МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования   
«САНКТ–ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой № 44

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| проф., д-р техн. наук, проф. |  |  |  | М.Б. Сергеев |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

БАКАЛАВРСКАЯ РАБОТА

|  |  |
| --- | --- |
| на тему | Разработка веб-интерфейса для отображения взаимодействия настроек на 3D |
| визуализацию цифрового телеуправляемого рентгеновского аппарата | |
|  | |

|  |  |
| --- | --- |
| выполнена | Никифоровым Никитой Сергеевичем |
| фамилия, имя, отчество студента в творительном падеже | |

|  |  |  |  |
| --- | --- | --- | --- |
| по направлению подготовки | 09.03.01 |  | Информатика и вычислительная |
|  | код |  | наименование направления |
| техника | | | |
| наименование направления | | | |
| направленности | 04 |  | Компьютерные технологии, системы и |
|  | код |  | наименование направленности |
| сети | | | |
| наименование направленности | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Студент группы № | 4142 |  |  |  | Н.С. Никифоров |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Руководитель

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| канд. техн. наук, доцент |  |  |  | Н.Н. Решетникова |
| должность, уч. Степень, звание |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования   
«САНКТ–ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

УТВЕРЖДАЮ

Заведующий кафедрой №44

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| проф., д-р техн. наук, проф. |  |  |  | М.Б. Сергеев |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

ЗАДАНИЕ НА ВЫПОЛНЕНИЕ БАКАЛАВРСКОЙ РАБОТЫ

|  |  |  |  |
| --- | --- | --- | --- |
| студенту группы | 4142 |  | Никифоров Никита Сергеевич |
|  | номер |  | фамилия, имя, отчество |

|  |  |
| --- | --- |
| на тему | Разработка веб-интерфейса для отображения взаимодействия настроек на 3D |
| визуализацию цифрового телеуправляемого рентгеновского аппарата | |
|  | |

|  |  |  |  |
| --- | --- | --- | --- |
| утвержденную приказом ГУАП от | 27.03.2025 | № | 11-387/25 |

|  |  |
| --- | --- |
| Цель работы: | Разработка веб-интерфейса для отображения взаимодействия |
| настроек на 3D визуализацию цифрового телеуправляемого рентгеновского аппарата | |
|  | |

|  |  |
| --- | --- |
| Задачи, подлежащие решению: | Анализ предметной области разработки веб-интерфейсов. |
| Выбор программных средств для разработки интерактивной 3D визуализации. Создание | |
| 3D-модели телеуправляемого рентгеновского аппарата. Создание пользовательского | |
| интерфейса для взаимодействия с 3D-моделью рентгеновского аппарата. Создание | |
| способа настройки взаимодействия с рентгеновским аппаратом. Создание базы данных для хранения данных 3D-моделей. Создание сервера эмулятора рентгеновского аппарата. | |

|  |  |
| --- | --- |
| Содержание работы (основные разделы): | 1. Анализ предметной области. |
| 1. Создание 3D-модели рентгеновского аппарата. 3) Отображение 3D-модели в | |
| веб-приложении. 4) Создание пользовательского интерфейса. 5) Создание сервера | |
| 6) Создание сервера эмулятора рентгеновского аппарата. 7) Проверка работоспособности веб-приложения. | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Срок сдачи работы « | 10 | » | Июня |  | 2025 |

Руководитель

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| канд. техн. наук, доцент |  | 28.03.2025 |  | Н.Н. Решетникова |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

Задание принял(а) к исполнению

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| студент группы № | 4142 |  | 28.03.2025 |  | Н.С. Никифоров |

**Оглавление**

[**ВВЕДЕНИЕ** 6](#_Toc199235841)

[1. Анализ предметной области 8](#_Toc199235842)

[1.1. Описание предметной области 8](#_Toc199235843)

[1.2. Описание исходных данных для создания 3D-модели телеуправляемого рентгеновского аппарата 8](#_Toc199235844)

[1.3. Описание исходных данных для создания веб интерфейса 9](#_Toc199235845)

[1.4. Сравнение с существующими аналогами веб-интерфейсов для отображения настроек в реальном времени. 10](#_Toc199235846)

[1.5. Выбор программных средств 12](#_Toc199235847)

[1.6. Общая архитектура приложения 14](#_Toc199235848)

[1.7. Выводы по анализу предметной области 15](#_Toc199235849)

[2. Создание 3D-модели рентгеновского аппарата 16](#_Toc199235850)

[2.1. Определение правил создания 3D-модели 16](#_Toc199235851)

[2.2. Создание 3D-модели 18](#_Toc199235852)

[2.3. Текстурирование 23](#_Toc199235853)

[2.4. Выводы 24](#_Toc199235854)

[3. Отображение 3D-модели рентгеновского аппарата 25](#_Toc199235855)

[3.1. Настройка фронтенд-приложения 25](#_Toc199235856)

[3.2. Хранение и обработка данных для модели 27](#_Toc199235857)

[3.3. Настройка сцены 31](#_Toc199235858)

[3.4. Загрузка модели 33](#_Toc199235859)

[3.5. Интеграция с пользовательским интерфейсом 35](#_Toc199235860)

[4. Создание пользовательского интерфейса 38](#_Toc199235861)

[4.1. Общее описание интерфейса 38](#_Toc199235862)

[4.2. Выбор технологий для интерфейса 39](#_Toc199235863)

[4.3. Архитектура пользовательского интерфейса 40](#_Toc199235864)

[4.4. Проектирование интерфейса 42](#_Toc199235865)

[4.5. Взаимодействие с элементами управления 44](#_Toc199235866)

[4.6. Управление подключением и режимами работы 45](#_Toc199235867)

[4.7. Обработка ошибок и визуализация статуса 46](#_Toc199235868)

[4.8. Выводы 48](#_Toc199235869)

[5. Создание сервера 49](#_Toc199235870)

[5.1. Архитектура серверной части 49](#_Toc199235871)

[5.2. Разработка REST API 50](#_Toc199235872)

[5.3. Реализация базы данных 53](#_Toc199235873)

[5.4. Интеграция с Socket.IO 55](#_Toc199235874)

[5.5. Выводы 56](#_Toc199235875)

[6. Создание сервера эмулятора рентгеновского аппарата. 57](#_Toc199235876)

[6.1. Архитектура эмулятора 57](#_Toc199235877)

[6.2. Настройка базы данных 58](#_Toc199235878)

[6.3. Реализация обработки команд 59](#_Toc199235879)

[6.4. Интеграция с сервером и фронтендом 63](#_Toc199235880)

[6.5. Выводы 64](#_Toc199235881)

[7. Проверка работоспособности приложения 65](#_Toc199235882)

[7.1. Общая стратегия тестирования 65](#_Toc199235883)

[7.2. Тестирование клиентской части 65](#_Toc199235884)

[7.3. Тестирование серверной части и REST API 66](#_Toc199235885)

[7.4. Тестирование WebSocket-взаимодействия 68](#_Toc199235886)

[7.5. Выявленные проблемы и их устранение 69](#_Toc199235887)

[7.6. Система логгирования и диагностики 69](#_Toc199235888)

[7.7. Выводы 70](#_Toc199235889)

[**ЗАКЛЮЧЕНИЕ** 71](#_Toc199235890)

[**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ** 72](#_Toc199235891)

**ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ**

**Blender** — программное обеспечение с открытым исходным кодом для 3D-моделирования, анимации и рендеринга, используемое для создания реалистичных трёхмерных объектов.

**Three.js** — JavaScript-библиотека, использующая WebGL для отображения и управления 3D-графикой в веб-браузерах.

**WebGL** — веб-стандарт для рендеринга 3D-графики в браузере без дополнительных плагинов.

**glTF (GL Transmission Format)** — формат файлов для эффективной передачи и загрузки 3D-моделей и сцен, разработанный Khronos Group.

**JSON (JavaScript Object Notation)** — текстовый формат обмена данными, используемый для конфигурации и хранения информации в приложениях.

**SQLite** — легковесная реляционная база данных, предназначенная для хранения данных с минимальными системными требованиями.

**bun.js** — серверная платформа для выполнения JavaScript-кода, используемая для создания серверных приложений.

**Express** — фреймворк для Node.js, упрощающий разработку веб-серверов и API.

**UI (User Interface)** — пользовательский интерфейс, обеспечивающий взаимодействие между пользователем и приложением.

**API (Application Programming Interface)** — программный интерфейс приложения, позволяющий взаимодействовать с внешними системами или компонентами.

**React** — JavaScript-библиотека с открытым исходным кодом для создания динамичных пользовательских интерфейсов, использующая компонентный подход и виртуальный DOM для эффективного обновления UI.

**Redux** — библиотека управления состоянием для JavaScript-приложений, обеспечивающая предсказуемое управление данными через централизованное хранилище (store).

**Хук (Hook)** — специальная функция в React, позволяющая использовать состояние и другие возможности React в функциональных компонентах без написания классов.

**TS(TypeScript)** — надстройка над JavaScript, добавляющая строгую типизацию и расширенные возможности для улучшения масштабируемости и читаемости кода.

# **ВВЕДЕНИЕ**

В современном мире цифровые технологии прочно интегрированы в нашу повседневную жизнь. Веб-приложения становятся инструментом для решения задач в разных областях. Особое место занимают приложения с интерактивными 3D-моделями, которые позволяют пользователям визуализировать сложные объекты и взаимодействовать с ними через сетевое соединение. Такие решения находят применение в медицине, образовании, промышленности и научных исследованиях, обеспечивая новые возможности для обучения, демонстрации и управления оборудованием. В связи с этим разработка веб-приложений с использованием 3D элементов становится актуальной задачей.

Разработка веб-интерфейса для отображения взаимодействия настроек с 3D визуализацией цифрового телеуправляемого рентгеновского аппарата представляет собой актуальную задачу в условиях роста спроса на дистанционные и интерактивные технологии. Телеуправляемые рентгеновские аппараты широко используются для диагностики, обеспечивая высокую безопасность за счёт дистанционного управления. Визуализация таких систем в веб-среде позволяет не только демонстрировать их конструкцию, но и моделировать работу, что особенно полезно для обучения медицинского персонала и презентации оборудования.

Цель данной дипломной работы заключается в разработке веб-интерфейса, который обеспечивает возможность интерактивной настройки 3D-модели телеуправляемого рентгеновского аппарата. Для моделирования используется именно телеуправляемый аппарат так как другие модели не имеют дистанционного управления. Особенность данной работы состоит в создании реалистичной 3D визуализации, интегрированной с пользовательским интерфейсом, а также в эмуляции работы аппарата через серверное решение. Это требует комплексного подхода, включающего моделирование, программирование и работу с данными.

В рамках дипломной работы проведен анализ современных технологий для 3D визуализации и веб-разработки, создана 3D-модель рентгеновского аппарата с использованием Blender, реализован веб-интерфейс с использованием библиотеки Three.js, а также разработаны сервер эмулятор и база данных SQLite для хранения данных конфигураций. Проект направлен на решение таких задач как обучение операторов и демонстрация возможностей оборудования для повышения доступности сложных систем.

Результаты данной работы могут быть использованы в образовательных учреждениях для подготовки специалистов в области медицинской техники. А также в системах управления рентгеновскими аппаратами. Эта разработка может способствовать развитию использования технологий 3D визуализации в веб-приложениях в областях, где может применятся сложная телеуправляемая техника.

Таким образом, данная дипломная работа направлена на создание инновационного решения, объединяющего передовые технологии и практическую значимость, а также на повышение интереса к изучению современных методов визуализации для управления сложными системами.

1. Анализ предметной области
   1. Описание предметной области

В современном мире, в условиях быстрого развития цифровых технологий веб-интерфейсы становятся неотъемлемой частью многих продуктов. 3D визуализации становятся важным инструментом в различных областях таких как медицина, образование, научные исследования. Такие приложения позволяют пользователям визуализировать сложное оборудование и взаимодействовать с ним изменяя настройки и наблюдая за изменениями в реальном времени.

Предметная область включает в себя создание реалистичных 3D-моделей, их интеграцию в веб-среду, разработку интуитивно понятного пользовательского интерфейса и обеспечение хранения данных. Для успешной реализации проекта необходимо сочетание знаний в области 3D-графики, веб-разработки и особенностей работы медицинского оборудования, такого как телеуправляемые рентгеновские аппараты.

Работа в данном проекте сосредоточена на телеуправляемых рентгеновских аппаратах, поскольку именно они поддерживают удаленное управление, что делает их идеальными для интеграции с веб-интерфейсами. Такие аппараты позволяют оператору изменять настройки, такие как положение стола или угол наклона излучателя, дистанционно через программное обеспечение, обеспечивая высокую точность и безопасность.

* 1. Описание исходных данных для создания 3D-модели телеуправляемого рентгеновского аппарата

Для создания 3D-модели телеуправляемого рентгеновского аппарата, в качестве исходных данных, используются схемы и технические характеристики реальных устройств, применяемых в медицинской практике. Телеуправляемый рентгеновский аппарат представляет собой комплексное оборудование, включающее рентгеновский излучатель, детектор, подвижный стол для пациента и систему управления. Рендер планируемого к выпуску рентгеновского аппарата представлен на рисунке 1.2.

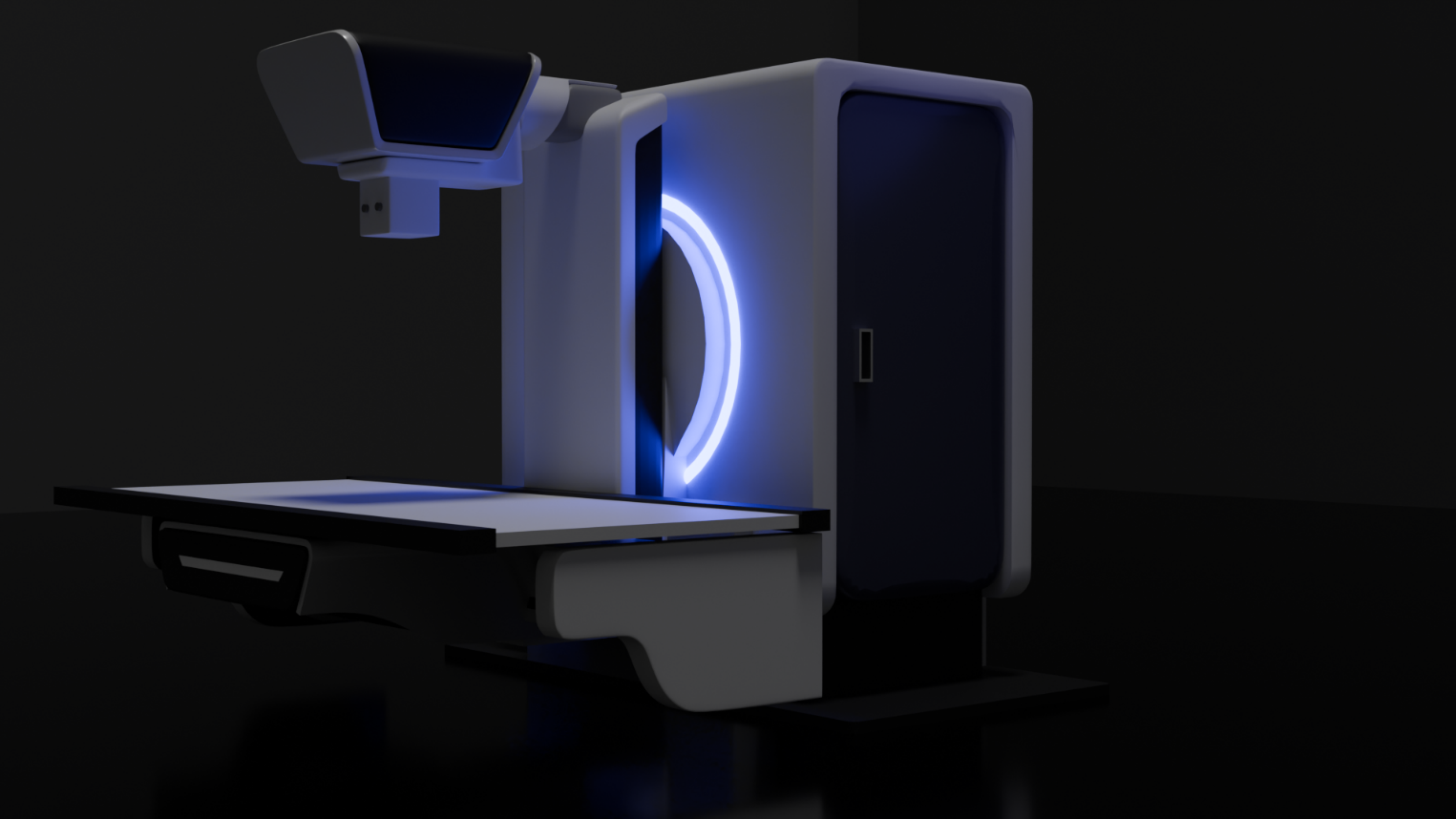


Рисунок 1.2 – Рендер предполагаемого к выпуску рентгеновского аппарата

Основные элементы, подлежащие моделированию, включают:

**Детектор**: фиксирует излучение, проходящее через объект, для формирования изображения.

**Подвижный стол**: позволяет изменять положение пациента относительно излучателя и детектора.

**Механизмы управления**: шарниры и приводы, обеспечивающие точное позиционирование.

Примерные размеры аппарата зависят от конкретной модели, но типичные параметры включают высоту около 2 м, ширину 1,5–2 м и длину стола 2–2,5 м. Технические характеристики, такие как диапазон углов наклона стола (от -90° до +90°).

* 1. Описание исходных данных для создания веб интерфейса

Для создания веб-интерфейса телеуправляемого рентгеновского аппарата в качестве исходных данных используются схемы пультов управления реальных устройств, применяемых в медицинской практике. Схема пульта управления представлена на рисунке 1.3.

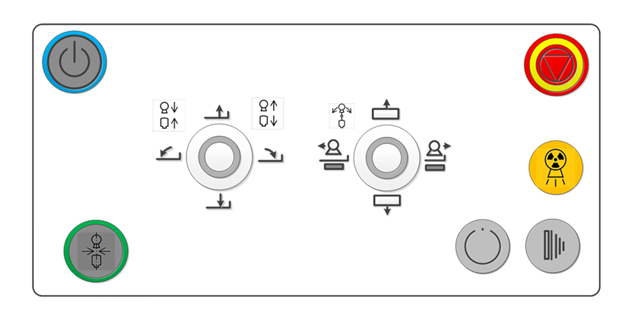


Рисунок 1.3 – Схема пульта управления

Для использования на кнопках предполагаются к использованию символы стандарта “ISO 7000 / IEC 60417 Graphical symbols for use on equipment”, это коллекция символов , которые могут быть размещены на оборудовании, чтобы указать, как правильно и безопасно его использовать. Она включает в себя символы для всех типов оборудования.

* 1. Сравнение с существующими аналогами веб-интерфейсов для отображения настроек в реальном времени.

На рынке не существует полных аналогов но из близких решений есть программа Voron для управления 3D принтером, она позволяет управлять им в реальном времени и просматривать создание 3D-модели, отправленной на печать по слоям. На рисунках 1.4.1-1.4.2 представлен интерфейс данной программы [1].

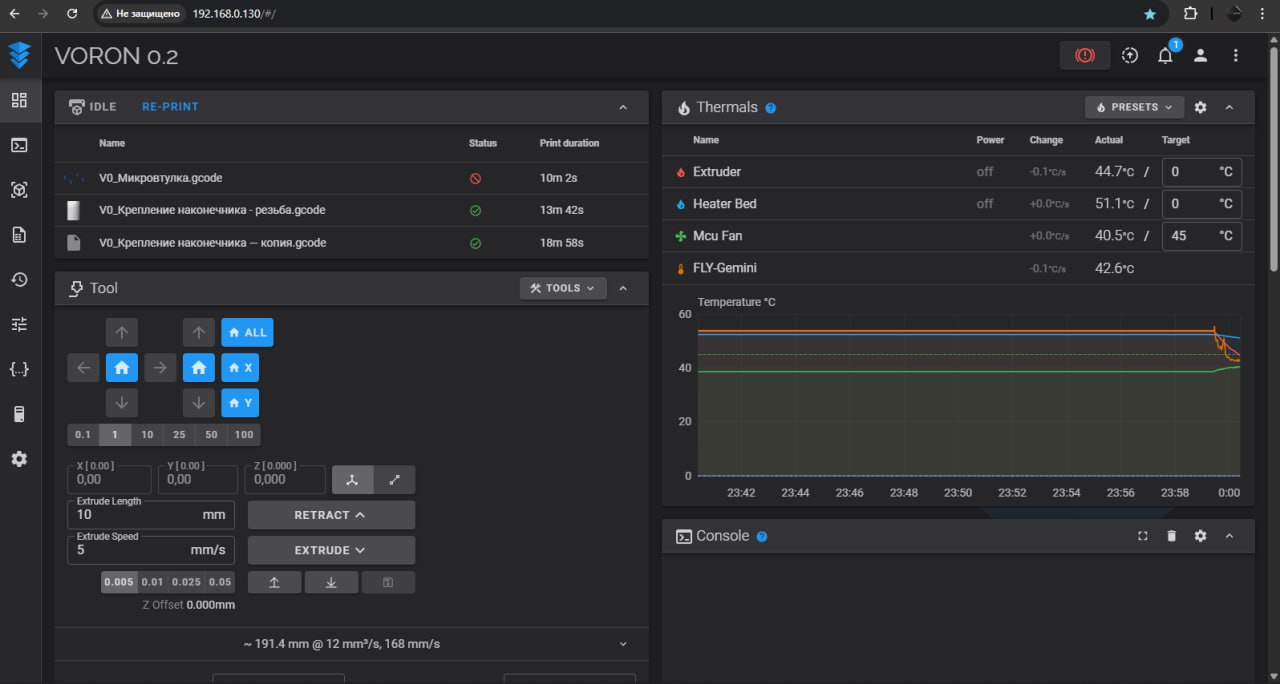


Рисунок 1.4.1 – Интерфейс управления 3D принтером

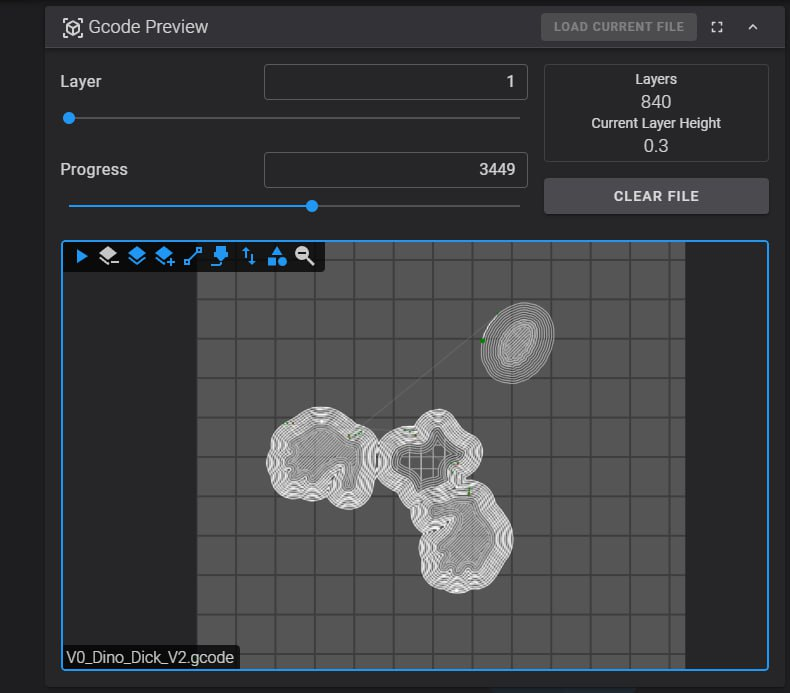


Рисунок 1.4.2 – Визуализация слоя печати

* 1. Выбор программных средств

Для реализации проекта выбраны следующие инструменты и технологии, исходя из их функциональности, совместимости и соответствия поставленным задачам:

**1. Blender**

Blender использован для создания реалистичных 3D-моделей для последующей загрузки в программу.

**Особенности**:

* Поддержка полигонального моделирования и создания реалистичных текстур;
* Возможность экспорта моделей в формат glTF 2.0 (glb), совместимый с Three.js;
* Широкий набор инструментов для детализированного воспроизведения сложных механизмов, таких как рентгеновский аппарат;
* Бесплатность и активное сообщество разработчиков.

**Причины выбора**: Blender выбран за его универсальность, бесплатность и совместимость с Three.js. Альтернативы, такие как Autodesk Maya, были отклонены из-за высокой стоимости и избыточной функциональности для задач проекта.

