стенок цилиндра
 */
double CombustionChamber::calculateWallArea(double height) const {
    const double bore = m_cylinderHead->getBank()->getCylinderDiameter();
    return height * constants::pi * bore + m_cylinderCrossArea * 2;
}

/**
 * Моделирует процессы газообмена через впуск и выпуск
 *
 * @param dt Временной шаг симуляции
 */
void CombustionChamber::simulateGasExchange(double dt) {
    // Получение ссылок на системы
    IntakeSystem* intake = m_cylinderHead->getIntakeSystem(m_piston->getPosi-
tionIndex());
    ExhaustSystem* exhaust = m_cylinderHead->getExhaustSystem(m_piston->get-
PositionIndex());

    // Начальное количество частиц для отслеживания изменений
    const double initialParticles = m_system.getParticleCount();

    // 1. Поток из впускного коллектора в камеру
    processIntakeFlow(intake, dt);

    // 2. Поток из камеры в выпускную систему
    processExhaustFlow(exhaust, dt);

    // Обновление скоростей потока
    updateFlowVelocities(intake, exhaust, dt);

    // Проверка сброса флага горения при наличии потока
    if (std::abs(m_lastIntakeFlow) > FLOW_THRESHOLD && m_isCombustionActive)
{
        m_isCombustionActive = false;
    }

    // Накопление суммарных расходов
    m_totalExhaustFlow += m_lastExhaustFlow;
    m_totalIntakeFlow += m_lastIntakeFlow;
}

/**
 * Обрабатывает процесс горения в камере
 *
 * @param dt Временной шаг симуляции
```

Продолжение листинга А.1

```
 */
void CombustionChamber::processCombustion(double dt) {
    CylinderBank* bank = m_cylinderHead->getBank();

    // Расчёт геометрии пламени
    const double maxFlameRadius = bank->getCylinderDiameter() / 2;
    const double maxFlameHeight = getCurrentVolume() / bank->getBoreArea();

    // Расчёт расширения пламени
    const double expansionRatio = getCurrentVolume() / m_combustionData.ini-
tialVolume;

    // Обновление положения фронта пламени
    updateFlameFrontPosition(dt, maxFlameRadius, maxFlameHeight, expansionRa-
tio);

    // Проверка завершения горения
    if (hasCombustionCompleted(maxFlameRadius, maxFlameHeight)) {
        m_isCombustionActive = false;
        return;
    }

    // Расчёт сгоревшего объёма и выделившейся энергии
    calculateBurnedVolumeAndEnergy(expansionRatio);

    // Обновление начального объёма для следующего шага
    m_combustionData.initialVolume = getCurrentVolume();
}

/**
 * Рассчитывает соотношение воздух-топливо для последнего события горения
 *
 * @return Соотношение масс воздуха к топливу (0 если топлива нет)
 */
double CombustionChamber::getLastCombustionAFR() const {
    // Расчет общего количества компонентов смеси
    const double fuelMass = m_combustionData.mixture.fuelPartialPressure *
                            m_combustionData.totalParticles;
    const double oxygenMass = m_combustionData.mixture.oxygenPartialPressure
*
                              m_combustionData.totalParticles;
    const double inertMass = m_combustionData.mixture.inertPartialPressure *
                             m_combustionData.totalParticles;

    // Молярные массы компонентов
    static constexpr double fuelMolarWeight = 114.23;  // г/моль (октан)
    static constexpr double oxygenMolarWeight = 31.9988;  // г/моль
    static constexpr double nitrogenMolarWeight = 28.014;  // г/моль
(основной инертный газ)

    if (fuelMass <= 0) return 0.0;
```

Продолжение листинга А.1

```
    // Расчет массового соотношения (воздух + азот) / топливо
    return (oxygenMolarWeight * oxygenMass + inertMass * nitrogenMolarWeight)
/
           (fuelMass * fuelMolarWeight);
}

/**
 * Вычисляет силу трения поршня о стенки цилиндра
 *
 * @param pistonVelocity Скорость поршня (м/с)
 * @return Сила трения (Н)
 */
double CombustionChamber::computePistonFriction(double pistonVelocity) const
{
    // Нормальная сила от давления на стенки цилиндра
    const double normalForce = m_piston->getWallContactForce();

    // Параметры модели трения
    const double coulombFriction = m_frictionParams.coefficient * normal-
Force;
    const double stribeckVelocity = m_frictionParams.stribeckVelocity * con-
stants::sqrt_two;
    const double coulombVelocity = m_frictionParams.stribeckVelocity / 10.0;
    const double stribeckFriction = m_frictionParams.stribeckForce;
    const double absVelocity = std::abs(pistonVelocity);

    // Компоненты модели трения
    const double stribeckComponent = constants::sqrt_two * constants::e *
                                     (stribeckFriction - coulombFriction);
    const double velocityRatio = absVelocity / stribeckVelocity;
    const double exponentialComponent = std::exp(-velocityRatio * velocityRa-
tio) * velocityRatio;
    const double coulombComponent = coulombFriction * std::tanh(absVelocity /
coulombVelocity);
    const double viscousComponent = m_frictionParams.viscousCoefficient * ab-
sVelocity;

    // Суммарная сила трения
    return stribeckComponent * exponentialComponent +
           coulombComponent +
           viscousComponent;
}

/**
 * Обновляет состояния цикла двигателя для текущего угла поворота коленвала
 */
void CombustionChamber::updateCycleParameters() {
    double crankAngle = m_engine->getCrankshaft()->getCurrentAngle();

    // Защита от некорректных значений
    if (!std::isfinite(crankAngle)) {
        crankAngle = 0.0;
    }
```

Продолжение листинга А.1

```
    // Нормализация угла и расчет индекса для кольцевого буфера
    const int bufferIndex = static_cast<int>(
        std::round((crankAngle / (4 * constants::pi)) *
        (CYCLE_STATE_SAMPLES - 1))
    );