**2. Three.js**

Three.js использован для рендеринга 3D графики в браузере, является JavaScript-библиотекой, использующий WebGL для рендеринга

**Особенности**:

* Простота интеграции с веб-приложениями и поддержка формата glTF;
* Возможность создания интерактивных сцен с управлением камерой и объектами;
* Высокая производительность при работе с 3D-моделями в реальном времени;
* Совместимость с современными браузерами без дополнительных плагинов.

**Причины выбора**: Three.js выбран за простоту интеграции и высокую производительность. Альтернативы, такие как Babylon.js, были отклонены из-за более сложной настройки для задач проекта.

**3. JSON**

JSON используется для конфигурации возможностей взаимодействия с моделью.

**Особенности**:

* Лёгкость парсинга в JavaScript, что упрощает интеграцию с Three.js;
* Компактность и низкая вероятность ошибок при настройке параметров;
* Высокая читаемость и поддержка в JavaScript-экосистеме.

**Причины выбора**: JSON выбран за его простоту и нативную поддержку в JavaScript. Альтернатива YAML была рассмотрена, но отклонена из-за избыточной сложности для небольшого числа моделей.

**4. SQLite**

SQLite использовался для хранения 3D-моделей и конфигураций как легковесная реляционная база данных для хранения данных

**Особенности**:

* Отсутствие необходимости в отдельном серверном процессе, что снижает нагрузку на систему;
* Простота интеграции с веб-приложением благодаря компактности и встраиваемости;
* Достаточная производительность для небольшого объёма данных;
* Поддержка параметризованных запросов для безопасности.

**Причины выбора**: SQLite выбрана за её лёгкость, компактность и отсутствие необходимости в серверной инфраструктуре. Альтернатива MongoDB была отклонена из-за минимальной ожидаемой нагрузки и избыточных требований к ресурсам.

**5. Bun.js с Express**

Bun.js и фреймворк Express выбраны для создания сервера-эмулятора рентгеновского аппарата.

**Особенности**:

* Высокая совместимость с JavaScript-экосистемой, включая Three.js и JSON;
* Простота реализации REST API для обработки запросов от интерфейса;
* Быстрое развертывание и поддержка асинхронных операций;
* Встроенная поддержка SQLite, обеспечивающая высокую скорость запросов.

**Причины выбора**: Bun.js с Express выбраны за высокую производительность и простоту интеграции с SQLite. Альтернатива Node.js была отклонена из-за меньшей скорости работы для задач, для которых оптимизировано приложение.

**6. Socket.IO**

Socket.IO использован для организации двунаправленного соединения в реальном времени между клиентом, сервером и эмулятором.

**Особенности**:

* Поддержка WebSocket с автоматическим переподключением, что обеспечивает стабильность соединения;
* Простота реализации событийной модели для передачи команд и состояний;
* Высокая совместимость с JavaScript-экосистемой, включая Bun.js и Express;
* Возможность рассылки обновлений всем подключённым клиентам через broadcast.

**Причины выбора**: Socket.IO выбран за простоту интеграции, поддержку реального времени и надёжность. Альтернативы, такие как WebSocket API, были отклонены: WebSocket API требует больше кода для обработки переподключений.

* 1. Общая архитектура приложения

Приложение состоит из трех частей: клиентской, серверной и эмулятора. Они общаются между собой по протоколам http/https и websocket. Http использован для передачи больших данных, например модели с сервера на клиент. Вебсокеты используются для сверхбыстрой передачи данных от клиента к эмулятору, сквозь сервер. Один сервер может взаимодействовать с несколькими эмуляторами, и множеством клиентов. Пользователь взаимодействует только с клиентом, путем управления манипуляторами: мышкой, джойстиком, клавиатурой.

На рисунке 1.6 представелна общая схема архитектуры.

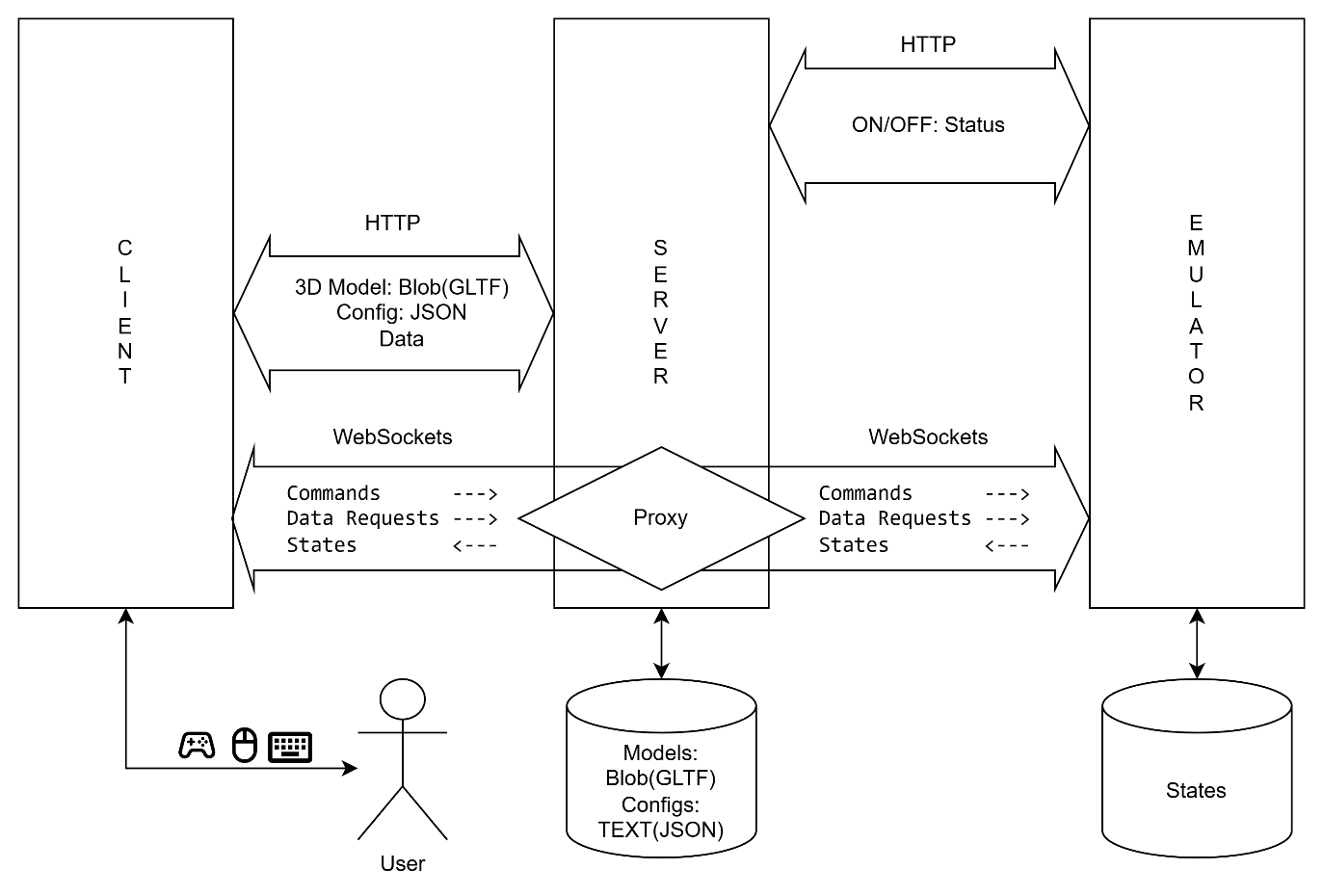


Рисунок 1.6 – Общая архитектура приложения

* 1. Выводы по анализу предметной области

Анализ предметной области показал, что разработка веб-интерфейса для телеуправляемого рентгеновского аппарата требует использования современных технологий 3D визуализации и веб-разработки. Выбор Blender, Three.js, JSON, SQLite и Bun.js, Socket.io обусловлен их функциональностью, производительностью и совместимостью. В отличие от существующих аналогов, проект фокусируется на интерактивности и эмуляции работы оборудования, что делает его уникальным решением для обучения и демонстрации.

1. Создание 3D-модели рентгеновского аппарата
   1. Определение правил создания 3D-модели

Создание 3D-модели рентгеновского аппарата для последующей интеграции в веб-интерфейс требует строгого соблюдения технических ограничений, обусловленных особенностями формата glTF и библиотеки Three.js. Данные ограничения обеспечивают корректное отображение модели, её интерактивность и производительность в веб-приложении. Ниже описаны основные требования и рекомендации, применяемые при разработке модели в Blender.

1. **Номенклатура элементов**

Все объекты в сцене Blender должны иметь уникальные имена на английском языке без пробелов. Для разделения слов используется нижнее подчёркивание (например, XRay\_Table, Detector\_Arm). Это требование обусловлено необходимостью корректного парсинга имён объектов библиотекой Three.js, которая использует их для привязки обработчиков событий, управляющих перемещением и вращением элементов модели. Например, в файле useModelLoader.ts имена объектов из glTF-сцены (child.name) сопоставляются с настройками управления (modelControls.models), что требует строгого соответствия имён.

1. **Использование Empty Objects вместо групп**

В Blender вместо групп вершин или объектов используются пустые объекты (Empty Objects). Группы, созданные в Blender, не экспортируются в формат glTF, что приводит к потере иерархической структуры при загрузке модели в Three.js. Пустые объекты позволяют сохранять иерархию и служат контейнерами для дочерних элементов, таких как части рентгеновского аппарата (стол, детектор, излучатель). Например, для управления движением стола создаётся Empty Object с именем Table\_Parent, к которому привязываются дочерние меши. Это обеспечивает возможность назначения обработчиков на всю группу через родительский объект, как указано в ModelControlsInputs.tsx.

1. **Ограничения формата glTF**

Формат glTF поддерживает ограниченный набор данных, поэтому в модели используются только совместимые элементы: меши, материалы, текстуры, трансформации (положение, вращение, масштаб) и базовые анимации. Элементы, такие как модификаторы, симуляции физики или специфические шейдеры Blender (например, Principled BSDF с неподдерживаемыми узлами), не экспортируются корректно. Для рентгеновского аппарата применялись простые PBR-материалы с текстурами (диффузная, нормали, металлическая), оптимизированные для WebGL. В файле useModelLoader.ts модель загружается с помощью GLTFLoader, который обрабатывает только glTF-совместимые данные.

1. **Центрирование объектов**

Центр каждого объекта должен быть выровнен по осям вращения, чтобы обеспечить корректное вращение в Three.js. Если центр объекта смещён, вращение вокруг осей (например, rotation.x, rotation.y) будет некорректным, что приведёт к ошибкам в интерфейсе управления. В Blender перед экспортом применяется операция «Set Origin to Geometry», а все трансформации (положение, вращение, масштаб) фиксируются через «Apply All Transforms». Это гарантирует, что модель в сцене Three.js правильно интерпретирует координаты и углы.

1. **Предварительное применение трансформаций**

Все операции масштабирования, перемещения и вращения должны быть применены в Blender перед экспортом. Неприменённые трансформации (например, масштаб, заданный в режиме редактирования) могут привести к искажениям при загрузке в Three.js. Для этого используется команда «Apply All Transforms» для каждого объекта. Например, модель стола рентгеновского аппарата была масштабирована до реальных размеров (в метрах) и экспортирована с применёнными трансформациями, чтобы обеспечить точное позиционирование в сцене.

1. **Оптимизация модели**

Для обеспечения высокой производительности в веб-приложении модель оптимизируется: количество полигонов минимизируется (до 50–100 тысяч для всей сцены), текстуры сжимаются (разрешение не превышает 2K), а ненужные элементы (внутренние детали, невидимые в интерфейсе) удаляются.

Таким образом, соблюдение перечисленных требований позволяет создать 3D-модель рентгеновского аппарата, соответствующую техническим ограничениям glTF и Three.js, а также обеспечивающую интерактивность и производительность в веб-интерфейсе.

* 1. Создание 3D-модели

Создание 3D-модели рентгеновского аппарата выполнялось в программном обеспечении Blender с учетом технических требований формата glTF и библиотеки Three.js, обеспечивающих корректную интеграцию модели в веб-интерфейс. Моделирование всех компонентов аппарата осуществлялось по единому алгоритму, включающему последовательное применение модификаторов и инструментов Blender для достижения необходимой геометрической точности и визуальной достоверности. Ниже описан процесс создания каждого элемента модели.

Для создания каждого компонента рентгеновского аппарата применялся следующий алгоритм:

1. **Создание базовой геометрии**: В качестве исходного объекта использовался примитив (например, куб или цилиндр), который служил основой для дальнейшего моделирования.
2. **Применение модификаторов**: Последовательно применялись модификаторы для формирования сложной геометрии:

* **Bevel** (скругление) с ограничением по весу ребер для создания плавных углов.
* **Subdivision Surface** для сглаживания поверхностей и повышения детализации.
* **Mirror** (при необходимости) для симметричного моделирования, что сокращало объем работы.
* **Smooth by Angle** для дополнительного сглаживания поверхностей с учетом углов.

1. **Редактирование геометрии**: Выполнялись операции по перемещению, созданию и редактированию вершин, ребер и граней. Для разделения геометрии применялась команда «Loop Cut» (Ctrl+R).
2. **Итеративная доработка**: Процесс повторялся до достижения соответствия модели реальному объекту.

Данный подход обеспечивал единообразие в создании всех компонентов модели, упрощая процесс и обеспечивая совместимость с требованиями экспорта в glTF.

Моделирование основания рентгеновского аппарата начиналось с создания куба. Верхняя грань куба была обработана с помощью операции **Inset** для создания выпуклости. Далее применялись модификаторы **Bevel** и **Subdivision Surface** для формирования скругленных краев и сглаживания поверхности.

На рисунке 2.2.1 представлено основание рентгена.

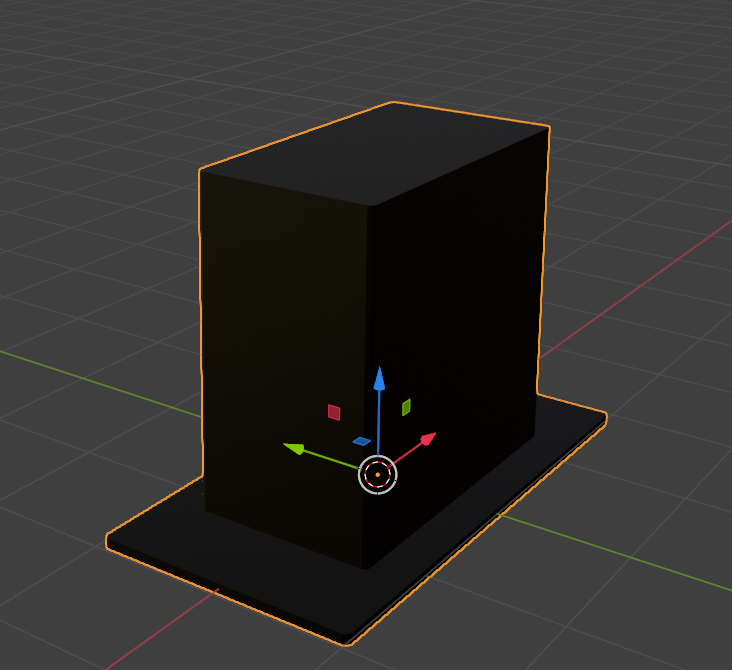


Рисунок 2.2.1 – Основание рентгена

Коробка управления состояла из двух элементов, созданных на основе кубов: черной и белой частей. Для каждой части применялись модификаторы в следующем порядке:

* **Mirror** для создания симметричной геометрии.
* **Bevel** для скругления углов.
* **Subdivision Surface** для сглаживания поверхностей.
* **Smooth by Angle** для дополнительной обработки угловых переходов.

На рисунке 2.2.2 представлена коробка управления рентгеновским аппаратом.



Рисунок 2.2.2 – Коробка управления

Ручка вращателя создавалась на основе цилиндра. Геометрия формировалась с использованием операции **Inset** для создания дополнительных полигонов, после чего применялись модификаторы **Bevel** и **Subdivision Surface** для получения необходимой формы и сглаживания поверхности. На рисунке 2.2.3 представлен вращатель.

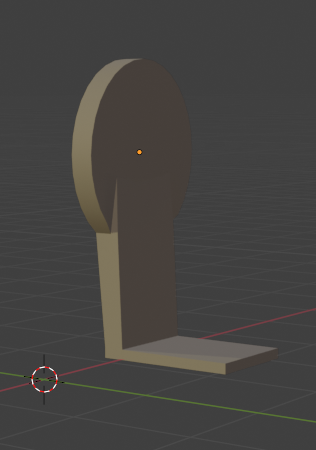


Рисунок 2.2.3 – Вращатель

Выдвижной стол в основном был создан из множества кубов. Из них создана пластина стола, и рельсы, по которым ездит детектор во время работы.

Бортик формировался из куба, который был разделен пополам, путем выдавливания (Extrude) отростков из грани куба. Для достижения финальной формы применялись модификаторы **Bevel**, **Subdivision Surface**, **Smooth by Angle** и **Mirror** для симметричного отображения. На рисунке 2.2.4 представлен подвижный стол.

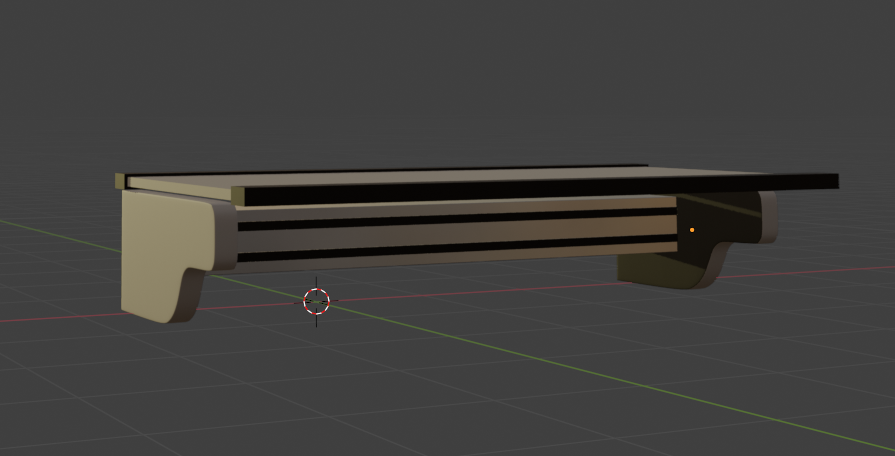


Рисунок 2.2.4 – подвижной стол

Моделирование детектора начиналось с создания внутренней части, после чего формировалась внешняя оболочка («крышка»). Процесс включал следующие этапы:

* Выделение контура внутренней части.
* Создание копии поверхности, предназначенной для формирования крышки.
* Выделение копии в отдельный объект.
* Применение модификатора **Solidify** для придания толщины плоской поверхности.

Данный подход позволил оптимизировать процесс создания сложных элементов с внутренней и внешней структурой.

На рисунке 2.2.5 представлен детектор.

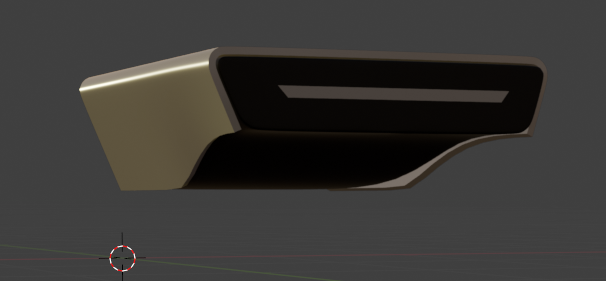


Рисунок 2.2.5 – Детектор

Основной крепеж состоял из двух частей: белой и черной. Белая часть создавалась из куба с применением модификаторов **Bevel** и **Subdivision Surface** для формирования скругленных форм. Черная часть, представляющая вставку, моделировалась аналогичным образом, но с меньшими размерами и дополнительной детализацией.

На рисунке 2.2.6 представлен основной крепеж рентгеновского аппарата.



Рисунок 2.2.6 – Основной крепеж

Моделирование коллиматора и рентгеновской трубки начиналось с создания внутренней части, аналогично процессу для детектора. Внешняя оболочка формировалась следующим образом:

* Выделение контура внутренней части.
* Создание копии поверхности для формирования внешней оболочки.
* Выделение копии в отдельный объект.

Применение модификаторов **Bevel** и **Subdivision Surface** обеспечивало сглаживание и реалистичность компонентов.

На рисунке 2.2.7 представлен основной коллиматор и рентгеновская трубка.

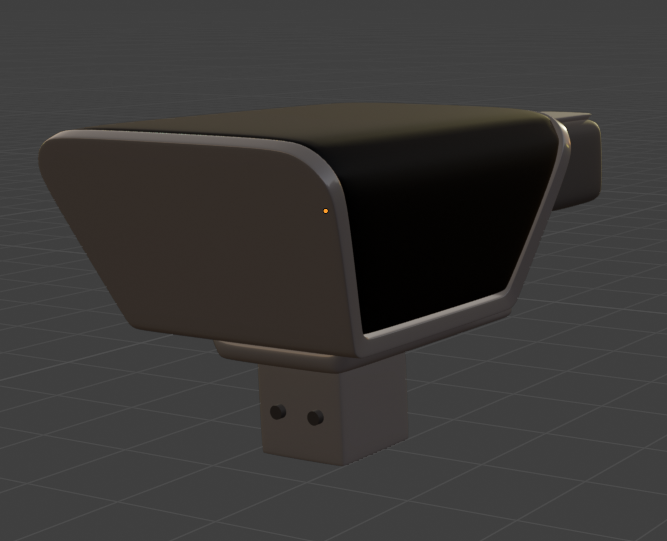


Рисунок 2.2.7 – Коллиматор и рентгеновская трубка

* 1. Текстурирование

Процесс текстурирования 3D-модели рентгеновского аппарата осуществлялся в программном обеспечении Blender с использованием стандартного шейдера Principled BSDF, встроенного в среду разработки. Данный шейдер применялся для всех компонентов модели, с учетом индивидуальной настройки параметров для достижения необходимого визуального эффекта и реалистичности текстур. Настройка параметров включала изменение значений Base Color, Metallic, Roughness, IOR, Alpha и других характеристик, что позволило адаптировать материал к геометрическим особенностям каждого элемента аппарата. Использование единого шейдера обеспечило согласованность текстурного оформления и упростило процесс интеграции модели в веб-интерфейс с учетом ограничений формата glTF и библиотеки Three.js.

На рисунке 2.3 представлен один из материалов.

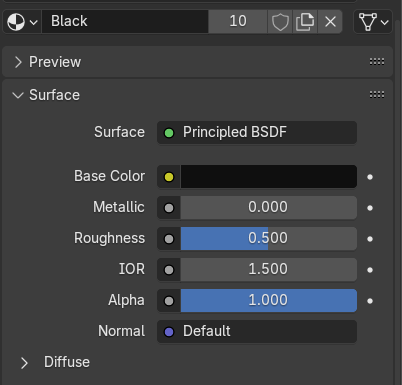


Рисунок 2.3 - Материал

* 1. Выводы

Созданная 3D-модель рентгеновского аппарата для последующей интеграции в веб-интерфейс соблюдает строгие технические ограничения, обусловленные особенностями формата glTF и библиотеки Three.js. Данные ограничения обеспечивают корректное отображение модели, её интерактивность и производительность в веб-приложении.

1. Отображение 3D-модели рентгеновского аппарата

Отображение 3D-модели телеуправляемого рентгеновского аппарата является центральным компонентом веб-интерфейса, обеспечивающим визуализацию подвижных элементов оборудования в реальном времени на основе параметров, получаемых от эмулятора. Для реализации использована библиотека Three.js, которая предоставляет инструменты для рендеринга 3D-графики в браузере, и библиотека socket.io-client для интеграции с серверной частью. Фронтенд-приложение отвечает за загрузку модели, настройку сцены, интерактивное управление камерой, синхронизацию параметров и обработку ошибок. В данной главе описаны архитектура фронтенда, инициализация сцены, интерактивное управление, синхронизация с эмулятором, а также управление анимацией и обработка исключительных ситуаций.

* 1. Настройка фронтенд-приложения

Фронтенд-приложение разработано как одностраничное веб-приложение (SPA), обеспечивающее рендеринг 3D-модели и взаимодействие с серверной частью с использованием фреймворка React и языка TypeScript, что обеспечивает компонентную архитектуру, строгую типизацию и удобство поддержки кода. Для управления состоянием приложения применяется Redux Toolkit (RTK), предоставляющий централизованное хранилище для данных модели, параметров и состояний соединения. Для сборки и разработки использован инструмент Vite, выбранный за высокую скорость горячей перезагрузки, поддержку модульного JavaScript и простоту интеграции с библиотеками. Vite обеспечивает автоматическую сборку и минимизацию ресурсов, что упрощает разработку и отладку.