    // Сохранение текущих параметров
    m_cycleState.velocityBuffer[bufferIndex] = std::abs(getPistonVelocity());
    m_cycleState.pressureBuffer[bufferIndex] = m_system.getPressure();
}

/**
 * Применяет силы к поршню в физической модели
 *
 * @param systemState Состояние физической системы
 */
void CombustionChamber::applyPistonForces(PhysicalSystemState* systemState) {
    CylinderBank* bank = m_cylinderHead->getBank();

    // Расчет площади поршня
    const double pistonArea = (bank->getBoreDiameter() * bank->getBoreDiame-
ter() / 4.0) *
                              constants::pi;

    // Проекция скорости поршня
    const double pistonVelocity =
        systemState->velocityX[m_piston->bodyId] * bank->getDirectionX() +
        systemState->velocityY[m_piston->bodyId] * bank->getDirectionY();

    // Сила от перепада давления
    const double pressureForce = -pistonArea *
                                 (m_system.getPressure() - m_crankcasePres-
sure);

    // Ограничение скорости для стабильности вычислений
    const double limitedVelocity = std::min(std::abs(pistonVelocity),
                                            VELOCITY_LIMIT);
    const double velocityAttenuation = limitedVelocity / VELOCITY_LIMIT;

    // Расчет и применение суммарной силы
    const double frictionForce = computePistonFriction(pistonVelocity) *
                                 velocityAttenuation;
    const double totalForce = pressureForce +
                              ((pistonVelocity > 0) ? -frictionForce : fric-
tionForce);

    // Применение силы к телу поршня
    systemState->applyForceToBody(
        totalForce * bank->getDirectionX(),
        totalForce * bank->getDirectionY(),
        m_piston->bodyId
```

Продолжение листинга А.1

```
    );
}

/**
 * Возвращает текущую силу трения поршня
 *
 * @return Сила трения (Н)
 */
double CombustionChamber::getCurrentFrictionForce() const {
    CylinderBank* bank = m_cylinderHead->getBank();
    const double pistonVelocity =
        m_piston->bodyVelocityX * bank->getDirectionX() +
        m_piston->bodyVelocityY * bank->getDirectionY();

    return computePistonFriction(pistonVelocity);
}

/**
 * Вычисляет скорость распространения пламени с учетом турбулентности
 *
 * @param turbulenceIntensity Уровень турбулентности (м/с)
 * @param actualAfr Фактическое соотношение воздух-топливо
 * @param temperature Температура смеси (К)
 * @param pressure Давление смеси (Па)
 * @param combustionPressure Давление при сгорании (Па)
 * @param motoringPressure Давление при прокрутке (Па)
 * @return Скорость пламени (м/с)
 */
double Fuel::computeFlameSpeed(
    double turbulenceIntensity,
    double actualAfr,
    double temperature,
    double pressure,
    double combustionPressure,
    double motoringPressure) const
{
    // Базовая скорость ламинарного горения
    const double baseSpeed = computeLaminarFlameSpeed(actualAfr, temperature,
pressure);

    // Поправочный коэффициент для давления
    const double pressureFactor = 1.0; // Может быть адаптирован в будущем

    // Расчет влияния турбулентности с использованием треугольного распреде-
ления
    const double turbulenceRatio = (turbulenceIntensity / baseSpeed) * pres-
sureFactor;
    const double turbulenceEffect = m_flameSpeedTurbulenceModel->evaluateTri-
angular(turbulenceRatio);

    return turbulenceEffect * baseSpeed;
}
```

Продолжение листинга А.1

```
/**
 * Вычисляет скорость ламинарного горения для заданных условий
 *
 * @param actualAfr Фактическое соотношение воздух-топливо
 * @param temperature Температура смеси (К)
 * @param pressure Давление смеси (Па)
 * @return Скорость ламинарного пламени (м/с)
 */
double Fuel::computeLaminarFlameSpeed(
    double actualAfr,
    double temperature,
    double pressure) const
{
    // Константы для расчета скорости горения
    static constexpr double REFERENCE_RATIO = 1.21;
    static constexpr double BASE_VELOCITY = 0.305; // м/с
    static constexpr double RATIO_COEFFICIENT = -0.549; // м/с

    // Нормированное соотношение воздух-топливо
    const double ratio = actualAfr / m_idealAfr;

    // Температурная и барическая чувствительность
    const double tempExponent = 2.4 - 0.271 * std::pow(ratio, 3.51);
    const double pressureExponent = -0.357 + 0.14 * std::pow(ratio, 2.77);

    // Базовая скорость при эталонных условиях
    const double referenceSpeed = BASE_VELOCITY +
                                  RATIO_COEFFICIENT *
                                  std::pow(ratio - REFERENCE_RATIO, 2);

    // Нормировка температуры и давления
    const double normalizedTemp = temperature / 298.0; // Относительно 298K
    const double normalizedPressure = pressure / 101325.0; // Относительно 1
атм

    // Коррекция скорости для текущих условий
    return referenceSpeed *
           std::pow(normalizedTemp, tempExponent) *
           std::pow(normalizedPressure, pressureExponent);
}


/**
 * Класс для моделирования коленчатого вала двигателя
 */
class Crankshaft {
public:
    /// Конструктор по умолчанию
    Crankshaft()
        : rodJournalAngles(nullptr),
          rodJournalCount(0),
```

Продолжение листинга А.1

```
          crankThrow(0.0),
          mass(0.0),
          inertia(0.0),
          flywheelMass(0.0),
          positionX(0.0),
          positionY(0.0),
          tdcPosition(0.0),
          friction(0.0) {}

    /// Деструктор
    ~Crankshaft() {
        // Проверка что память уже освобождена
        assert(rodJournalAngles == nullptr);
    }

    /**
     * Инициализация параметров коленвала
     * @param params Параметры коленчатого вала
     */
    void initialize(const CrankshaftParams &params) {
        mass = params.mass;
        flywheelMass = params.flywheelMass;
        inertia = params.momentOfInertia;
        crankThrow = params.crankThrow;
        rodJournalCount = params.rodJournals;

        // Выделение памяти под углы шатунных шеек
        rodJournalAngles = new double[rodJournalCount];

        positionX = params.pos_x;
        positionY = params.pos_y;
        tdcPosition = params.tdc;
        friction = params.frictionTorque;
    }

    /// Освобождение ресурсов
    void cleanup() {
        delete[] rodJournalAngles;
        rodJournalAngles = nullptr;
    }

    /**
     * Получить локальные координаты шатунной шейки
     * @param journalIndex Индекс шейки
     * @param[out] x Координата X
     * @param[out] y Координата Y
     */
    void getLocalJournalPosition(int journalIndex, double *x, double *y)
const {
        const double angle = rodJournalAngles[journalIndex];
        *x = std::cos(angle) * crankThrow;
        *y = std::sin(angle) * crankThrow;
```

Продолжение листинга А.1

```
    }

    /**
     * Получить глобальные координаты шатунной шейки
     * @param journalIndex Индекс шейки
     * @param[out] x Координата X
     * @param[out] y Координата Y
     */
    void getGlobalJournalPosition(int journalIndex, double *x, double *y)
const {
        double localX, localY;
        getLocalJournalPosition(journalIndex, &localX, &localY);

        *x = localX + body.positionX;
        *y = localY + body.positionY;
    }

    /// Нормализовать угол коленвала (0-4π)
    void normalizeAngle() {
        body.angle = std::fmod(body.angle, 4 * constants::pi);
    }

    /**
     * Установить угол для шатунной шейки
     * @param journalIndex Индекс шейки
     * @param angle Угол в радианах
     */
    void setJournalAngle(int journalIndex, double angle) {
        assert(journalIndex >= 0 && journalIndex < rodJournalCount);
        rodJournalAngles[journalIndex] = angle;
    }

    /**
     * Получить текущий угол коленвала относительно ВМТ
     * @return Угол в радианах
     */
    double getCurrentAngle() const {
        return body.angle - tdcPosition;
    }

    /**
     * Получить угол цикла с возможным смещением
     * @param offset Смещение угла
     * @return Нормализованный угол цикла (0-4π)
     */
    double getCycleAngle(double offset = 0.0) const {
        double angle = std::fmod(-getCurrentAngle() + offset, 4 * con-
stants::pi);
        return angle < 0 ? angle + 4 * constants::pi : angle;
    }
private:
    double* rodJournalAngles;       // Углы шатунных шеек
    int rodJournalCount;            // Количество шатунных шеек
    double crankThrow;              // Радиус кривошипа
    double mass;                    // Масса коленвала
    double inertia;                 // Момент инерции
```

Окончание листинга А.1

```
    double flywheelMass;            // Масса маховика
    double positionX, positionY;    // Позиция коленвала
    double tdcPosition;             // Угол ВМТ
    double friction;                // Момент трения

    struct {
        double positionX, positionY;
        double angle;
    } body;                         // Физическое тело коленвала
};
void RightGaugeCluster::render() {
    drawFrame(m_bounds, 1.0, m_app->getForegroundColor(), m_app->getBack-
groundColor());

    const Bounds tachSpeedCluster = m_bounds.verticalSplit(0.5f, 1.0f);
    renderTachSpeedCluster(tachSpeedCluster);

    const Bounds fuelAirCluster = m_bounds.verticalSplit(0.0f, 0.5f);
    renderFuelAirCluster(fuelAirCluster);

    UiElement::render();
}
       void RightGaugeCluster::update(float dt) {
    m_combusionChamberStatus->m_engine = m_engine;
    m_throttleDisplay->m_engine = m_engine;
    m_afrCluster->m_engine = m_engine;
    m_fuelCluster->m_engine = m_engine;
    m_fuelCluster->m_simulator = m_simulator;

    UiElement::update(dt);
}
```

Листинг А.2 – Реализация графической системы

```
/**
 * @file animation_export_data.cpp
 * @brief Реализация класса для экспорта данных анимации
 */

#include "../include/animation_export_data.h"

namespace dbasic {

// ==================== AnimationExportData ====================

/**
 * @brief Конструктор класса AnimationExportData
 */
AnimationExportData::AnimationExportData()
    : ysObject("AnimationExportData"),
      m_referenceFrame(0)
{
    Reset();
}

/**
 * @brief Деструктор класса AnimationExportData
 */
AnimationExportData::~AnimationExportData() {
    // Автоматическое освобождение ресурсов
}

/**
 * @brief Сброс всех данных анимации
 */
void AnimationExportData::Reset() {
    m_referenceFrame = 0;
    m_keyframes.Clear();
    m_poses.Clear();
    m_motions.Clear();
}

/**
 * @brief Добавляет данные ключевых кадров для объекта
 * @param objectName Имя объекта анимации
 * @return Указатель на добавленные данные ключевых кадров
 */
ObjectKeyframeDataExport* AnimationExportData::AddObjectKeyframeData(const
char* objectName) {
    ObjectKeyframeDataExport& newKeyframe = m_keyframes.AddNew();
    strncpy_s(newKeyframe.m_objectName, sizeof(newKeyframe.m_objectName),
              objectName, _TRUNCATE);
    return &newKeyframe;
}
```

Продолжение листинга А.2

```
/**
 * @brief Добавляет новый сегмент анимации (motion)
 * @param name Имя сегмента
 * @param startFrame Начальный кадр
 * @param endFrame Конечный кадр
 */
void AnimationExportData::AddMotionSegment(const char* name, int startFrame,
int endFrame) {
    MotionExport& motion = m_motions.AddNew();
    motion.m_startFrame = startFrame + m_referenceFrame;
    motion.m_endFrame = endFrame + m_referenceFrame;
    strncpy_s(motion.m_name, sizeof(motion.m_name), name, _TRUNCATE);
}