Основные зависимости, указанные в package.json, включают:

* react, react-dom: для создания компонентной структуры интерфейса.
* three, @types/three: для рендеринга 3D-графики.
* socket.io-client: для взаимодействия с бэкендом в реальном времени.
* typescript, vite: для типизации и сборки.
* redux, rtk: для хранения состояний

Основной библиотекой для рендеринга является Three.js, которая обеспечивает кроссбраузерную поддержку WebGL и богатый API для работы с 3D-объектами. Для обмена данными с бэкендом в реальном времени применена библиотека socket.io-client, совместимая с серверной реализацией Socket.IO (раздел 5.4).

Основные компоненты приложения:

* **ModelViewer** (src\components\ModelViewer.tsx): центральный компонент, отвечающий за рендеринг 3D-сцены и интеграцию элементов управления. Он объединяет хуки для настройки сцены и загрузки модели, а также отображает ошибки загрузки.
* **useThreeSetup** (src\hooks\useThreeSetup.ts): хук, инициализирующий сцену, рендер, камеру и управление камерой с использованием Three.js и TrackballControls.
* **useModelLoader** (src\hooks\useModelLoader.ts): хук для асинхронной загрузки glTF-модели, ее парсинга и синхронизации с состоянием приложения.
* **socket.ts** (src\socket.ts): модуль для взаимодействия с сервером через Socket.IO, обеспечивающий передачу команд и состояний в онлайн-режиме.

Взаимодействие компонентов:

Фронтенд взаимодействует с сервером через REST API для получения данных модели (/api/models/:name) и настроек (/api/json/:name). Состояние модели, включая позиции и повороты компонентов, хранится в Redux (state.model). В оффлайн-режиме управление осуществляется локально с использованием JSON-конфигурации, а в онлайн-режиме команды передаются на сервер через Socket.IO. Например, изменение позиции детектора отправляется как команда { command: "set", path: "detector/position/x", value: 5 }.

Преимущества архитектуры:

* **Модульность**: разделение логики рендеринга, загрузки и управления упрощает поддержку и расширение.
* **Масштабируемость**: JSON-конфигурация позволяет добавлять новые элементы управления без изменения кода.
* **Производительность**: использование мемоизации в React и оптимизированных хуков минимизирует лишние перерисовки.

Файл src/vite-env.d.ts определяет типы для импорта ресурсов используемых библиотек. Основной входной точкой является index.html, который загружает компонент App через main.tsx. Компонент App монтирует канвас для рендеринга сцены и устанавливает соединение с бэкендом через client-socket.io. Настройка CORS в бэкенде (описана в разделе 5.4) позволяет фронтенду безопасно взаимодействовать с сервером.

На рисунке 3.1 представлена архитектура клиентской части приложения.

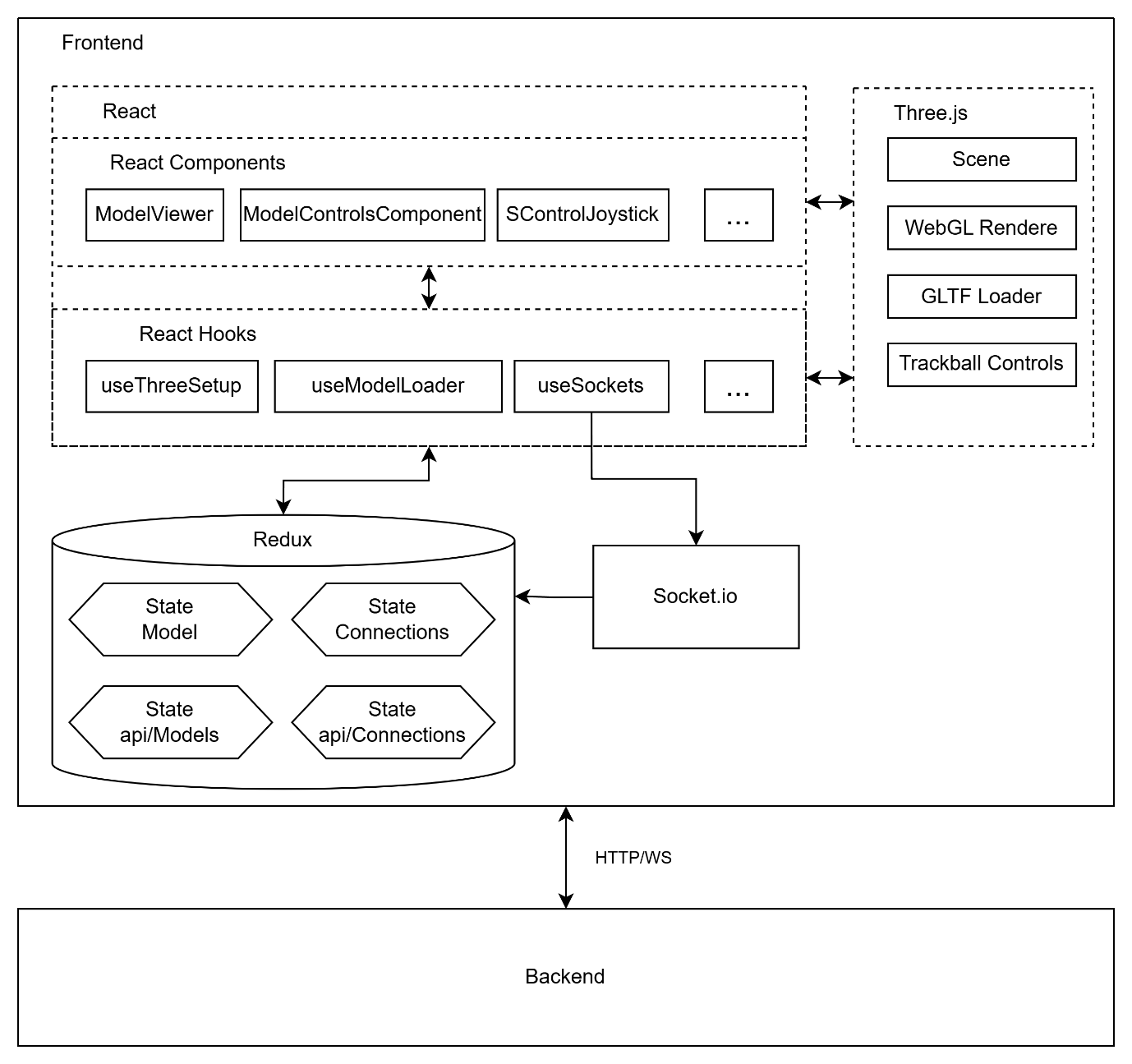


Рисунок 3.1 – Архитектура клиентской части приложения

* 1. Хранение и обработка данных для модели

Для обеспечения корректного отображения и управления 3D-моделью рентгеновского аппарата необходима эффективная организация хранения и обработки данных. Данные модели и настройки хранятся как на сервере, так и на клиенте, с учетом требований к производительности и гибкости.

Формат и хранение данных:

* **Формат модели**: Используется glTF (GL Transmission Format), который обеспечивает компактность, поддержку иерархии объектов, материалов и анимаций. Этот формат оптимален для веб-приложений благодаря малому размеру файлов и совместимости с Three.js.
* **Хранение на сервере**: Модели хранятся в базе данных SQLite (data.db) как бинарные файлы. Доступ осуществляется через REST API (/api/models/:name). Настройки управления хранятся в виде JSON-файлов (/api/json/:name).
* **Хранение на клиенте**: Текущие позиции и повороты компонентов модели сохраняются в состоянии Redux (state.model.positions). Настройки управления (state.model.modelControls) загружаются из JSON-файлов и также хранятся в Redux.

Структура данных:

1. **ModelControls**: Объект, содержащий конфигурацию компонентов модели и элементы управления. Пример представлен в листинге 3.2.1:

Листинг 3.2.1 - ModelControls

|  |
| --- |
| {  "models": {  "detector": {  "position": { "x": [-10, 10], "y": [-5, 5], "z": [-15, 15] },  "rotation": { "x": [-1.57, 1.57] }  }  },  "controlElements": [  { "name": "Move Detector", "element": "Joystick", "position": "Center", "props": { "xpath": "detector/position/x", "ypath": "detector/position/y" } }  ]  } |

Здесь models определяет допустимые диапазоны позиций и поворотов, а controlElements — элементы управления (джойстик, кнопки).

1. **ModelPositions**: Объект с текущими значениями позиций и поворотов, например:

Листинг 3.2.2 – **ModelPositions**

|  |
| --- |
| {  "detector": { "position": { "x": 0, "y": 0, "z": 0 }, "rotation": { "x": 0 } },  "table": { "position": { "y": 0 } }  } |

1. **JSON-настройки**: Определяют типы и параметры элементов управления, такие как пути (path), значения и позиции на интерфейсе.

Обработка данных:

* **Парсинг модели**: Модель в формате glTF загружается с помощью GLTFLoader. После парсинга объекты модели сопоставляются с конфигурацией modelControls для извлечения начальных позиций и поворотов.
* **Обновление данных**: Изменения позиций и поворотов обрабатываются через действия Redux (updateModelPositionLocal, updatePositionsLocal). В оффлайн-режиме обновления применяются локально, в онлайн-режиме — отправляются на сервер через Socket.IO.
* **Пример обработки**: Для команды { command: "set", path: "detector/position/x", value: 5 } значение проверяется на соответствие диапазону (например, [-10, 10]) и применяется к состоянию (state.model.positions.detector.position.x).

На рисунках 3.2.1 - 3.2.2 представлена визуалицзация данных содержащихся в хранилище redux во время работы с одной из моделей. Для визуализации использован Redux DevTools.

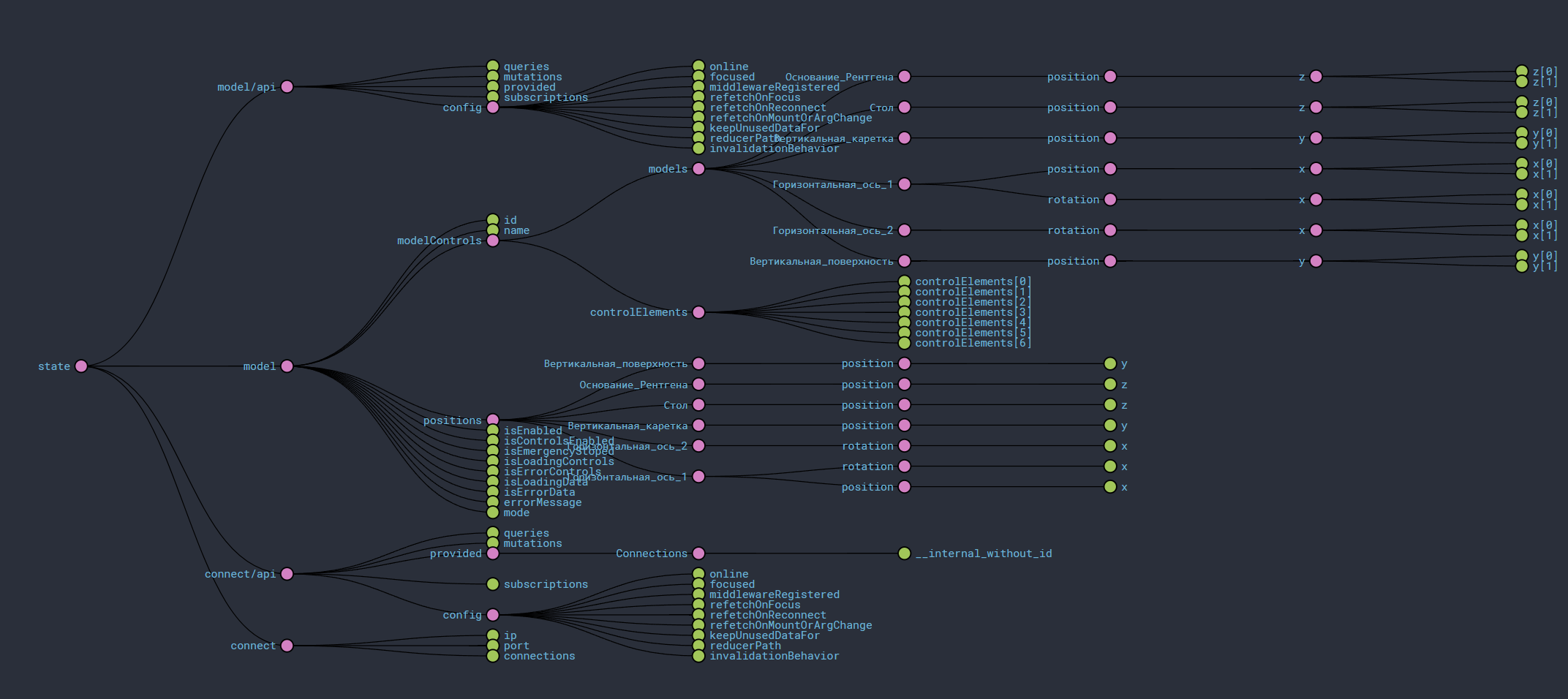


Рисунок 3.2.1 – Хранилище redux часть 1

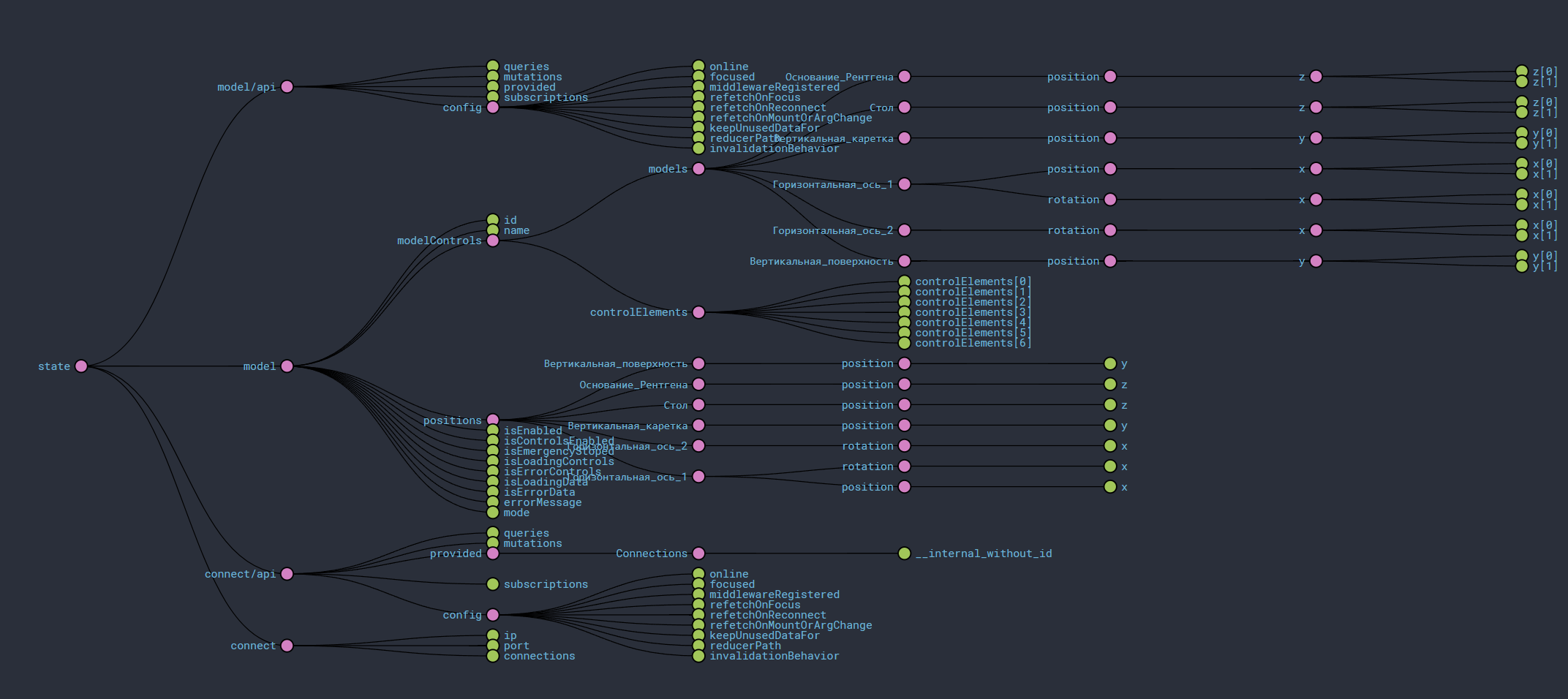


Рисунок 3.2.1 – Хранилище redux часть 2

* 1. Настройка сцены

Настройка сцены является критически важным этапом для корректного отображения 3D-модели. Она включает инициализацию сцены, настройку рендера, камеры, управления и освещения, выполненную с использованием библиотеки Three.js.

Инициализация и рендеринг:

* **Сцена**: Создается объект THREE.Scene, который служит контейнером для модели, камеры и источников света.
* **Рендер**: Используется WebGLRenderer с включенным сглаживанием (antialias: true) для повышения качества изображения. Поддержка теней активируется через shadowMap.enabled = true. Цвет фона устанавливается как серый (0x3f3f3f, непрозрачность 1). Разрешение рендера по умолчанию составляет 600x600 пикселей, но адаптируется под размер контейнера.

Настройка камеры:

* Используется PerspectiveCamera с углом обзора 75°, соотношением сторон 1:1 и дальностью рендера от 0.1 до 1000 единиц.
* Начальная позиция камеры: (20, 20, -5), с фокусом на центр сцены (lookAt(scene.position)), что обеспечивает оптимальный обзор модели.

Управление камерой:

Для интерактивного управления используется TrackballControls, позволяющий вращать, масштабировать и панорамировать сцену. Параметры настроены следующим образом:

* Скорость вращения: 1.0.
* Скорость масштабирования: 1.2.
* Скорость панорамирования: 0.8.
* Динамическое затухание: 0.2 для плавности движений.
* Разрешены масштабирование и панорамирование (noZoom = false, noPan = false).

Освещение:

* **AmbientLight**: Добавлен источник окружающего света с интенсивностью 1.6 и цветом 0xffffff для равномерного освещения сцены.
* **DirectionalLight**: Направленный источник света с интенсивностью 2.3, расположенный в позиции (140, 140, -15). Поддерживает тени с настройками
* Размер карты теней: 1024x1024 пикселей.
* Область охвата камеры теней: от -10 до 10 по осям X и Y.
* Дальность теней: от 1 до 1000 единиц.
* Смещение теней: -0.0002 для устранения артефактов рендеринга.

На рисунке 3.3.1 представлен эскиз расположения обьектов на сцене.

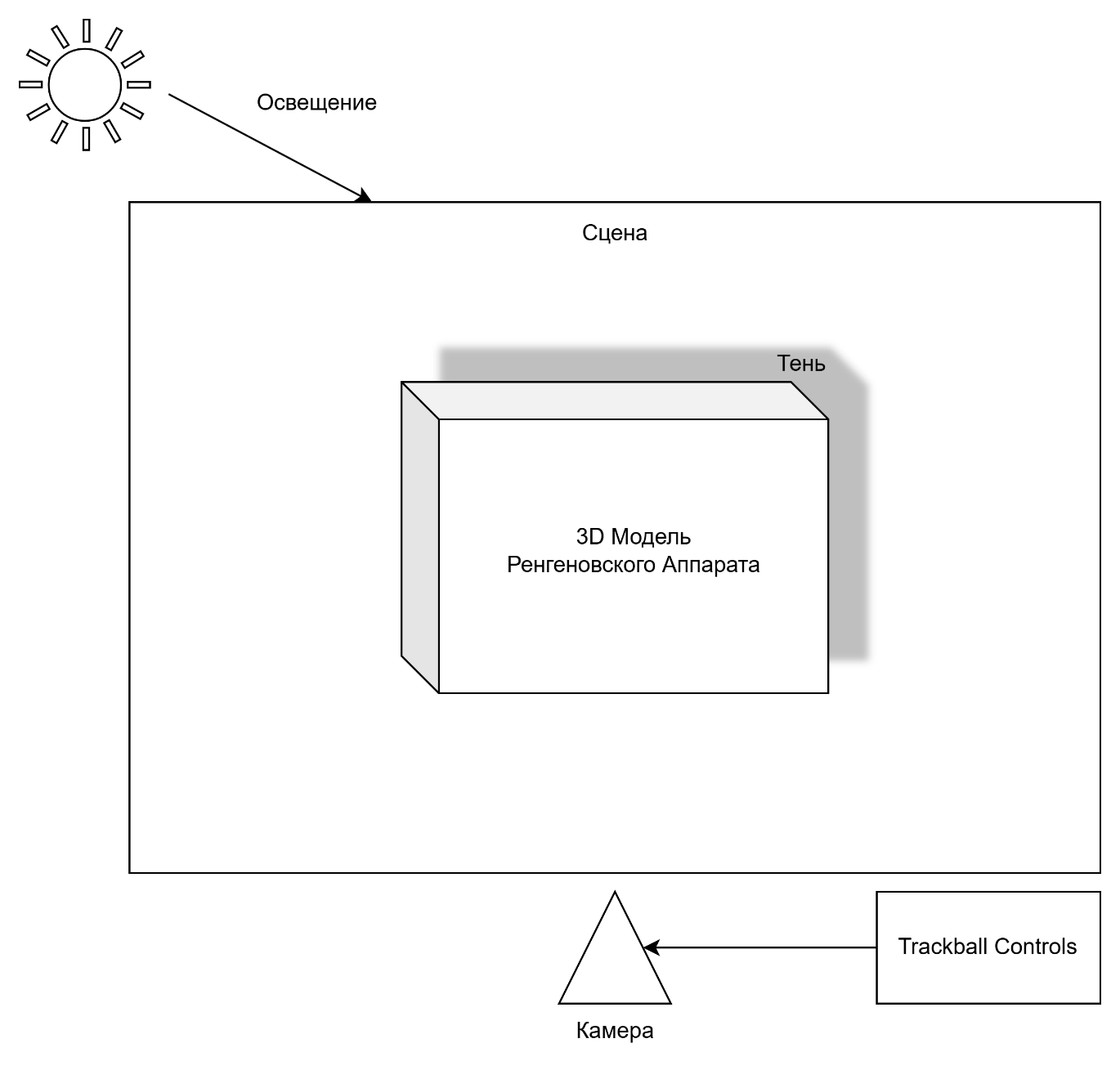


Рисунок 3.3.1 – Эскниз обьекотов на сцене

На рисунке 3.3.2 представлена модель с тенями, настроенными данными приведенными выше параметрами.

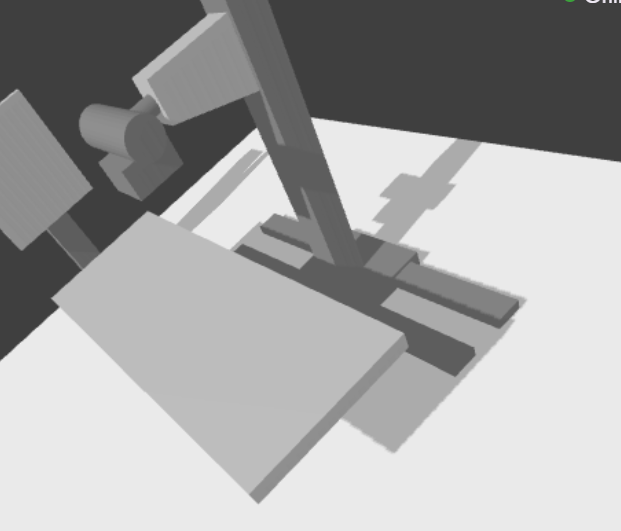


Рисунок 3.3.2 - Модель с тенями

Настройка сцены реализована в функции setupLCC, которая инкапсулирует все описанные параметры.

* 1. Загрузка модели

Загрузка 3D-модели рентгеновского аппарата представляет собой важный этап реализации веб-интерфейса, обеспечивая корректное отображение и взаимодействие с моделью. Для достижения высокой производительности и гибкости процесс загрузки поддерживает два сценария: получение данных с сервера через REST API и обработку локальных файлов, загруженных пользователем. Основным инструментом для этого служит хук useModelLoader, реализованный в файле src\hooks\useModelLoader.ts, который асинхронно обрабатывает данные модели в формате glTF, выбранном за компактность и совместимость с библиотекой Three.js.

При серверной загрузке хук инициирует запрос к API по адресу /api/models/:name с использованием асинхронного действия updateModelDataAsync, определенного в срезе Redux (model.slice.ts). Сервер возвращает бинарные данные модели в виде ArrayBuffer, которые затем передаются в GLTFLoader для парсинга. В случае локальной загрузки, например, на страницах AddModel или EditModel, пользователь выбирает файл через интерфейс, а обработчик handleFileChange считывает его содержимое как ArrayBuffer, передавая данные непосредственно в useModelLoader. Такой подход позволяет предварительно просматривать или редактировать модель без обращения к серверу.

После получения данных GLTFLoader.parse преобразует их в сцену с иерархией объектов, готовых для рендеринга. Чтобы избежать конфликтов, хук сначала удаляет предыдущую модель из сцены с помощью метода scene.remove(modelRef.current), а затем добавляет новую, вызывая scene.add(gltf.scene). Для обеспечения визуальной реалистичности метод gltf.scene.traverse рекурсивно обходит объекты модели, активируя тени через свойства castShadow и receiveShadow. Кроме того, из модели извлекаются начальные позиции и повороты компонентов, таких как детектор или стол, которые сопоставляются с конфигурацией modelControls.models и сохраняются в состоянии Redux через действие updatePositionsLocal. Это позволяет синхронизировать отображение модели с текущими настройками, например, устанавливая part.position.x в соответствии с state.model.positions.detector.position.x.

Особое внимание уделено обработке ошибок. Если парсинг модели завершается неудачно, например, из-за некорректного формата glTF, в состояние state.model.errorMessage записывается сообщение, такое как «Error parsing model», которое отображается в интерфейсе. Успешная загрузка, напротив, сбрасывает ошибки через действие resetErrors, обеспечивая чистое состояние.

На рисунке 3.4 представлена диаграмма последовательности для загрузки серверной и локальной загрузки модели. При серверной загрузке, модель загружается с сервера. При локальной загрузки пользователь сам загружает модель со своего устройства. Локальная загрузка по функционалу не отличается от серверной, и создана для предпросмотра модели перед загрузкой ее на сервер.

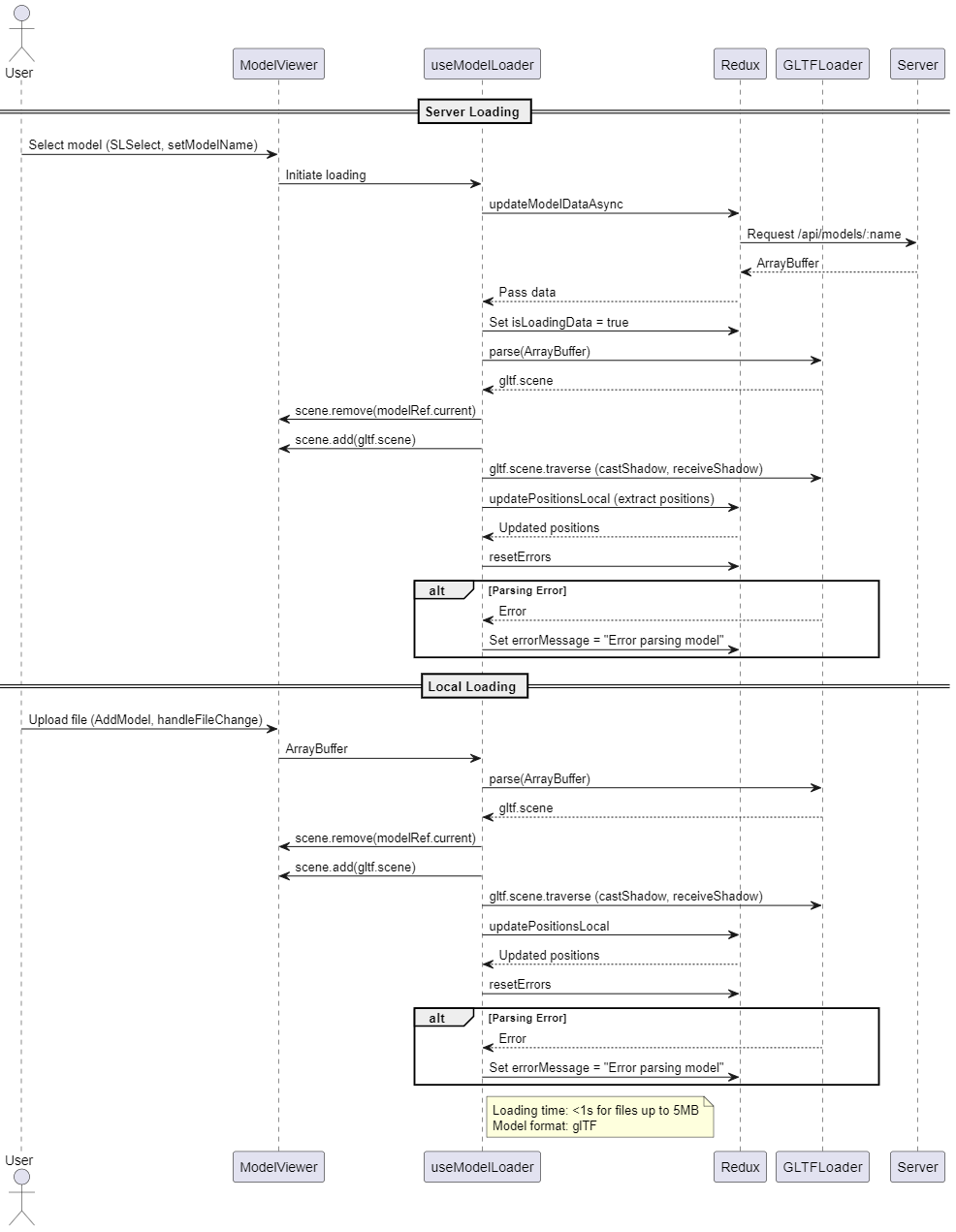


Рисунок 3.4 – Диаграмма последовательности загрузки модели

* 1. Интеграция с пользовательским интерфейсом

Интеграция 3D визуализации с пользовательским интерфейсом обеспечивает интуитивное и функциональное управление рентгеновским аппаратом, поддерживая как оффлайн-, так и онлайн-режимы работы. Центральную роль играет компонент ModelViewer. Этот компонент отвечает за рендеринг сцены, отображение динамических элементов управления и обработку ошибок. Он использует хуки useThreeSetup для инициализации сцены и useModelLoader для загрузки модели, а также поддерживает адаптивное изменение размеров сцены через свойство size. При возникновении ошибок, таких как сбой загрузки, ModelViewer отображает уведомления, основанные на значениях ошибок, присылаемых с серверва или загрузчиком модели, а при успешной загрузке активирует элементы управления, заданные в modelControls.

Приложение функционирует в двух режимах. В оффлайн-режиме управление моделью осуществляется локально, опираясь на JSON-конфигурацию, загруженную через API. Позиции и повороты компонентов обновляются через действие Redux updateModelPositionLocal, что позволяет мгновенно отражать изменения в 3D сцене. В онлайн-режиме взаимодействие с аппаратным обеспечением реализовано через Socket.IO. Команды, такие как clientCommand, и состояния, такие как clientState, передаются в реальном времени, а параметры, полученные через clientSetParameters, обновляют стейт позиций модели, синхронизируя модель с физическим устройством.

Для управления моделью разработаны специализированные компоненты, которые динамически размещаются в интерфейсе на основе конфигурации modelControls.controlElements. Джойстик, реализованный в SControlJoystick с использованием библиотеки react-joystick-component, позволяет перемещать компоненты модели, например, детектор, по двум осям. Стрелочные кнопки, представленные компонентом SArrowButtons, обеспечивают пошаговое изменение позиции по одной оси, поддерживая направления «вверх/вниз» или «влево/вправо» с аналогичным шагом. Кнопка установки, реализованная в SSetButton, применяет предустановленные значения позиций, например, для возврата в «домашнее» состояние, с плавным переходом в оффлайн-режиме. Кнопка питания (SPowerButton) управляет включением и выключением модели, обновляя состояние включения/выключения модели и выполняя анимацию перехода. В случае необходимости немедленной остановки предусмотрена кнопка аварийной остановки (SEmergencyStopButton), которая устанавливает isEmergencyStoped в значение true, блокируя управление.

Размещение элементов управления организовано через CSS-сетку, определенную в файле ControlGrid.css. Возможные позиции включают TopLeft, Center, BottomRight и другие, что позволяет гибко компоновать интерфейс. Компонент ModelControlsComponent автоматически распределяет элементы по этим позициям, основываясь на конфигурации modelControls. Для отображения состояния аппарата используется компонент SStatusIndicator, который показывает текущий статус: «Online» для активного управления, «Working» при временной блокировке, «Emergency Stopped» при аварийной остановке или «Offline» при выключенной модели. Статус обновляется через состояние Redux, включая параметры isEnabled, isControlsEnabled и isEmergencyStoped.

Пользователь может взаимодействовать с моделью не только через элементы управления, но и через ручной ввод. Компонент ModelControlsInputs предоставляет числовые поля для корректировки позиций и поворотов, напрямую обновляя state.model.positions. На странице Viewer предусмотрен компонент SLSelect, который позволяет выбрать модель из списка, полученного через API /api/models. Выбор модели инициирует действия setModelName и updateModelControlsAsync, обновляя сцену и настройки. Интерфейс адаптирован для удобства благодаря использованию UnoCSS и поддерживает динамическую компоновку, что делает управление интуитивным.

1. Создание пользовательского интерфейса
   1. Общее описание интерфейса

Пользовательский веб-интерфейс представляет собой интуитивно понятный инструмент, разработанный для медицинского персонала, чтобы упростить управление телеуправляемым рентгеновским аппаратом. Он позволяет оператору взаимодействовать с 3D-моделью аппарата, настраивать параметры, такие как положение стола или угол поворота, и получать немедленную обратную связь о состоянии системы. Интерфейс адаптирован под пользователей без глубоких технических знаний, что отличает его от инженерных программ, таких как Voron, рассмотренная в главе 1. Основное внимание уделено эргономике, соответствующей стандартам ISO 9241-210, которые требуют минимизации когнитивной нагрузки и обеспечения безопасности при работе с медицинским оборудованием.

Интерфейс поддерживает два режима: онлайн, где оператор управляет реальным аппаратом или эмулятором через сетевое соединение, и оффлайн, позволяющий тренироваться с локальной моделью без подключения. Это делает систему универсальной, подходящей как для реальных процедур, так и для обучения. Визуальная структура повторяет логику физических пультов рентгеновских аппаратов: ключевые элементы управления, такие как джойстик и кнопка аварийной остановки, расположены для быстрого доступа, а их дизайн исключает случайные ошибки. Например, критические действия, такие как удаление модели или соединения, требуют подтверждения через всплывающие окна, что повышает безопасность.

Адаптивный дизайн, реализованный с учетом разных разрешений экрана (от 640x480 до 1920x1080), обеспечивает комфортную работу на настольных компьютерах, планшетах и мобильных устройствах. Это особенно важно для мобильных медицинских станций, где операторы могут использовать планшеты. Читаемость текста, цветовые акценты и интуитивная навигация делают интерфейс удобным даже в условиях ограниченного времени или сложного освещения, типичных для медицинских учреждений. Интерфейс не только упрощает управление сложным оборудованием, но и сокращает время обучения, позволяя медперсоналу сосредоточиться на выполнении процедур.

На рисунке 4.1 представлен интерфейс с пояснениями к управлению.

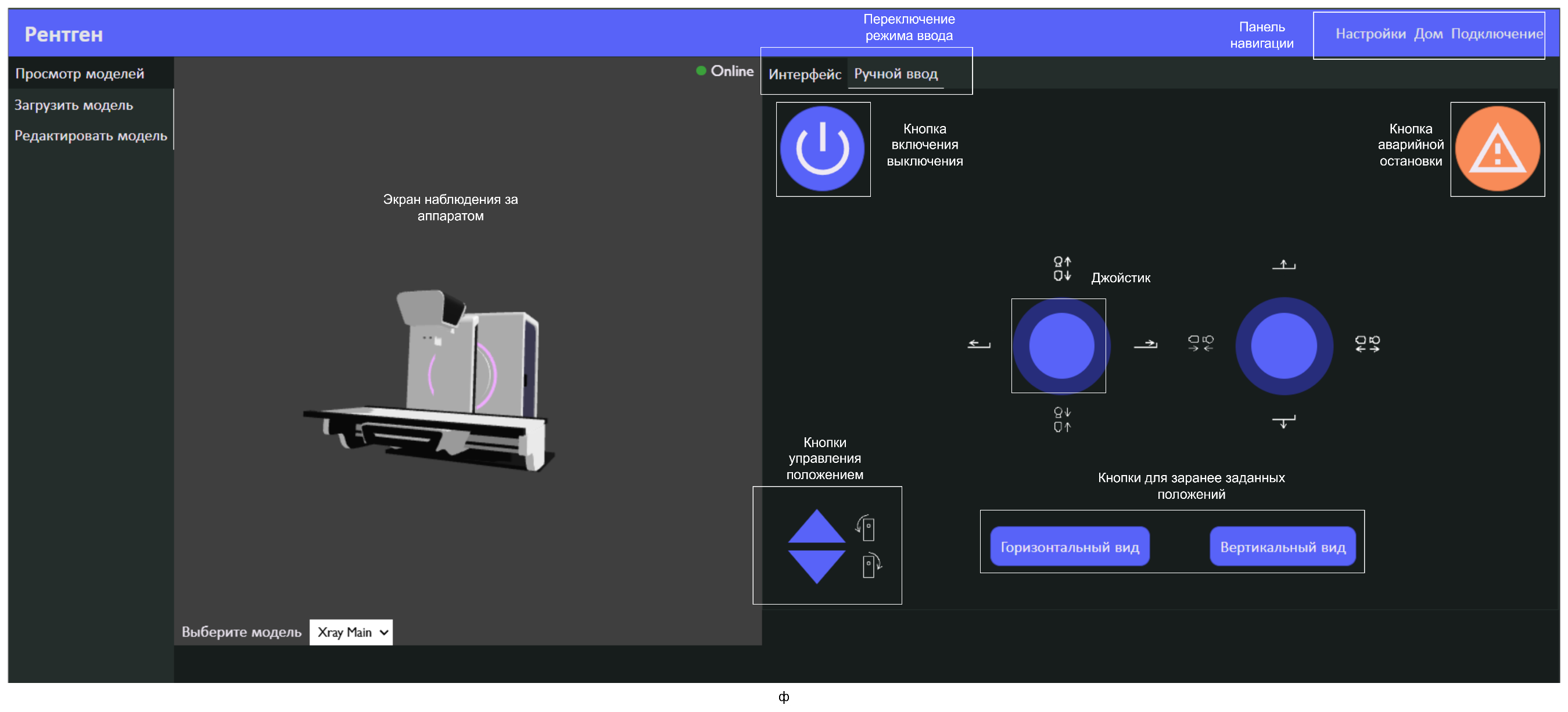


Рисунок 4.1 – интерфейс с пояснениями к управлению.

* 1. Выбор технологий для интерфейса

Для создания интерфейса выбраны технологии, которые обеспечивают высокую производительность, надежность и удобство разработки. Основой служит библиотека React, использующая компонентный подход и виртуальный DOM для быстрого обновления интерфейса. Это позволяет мгновенно отображать изменения, например, при переключении статуса системы. TypeScript добавляет строгую типизацию, снижая вероятность ошибок в коде, что критично для медицинских приложений, где надежность имеет приоритет.

Состояние приложения управляется через Redux Toolkit, который централизует данные о подключении, настройках и статусе, упрощая взаимодействие между компонентами. Для стилизации используется UnoCSS — утилитарный CSS-фреймворк, обеспечивающий компактные и адаптивные стили. В отличие от Tailwind CSS, UnoCSS генерирует меньший объем CSS, что ускоряет загрузку на мобильных устройствах. Инструмент сборки Vite поддерживает быструю разработку благодаря горячей перезагрузке и оптимизирует итоговую сборку для продакшена. React Router отвечает за навигацию между страницами, такими как настройки, подключение и рабочий режим. Для реализации джойстика, ключевого элемента управления, используется библиотека react-joystick-component, которая обеспечивает точное и плавное взаимодействие.

Эти технологии выбраны за их способность поддерживать сложные интерактивные интерфейсы, соответствующие требованиям медицинских систем. Они позволяют создать масштабируемое решение, готовое к добавлению новых функций, таких как многоязычность или поддержка дополнительных моделей аппаратов.

На рисунке 4.2 показана схема технлогического стека проекта.

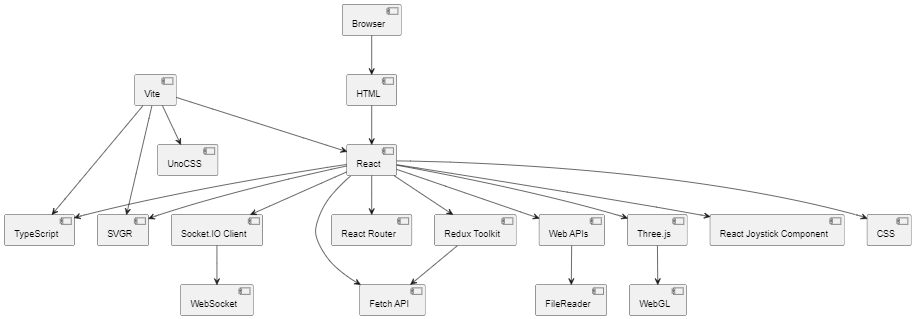


Рисунок 4.2 – Схема технологического стека проекта.

* 1. Архитектура пользовательского интерфейса

Интерфейс построен как одностраничное приложение (SPA), что обеспечивает плавную навигацию и минимизирует задержки при переключении между функциями. Основной компонент App.tsx управляет маршрутизацией через React Router, направляя пользователя к страницам настроек (Settings.tsx), подключения (Connect.tsx), рабочего режима (Working.tsx), а также добавления и редактирования моделей (AddModel.tsx, EditModel.tsx). Каждая страница решает конкретную задачу: настройка параметров, установление соединения или управление аппаратом.

Ключевые элементы управления, такие как джойстик и кнопки, реализованы в компонентах ModelControls.tsx и ModelControlsInputs.tsx. Они динамически формируются на основе JSON-настроек, что позволяет адаптировать интерфейс под разные модели аппаратов без изменения кода. Например, компонент ModelControls.tsx размещает джойстик (SControlJoystick.tsx), кнопки установки (SSetButton.tsx), стрелочные кнопки (SArrowButtons.tsx), кнопку питания (SPowerButton.tsx) и аварийную остановку (SEmergencyStopButton.tsx) в сетке, заданной в ControlGrid.css.

Дополнительные компоненты улучшают пользовательский опыт: навигационная панель (Navigation.tsx) упрощает доступ к основным функциям, индикатор статуса (SStatusIndicator.tsx) показывает текущее состояние системы, а всплывающие окна (SPopUp.tsx) информируют об ошибках или запрашивают подтверждения. Вкладки (STabViewer.tsx, STab.tsx) позволяют переключаться между режимами управления, например, между ручным вводом параметров и использованием джойстика. Хук useActions.ts облегчает взаимодействие с Redux, а useSocket.ts управляет WebSocket-соединением для онлайн-режима.

Такая архитектура делает интерфейс модульным и удобным для расширения, например, для добавления новых элементов управления или страниц.

На рисунке 4.3 представлена диаграмма компонентов интерфейса

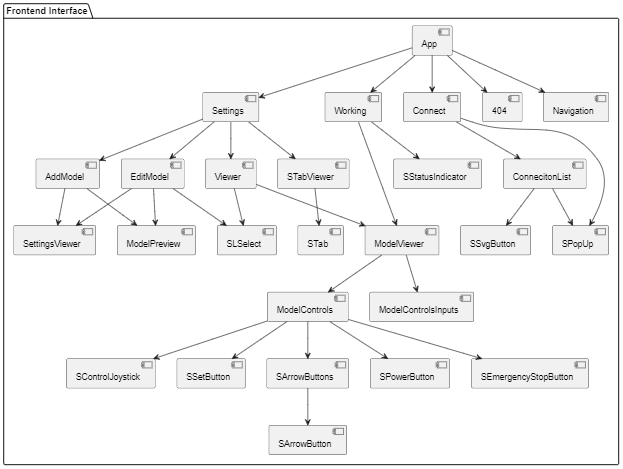


Рисунок 4.3 – Схема технологического стека проекта.

* 1. Проектирование интерфейса

Дизайн интерфейса разработан с учетом принципов эргономики и usability, чтобы медицинский персонал мог быстро освоить управление аппаратом. Визуальная структура вдохновлена физическими пультами рентгеновских аппаратов, где ключевые элементы, такие как джойстик и кнопка аварийной остановки, расположены для моментального доступа. Стандарты ISO 7000/IEC 60417 использованы для выбора форм и цветов: например, кнопка аварийной остановки (SEmergencyStopButton.tsx) выделена красным цветом и увеличенным размером, чтобы исключить случайное игнорирование.

Адаптивность интерфейса реализована через UnoCSS, который автоматически подстраивает размеры элементов под разрешения экрана от 640x480 до 1920x1080. На странице Working.tsx элементы управления размещены в сетке (CSS Grid, файл ControlGrid.css), что позволяет пользователю легко ориентироваться даже на небольших экранах. Например, джойстик занимает центральную зону, а кнопки установки и стрелки располагаются вокруг, создавая интуитивно понятную компоновку.

Взаимодействие с интерфейсом начинается с навигационной панели, где пользователь выбирает нужную функцию: настройку, подключение или рабочий режим. На странице Working.tsx оператор видит вкладки (STabViewer.tsx), позволяющие переключаться между управлением джойстиком и ручным вводом параметров через поля (ModelControlsInputs.tsx). Визуальная иерархия подчеркивается цветами: зеленый индикатор (SStatusIndicator.tsx) сигнализирует об успешном подключении, а красный — об аварийной остановке.

Обратная связь реализована через всплывающие окна (SPopUp.tsx), которые появляются при ошибках или для подтверждения действий. Например, при попытке удалить модель в EditModel.tsx пользователь должен подтвердить действие, что предотвращает случайные ошибки. Читаемость текста оптимизирована за счет крупных шрифтов и контрастных цветов, что особенно важно в условиях медицинских учреждений с разным освещением.