/**
 * @brief Добавляет новую позу
 * @param name Имя позы
 * @param frame Номер кадра позы
 * @return Указатель на добавленную позу
 */
PoseExport* AnimationExportData::AddAnimationPose(const char* name, int
frame) {
    PoseExport& pose = m_poses.AddNew();
    pose.m_frame = frame + m_referenceFrame;
    strncpy_s(pose.m_name, sizeof(pose.m_name), name, _TRUNCATE);
    return &pose;
}

/**
 * @brief Находит сегмент анимации по имени
 * @param name Имя сегмента анимации
 * @return Указатель на найденный сегмент или nullptr
 */
MotionExport* AnimationExportData::FindMotionByName(const char* name) const {
    for (int i = 0; i < m_motions.GetCount(); ++i) {
        if (strcmp(name, m_motions[i].GetName()) == 0) {
            return &m_motions[i];
        }
    }
    return nullptr;
}

/**
 * @brief Загружает данные анимации объекта из инструментального формата
 * @param animData Данные анимации объекта
 * @return Код ошибки
 */
ysError AnimationExportData::ImportObjectAnimation(const ysObjectAnima-
tionData* animData) {
    YDS_ERROR_DECLARE("ImportObjectAnimation");

    if (!animData) return YDS_ERROR_RETURN(ysError::InvalidParameter);
```

Продолжение листинга А.2

```
    // Создаем контейнер для ключевых кадров объекта
    ObjectKeyframeDataExport* keyData = AddObjectKeyframeData(animData->m_ob-
jectName);

    // Импортируем ключи позиции
    for (int i = 0; i < animData->m_positionKeys.NumKeys; ++i) {
        auto& posKey = animData->m_positionKeys.Keys[i];
        auto* key = keyData->AddKeyframe(
            ObjectKeyframeDataExport::POSITION_KEY,
            posKey.Frame);
        key->m_position = posKey.Position;
    }

    // Импортируем ключи вращения
    for (int i = 0; i < animData->m_rotationKeys.NumKeys; ++i) {
        auto& rotKey = animData->m_rotationKeys.Keys[i];
        auto* key = keyData->AddKeyframe(
            ObjectKeyframeDataExport::ROTATION_KEY,
            rotKey.Frame);
        key->m_rotation = rotKey.RotationQuaternion;
    }

    return YDS_ERROR_RETURN(ysError::None);
}

/**
 * @brief Загружает данные временных меток
 * @param timeTagData Данные временных меток
 * @return Код ошибки
 */
ysError AnimationExportData::ImportTimeTags(const ysTimeTagData* timeTagData)
{
    YDS_ERROR_DECLARE("ImportTimeTags");

    if (!timeTagData) return YDS_ERROR_RETURN(ysError::InvalidParameter);

    for (int i = 0; i < timeTagData->m_timeTagCount; ++i) {
        const auto& tag = timeTagData->m_timeTags[i];
        char name[128];
        bool isStart;

        if (ParseMotionTag(tag.Name, name, isStart)) {
            MotionExport* motion = FindMotionByName(name);
            if (motion) {
                // Обновляем существующий сегмент
                if (isStart) motion->m_startFrame = tag.Frame;
                else motion->m_endFrame = tag.Frame;
            } else {
                // Создаем новый сегмент
                if (isStart) AddMotionSegment(name, tag.Frame, -1);
                else AddMotionSegment(name, -1, tag.Frame);
```

Продолжение листинга А.2

```
            }
        }
        else if (ParsePoseTag(tag.Name, name)) {
            AddAnimationPose(name, tag.Frame);
        }
    }

    return YDS_ERROR_RETURN(ysError::None);
}

// ==================== Вспомогательные методы парсинга ====================

/**
 * @brief Парсит тег сегмента анимации
 * @param tag Строка тега
 * @param[out] name Имя сегмента
 * @param[out] isStart Флаг начала сегмента
 * @return Успешность парсинга
 */
bool AnimationExportData::ParseMotionTag(const char* tag, char* name, bool&
isStart) {
    char buffer[1024];
    int pos = 0;

    if (!ReadTagComponent(tag, buffer, pos)) return false;
    if (strcmp(buffer, "MOTION") != 0) return false;

    if (!SkipTagSeparator(tag, pos)) return false;
    if (!ReadTagComponent(tag, name, pos)) return false;
    if (!SkipTagSeparator(tag, pos)) return false;
    if (!ReadTagComponent(tag, buffer, pos)) return false;

    if (strcmp(buffer, "START") == 0) isStart = true;
    else if (strcmp(buffer, "END") == 0) isStart = false;
    else return false;

    return true;
}

/**
 * @brief Парсит тег позы
 * @param tag Строка тега
 * @param[out] name Имя позы
 * @return Успешность парсинга
 */
bool AnimationExportData::ParsePoseTag(const char* tag, char* name) {
    char buffer[1024];
    int pos = 0;

    if (!ReadTagComponent(tag, buffer, pos)) return false;
    if (strcmp(buffer, "POSE") != 0) return false;
```

Продолжение листинга А.2

```
    if (!SkipTagSeparator(tag, pos)) return false;
    if (!ReadTagComponent(tag, name, pos)) return false;

    return true;
}

/**
 * @brief Пропускает разделитель тега
 * @param tag Строка тега
 * @param[in,out] pos Текущая позиция в строке
 * @return Успешность операции
 */
bool AnimationExportData::SkipTagSeparator(const char* tag, int& pos) {
    if (tag[pos++] != ':' || tag[pos++] != ':')
        return false;
    return true;
}

/**
 * @brief Читает компонент тега
 * @param tag Строка тега
 * @param[out] component Буфер для компонента
 * @param[in,out] pos Текущая позиция в строке
 * @return Успешность операции
 */
bool AnimationExportData::ReadTagComponent(const char* tag, char* component,
int& pos) {
    int len = 0;
    while (tag[pos] && tag[pos] != ':') {
        component[len++] = tag[pos++];
    }
    component[len] = '\0';
    return len > 0;
}

// ==================== PoseExport ====================

PoseExport::PoseExport()
    : ysObject("PoseExport"),
      m_frame(-1)
{
    m_name[0] = '\0';
}