На рисунках 4.4.1 – 4.4.2 показан интерфейс программы

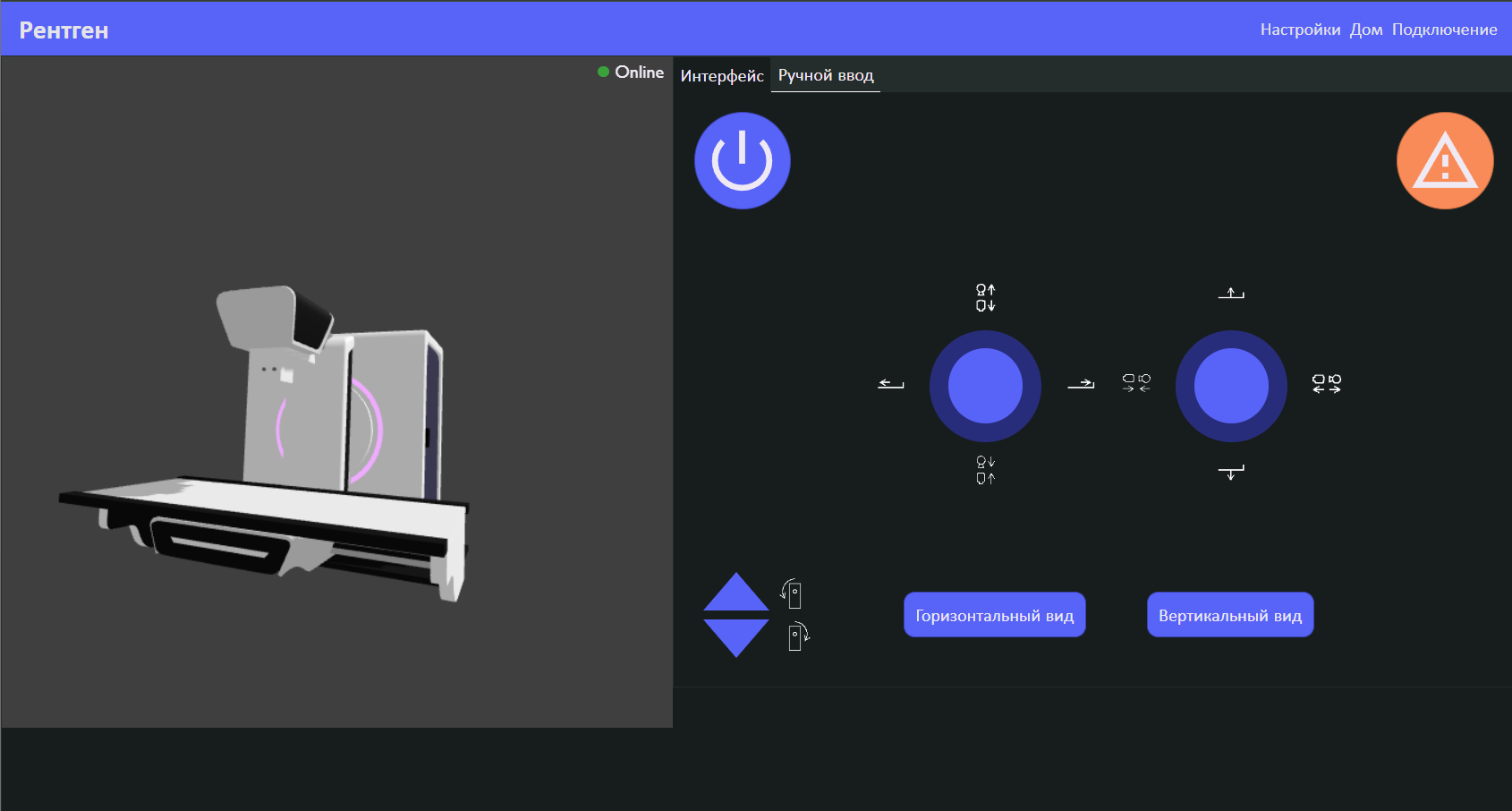


Рисунок 4.4.1 – Интерфейс программы для горизонтальных устройств

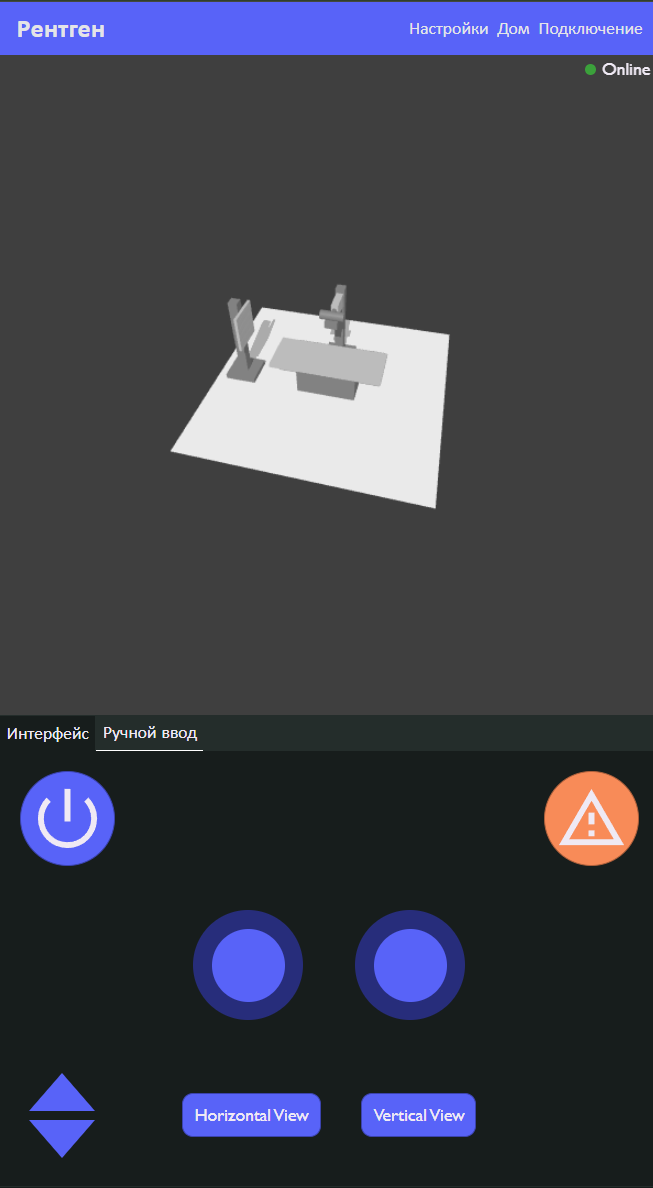


Рисунок 4.4.2 – Интерфейс программы для вертикальных устройств

* 1. Взаимодействие с элементами управления

Взаимодействие с интерфейсом сосредоточено на элементах управления, которые имитируют реальный пульт рентгеновского аппарата, обеспечивая оператору привычный опыт. На странице Working.tsx пользователь управляет аппаратом через джойстик (SControlJoystick.tsx), который позволяет перемещать стол или другие части модели по осям X и Y. Перемещение джойстика влево или вправо изменяет позицию с шагом 0.1, а движение вверх или вниз корректирует высоту. Плавность управления достигается за счет обработки событий move с интервалом 100 мс, что создает ощущение непрерывного движения. Если оператор отпускает джойстик, движение прекращается, а текущая позиция сохраняется в Redux.

Рядом с джойстиком расположены стрелочные кнопки (SArrowButtons.tsx), которые позволяют точно корректировать параметры, например, угол поворота или положение стола. Нажатие на кнопку «вверх» увеличивает значение на 0.1, а «вниз» — уменьшает. Это удобно для ситуаций, требующих высокой точности, например, при позиционировании пациента. Кнопки установки (SSetButton.tsx) предлагают быстрый доступ к предопределенным позициям, заданным в JSON-настройках. Нажатие на такую кнопку автоматически перемещает модель в нужное положение, что экономит время при выполнении стандартных процедур.

Кнопка питания (SPowerButton.tsx) переключает аппарат между активным и неактивным состояниями. При активации она подсвечивается зеленым, а элементы управления становятся доступными. В неактивном состоянии управление блокируется, предотвращая случайные действия. Кнопка аварийной остановки (SEmergencyStopButton.tsx) выделена красным и вызывает немедленную блокировку всех операций.

Все действия сопровождаются визуальной обратной связью. Например, при перемещении джойстика индикатор статуса (SStatusIndicator.tsx) показывает «Working», а при аварийной остановке — «Emergency Stopped». Это делает управление прозрачным и безопасным.

* 1. Управление подключением и режимами работы

Страница Connect.tsx позволяет оператору настроить подключение к эмулятору или реальному аппарату. Пользователь открывает форму, нажав кнопку «Подключиться», и вводит IP-адрес (например, «localhost») и порт (по умолчанию 12537). После нажатия кнопки «Подключиться» система отправляет запрос ping через REST API (connect.api.ts). Если соединение успешно, данные сохраняются в базе SQLite, и пользователь перенаправляется на страницу Working.tsx. В случае ошибки, например, если эмулятор недоступен, всплывающее окно (SPopUp.tsx) показывает сообщение, такое как «Connection refused», и предлагает исправить IP или порт.

Сохраненные соединения отображаются в таблице (ConnecitonList.tsx) на той же странице. Оператор может выбрать ранее сохраненное соединение, отредактировать его (изменив IP/порт) или удалить, подтвердив действие через SPopUp.tsx. Это упрощает повторное подключение, например, при работе с несколькими аппаратами в клинике.

Переключение между онлайн- и оффлайн-режимами происходит на страницах Settings.tsx и Working.tsx. В оффлайн-режиме пользователь выбирает модель из выпадающего списка (SLSelect.tsx) на странице Settings.tsx. После выбора модель загружается локально, и оператор может управлять ею через джойстик или кнопки, как в реальном режиме, но без связи с эмулятором. В онлайн-режиме, доступном на Working.tsx, интерфейс синхронизируется с эмулятором через WebSocket (хук useSocket.ts). Статус соединения отображается в SStatusIndicator.tsx: зеленый для «Online», серый для «Offline».

Такой подход делает интерфейс гибким: оператор может тренироваться в оффлайн-режиме или работать с реальным аппаратом, не меняя привычный способ управления. Взаимодействие интуитивно: выбор режима занимает пару кликов, а подключение подтверждается визуально.

На рисунках 4.6.1 и 4.6.2 предсталены таблица подключений к аппаратам, к которым подключались ранее и подключение к новому аппарату соответсвенно.

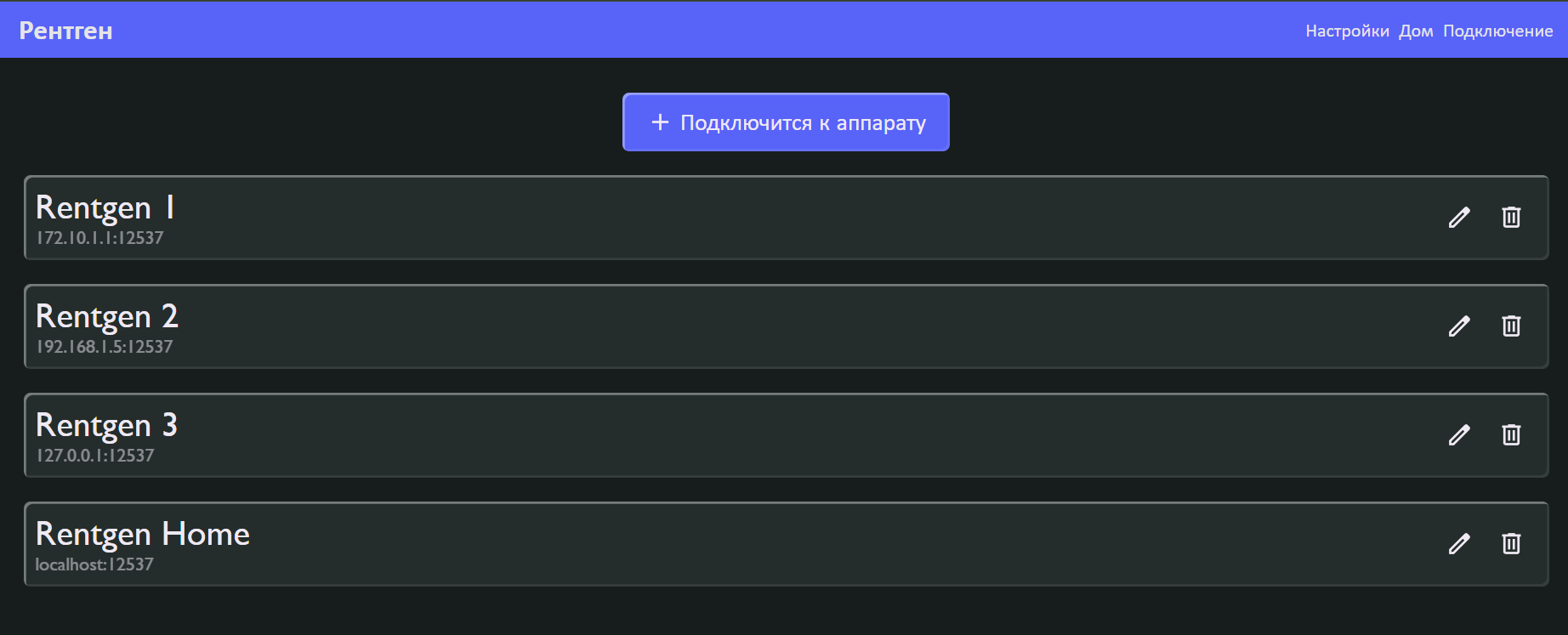


Рисунок 4.6.1 – Таблица подключений

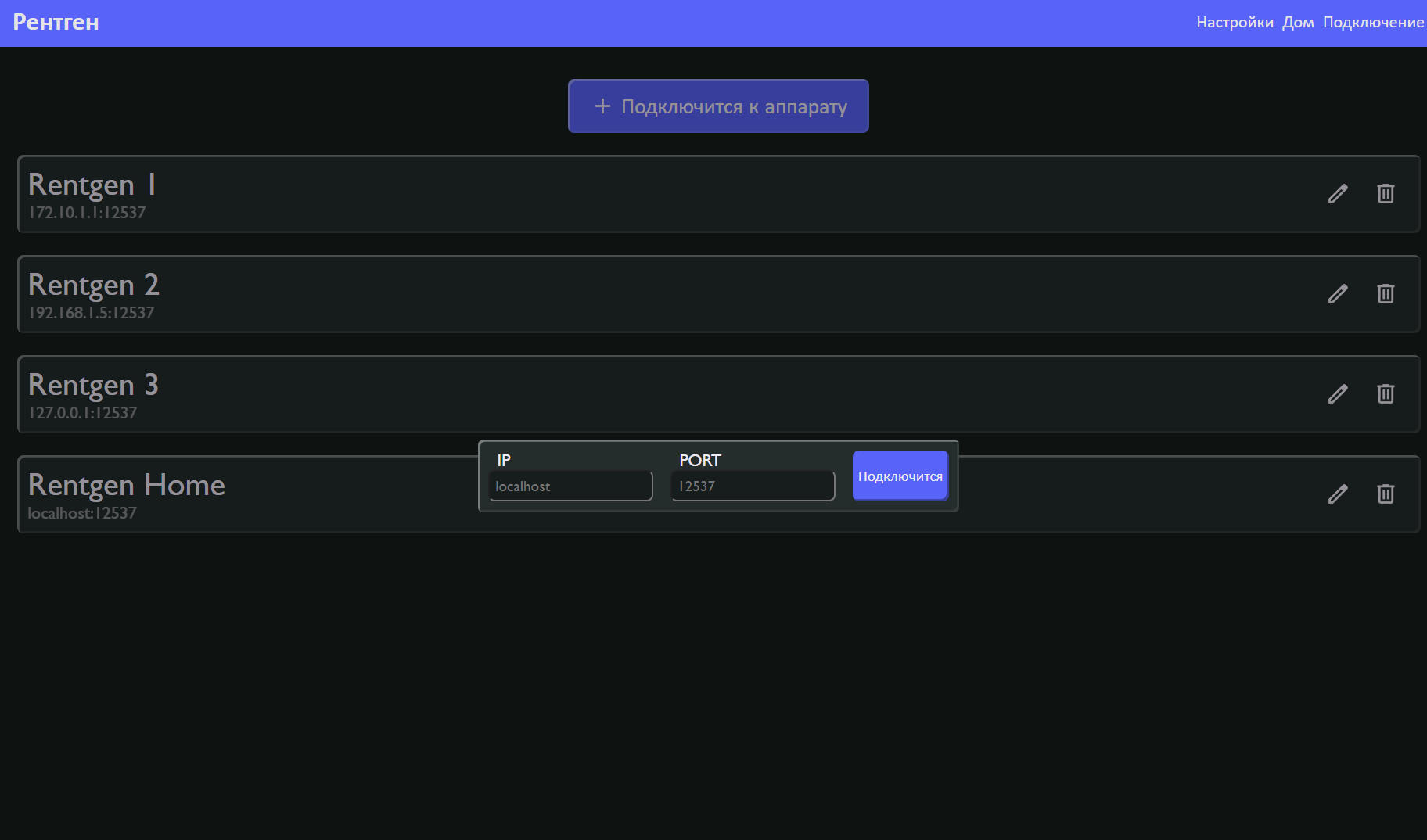


Рисунок 4.6.2 – Подключение к новому аппарату

* 1. Обработка ошибок и визуализация статуса

Интерфейс спроектирован так, чтобы оператор всегда понимал состояние системы и мог быстро реагировать на проблемы. Ошибки отображаются через всплывающие окна (SPopUp.tsx), которые появляются в критических ситуациях. Например, на странице Connect.tsx, если пользователь вводит неверный IP, окно показывает сообщение «Error: Invalid IP address» и предлагает проверить данные. Аналогично, при попытке загрузить неподдерживаемый файл модели в AddModel.tsx, оператор видит сообщение «Unsupported file format» с указанием допустимых форматов (например, glTF). Эти окна ненавязчивы: они закрываются по клику и не блокируют интерфейс, позволяя сразу исправить ошибку.

Для подтверждения критических действий, таких как удаление модели в EditModel.tsx или активация аварийной остановки, всплывающее окно запрашивает согласие. Например, перед удалением модели пользователь видит вопрос «Удалить модель ‘Table1’?» с кнопками «Да» и «Нет». Это снижает риск случайных ошибок, что особенно важно в медицинском контексте.

Статус системы визуализируется через компонент SStatusIndicator.tsx, расположенный в верхней части страницы Working.tsx. Он отображает четыре состояния: «Online» (зеленый, активное соединение с эмулятором), «Working» (желтый, выполняется команда, например, перемещение джойстиком), «Emergency Stopped» (красный, аппарат заблокирован) и «Offline» (серый, говорит о том что аппарат выключен или недоступен). Цветовая кодировка интуитивна и соответствует стандартам медицинского оборудования, где красный сигнализирует об опасности. При переключении статуса индикатор обновляется мгновенно, что помогает оператору отслеживать изменения в реальном времени.

Асинхронные операции, такие как загрузка модели или установление соединения, сопровождаются текстовыми уведомлениями. Например, на странице AddModel.tsx во время загрузки файла отображается сообщение «Загрузка модели...», а при успехе — «Модель загружена».

На рисунках 4.7.1 и 4.7.2 предсталены визуализированные статусы ошибок возникающие при работе с программой.

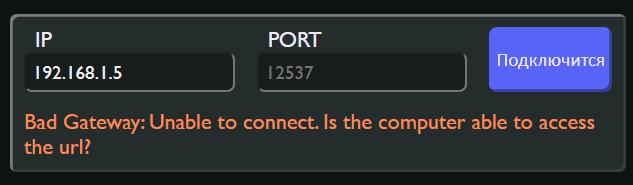


Рисунок 4.7.1 – Ошибка подключения к аппарату

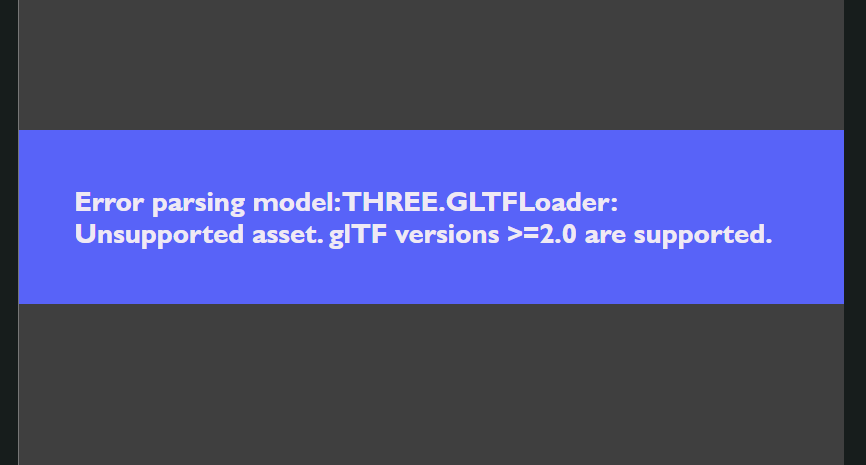


Рисунок 4.7.2 – Ошибка загрузки модели

* 1. Выводы

Разработанный веб-интерфейс успешно решает задачу управления телеуправляемым рентгеновским аппаратом, обеспечивая медицинскому персоналу удобный и безопасный инструмент. Элементы управления, такие как джойстик и кнопки, повторяют логику реальных пультов, что сокращает время обучения. Адаптивный дизайн, реализованный через UnoCSS, позволяет использовать интерфейс на разных устройствах, от настольных компьютеров до планшетов, что актуально для мобильных медицинских станций.

Визуальная обратная связь, включая индикаторы статуса и всплывающие окна, делает взаимодействие прозрачным: оператор всегда знает, выполняется ли команда, активен ли аппарат или возникла ошибка. Поддержка онлайн- и оффлайн-режимов добавляет гибкости: пользователь может тренироваться локально или управлять реальным оборудованием без изменения привычного интерфейса. Использование React, TypeScript и Redux Toolkit гарантирует надежность и масштабируемость, позволяя в будущем расширить функциональность, например, добавить поддержку новых моделей или языков.

1. Создание сервера

Серверная часть является ключевым компонентом системы, обеспечивающим взаимодействие между веб-интерфейсом, базой данных и эмулятором телеуправляемого рентгеновского аппарата. Бэкенд отвечает за обработку запросов на загрузку и управление 3D-моделями, настройку подключений к эмулятору и передачу данных в реальном времени. Для реализации использованы Bun.js как высокопроизводительная среда выполнения JavaScript, фреймворк Express для создания REST API и библиотека Socket.IO для двунаправленного взаимодействия. В данной главе описаны архитектура сервера, разработка API, интеграция с базой данных, реализация реального времени и результаты тестирования.

* 1. Архитектура серверной части

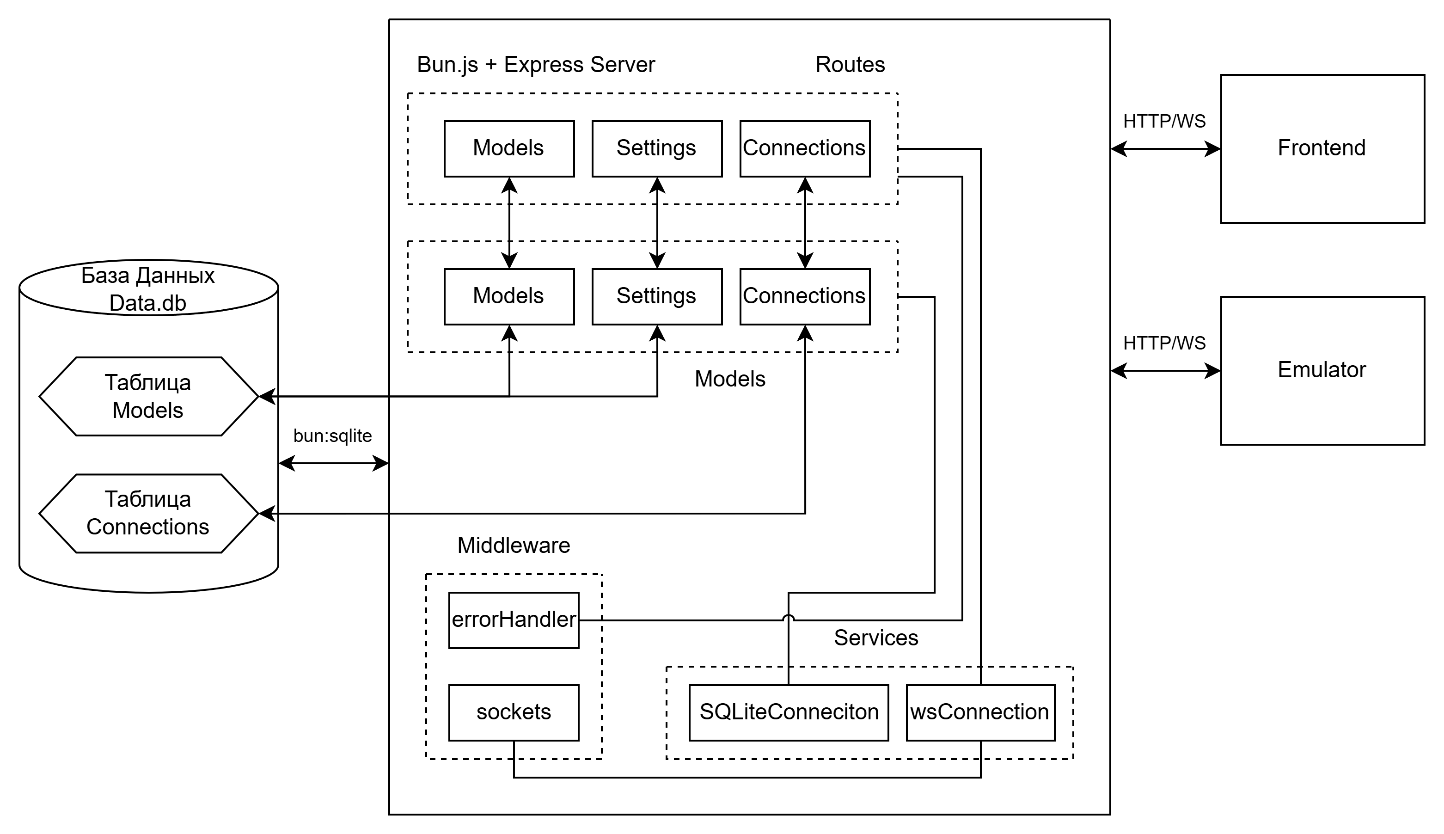
Бэкенд реализован на основе Bun.js, обеспечивающего высокую производительность за счёт нативной поддержки SQLite. Основным фреймворком выбран Express из-за простоты его интеграции с JavaScript-экосистемой и широкой поддержки паттерна middleware. Для взаимодействия в реальном времени между фронтендом и эмулятором используется Socket.IO, позволяющее передавать команды и состояния с минимальной задержкой. Архитектура сервера организована по модульному принципу, разделяя функциональность на маршруты (routes), модели (models), сервисы (services) и middleware (middleware). Основной входной точкой является файл src/index.ts, который инициализирует сервер и настраивает middleware, включая CORS, body-parser и cookie-parser.