PoseExport::~PoseExport() {
    // Автоматическое освобождение ресурсов
}

// ================== ObjectKeyframeDataExport ==================
ObjectKeyframeDataExport::ObjectKeyframeDataExport() {
    m_objectName[0] = '\0';
}

ObjectKeyframeDataExport::~ObjectKeyframeDataExport() {
    // Автоматическое освобождение ресурсов
}
```

Продолжение листинга А.2

```
/**
 * @brief Добавляет ключевой кадр
 * @param type Тип ключа (позиция/вращение)
 * @param frame Номер кадра
 * @return Указатель на данные ключевого кадра
 */
ObjectKeyframeDataExport::KeyFrameData* ObjectKeyframeDataExport::AddK-
eyframe(
    KeyType type, int frame)
{
    // Поиск позиции для вставки с сохранением порядка кадров
    int insertPos = 0;
    for (; insertPos < m_keyData.GetCount(); ++insertPos) {
        if (frame < m_keyData[insertPos].m_frame) break;
        if (frame == m_keyData[insertPos].m_frame) {
            // Обновляем существующий ключ
            m_keyData[insertPos].m_type |= type;
            return &m_keyData[insertPos];
        }
    }

    // Вставляем новый ключ
    KeyFrameData& newKey = m_keyData.InsertNew(insertPos);
    newKey.m_type = type;
    newKey.m_frame = frame;
    return &newKey;
}

// ==================== MotionExport ====================

MotionExport::MotionExport() {
    Clear();
}

MotionExport::~MotionExport() {
    // Автоматическое освобождение ресурсов
}

/**
 * @brief Сбрасывает данные сегмента анимации
 */
void MotionExport::Clear() {
    m_name[0] = '\0';
    m_startFrame = -1;
    m_endFrame = -1;
}

} // namespace dbasic

#include "../include/animation_group.h"

dbasic::AnimationGroup::AnimationGroup() : ysObject("AnimationGroup") {
    // Инициализация имени группы нулевым символом
    m_groupName[0] = '\0';
}
```

Продолжение листинга А.2

```
dbasic::AnimationGroup::~AnimationGroup() {
    // Деструктор по умолчанию, ничего не освобождаем вручную
}

void dbasic::AnimationGroup::Update() {
    // Обновление всех контроллеров анимации
    int totalControllers = m_animationControllers.GetNumObjects();
    for (int idx = 0; idx < totalControllers; ++idx) {
        dbasic::AnimationObjectController *currentController = m_animation-
Controllers[idx];
        currentController->Update();
    }

    // Рекурсивное обновление всех вложенных групп анимации
    int totalGroups = m_animationGroups.GetNumObjects();
    for (int idx = 0; idx < totalGroups; ++idx) {
        dbasic::AnimationGroup *childGroup = m_animationGroups.Get(idx);
        childGroup->Update();
    }
}

void dbasic::AnimationGroup::AddAnimationController(dbasic::AnimationOb-
jectController *newController) {
    // Добавление нового контроллера анимации в текущую группу
    m_animationControllers.New() = newController;
}

dbasic::AnimationGroup *dbasic::AnimationGroup::AddAnimationGroup(const char
*childGroupName) {
    // Создание новой подгруппы анимации
    AnimationGroup *childGroup = m_animationGroups.NewGeneric<Anima-
tionGroup>();

    if (childGroupName != nullptr) {
        childGroup->SetName(childGroupName);
    }

    return childGroup;
}
void dbasic::AnimationGroup::SetName(const char *groupName) {
    // Присваиваем имя группе, копируя его в буфер
    strcpy_s(m_groupName, sizeof(m_groupName), groupName);
}

void dbasic::AnimationGroup::SetFrame(int targetFrame) {
    // Установка текущего кадра для всех контроллеров
    int totalControllers = m_animationControllers.GetNumObjects();
    for (int i = 0; i < totalControllers; ++i) {
        m_animationControllers[i]->SetFrame(targetFrame);
    }

    // Установка кадра для всех вложенных групп
    int totalGroups = m_animationGroups.GetNumObjects();
    for (int i = 0; i < totalGroups; ++i) {
        m_animationGroups.Get(i)->SetFrame(targetFrame);
```

Продолжение листинга А.2

```
}

void dbasic::AnimationGroup::SetTimeOffset(float offsetSeconds) {
    // Установка временного смещения для всех контроллеров
    int controllerNum = m_animationControllers.GetNumObjects();
    for (int i = 0; i < controllerNum; ++i) {
        m_animationControllers[i]->SetTimeOffset(offsetSeconds);
    }

    // Установка смещения времени во всех дочерних группах
    int groupNum = m_animationGroups.GetNumObjects();
    for (int i = 0; i < groupNum; ++i) {
        m_animationGroups.Get(i)->SetTimeOffset(offsetSeconds);
    }
}

#include "../include/animation_object_controller.h"
#include "../include/delta_physics.h"

dbasic::AnimationObjectController::AnimationObjectController() : ysOb-
ject("AnimationObjectController") {
    // Инициализация значений по умолчанию
    m_controlledObject = nullptr;
    m_animationFramerate = 30.0f;
    m_currentFrame = 0;
    m_startTimeOffset = 0.0f;
}

dbasic::AnimationObjectController::~AnimationObjectController() {
    // Пустой деструктор - управление памятью не требуется
}

void dbasic::AnimationObjectController::Update() {
    // Получаем текущую ориентацию объекта в родительском пространстве
    ysQuaternion currentQuat = m_controlledObject->GetOrientation-
ParentSpace();
    ysQuaternion resultQuat;

    int nearestPrev = -1;
    int nearestNext = -1;

    int totalKeyframes = m_keyframeTimeline.GetNumObjects();

    // Поиск ближайших ключевых кадров по вращению
    for (int index = 0; index < totalKeyframes; ++index) {
        if (m_keyframeTimeline[index].GetKeyType() & ObjectKeyframeDataEx-
port::KEY_FLAG_ROTATION_KEY) {
            if (m_keyframeTimeline[index].GetFrame() <= m_currentFrame) {
                nearestPrev = index;
            }
            else {
                nearestNext = index;
                break;
            }
        }
```

Продолжение листинга А.2

```
    // Интерполяция вращения между двумя ближайшими ключевыми кадрами
    if (nearestPrev == -1 && nearestNext != -1) {
        resultQuat = m_keyframeTimeline[nearestNext].GetRotationKey();
    }
    else if (nearestNext == -1 && nearestPrev != -1) {
        resultQuat = m_keyframeTimeline[nearestPrev].GetRotationKey();
    }
    else if (nearestPrev != -1 && nearestNext != -1) {
        float timeDelta = m_startTimeOffset - (m_keyframeTimeline[near-
estPrev].GetFrame() / m_animationFramerate);
        float totalDuration = (m_keyframeTimeline[nearestNext].GetFrame() -
m_keyframeTimeline[nearestPrev].GetFrame()) / m_animationFramerate;

        float interpolationFactor = timeDelta / totalDuration;

        ysVector weightPrev = ysMath::LoadScalar(1.0f - interpolationFactor);
        ysVector weightNext = ysMath::LoadScalar(interpolationFactor);

        ysQuaternion quatPrev = m_keyframeTimeline[nearestPrev].GetRota-
tionKey();
        ysQuaternion quatNext = m_keyframeTimeline[nearestNext].GetRota-
tionKey();

        resultQuat = ysMath::Add(
            ysMath::Mul(weightPrev, quatPrev),
            ysMath::Mul(weightNext, quatNext)
        );
    }

    // Применение рассчитанного вращения к целевому объекту
    m_controlledObject->SetOrientation(resultQuat);
}


#include "../include/animation.h"

dbasic::Animation::Animation() {
    // Инициализация значений по умолчанию
    m_frameTextures = nullptr;
    m_currentState = PAUSED;
    m_timer = 0.0f;

    m_currentFrame = 0;
    m_totalFrames = 0;
}

dbasic::Animation::~Animation() {
    // Пустой деструктор - освобождение ресурсов не требуется
}

void dbasic::Animation::SetRate(float frameRate) {
    // Устанавливаем скорость анимации (инвертированная частота)
    m_inverseRate = 1.0f / frameRate;
}

void dbasic::Animation::SetMode(ANIMATION_MODE newMode) {
```

Окончание листинга А.2

```
    // Устанавливаем текущий режим воспроизведения
    m_currentState = newMode;
}

ysTexture *dbasic::Animation::GetCurrentFrame() {
    // Получаем текстуру текущего кадра
    return m_frameTextures[m_currentFrame];
}

void dbasic::Animation::SetFrame(int frameIndex) {
    // Принудительно устанавливаем конкретный кадр
    m_currentFrame = frameIndex;
}

void dbasic::Animation::Update(float deltaTime) {
    // Не обновляем, если воспроизведение приостановлено
    if (m_currentState == PAUSED) return;

    // Прибавляем прошедшее время к таймеру
    m_timer += deltaTime;

    // Если набралось достаточно времени для перехода к следующему кадру
    if (m_timer >= m_inverseRate) {
        int framesPassed = static_cast<int>(floor(m_timer / m_inverseRate) +
0.5f);
        m_timer -= m_inverseRate * framesPassed;
        m_currentFrame += framesPassed;

        // Обработка переполнения кадра в зависимости от режима
        if (m_currentFrame >= m_totalFrames) {
            switch (m_currentState) {
                case PLAY:
                    m_currentFrame = m_totalFrames - 1;
                    break;
                case LOOP:
                    m_currentFrame = m_currentFrame % m_totalFrames;
                    break;
                default:
                    break;
            }
        }

        // Защита от отрицательных индексов
        if (m_currentFrame < 0) {
            m_currentFrame = 0;
        }
    }
}
```

Листинг А.3 – Реализация пользовательского интерфейса

```
#include "../include/default_ui_shaders.h"

dbasic::DefaultUiShaders::DefaultUiShaders() {
    // Начальные значения для параметров камеры и экрана
    m_rotationAngle = 0.0f;
    m_viewportWidth = 1.0f;
    m_viewportHeight = 1.0f;
}

dbasic::DefaultUiShaders::~DefaultUiShaders() {
    // Пустой деструктор - очистка вручную не требуется
}

ysError dbasic::DefaultUiShaders::Initialize(ysDevice *device, ysRenderTarget
*target, ysShaderProgram *program, ysInputLayout *layout) {
    YDS_ERROR_DECLARE("Initialize");

    // Инициализация набора шейдеров
    m_shaderManager.Initialize(device);

    ShaderStage *uiStage = nullptr;

    // Создание основного шейдерного этапа
    YDS_NESTED_ERROR_CALL(m_shaderManager.NewStage("UIShaders::Stage",
&uiStage));

    uiStage->SetInputLayout(layout);
    uiStage->SetRenderTarget(target);
    uiStage->SetShaderProgram(program);
    uiStage->SetType(ShaderStage::Type::FullPass);

    // Привязка буферов констант
    uiStage->NewConstantBuffer<ShaderScreenVariables>(
        "UIShaders::ScreenVars", 0, ShaderStage::ConstantBufferBind-
ing::BufferType::SceneData, &m_screenVars);
    uiStage->NewConstantBuffer<ShaderObjectVariables>(
        "UIShaders::ObjectVars", 1, ShaderStage::ConstantBufferBind-
ing::BufferType::ObjectData, &m_objectVars);
    uiStage->NewConstantBuffer<LightingControls>(
        "UIShaders::LightVars", 3, ShaderStage::ConstantBufferBind-
ing::BufferType::SceneData, &m_lightingVars);

    return YDS_ERROR_RETURN(ysError::None);
}

ysError dbasic::DefaultUiShaders::Destroy() {
    YDS_ERROR_DECLARE("Destroy");