Сервер выполняет следующие функции:

* Обработка HTTP-запросов для управления моделями и подключениями через REST API;
* Хранение данных моделей и подключений в базе данных SQLite (data.db);
* Пересылка команд и состояний между фронтендом и эмулятором через Socket.IO;
* Проверка доступности эмулятора через HTTP-запросы;

Модульная структура упрощает поддержку и масштабирование системы, позволяя независимо обновлять отдельные компоненты, такие как маршруты или обработчики Socket.IO.

На рисунке 5.1 представлена архитектура сервеной части. Отдельно выделены блоки маршрутов (routes), моделей (models), сервисов (services) и промежуточное программное обеспечение (middleware). Показано их взаимодействие междну собой.

****Рисунок 5.1 – Архитектура серверной части

Маршруты – предостовляют функциональность путей для HTTP запросов

Модели – представляют набор функций для работы с базой данны

Сервисы – обеспечивают соединение с базой данных, и веб-сокет соединение между сервером, клиентами и эмуляторами.

Промежуточное программное обеспечение – предоставляет функции для обработки данных или ошибок, внутри маршрутов, моделей и сервисов.

* 1. Разработка REST API

Для управления данными 3D-моделей и подключениями к эмулятору разработан REST API, реализованный с использованием Express. API включает три основных маршрута, определённых в файлах src/routes/models.ts, src/routes/connections.ts и src/routes/settings.ts. Основные эндпоинты обеспечивают CRUD-операции (создание, чтение, обновление, удаление) для моделей и подключений, а также проверку доступности эмулятора.

Основные маршруты:

**/api/models**:

* GET /: получение списка имён моделей с их идентификаторами;
* GET /:modelName: получение бинарных данных модели в формате application/octet-stream;
* POST /: загрузка новой модели с использованием multer для обработки бинарных файлов;
* PUT /:modelName: обновление модели (данных или настроек);
* DELETE /:modelName: удаление модели.

**/api/connect**:

* GET /?page=X&perPage=Y: постраничное получение списка подключений;
* POST /: добавление нового подключения (имя, IP, порт);
* PATCH /:id: обновление подключения по идентификатору;
* DELETE /:id: удаление подключения;
* POST /ping: проверка доступности эмулятора по IP и порту.

**/api/json**

* GET /:modelName: получение JSON-настроек модели для управления взаимодействием в веб-интерфейсе.

Для обработки файлов в POST /api/models используется библиотека multer, которая позволяет загружать бинарные данные модели с ограничением размера 50 МБ. Ошибки обрабатываются централизованным промежуточным программным обеспечением errorHandler возвращающим стандартизированный JSON-ответ с кодом 400 и описанием ошибки. Безопасность запросов обеспечена использованием CORS с белым листом (whitelist) для доменов используемых клиентской частью и эмуляторами, а также проверкой входных данных в маршрутах.

Пример запроса для загрузки модели представлен в листинге 5.2

Листинг 5.1 – Пример запроса для загрузки

|  |
| --- |
| POST /api/models  Content-Type: multipart/form-data  {  "name": "xray\_model",  "settings": "{\"controls\": {\"joystick\": {\"x\": 0, \"y\": 0}}}",  "data": [binary glb file]  } |

Древовидная диаграмма, показывающая иерархию маршрутов API представлена на рисунке 5.2

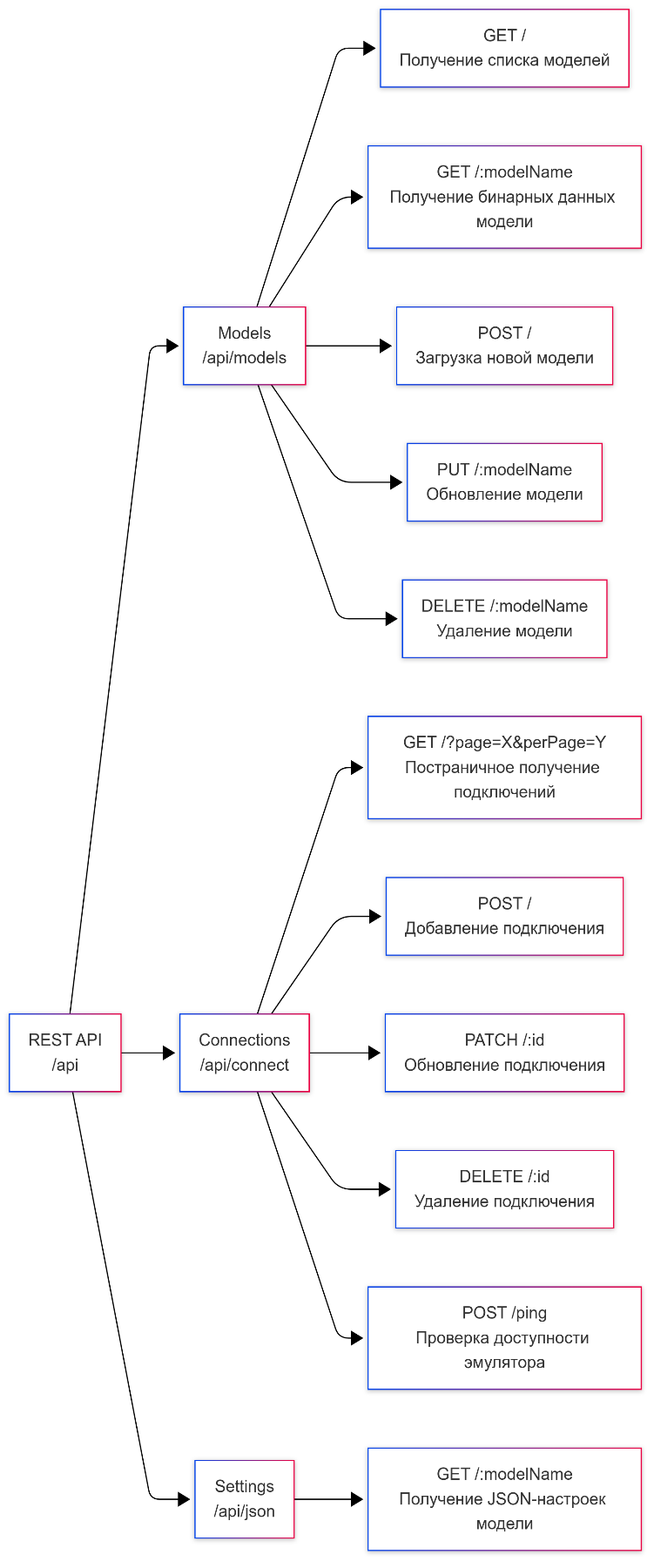


Рисунок 5.2 – Древовидная диаграмма, показывающая иерархию маршрутов API

* 1. Реализация базы данных

Для хранения данных 3D-моделей и подключений используется реляционная база данных SQLite, размещённая в файле data.db. Инициализация базы выполняется в файле db/init-db.ts, который создаёт две таблицы:

**Таблица models**:

* id (INTEGER, автоинкремент, первичный ключ): уникальный идентификатор модели;
* name (TEXT, уникальное): имя модели;
* settings (TEXT): JSON-строка с настройками взаимодействия (например, диапазоны для джойстика);
* data (BLOB): бинарные данные модели в формате glTF;

**Таблица connections**:

* id (INTEGER, автоинкремент, первичный ключ): идентификатор подключения;
* name (TEXT): название подключения;
* ip (TEXT): IP-адрес эмулятора;
* port (TEXT): порт эмулятора;
* Уникальный индекс на поля ip и port для предотвращения дублирования.

Доступ к базе осуществляется через модуль src/services/SQLiteConnection.ts, который использует библиотеку bun:sqlite для выполнения параметризованных запросов. Это обеспечивает защиту от SQL-инъекций, как, например, в функциях addConnection и updateModelByName. Модули в src/models (например, connections.model.ts, models.model.ts) предоставляют функции для CRUD-операций, такие как getConnectionsPaginated для постраничного вывода подключений или getModelByName для получения бинарных данных модели.

Пример параметризованного запроса для добавления подключения представлен в листинге 5.3

Листинг 5.3 – Пример запроса для загрузки

|  |
| --- |
| const query = db.query(`INSERT INTO connections (name, ip, port) VALUES (?, ?, ?)`)*;*  query.run(name, ip, port)*;* |

База данных оптимизирована для небольшого объёма данных, характерного для демонстрационного приложения, обеспечивая время выполнения запросов менее 10 мс для таблиц с до 1000 записей.

ER-диаграмма, показывающая таблицы models и connections представлена на рисунке 5.3

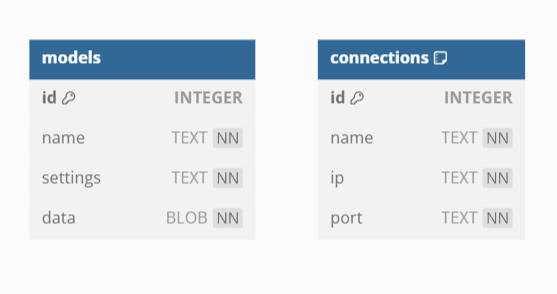


Рисунок 5.3 – ER-диаграмма, показывающая таблицы models и connections

* 1. Интеграция с Socket.IO

Для обеспечения взаимодействия в реальном времени между фронтендом и эмулятором сервер использует библиотеку Socket.IO, реализованную в файле src/middleware/sockets.ts. Socket.IO настроен с поддержкой CORS для тех же доменов (localhost:8045, 8046) и обрабатывает подключения через функцию socketManager. Основные события, обрабатываемые сервером:

* **connectHardware**: устанавливает соединение с эмулятором по указанным IP и порту, используя WsConnect из src/services/wsConnection.ts. Применяется async-mutex для предотвращения конфликтов при множественных подключениях;
* **clientCommand**: пересылает команды (например, { command: "set", path: "table\_x", value: 10.5 }) от фронтенда к эмулятору;
* **clientState**: передаёт состояния оборудования (например, isControlsEnabled) от фронтенда к эмулятору;
* **getCurrentParameters**, **getModel**: запрашивают текущие параметры или имя модели у эмулятора.

**События от эмулятора**: command, state, setParameters пересылаются фронтенду как clientCommand, clientState, clientSetParameters.

Механизм async-mutex гарантирует, что только одно соединение с эмулятором активно в любой момент времени, предотвращая дублирование команд. Задержка передачи данных через Socket.IO составляет менее 100 мс, что обеспечивает отзывчивость интерфейса при управлении 3D-моделью.

Socket.IO также поддерживает автоматическое переподключение, что повышает надёжность системы при временных сбоях сети.

На рисунке 5.4 представлена диаграмма последовательности (Sequence Diagram), показывающая взаимодействие фронтенда, бэкенда и эмулятора с помощью веб-сокетов. Этапы: 1) Фронтенд отправляет clientCommand на бэкенд; 2) Бэкенд пересылает command эмулятору; 3) Эмулятор возвращает setParameters; 4) Бэкенд отправляет clientSetParameters фронтенду.

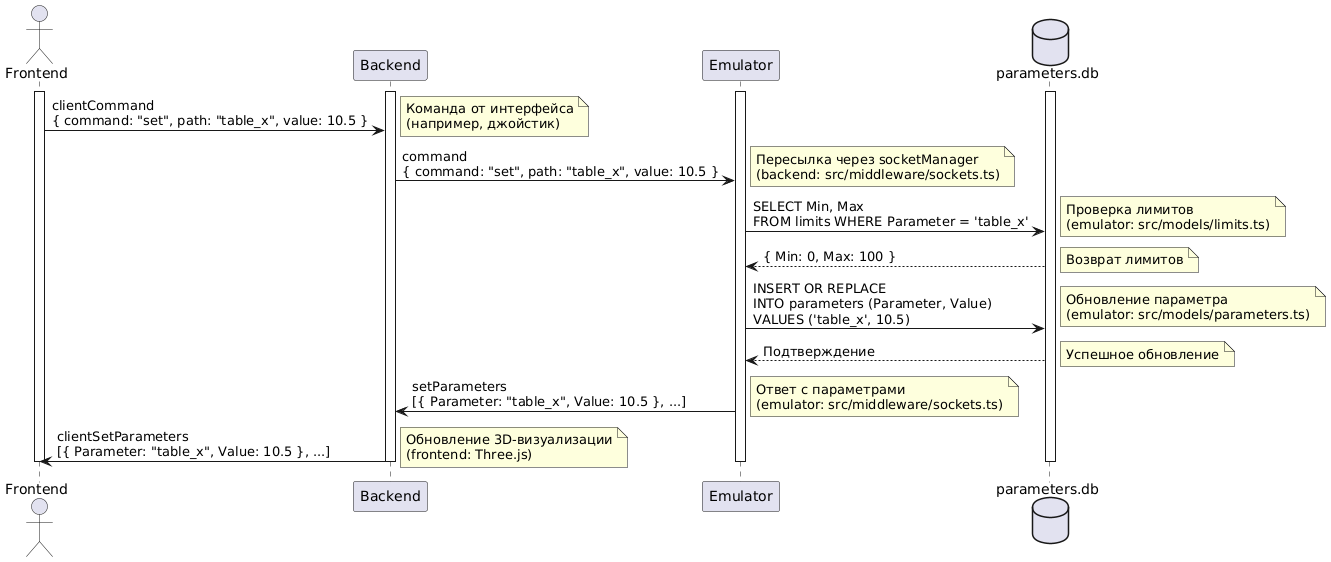


Рисунок 5.4 - Диаграмма последовательности

* 1. Выводы

Разработанный сервер обеспечивает надёжное управление 3D-моделями и подключениями, а также время взаимодействия в реальном времени с эмулятором рентгеновского аппарата. REST API, реализованное на Express, поддерживает CRUD-операции с высокой производительностью, а Socket.IO гарантирует отзывчивость системы с задержкой менее 100 мс. SQLite база данных эффективно хранит данные, а параметризованные запросы обеспечивают безопасность.

1. Создание сервера эмулятора рентгеновского аппарата.

Эмулятор рентгеновского аппарата является ключевым компонентом системы, обеспечивающим реалистичную имитацию работы телеуправляемого медицинского оборудования. Он обрабатывает команды и состояния, поступающие от серверной части (бэкенда), обновляет параметры в базе данных с учётом физических ограничений и возвращает результаты для синхронизации с 3D визуализацией в веб-интерфейсе. Эмулятор реализован на Bun.js, используя Express для минимального HTTP API и Socket.IO для взаимодействия в реальном времени. Для хранения текущих параметров и их лимитов применяется база данных SQLite (parameters.db). В данной главе описаны архитектура эмулятора, настройка базы данных, реализация обработки команд, интеграция с системой и результаты тестирования.

* 1. Архитектура эмулятора

Эмулятор реализован как отдельный сервер, работающий на Bun.js, который обеспечивает высокую производительность нативной поддержке SQLite. Для обработки HTTP-запросов используется фреймворк Express, а для реального времени — библиотека Socket.IO. Архитектура эмулятора организована модульно, с разделением функциональности на следующие компоненты:

* Маршруты (src/index.ts): минимальное HTTP API для проверки доступности (GET /ping).
* Обработчики Socket.IO (src/middleware/sockets.ts): управление командами, состояниями и параметрами.
* Модели базы данных (src/models/limits.ts, src/models/parameters.ts): функции для работы с таблицами limits и parameters.
* Сервисы (src/services/CommandProcess.ts, src/services/SQLiteConnection.ts): логика обработки команд и подключение к базе данных.

Основной файл src/index.ts инициализирует сервер на порту 12537 (или из переменной окружения PORT), настраивает middleware (CORS, body-parser, cookie-parser) и запускает Socket.IO с обработкой соединений через socketManager. Эмулятор выступает как имитатор оборудования, принимая команды (например, установка позиции стола) от бэкенда, проверяя их на соответствие лимитам и возвращая обновлённые параметры для отображения в веб-интерфейсе.

Эмулятор поддерживает следующие функции:

* Обработка команд set (установка значения) и add (инкремент параметра).
* Управление состояниями оборудования (isEnabled, isControlsEnabled, isEmergencyStoped).
* Хранение параметров и лимитов в базе данных SQLite.
* Передача данных в реальном времени через Socket.IO.

Модульная структура упрощает добавление новых функций, таких как симуляция ошибок оборудования или поддержка сложных сценариев движения.

На рисунке 6.1 представлена архитектура эмулятора. Отдельно выделены блоки маршрутов моделей (models), сервисов (services) и промежуточное программное обеспечение (middleware), а также блок Command Processor, ответсвенный за исполнение и обработку команд. Показано их взаимодействие междну собой.

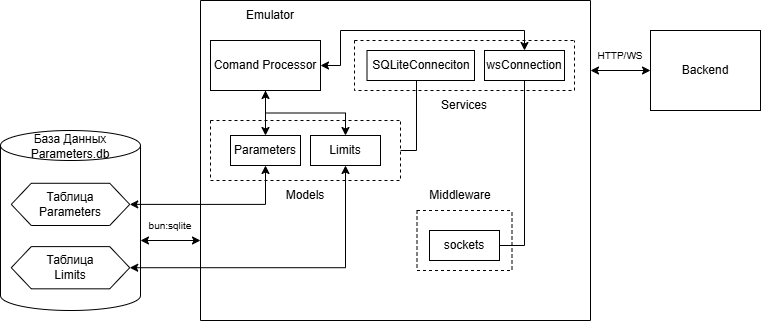


Рисунок 6.1 – Архитектура эмулятора

* 1. Настройка базы данных

Для хранения текущих параметров рентгеновского аппарата и их физических ограничений используется база данных SQLite, размещённая в файле parameters.db. Инициализация базы выполняется скриптами db/init-db.ts, db/init-limits-table.ts и db/init-parameters-table.ts, которые создают две таблицы:

**Таблица parameters**:

* Parameter (TEXT, первичный ключ): имя параметра (например, table\_x).
* Value (REAL): текущее значение параметра (например, 10.5).

**Таблица limits**:

* Parameter (TEXT, первичный ключ): имя параметра.
* Min (REAL): минимальное допустимое значение (например, 0).
* Max (REAL): максимальное допустимое значение (н parameters.

Доступ к базе осуществляется через модуль SQLiteConnection, который инициализирует соединение с parameters.db. Модули лимитов и параметров предоставляют функции для работы с данными, такие как getAllLimits и setParameter.

ER-диаграмма, показывающая таблицы parmeters и limits представлена на рисунке 6.2

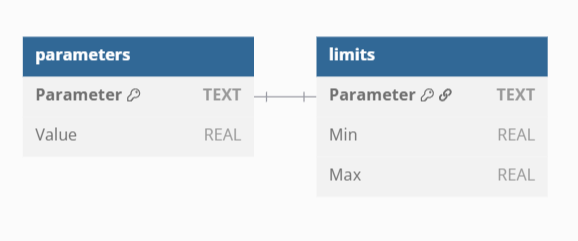


Рисунок 6.2 – ER-диаграмма, показывающая таблицы models и connections

* 1. Реализация обработки команд

Логика обработки команд реализована в модуле CommandProcess, который управляет командами set (установка значения параметра) и add (добавление к текущему значению). Команды поступают от бэкенда через Socket.IO (событие command) и содержат структуру { command, path, value }, определённую в списке типов комманд.

Пример команды представлен листингом 6.3.1:

Листинг 6.3.1 – Пример запроса для загрузки

|  |
| --- |
| { "command": "set", "path": "table\_x", "value": 10.5 } |

Процесс обработки включает следующие шаги:

**Проверка лимитов**: Функция setParameter запрашивает лимиты для параметра (getLimits) и проверяет, находится ли новое значение в допустимом диапазоне (Min, Max). Если значение выходит за пределы, команда отклоняется.

**Плавное изменение**: Для больших изменений (разница больше шага step=0.1) используется интервальная обработка через setInterval с таймаутом 100 мс, что имитирует плавное движение оборудования. Управление отключается (isControlsEnabled=0) на время выполнения.

**Обновление базы данных**: Новое значение записывается в таблицу parameters с помощью INSERT OR REPLACE.

**Отправка результата**: Обновлённые параметры возвращаются через событие setParameters.

Пример кода обработки команды set представлен листингом 6.3.2:

Листинг 6.3.2 – Пример кода обработки команды

|  |
| --- |
| if (command === "set" && Math.abs(currentValue - value) > step) {  setParameter("isControlsEnabled", 0);  const intervalID = setInterval(async () => {  const currentValue = await getParameter(path);  if (currentValue < value) {  const result = await setParameter(path, Math.min(currentValue + step, value));  socket.emit("command", result);  }  }, timeout);  } |

Состояния оборудования (isEnabled, isControlsEnabled, isEmergencyStoped) обрабатываются через событие state в функции setState, которая обновляет соответствующие параметры в базе данных и рассылает изменения всем клиентам через socket.broadcast.emit. Аварийная остановка (isEmergencyStoped=1) прерывает выполнение команд, имитируя реальную защиту оборудования.

На рисунке 6.3 представлена блок схема обработчика комманды set модуля CommandProcess

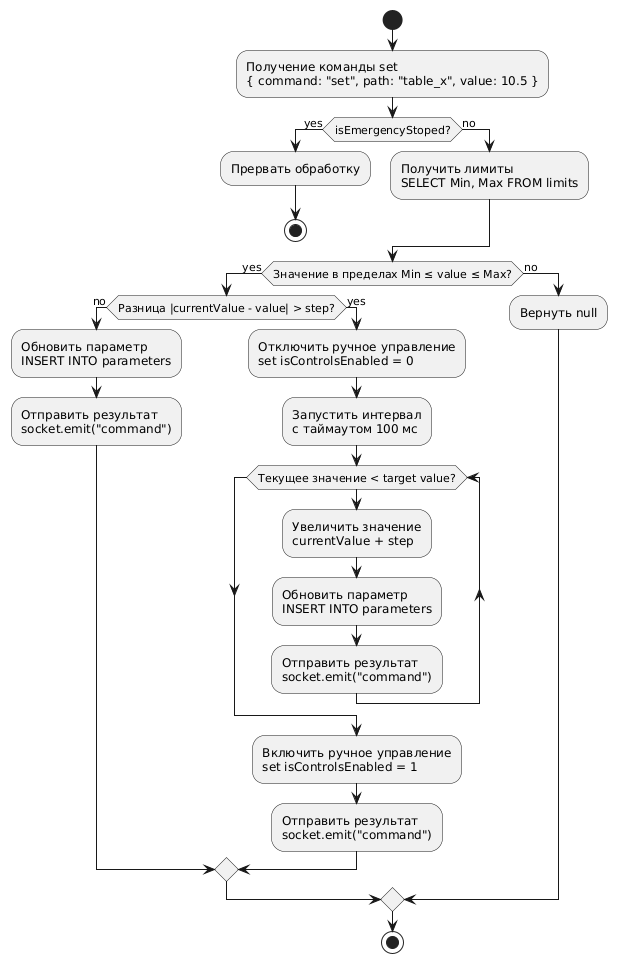


Рисунок 6.3 - блок схема обработчика комманды set

* 1. Интеграция с сервером и фронтендом

Эмулятор взаимодействует с системой через бэкенд, используя Socket.IO для реального времени и HTTP API для проверки доступности. Основные события Socket.IO, обрабатываемые эмулятором, включают:

* **getModel**: Возвращает имя модели (по умолчанию Xray или из process.env.model);
* **getCurrentParameters**: Отправляет все параметры из таблицы parameters (функция getAllParameters);
* **command**: Выполняет команду через runCommand, обновляя параметры;
* **state**: Обновляет состояния через setState и рассылает их клиентам;
* **disconnect**: Логирует отключение клиента.

Бэкенд пересылает команды от фронтенда (clientCommand) эмулятору как command и возвращает ответы (setParameters) как clientSetParameters. Для проверки доступности эмулятора бэкенд использует маршрут GET /ping. Интеграция с фронтендом обеспечивает синхронизацию параметров с 3D-моделью: например, изменение table\_x обновляет положение стола в Three.js через GLTFLoader.

Пример взаимодействия:

Фронтенд отправляет clientCommand: { command: "add", path: "table\_x", value: 0.1 }.

Бэкенд пересылает комнанду эмулятору.

Эмулятор обновляет table\_x в базе данных и возвращает setParameters: { table\_x: 0.1}.

Бэкенд отправляет clientSetParameters фронтенду, обновляя визуализацию.

На рисунке 6.4 представлена диаграмма последовательности (Sequence Diagram), показывающая взаимодействие фронтенда, бэкенда и эмулятора с помощью веб-сокетов.

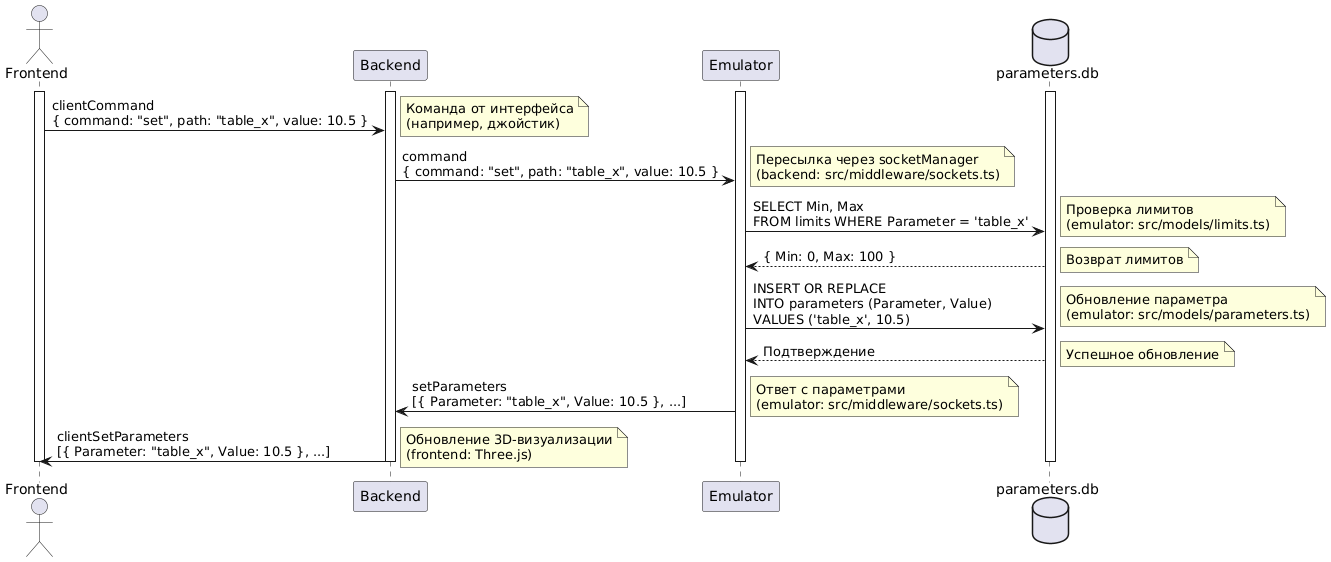


Рисунок 6.4 - диаграмма последовательности

* 1. Выводы

Эмулятор рентгеновского аппарата обеспечивает реалистичную имитацию работы телеуправляемого оборудования, обрабатывая команды и состояния с задержкой менее 100 мс. Использование Bun.js и Socket.IO гарантирует высокую производительность и надёжность, а SQLite (parameters.db) эффективно хранит параметры и лимиты. Реализация команд set и add с учётом физических ограничений и интервальной обработки моделирует плавное движение оборудования.

1. Проверка работоспособности приложения

Разработка информационной системы, особенно связанной с медицинским оборудованием, требует обязательного прохождения этапа тестирования. Целью данного этапа является не только обнаружение ошибок, но и верификация соответствия системы заявленным требованиям, а также оценка её устойчивости при различных сценариях использования. В данной главе изложены принципы и результаты тестирования разработанной системы визуализации и управления цифровым рентгеновским аппаратом.

* 1. Общая стратегия тестирования

Тестирование проводилось на всех ключевых уровнях системы: клиентском, серверном и в рамках двустороннего взаимодействия по WebSocket. Использовались как ручные, так и полуавтоматические методы тестирования. Отдельное внимание уделялось интеграции всех компонентов, обработке ошибок и пользовательскому опыту.

На этапе ручного тестирования в первую очередь проверялась корректность визуализации 3D-модели, доступность и работоспособность всех интерфейсных элементов, адекватная реакция на управляющие действия и сохранение модели в различных форматах. Для backend части проводилась проверка REST API с использованием Postman, а также анализ логов при запуске и эмуляции событий.

* 1. Тестирование клиентской части

Клиентская часть реализована с использованием библиотеки React, взаимодействующей с сервером через REST и WebSocket. Особое внимание уделялось проверке работоспособности следующих функций:

* загрузка 3D-модели и её визуализация в компоненте ModelViewer;
* управление положением и поворотом элементов модели с помощью пользовательских элементов: кнопок, джойстика, стрелок;
* отображение статуса системы: подключена ли модель, активно ли управление, активирована ли аварийная остановка;
* отображение ошибок при недопустимых действиях пользователя или сбоях во взаимодействии с сервером.

Для тестирования использовалась страница взаимодействия с рентгеновским аппаратом. На рисунке 7.2.1 представлен аппарат в положении по умолчанию. На рисунке 7.2.2 представлен аппарат после взаимодейсвия (перевод в вертикльный вид).

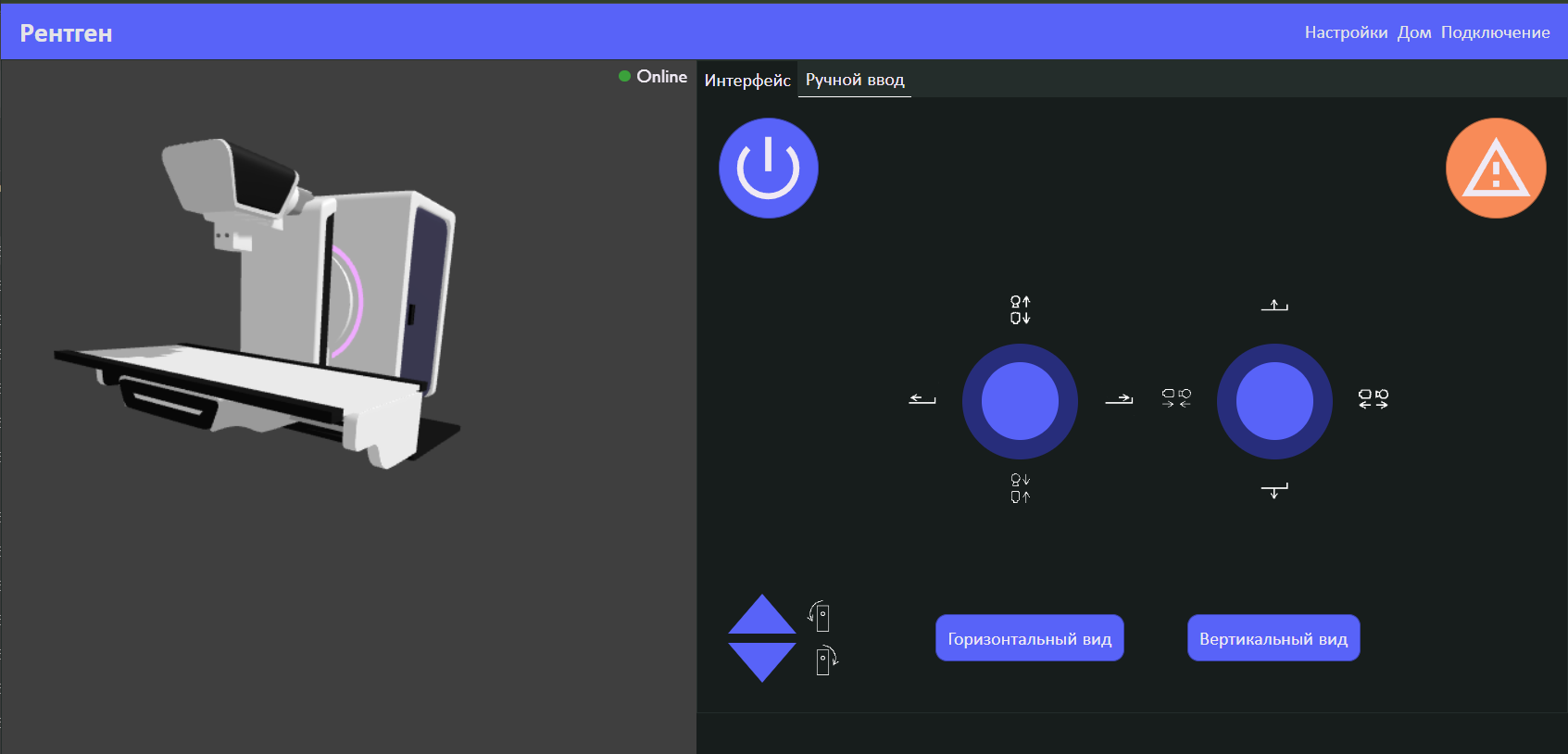


Рисунок 7.2.1 – Рентгеновский аппарат в положении по умолчанию

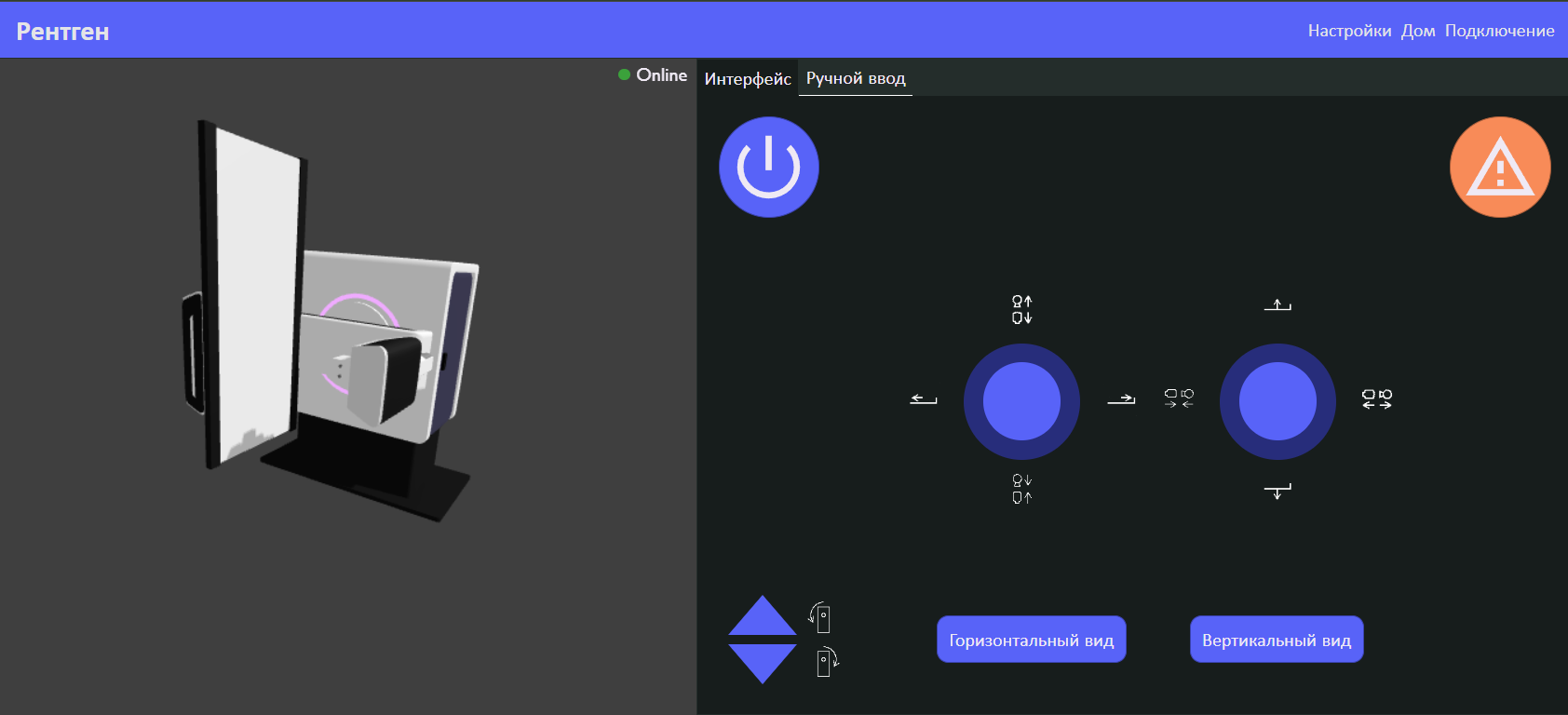


Рисунок 7.2.2 – Рентгеновский аппарат после взаимодействия

* 1. Тестирование серверной части и REST API

Серверное приложение было протестировано на предмет обработки запросов от клиента и взаимодействия с базой данных. Были проверены все маршруты API:

* /api/models — добавление, удаление, обновление моделей;
* /api/connect — управление подключениями к внешнему оборудованию;
* /api/json/:modelName — получение настроек модели;
* /api/connect/ping — проверка доступности внешнего устройства.

Для каждого маршрута формировались как корректные, так и заведомо некорректные запросы с целью оценить поведение системы в условиях нестандартных ситуаций.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Эндпоинт | Метод | Параметры | Ожидаемый результат | Фактический результат |
| /api/models | GET | - | Статус 200, JSON с массивом { id, name } всех моделей | Возвращает список моделей, например, [{ id: 1, name: "xray\_model" }] |
| /api/models/:modelName | GET | modelName: string | Статус 200, бинарные данные модели в формате application/octet-stream | Возвращает glTF-файл для корректного modelName, ошибка 400 при отсутствии модели |
| /api/models | POST | name: string, settings: string, data: file | Статус 200, JSON { status: "success", message: "Model added" } | Модель успешно добавляется, ошибка 400 при отсутствии файла или дублировании имени |
| /api/models/:modelName | PUT | name: string, settings: string, data: file | Статус 200, JSON { status: "success", message: "Model updated" } | Модель обновляется, ошибка 400 при отсутствии данных или некорректном имени |
| /api/models/:modelName | DELETE | modelName: string | Статус 200, JSON { status: "success", message: "Model deleted" } | Модель удаляется, ошибка 400 при отсутствии модели |
| /api/connect | GET | page: number, perPage: number | Статус 200, JSON с массивом подключений { id, name, ip, port } | Возвращает постраничный список подключений, ошибка 400 при page < 1 или perPage < 1 |
| /api/connect | POST | name: string, ip: string, port: string | Статус 200, JSON { status: "success", message: "Connection added" } | Подключение добавляется, ошибка 400 при дублировании ip и port |
| /api/connect/:id | PATCH | id: number, name: string, ip: string, port: string | Статус 200, JSON { status: "success", message: "Connection updated" } | Подключение обновляется, ошибка 400 при неверном id |
| /api/connect/:id | DELETE | id: number | Статус 200, JSON { status: "success", message: "Connection deleted" } | Подключение удаляется, ошибка 400 при неверном id |
| /api/connect/ping | POST | ip: string, port: string | Статус 200, JSON { status: "success", message: "pong" } | Возвращает «pong» при доступном эмуляторе, статус 502 при ошибке подключения |
| /api/json/:modelName | GET | modelName: string | Статус 200, JSON с настройками модели | Возвращает настройки, ошибка 400 при отсутствии модели |

Дополнительно была проверена валидация данных на стороне сервера, например, невозможность создать два подключения с одинаковым IP и портом или загрузить модель с уже существующим именем.

* 1. Тестирование WebSocket-взаимодействия

Для обмена командами с внешним оборудованием (или его эмулятором) используется WebSocket-протокол. Подключение к серверу осуществляется по IP-адресу и порту, указанным пользователем, а все последующие команды передаются в формате событий.

Тестировались следующие события:

* connectHardware — подключение к эмулятору;
* clientCommand и clientState — управление моделью;
* getModel, getCurrentParameters — получение данных от эмулятора.

Со стороны клиента события обрабатываются через socket.ts и отображаются в пользовательском интерфейсе. На сервере предусмотрено безопасное переключение между соединениями, реализованное с использованием блокировок (async-mutex), что особенно важно при одновременном доступе нескольких клиентов.

В процессе тестирования производилась эмуляция нестабильного подключения и отключения эмулятора. Система в этих условиях корректно восстанавливала подключение после повторной попытки.

* 1. Выявленные проблемы и их устранение

В ходе тестирования были выявлены следующие ошибки, которые были своевременно устранены:

* отсутствие проверки на наличие файла настроек при загрузке модели;
* неочевидные ошибки при некорректном JSON — была добавлена система отображения текстовых сообщений в интерфейсе;
* повторное подключение к эмулятору без отключения предыдущего соединения — решено через дополнительную логику отключения WebSocket;
* некорректная работа ползунков и стрелок при выходе за допустимые пределы — решено добавлением ограничений min/max.
  1. Система логгирования и диагностики

Серверная часть содержит встроенную централизованную обработку ошибок (errorHandler.ts), позволяющую не прерывать выполнение программы при исключениях. Все ошибки возвращаются в клиент с понятным описанием.

На клиентской части используется визуальный компонент SStatusIndicator, отображающий текущее состояние модели и управления:

* зелёный — Online, Working;
* серый — Offline;
* оранжевый — Emergency Stopped.

На рисунке 7.6 представлены виды состояний

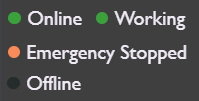


Рисунок 7.6 – Виды состояний

Такая система визуализации позволяет оперативно оценить текущее состояние системы без необходимости обращения к логам.

* 1. Выводы

По итогам тестирования можно сделать следующие выводы:

* Все ключевые компоненты системы прошли тестирование без критических сбоев.
* Интерфейс устойчив к ошибочным действиям пользователя и отображает корректные сообщения.
* Связь с внешним устройством по WebSocket стабильна, система надёжно работает как в режиме эмуляции, так и в условиях разрыва соединения.
* Все обнаруженные ошибки были устранены, а архитектура системы позволяет легко расширять покрытие автоматическими тестами (например, с использованием Jest и Mocha).

Таким образом, система готова к демонстрации и возможному внедрению в тестовую эксплуатацию.

# **ЗАКЛЮЧЕНИЕ**

Разработанное веб-приложение успешно решает задачу отображения взаимодействия настроек с 3D визуализацией телеуправляемого рентгеновского аппарата. В ходе выполнения работы были достигнуты следующие результаты:

* Создана реалистичная 3D-модель рентгеновского аппарата в Blender с экспортом в формат glTF 2.0.
* Реализован веб-интерфейс на основе React и Three.js, обеспечивающий интерактивное управление моделью через джойстик, кнопки и числовые поля.
* Разработан сервер-эмулятор на Node.js с Express, поддерживающий API для работы с моделями и их конфигурациями.
* Настроена база данных SQLite для хранения 3D-моделей и их настроек, обеспечивающая простоту и производительность.
* Проведено тестирование, подтвердившее работоспособность всех компонентов приложения.

Использование современных технологий, таких как Three.js, WebGL и Redux Toolkit, позволило создать масштабируемое и производительное решение. Применение JSON для конфигурации управления обеспечило гибкость и простоту настройки, а выбор SQLite минимизировал требования к серверным ресурсам.

Приложение имеет практическую ценность для образовательных целей, обучения медицинского персонала и демонстрации возможностей телеуправляемых рентгеновских систем.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Сайт Voron Design [Электронный ресурс]. — URL: <https://vorondesign.com/> (дата обращения 20.05.2025).
2. Сайт React [Электронный ресурс]. — URL: <https://react.dev/> (дата обращения 20.05.2025).
3. Сайт Redux [Электронный ресурс]. — URL: <https://redux.js.org/> (дата обращения 20.05.2025).
4. Сайт RTK [Электронный ресурс]. — URL: <https://redux-toolkit.js.org/rtk-query/overview> (дата обращения 20.05.2025).
5. Сайт Three.js [Электронный ресурс]. — URL: <https://threejs.org/> (дата обращения 20.05.2025).
6. Сайт Express [Электронный ресурс]. — URL: <https://expressjs.com/> (дата обращения 20.05.2025).
7. Сайт Socket.io [Электронный ресурс]. — URL: <https://socket.io/> (дата обращения 20.05.2025).
8. Сайт Sqlite [Электронный ресурс]. — URL: <https://www.sqlite.org/docs.html> (дата обращения 20.05.2025).
9. Сайт Blender [Электронный ресурс]. — URL: <https://www.blender.org/> (дата обращения 20.05.2025).
10. Полный код проекта [Электронный ресурс]. — URL: <https://github.com/NixNi/Diplom-2025> (дата обращения 20.05.2025).

**ПРИЛОЖЕНИЕ A**

**Содержимое файла CommandProcess.ts ответсвенного за обработку комманд эмулятора**

import type { Command, HardwareState } from "./../types/command";

import {

changeParameter,

getParameter,

setParameter,

} from "../models/parameters";

import type { Socket } from "socket.io";

const step = Number(process.env.step) || 0.1;

const timeout = Number(process.env.timeout) || 100;

let proccesses = 0;

export async function runCommand(socket: Socket, commandObject: Command) {

const { path, value, command } = commandObject;

if (command === "add") {

if (Math.abs(value) <= step) {

const result = await changeParameter(path, value);

if (result) {

socket.emit("command", result);

socket.broadcast.emit("command", result);

}

} else {

let valueI = value + ((await getParameter(path)) || 0);

const intervalID = setInterval(async () => {

const currentValue = (await getParameter(path)) || 0;

if (currentValue !== valueI) {

if (currentValue > valueI) {

const result = await setParameter(

path,

Math.max(currentValue - step, valueI)

);

if (result) {

socket.emit("command", result);

socket.broadcast.emit("command", result);

} else clearInterval(intervalID);

}

if (currentValue < valueI) {

const result = await setParameter(

path,

Math.min(currentValue + step, valueI)

);

if (result) {

socket.emit("command", result);

socket.broadcast.emit("command", result);

} else clearInterval(intervalID);

}

} else {

clearInterval(intervalID);

}

}, timeout);

}

}

if (command === "set") {

const currentValue = (await getParameter(path)) || 0;

if (Math.abs(currentValue - value) <= step) {

const result = await setParameter(path, value);

if (result) {

socket.emit("command", result);

socket.broadcast.emit("command", result);

}

} else {

setParameter("isControlsEnabled", 0);

socket.emit("state", { isControlsEnabled: 0 });

socket.broadcast.emit("state", { isControlsEnabled: 0 });

proccesses += 1;

const intervalID = setInterval(async () => {

if (await getParameter("isEmergencyStoped")) {

proccesses -= 1;

clearInterval(intervalID);

return;

}

const currentValue = (await getParameter(path)) || 0;

// console.log(path, currentValue, value);

if (currentValue !== value) {

if (currentValue > value) {

const result = await setParameter(

path,

Math.max(currentValue - step, value)

);

if (result) {

socket.emit("command", result);

socket.broadcast.emit("command", result);

} else {

proccesses -= 1;

if (!proccesses) {

setParameter("isControlsEnabled", 1);

socket.emit("state", { isControlsEnabled: 1 });

socket.broadcast.emit("state", { isControlsEnabled: 1 });

}

clearInterval(intervalID);

}

}

if (currentValue < value) {

const result = await setParameter(

path,

Math.min(currentValue + step, value)

);

if (result) {

socket.emit("command", result);

socket.broadcast.emit("command", result);

} else {

proccesses -= 1;

if (!proccesses) {

setParameter("isControlsEnabled", 1);

socket.emit("state", { isControlsEnabled: 1 });

socket.broadcast.emit("state", { isControlsEnabled: 1 });

}

clearInterval(intervalID);

}

}

} else {

proccesses -= 1;

if (!proccesses) {

setParameter("isControlsEnabled", 1);

socket.emit("state", { isControlsEnabled: 1 });

socket.broadcast.emit("state", { isControlsEnabled: 1 });

}

clearInterval(intervalID);

}

}, timeout);

}

}

}

export async function setState(socket: Socket, commandObject: HardwareState) {

if (!(commandObject.isControlsEnabled === undefined)) {

await setParameter(

"isControlsEnabled",

Number(commandObject.isControlsEnabled)

);

}

if (!(commandObject.isEmergencyStoped === undefined)) {

await setParameter(

"isEmergencyStoped",

Number(commandObject.isEmergencyStoped)

);

}

if (!(commandObject.isEnabled === undefined)) {

await setParameter("isEnabled", Number(commandObject.isEnabled));

}

socket.emit("state", commandObject);

socket.broadcast.emit("state", commandObject);

}

**Содержимое файла server/index.ts ответвенного за инициализацию сервера**

import express from "express";

import cors from "cors";

import bodyParser from "body-parser";

import cookieParser from "cookie-parser";

import { createServer } from "http";

import { Server } from "socket.io";

import modelsRouter from "./routes/models";

import settingsRouter from "./routes/settings";

import connectionsRouter from "./routes/connections";

import socketManager from "./middleware/sockets";

const app = express();

const server = createServer(app);

const whitelist = ["http://localhost:8046", "http://localhost:8045"];

const io = new Server(server, {

cors: {

origin: whitelist,

},

});

const port = process.env.PORT || 8046;

const corsOptions = {

origin: (origin: any, callback: any) => {

if (whitelist.indexOf(origin) !== -1) {

callback(null, true);

} else {

callback(null, true);

// callback(new Error("Not allowed by CORS"));