    // Освобождение ресурсов шейдеров
    YDS_NESTED_ERROR_CALL(m_shaderManager.Destroy());

    return YDS_ERROR_RETURN(ysError::None);
}

void dbasic::DefaultUiShaders::SetScreenDimensions(float width, float height)
{
```

Продолжение листинга А.3

```
    // Установка текущих размеров экрана
    m_viewportWidth = width;
    m_viewportHeight = height;
}

void dbasic::DefaultUiShaders::CalculateCamera() {
    // Создание ортографической проекции
    m_screenVars.Projection = ysMath::Transpose(
        ysMath::OrthographicProjection(
            m_viewportWidth,
            m_viewportHeight,
            0.001f,
            500.0f));

    // Расчёт угла поворота камеры
    float radians = m_rotationAngle * ysMath::Constants::PI / 180.0f;
    float sinA = sin(radians);
    float cosA = cos(radians);

    // Определение позиции и направления камеры
    ysVector cameraPos = ysMath::LoadVector(0.0f, 0.0f, 10.0f, 1.0f);
    ysVector targetPos = ysMath::LoadVector(0.0f, 0.0f, 0.0f, 1.0f);
    ysVector upVector = ysMath::LoadVector(-sinA, cosA);

    // Расчёт и сохранение матрицы вида камеры
    m_screenVars.CameraView = ysMath::Transpose(ysMath::CameraTarget(cam-
eraPos, targetPos, upVector));

    // Сохранение положения камеры
    m_screenVars.Eye = ysMath::LoadVector(cameraPos);
}

#include "../include/ui_renderer.h"
#include "../include/delta_basic_engine.h"

dbasic::UiRenderer::UiRenderer() {
    // Инициализация указателей и размеров по умолчанию
    m_coreEngine = nullptr;
    m_capacity = 0;
    m_usedVertices = 0;

    m_fontAsset = nullptr;
    m_cpuIndexBuffer = nullptr;
    m_cpuVertexBuffer = nullptr;
    m_gpuIndexBuffer = nullptr;
    m_gpuVertexBuffer = nullptr;
    m_usedIndices = 0;
}

dbasic::UiRenderer::~UiRenderer() {
    // Проверка, что GPU-буферы были корректно уничтожены
    assert(m_gpuIndexBuffer == nullptr);
    assert(m_gpuVertexBuffer == nullptr);
}

ysError dbasic::UiRenderer::Initialize(int bufferCapacity) {
```

Продолжение листинга А.3

```
    YDS_ERROR_DECLARE("Initialize");

    // Создание и инициализация геометрии
    YDS_NESTED_ERROR_CALL(SetupBuffers(bufferCapacity));

    return YDS_ERROR_RETURN(ysError::None);
}

ysError dbasic::UiRenderer::RenderUI() {
    YDS_ERROR_DECLARE("RenderUI");

    if (m_usedVertices == 0) return YDS_ERROR_RETURN(ysError::None);

    // Обновление размеров экрана и камеры
    m_shaders.SetScreenDimensions(
        static_cast<float>(m_coreEngine->GetScreenWidth()),
        static_cast<float>(m_coreEngine->GetScreenHeight()));
    m_shaders.CalculateCamera();

    // Отправка буферов с CPU в GPU
    m_coreEngine->GetDevice()->EditBufferDataRange(
        m_gpuVertexBuffer,
        reinterpret_cast<char *>(m_cpuVertexBuffer),
        sizeof(ConsoleVertex) * m_usedVertices,
        0);
    m_coreEngine->GetDevice()->EditBufferDataRange(
        m_gpuIndexBuffer,
        reinterpret_cast<char *>(m_cpuIndexBuffer),
        sizeof(unsigned short) * m_usedIndices,
        0);

    // Установка текстуры шрифта
    m_shaders.SetTexture(m_fontAsset->GetTexture());

    // Выполнение рендера
    m_coreEngine->DrawGeneric(
        m_shaders.GetFlags(),
        m_gpuIndexBuffer,
        m_gpuVertexBuffer,
        sizeof(ConsoleVertex),
        0,
        0,
        m_usedIndices / 3,
        false);

    return YDS_ERROR_RETURN(ysError::None);
}

ysError dbasic::UiRenderer::Destroy() {
    YDS_ERROR_DECLARE("Destroy");

    // Удаление GPU-буферов
    m_coreEngine->GetDevice()->DestroyGPUBuffer(m_gpuIndexBuffer);
    m_coreEngine->GetDevice()->DestroyGPUBuffer(m_gpuVertexBuffer);

    // Очистка CPU-буферов
```

Продолжение листинга А.3

```
    delete[] m_cpuIndexBuffer;
    delete[] m_cpuVertexBuffer;

    return YDS_ERROR_RETURN(ysError::None);
}

dbasic::ConsoleVertex *dbasic::UiRenderer::AllocateQuads(int count) {
    // Подготовка индексов для каждого квадрата
    for (int i = 0; i < count; ++i) {
        int vertexBase = m_usedVertices + i * 4;
        int indexBase = m_usedIndices + i * 6;

        m_cpuIndexBuffer[indexBase + 0] = vertexBase + 2;
        m_cpuIndexBuffer[indexBase + 1] = vertexBase + 1;
        m_cpuIndexBuffer[indexBase + 2] = vertexBase + 0;
        m_cpuIndexBuffer[indexBase + 3] = vertexBase + 3;
        m_cpuIndexBuffer[indexBase + 4] = vertexBase + 2;
        m_cpuIndexBuffer[indexBase + 5] = vertexBase + 0;
    }

    ConsoleVertex *start = m_cpuVertexBuffer + m_usedVertices;

    m_usedIndices += count * 6;
    m_usedVertices += count * 4;

    return start;
}

dbasic::ConsoleVertex *dbasic::UiRenderer::AllocateTriangles(int count) {
    // Подготовка индексов для треугольников
    for (int i = 0; i < count; ++i) {
        int vertexBase = m_usedVertices + i * 3;
        int indexBase = m_usedIndices + i * 3;

        m_cpuIndexBuffer[indexBase + 0] = vertexBase + 2;
        m_cpuIndexBuffer[indexBase + 1] = vertexBase + 1;
        m_cpuIndexBuffer[indexBase + 2] = vertexBase + 0;
    }