}

},

credentials: true,

};

const middleware = [

cors(corsOptions),

bodyParser.urlencoded({

limit: "50mb",

extended: true,

parameterLimit: 50000,

}),

bodyParser.json({ limit: "50mb" }),

cookieParser(),

];

middleware.forEach((it) => app.use(it));

app.get("/", (request, response) => {

response.send("It is working api, you check manually");

});

app.use("/api/models", modelsRouter);

app.use("/api/json", settingsRouter);

app.use("/api/connect", connectionsRouter);

io.on("connection", socketManager);

server.listen(port, () => console.log(`Running on port ${port}`));

**Содержимое файла IO.ts ответсвенного за создание socket.io сервера на эмуляторе**

import { Server } from "socket.io";

import { Server as HttpServer } from "http";

let io: Server | null = null;

export function setIO(httpServer: HttpServer, whitelist: string | string[]) {

io = new Server(httpServer, {

cors: {

origin: whitelist,

methods: ["GET", "POST"],

credentials: true,

},

});

return io;

}

export default io;

**Содержимое файла model.api.ts ответственного за отправку команд с клиента серверу.**

import { createApi, fetchBaseQuery } from "@reduxjs/toolkit/query/react";

import { ServerDataResponse, ServerResponse } from "../types/server";

export const modelApi = createApi({

reducerPath: "model/api",

baseQuery: fetchBaseQuery({

baseUrl: "/api/models",

}),

tagTypes: ["Models"],

endpoints: (build) => ({

getAllModelNames: build.query<

ServerDataResponse<{ id: number; name: string }[]>,

void

>({

query: () => ({

url: "/",

}),

providesTags: ["Models"],

}),

addModel: build.mutation<

ServerResponse,

{ name: string; data: ArrayBuffer; settings: string }

>({

query: (model) => {

const formData = new FormData();

formData.append("name", model.name);

formData.append("settings", model.settings);

formData.append(

"data",

new Blob([model.data], { type: "application/octet-stream" })

);

return {

url: "/",

method: "POST",

body: formData,

};

},

invalidatesTags: ["Models"],

}),

updateModelByName: build.mutation<

ServerResponse,

{ name: string; data?: ArrayBuffer | null; settings?: string | null }

>({

query: ({ name, data, settings }) => {

const formData = new FormData();

formData.append("name", name);

if (settings) formData.append("settings", settings);

if (data)

formData.append(

"data",

new Blob([data], { type: "application/octet-stream" })

);

return {

url: `/${name}`,

method: "PUT",

body: formData,

};

},

invalidatesTags: ["Models"],

}),

deleteModelByName: build.mutation<ServerResponse, string>({

query: (modelName) => ({

url: `/${modelName}`,

method: "DELETE",

}),

invalidatesTags: ["Models"],

}),

}),

});

export const {

useGetAllModelNamesQuery,

useAddModelMutation,

useUpdateModelByNameMutation,

useDeleteModelByNameMutation,

} = modelApi;

**Содержимое файла model.slice.ts ответственного за обработку команд на стороне клиента**

import { createAsyncThunk, createSlice, PayloadAction } from "@reduxjs/toolkit";

import {

CommandResponse,

HardwareState,

ModelControls,

ModelPositions,

ValuesArray,

} from "../../types/models";

// import { sendCommand } from "../../hooks/socket";

import { sendCommand, sendState } from "../../socket";

interface ModelState {

id: number;

name: string;

modelControls: ModelControls;

positions: ModelPositions;

isEnabled: boolean;

isControlsEnabled: boolean;

isEmergencyStoped: boolean;

isLoadingControls: boolean;

isErrorControls: boolean;

isLoadingData: boolean;

isErrorData: boolean;

errorMessage: string | null;

mode: "online" | "offline";

}

const initialState: ModelState = {

id: 0,

name: "default",

modelControls: { models: {}, controlElements: [] },

positions: {},

isEnabled: true,

isControlsEnabled: true,

isEmergencyStoped: false,

isLoadingControls: false,

isErrorControls: false,

isLoadingData: false,

isErrorData: false,

errorMessage: null,

mode: "offline",

};

export const updateModelDataAsync = createAsyncThunk<ArrayBuffer, void>(

"model/updateModelDataAsync",

async (\_, { getState, signal }) => {

const state = getState() as { model: ModelState };

const modelName = state.model.name;

if (!modelName || modelName === "default") {

throw new Error("Choose a model");

}

try {

const response = await fetch(`/api/models/${modelName}`, { signal });

if (!response.ok) {

throw new Error(`Failed to fetch model data: ${response.statusText}`);

}

return await response.arrayBuffer();

} catch (error) {

throw new Error(`Error fetching model data: ${(error as Error).message}`);

}

}

);

export const updateModelControlsAsync = createAsyncThunk(

"model/updateModelControlsAsync",

async (\_, { getState, signal }) => {

const state = getState() as { model: ModelState };

const modelName = state.model.name;

if (!modelName || modelName === "default") {

throw new Error("Choose a model");

}

try {

const response = await fetch(`/api/json/${modelName}`, { signal });

if (!response.ok) {

throw new Error(

`Failed to fetch model controls: ${response.statusText}`

);

}

const updatedModelData = await response.json(); // Используем response.json() вместо arrayBuffer

return updatedModelData;

} catch (error) {

throw new Error(

`Error fetching model controls: ${(error as Error).message}`

);

}

}

);

export const modelSlice = createSlice({

name: "model",

initialState,

reducers: {

//################\_\_\_\_\_\_\_OFFLINE MODE\_\_\_\_\_\_\_####################

setModelName: (state, action: PayloadAction<string>) => {

state.name = action.payload;

state.modelControls = { models: {}, controlElements: [] };

state.positions = {};

state.isControlsEnabled = true;

state.isEnabled = true;

state.isEmergencyStoped = false;

state.isLoadingControls = false;

state.isErrorControls = false;

state.isLoadingData = false;

state.isErrorData = false;

state.errorMessage = null;

},

setMode: (state, action: PayloadAction<"online" | "offline">) => {

state.mode = action.payload;

},

resetErrors: (state) => {

state.isErrorControls = false;

state.isErrorData = false;

state.errorMessage = null;

},

resetModelState: (state) => {

state.modelControls = { models: {}, controlElements: [] };

state.positions = {};

state.isControlsEnabled = true;

state.isEnabled = true;

state.isEmergencyStoped = false;

state.isLoadingControls = false;

state.isErrorControls = false;

state.isLoadingData = false;

state.isErrorData = false;

state.errorMessage = null;

state.mode = "offline";

state.name = "default";

},

updatePositionsLocal: (state, action: PayloadAction<ModelPositions>) => {

state.positions = action.payload;

},

updateState: (state, actions: PayloadAction<HardwareState>) => {

state.isControlsEnabled =

actions.payload.isControlsEnabled === undefined

? state.isControlsEnabled

: actions.payload.isControlsEnabled;

state.isEmergencyStoped =

actions.payload.isEmergencyStoped === undefined

? state.isEmergencyStoped

: actions.payload.isEmergencyStoped;

state.isEnabled =

actions.payload.isEnabled === undefined

? state.isEnabled

: actions.payload.isEnabled;

},

//##############################################################

//#################\_\_\_\_\_\_\_ONLINE MODE\_\_\_\_\_\_\_####################

updateParametersFromHardware: (

state,

action: PayloadAction<{ Parameter: string; Value: number }[]>

) => {

const StatesArray = action.payload;

for (let item of StatesArray) {

if (item.Parameter === "isEnabled") {

state.isEnabled = Boolean(item.Value);

// console.log(item.Value);

continue;

}

if (item.Parameter === "isEmergencyStoped") {

state.isEmergencyStoped = Boolean(item.Value);

continue;

}

if (item.Parameter === "isControlsEnabled") {

state.isControlsEnabled = Boolean(item.Value);

continue;

}

const path\_spl = item.Parameter.split("/");

if (path\_spl.length === 0) return;

let current: any = state.positions;

for (let i = 0; i < path\_spl.length - 1; i++) {

const key = path\_spl[i];

if (!current[key]) {

current[key] = {};

}

current = current[key];

}

const lastKey = path\_spl[path\_spl.length - 1];

current[lastKey] = item.Value;

}

},

updateSetModelPositionOnline: (

state,

action: PayloadAction<ValuesArray>

) => {

if (state.mode === "online") {

for (let valPath of action.payload) {

const command: CommandResponse = {

command: "set",

path: valPath.path,

value: valPath.value,

};

sendCommand(command);

}

return;

}

},

switchEanbled: (state) => {

const TemporaryState: HardwareState = {};

TemporaryState.isEnabled = !state.isEnabled;

if (TemporaryState.isEnabled) {

TemporaryState.isControlsEnabled = true;

TemporaryState.isEmergencyStoped = false;

}

if (state.mode === "online") {

sendState(TemporaryState);

return;

}

state.isEnabled = TemporaryState.isEnabled;

state.isControlsEnabled =

TemporaryState.isControlsEnabled === undefined

? state.isControlsEnabled

: TemporaryState.isControlsEnabled;

state.isEmergencyStoped =

TemporaryState.isEmergencyStoped === undefined

? state.isEmergencyStoped

: TemporaryState.isEmergencyStoped;

},

setControlsEnabled: (state, action: PayloadAction<boolean>) => {

if (state.mode === "online") {

sendState({ isControlsEnabled: action.payload });

return;

}

state.isControlsEnabled = action.payload;

},

setEmergency: (state) => {

if (state.mode === "online") {

sendState({ isEmergencyStoped: true });

return;

}

state.isEmergencyStoped = true;

},

updateModelPositionLocal: (

state,

action: PayloadAction<CommandResponse>

) => {

const { command, path, value, isNeedOnlineCheck = true } = action.payload;

if (state.mode === "online" && isNeedOnlineCheck) {

sendCommand(action.payload);

return;

}

const path\_spl = path.split("/");

if (path\_spl.length === 0) return;

let current: any = state.positions;

let currentControls: any = state.modelControls.models;

for (let i = 0; i < path\_spl.length - 1; i++) {

const key = path\_spl[i];

if (!current[key]) {

current[key] = {};

}

if (!currentControls[key]) {

currentControls[key] = {};

}

current = current[key];

currentControls = currentControls[key];

}

// Имя конечного свойства

const lastKey = path\_spl[path\_spl.length - 1];

if (!currentControls[lastKey] || currentControls[lastKey].length !== 2)

currentControls[lastKey] = [-10, 10];

const [min, max] = currentControls[lastKey] as [number, number];

// Обновляем значение в зависимости от команды

if (command === "set") {

current[lastKey] = Math.max(Math.min(value, max), min);

} else if (command === "add") {

current[lastKey] = Math.max(

Math.min((current[lastKey] || 0) + value, max),

min

);

}

},

//##############################################################

},

extraReducers: (builder) => {

builder

// Обработка updateModelDataAsync (modelData)

.addCase(updateModelDataAsync.pending, (state) => {

state.isLoadingData = true;

state.isErrorData = false;

state.errorMessage = null;

})

.addCase(updateModelDataAsync.fulfilled, (state) => {

state.isLoadingData = false;

// Не сохраняем modelData в состоянии

})

.addCase(updateModelDataAsync.rejected, (state, action) => {

state.isLoadingData = false;

state.isErrorData = true;

state.errorMessage =

action.error.message || "Failed to fetch model data";

})

.addCase(updateModelControlsAsync.pending, (state) => {

state.isLoadingControls = true;

state.isErrorControls = false;

state.errorMessage = null;

})

.addCase(updateModelControlsAsync.fulfilled, (state, action) => {

state.isLoadingControls = false;

state.modelControls = action.payload;

})

.addCase(updateModelControlsAsync.rejected, (state, action) => {

state.isLoadingControls = false;

state.isErrorControls = true;

state.errorMessage =

action.error.message || "Failed to fetch model controls";

});

},

});

export const modelActions = {

...modelSlice.actions,

updateModelControlsAsync,

updateModelDataAsync,

};

export const modelReducer = modelSlice.reducer;

**Содержимое файла ModelControls.tsx ответсвенного за отображение элементов взаимодействия.**

import { controlElement } from "../types/models";

import { useAppSelector } from "../hooks/redux";

import SControlJoystic from "./shared/SControlJoystick";

import SSetButton from "./shared/SSetButton";

import SArrowButtons from "./shared/SArrowButtons";

import SPowerButton from "./shared/SPowerButton";

import SEmergencyStopButton from "./shared/SEmergencyStopButton";

import "./css/ControlGrid.css";

export const ModelControlsComponent = () => {

const modelControls = useAppSelector((state) => state.model.modelControls);

function chooseElement(el: controlElement) {

if (

![

"Joystick",

"setButton",

"ArrowButtons",

"PowerButton",

"EmergencyStop",

].includes(el.element)

)

return <div>Element Not Found</div>;

return (

<div key={el.name} className="p-2">

{/\* <p>{el.name}</p> \*/}

{(el.element === "Joystick" && (

<SControlJoystic key={el.name} element={el} />

)) ||

(el.element === "setButton" && (

<SSetButton key={el.name} element={el} />

)) ||

(el.element === "ArrowButtons" && (

<SArrowButtons key={el.name} element={el} />

)) ||

(el.element === "PowerButton" && (

<SPowerButton key={el.name} element={el} />

)) ||

(el.element === "EmergencyStop" && (

<SEmergencyStopButton key={el.name} />

))}

</div>

);

}

const positions: { [key: string]: controlElement[] } = {

TopLeft: [],

Top: [],

TopRight: [],

Left: [],

Center: [],

Right: [],

BottomLeft: [],

Bottom: [],

BottomRight: [],

};

modelControls.controlElements?.forEach((el) => {

const position =

el.position && positions[el.position] ? el.position : "Center";

positions[position].push(el);

});

return (

<div className="control-grid">

{Object.keys(positions).map((position) => (

<div key={position} className={`flex`}>

{positions[position].map((el) => chooseElement(el))}

</div>

))}

</div>

);

};

**Содержимое файла models types.ts содержащиего интерфейсы с которыми взаимодействует программа**

export interface xyz {

x?: number;

y?: number;

z?: number;

}

export interface xyzController {

x?: [number, number];

y?: [number, number];

z?: [number, number];

}

export interface ModelPositions {

[key: string]: {

position?: xyz;

rotation?: xyz;

};

}

export type modelValuePath = ["position" | "rotation", keyof xyz];

export type interfacePositions =

| "TopLeft"

| "Top"

| "TopRight"

| "Left"

| "Center"

| "Right"

| "BottomLeft"

| "Bottom"

| "BottomRight"

| undefined;

export interface JoystickControlElement {

name: string;

element: "Joystick";

position: interfacePositions;

props: {

xpath: string;

ypath: string;

topImg?: string;

bottomImg?: string;

leftImg?: string;

rightImg?: string;

};

}

export type ValuesArray = {

path: string;

value: number;

}[];

export interface SetButtonControlElement {

name: string;

element: "setButton";

position: interfacePositions;

props: {

values: ValuesArray;

};

}

export interface SArrowButtonsElement {

name: string;

element: "ArrowButtons";

position: interfacePositions;

props: {

type: "up/down" | "left/right";

path: string;

topImg?: string;

bottomImg?: string;

leftImg?: string;

rightImg?: string;

};

}

export interface SPowerButtonElement {

name: string;

element: "PowerButton";

position: interfacePositions;

props: {

values: ValuesArray;

};

}

export interface SEmergencyStopElement {

name: string;

element: "EmergencyStop";

position: interfacePositions;

}

export type controlElement =

| JoystickControlElement

| SetButtonControlElement

| SArrowButtonsElement

| SPowerButtonElement

| SEmergencyStopElement;

export interface ModelControls {

models: {

[key: string]: { position?: xyzController; rotation?: xyzController };

};

controlElements: Array<controlElement>;

}

export interface HardwareState {

isEnabled?: boolean;

isControlsEnabled?: boolean;

isEmergencyStoped?: boolean;

}

export interface CommandResponse {

command: "set" | "add";

path: string;

value: number;

isNeedOnlineCheck?: boolean;

}

**Содержимое файла server/routes/models.ts ответсвенного за обработку на сервере комманд поступающих с клиента**

import express from "express";

import multer from "multer";

import {

getModelByName,

addModel,

updateModelByName,

deleteModelByName,

getAllModelNames,

} from "../models/models.model";

import errorHandler from "src/hooks/errorHandler";

const modelsRouter = express.Router();

const upload = multer();

// Получение всех имен моделей с их ID

modelsRouter.get("/", async (req, res) => {

await errorHandler(res, async () => {

const models = await getAllModelNames();

res.json({ status: "success", data: models });

});

});

// Получение модели по имени

modelsRouter.get("/:modelName", async (req, res) => {

const modelName = req.params.modelName;

await errorHandler(res, async () => {

const model = await getModelByName(modelName);

res.setHeader("Content-Type", "application/octet-stream");

res.send(Buffer.from(model));

});

});

// Добавление новой модели

modelsRouter.post("/", upload.single("data"), async (req, res) => {

const { name, settings } = req.body;

await errorHandler(res, async () => {

if (!req.file?.buffer) throw new Error("Model is missing");

const data = req.file.buffer; // Получаем бинарные данные из файла

await addModel(name, data, settings);

res.json({

status: "success",

message: `Model ${name} added successfully`,

});

});

});

modelsRouter.put("/:modelName", upload.single("data"), async (req, res) => {

const { name, settings } = req.body;

await errorHandler(res, async () => {

// if (!req.file?.buffer) throw new Error("Model is missing");

const data = req.file?.buffer; // Получаем бинарные данные из файла

await updateModelByName(name, { newData: data, settings });

res.json({

status: "success",

message: `Model ${name} updated successfully`,

});

});

});

// Удаление модели по имени

modelsRouter.delete("/:modelName", async (req, res) => {

const modelName = req.params.modelName;

await errorHandler(res, async () => {

await deleteModelByName(modelName);

res.json({

status: "success",

message: `Model ${modelName} deleted successfully`,

});

});

});

export default modelsRouter;

**Содержимое файла ModelViewer.tsx ответсвенного за отображение модели в браузере.**

import { useRef } from "react";

import { useThreeSetup } from "../hooks/useThreeSetup";

import { useModelLoader } from "../hooks/useModelLoader";

import { ModelControlsInputs } from "./ModelControlsInputs";

import { ModelControlsComponent } from "./ModelControls";

import { useAppSelector } from "../hooks/redux";

import STabViewer from "./shared/STabViewer";

import STab from "./shared/STab";

interface ModelViewerProps {

size?: { x: number; y: number };

modelControlsEnable: boolean;

children?: React.ReactNode;

}

const ModelViewer = ({

size,

modelControlsEnable,

children,

}: ModelViewerProps) => {

const model = useAppSelector((state) => state.model);

const mountRef = useRef<HTMLDivElement>(null);

const { scene } = useThreeSetup(mountRef, size);

const { errorMessage, modelLoaded } = useModelLoader(scene);

const loaderError = model.errorMessage;

return (

<div className="relative flex flex-wrap">

<div

className="relative flex"

style={{ width: size?.x || 600, height: size?.y || 600 }}

>

<div

ref={mountRef}

style={{

visibility: modelLoaded ? "visible" : "hidden",

position: modelLoaded ? "initial" : "absolute",

}}

/>

{(errorMessage || loaderError) && (

<div

style={{ width: size?.x || 600, height: size?.y || 600 }}

className="flex filler absolute"

>

<div className="secondary m-auto p-10 font-bold text-5">

{loaderError || errorMessage}

</div>

</div>

)}

{children}

</div>

{modelLoaded && modelControlsEnable && model.modelControls?.models && (

<STabViewer>

<STab title="Интерфейс" default>

<ModelControlsComponent />

</STab>

<STab title="Ручной ввод">

<ModelControlsInputs />

</STab>

</STabViewer>

)}

</div>

);

};

export default ModelViewer;

**Содержимое файла SQLiteConnection.ts ответсвенного за подключение сервера к базе данных**

import initDB from "db/init-db";

import path from "path";

const dbPath = path.resolve(\_\_dirname, "../../db/data.db");

console.log(`Попытка подключения к базе данных по пути: ${dbPath}`);

const db = initDB();

export default db;

**Содержимое файла useModelLoader.ts ответсвенного за загрузку и парсинг модели**

import { useAppSelector } from "./redux";

import { useState, useRef, useEffect } from "react";

import { GLTFLoader } from "three/examples/jsm/Addons.js";

import \* as THREE from "three";

import { useAppDispatch } from "../store";

import { xyz, ModelPositions } from "../types/models";

import { updateModelDataAsync } from "../store/model/model.slice";

import { useActions } from "./actions";

export const useModelLoader = (

scene: THREE.Scene,

{ model: modelPassed }: { model?: ArrayBuffer | null } = {}

) => {

const modelRef = useRef<THREE.Object3D | null>(null);

const [modelData, setModelData] = useState<ArrayBuffer | null>(null);

const [errorMessage, setErrorMessage] = useState<string | null>(null);

const [modelLoaded, setModelLoaded] = useState(false);

const dispatch = useAppDispatch();

const actions = useActions();

const model = useAppSelector((state) => state.model);

const isLoading = model.isLoadingData || model.isLoadingControls;

const isError = model.isErrorData || model.isErrorControls;

const isOnline = model.mode === "online";

const modelControls = model.modelControls;

const positions = model.positions;

useEffect(() => {

setErrorMessage(null);

async function modelLoad() {

try {

const result = await dispatch(updateModelDataAsync()).unwrap();

setModelData(result);

} catch (error: any) {

if (error.message !== "Choose a model")

console.error("Failed to load model data:", error);

}

}

if (modelPassed) {

actions.resetErrors();

setModelData(modelPassed);

} else modelLoad();

}, [model.name, modelPassed]);

useEffect(() => {

setModelLoaded(false);

if (!modelData || isLoading || isError) return;

const loader = new GLTFLoader();

if (modelRef.current) {

scene.remove(modelRef.current);

modelRef.current = null;

}

loader.parse(

modelData,

"",

(gltf) => {

// console.log(gltf);

if (modelRef.current) {

scene.remove(modelRef.current);

modelRef.current = null;

}

modelRef.current = gltf.scene;

scene.add(gltf.scene);

setModelLoaded(true);

const newPositions: ModelPositions = {};

gltf.scene.traverse((child) => {

if (child.type === "Mesh") {

child.castShadow = true;

child.receiveShadow = true;

}

const controlEntry = modelControls?.models?.[child.name];

if (controlEntry) {

const position: xyz = {};

const rotation: xyz = {};

if (controlEntry.position) {

Object.keys(controlEntry.position).forEach((axis) => {

if (["x", "y", "z"].includes(axis)) {

position[axis as keyof xyz] =

child.position[axis as "x" | "y" | "z"];

}

});

}

if (controlEntry.rotation) {

Object.keys(controlEntry.rotation).forEach((axis) => {

if (["x", "y", "z"].includes(axis)) {

rotation[axis as keyof xyz] =

child.rotation[axis as "x" | "y" | "z"];

}

});

}

newPositions[child.name] = {

position: Object.keys(position).length ? position : undefined,

rotation: Object.keys(rotation).length ? rotation : undefined,

};

}

});

if (!isOnline) actions.updatePositionsLocal(newPositions);

},

(error) => {

setErrorMessage("Error parsing model: " + error.message);

}

);

return () => {

if (modelRef.current) {

scene.remove(modelRef.current);

modelRef.current = null;

}

};

}, [modelData, scene, modelControls]);

useEffect(() => {

if (modelLoaded && modelRef.current) {

Object.keys(positions).forEach((it) => {

const part = modelRef.current?.getObjectByName(it);

if (part) {

if (positions[it].position) {

if (positions[it].position.x)

part.position.x = positions[it].position.x;

if (positions[it].position.y)

part.position.y = positions[it].position.y;

if (positions[it].position.z)

part.position.z = positions[it].position.z;

}

if (positions[it].rotation) {

if (positions[it].rotation.x)

part.rotation.x = positions[it].rotation.x;

if (positions[it].rotation.y)

part.rotation.y = positions[it].rotation.y;

if (positions[it].rotation.z)

part.rotation.z = positions[it].rotation.z;

}

}

});

}

}, [positions, modelLoaded]);

return { modelRef, errorMessage, modelLoaded };

};

**Содержимое файла wsConnection.ts ответственного за подключение сервера к эмулятору с помощью протокола web socket**

import { io as ioClient, Socket } from "socket.io-client";

export function WsConnect(ip: string, port: number) {

return new Promise<Socket>((resolve, reject) => {

const url = `http://${ip}:${port}`;

const serverSocket = ioClient(url);

serverSocket.on("connect", () => {

console.log("Connected to Emulator");

resolve(serverSocket);

});

serverSocket.on("connect\_error", (err) => {

serverSocket.disconnect();

console.log(url);

reject(new Error("Server Not Connected"));

});

serverSocket.on("disconnect", () => {

console.log("Disconnected from Emulator");

});

});

}