    ConsoleVertex *start = m_cpuVertexBuffer + m_usedVertices;

    m_usedIndices += count * 3;
    m_usedVertices += count * 3;

    return start;
}

ysError dbasic::UiRenderer::Clear() {
    YDS_ERROR_DECLARE("Clear");

    // Сброс текущих оффсетов
    m_usedIndices = 0;
    m_usedVertices = 0;

    return YDS_ERROR_RETURN(ysError::None);
}
```

Окончание листинга А.3

```
ysError dbasic::UiRenderer::SetupBuffers(int bufferCapacity) {
    YDS_ERROR_DECLARE("SetupBuffers");

    m_capacity = bufferCapacity;

    // Создание CPU-буферов
    m_cpuIndexBuffer = new unsigned short[bufferCapacity * 2];
    m_cpuVertexBuffer = new ConsoleVertex[bufferCapacity];

    ysDevice *device = m_coreEngine->GetDevice();

    // Создание GPU-буферов
    YDS_NESTED_ERROR_CALL(
        device->CreateVertexBuffer(
            &m_gpuVertexBuffer,
            sizeof(ConsoleVertex) * bufferCapacity,
            reinterpret_cast<char *>(m_cpuVertexBuffer)));

    YDS_NESTED_ERROR_CALL(
        device->CreateIndexBuffer(
            &m_gpuIndexBuffer,
            sizeof(unsigned short) * bufferCapacity * 2,
            reinterpret_cast<char *>(m_cpuIndexBuffer)));

    return YDS_ERROR_RETURN(ysError::None);
}
```

Листинг А.4 – Реализация звуковой системы

```
#include "../include/audio_buffer.h"
#include <assert.h>
#include <cmath>
#include <cstring>

AudioDataBuffer::AudioDataBuffer() {
    m_cursor = 0;
    m_rate = 0;
    m_data = nullptr;
    m_capacity = 0;
    m_timeStep = 0.0;
}

AudioDataBuffer::~AudioDataBuffer() {
    assert(m_data == nullptr); // Убедиться, что ресурсы освобождены
}

void AudioDataBuffer::setup(int samplingRate, int maxSamples) {
    m_cursor = 0;
    m_rate = samplingRate;

    m_capacity = maxSamples;
    m_data = new int16_t[m_capacity];
    std::memset(m_data, 0, sizeof(int16_t) * m_capacity);

    m_timeStep = 1.0 / static_cast<double>(m_rate);
}

void AudioDataBuffer::cleanup() {
    delete[] m_data;
    m_data = nullptr;
    m_capacity = 0;
}

bool AudioDataBuffer::hasDiscontinuity(int jumpLimit) const {
    for (int i = 0; i < m_capacity - 1; ++i) {
        int indexA = resolveIndex(i + m_cursor);
        int indexB = resolveIndex(indexA + 1);

        if (std::abs(m_data[indexA] - m_data[indexB]) >= jumpLimit) {
            return true;
        }
    }

    return false;
}

#include "../include/exhaust_system.h"
#include "../include/units.h"
EngineExhaust::EngineExhaust() {
    m_flowInput = 0;
    m_flowOutput = 0;
    m_crossSection = 0;
    m_pipeLength = 0;
    m_tubeLength = 0;
    m_soundVolume = 0;
```

Продолжение листинга А.4

```
    m_velocityDrop = 0;
    m_resultFlow = 0;
    m_instanceId = -1;
    m_irProfile = nullptr;
}

EngineExhaust::~EngineExhaust() {
    // Нет необходимости в специальных действиях при удалении
}

void EngineExhaust::configure(const Config &config) {
    double collectorWidth = std::sqrt(config.crossSection);
    double collectorVolume = config.crossSection * config.pipeLength;

    m_chamber.initialize(
        units::pressure(1.0, units::atm),
        collectorVolume,
        units::celcius(25.0)
    );
    m_chamber.setGeometry(
        config.pipeLength,
        collectorWidth,
        1.0,
        0.0
    );

    m_environment.initialize(
        units::pressure(1.0, units::atm),
        units::volume(1000.0, units::m3),
        units::celcius(25.0)
    );
    m_environment.setGeometry(
        units::distance(10.0, units::m),
        units::distance(10.0, units::m),
        1.0,
        0.0
    );

    m_flowInput = config.flowInput;
    m_soundVolume = config.soundVolume;
    m_flowOutput = config.flowOutput;
    m_crossSection = config.crossSection;
    m_velocityDrop = config.velocityDrop;
    m_irProfile = config.irProfile;
    m_pipeLength = config.pipeLength;
    m_tubeLength = config.tubeLength;
}

void EngineExhaust::release() {
    // Заглушка для возможной будущей логики очистки
}

void EngineExhaust::simulateStep(double deltaTime) {
    GasSystem::Mix composition;
    composition.p_fuel = 0.0;
    composition.p_inert = 1.0;
```

Окончание листинга А.4

```
    composition.p_o2 = 0.0;

    m_environment.reset(
        units::pressure(1.0, units::atm),
        units::celcius(25.0),
        composition
    );

    GasSystem::FlowParameters params;
    params.crossSectionArea_0 = m_crossSection;
    params.crossSectionArea_1 = units::area(10.0, units::m2);
    params.direction_x = 1.0;
    params.direction_y = 0.0;
    params.dt = deltaTime;
    params.system_0 = &m_environment;
    params.system_1 = &m_chamber;
    params.k_flow = m_flowOutput;

    m_resultFlow = m_chamber.flow(params);

    m_chamber.dissipateExcessVelocity();
    m_chamber.updateVelocity(deltaTime, m_velocityDrop);
}
```