Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

Московский государственный технический университет имени Н.Э. Баумана
(МГТУ им. Н.Э. Баумана)

Факультет: «Информатика и системы управления»
Кафедра: «Теоретическая информатика и компьютерные технологии»

Расчетно-пояснительная записка к выпускной квалификационной работе

Редактирование STL моделей

Научный руководитель: _____ (А. Б. Домрачева)
(подпись, дата)

Студент группы ИУ9-82: _____ (А. В. Беляев)
(подпись, дата)

Москва, 2018

АННОТАЦИЯ

Работа посвящена созданию редактора для преобразования STL-моделей с использованием аппарата булевых функций. Обработка моделей рассматривается в контексте биомедицинских технологий и в рамках технологического процесса по моделированию имплантов.

Проведено исследование как общей предметной области — 3D-моделирования, так и специфики, накладываемой биомедицинской тематикой. Изучена имеющаяся на данный момент инфраструктура, и на основе результатов анализа сформирован ряд требований к разрабатываемому программному обеспечению.

Проведено проектирование программного комплекса на основе сформулированных требований и выполнена программная реализация в виде распределенной системы обработки моделей. Разработано пользовательское приложение для манипуляции 3D-объектами в STL формате и серверная часть, выполняющая преобразование объектов с использованием булевых функций.

Прототип системы, построенной на основе клиент-серверной архитектуры, изучен на предмет недостатков «наивной» реализации. Произведена оптимизация компонентов системы. Выполнена оценка возможности применения оптимизации и определены направления для дальнейшего развития системы.

Пояснительная записка к выпускной квалификационной работе содержит 80 страниц текста формата А4, 28 рисунков, 2 таблицы, 28 листингов и список используемой литературы, включающий 10 библиографических источников.

Содержание

ВВЕДЕНИЕ

Последнее десятилетие связано с разработкой программного обеспечения (далее – ПО) для поддержки исследований в области медицины, в том числе моделирования имплантов и протезов с использованием 3D-печати. Благодаря современному развитию технологий это становится возможно и начинает все более активно применяться в реальных условиях.

Использование специализированного программного обеспечения в предоперационный период позволяет существенно повысить точность производимых манипуляций в интероперационный период. В предоперационный период осуществляется сбор данных о пациенте, их анализ и моделирование необходимых пациенту искусственных компонентов. Этими компонентами могут быть всевозможные импланты и протезы. Точность изготовляемых протезов напрямую зависит от этапа моделирования, а автоматизация этого процесса ведет как к повышению качества, так и повышению количества проводимых операций.

Между тем, к имплантируемым и протезируемым моделям предъявляются повышенные требования, как с точки зрения точности, так и используемых материалов, и, особенно, к показателям прочности и износостойкости моделей. Неверное решение специалиста способно навредить пациенту.

В МГТУ им. Баумана разрабатывается комплекс средств в области автоматизации биомедицинских технологий. Так, совместная работа факультетов ИУ (информатики и систем управления) и РК (робототехники и комплексной автоматизации) способствовала достижению результатов в сфере проектирования челюстно-лицевых имплантов. Ведутся разработки в области автоматизации как подготовки, так и проведения операции.

Предполагается разработка программного обеспечения для автоматизации проектирования 3D-модели импланта или протеза специалистом-технологом, так как в данный момент частично этот этап происходит с использованием восковых и пластиковых моделей, что снижает качество полученной модели. Входными данными, поступающими на обработку в программный комплекс являются 3D-модели костей пациента, представленные в специализированном формате для 3D-печати – STL. Выходными данными являются модель или набор моделей в формате STL, обработанных программным комплексом и готовых к 3D-печати.

Преобразование STL-моделей производится с помощью библиотеки, разработанной ранее в МГТУ им. Баумана в рамках совместной работы факультетов, описанной выше. Предполагается тестирование программного обеспечения на наборе STL-моделей, представленных в свободном доступе на сайте Брненского технического университета (http://biomechanika.fme.vutbr.cz/). Представленные модели составляют скелетную систему человека.

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 STL формат

Формат STL был разработан в 1987 году как способ представления необработанной («сырой») триангулированной поверхности с помощью набора вершин, и нормалей, представленных вещественными числами в декартовой системе координат. В первоначальной спецификации устанавливалось ограничение на положительность значение координат, но к настоящему моменту это ограничение не учитывается и вероятность нахождения отрицательных координат в файле STL формата достаточно велика. В общем случае можно говорить, что **объект в формате STL есть результат триангуляции матрицы высот** или **представление геометрии поверхности тесселяцией**. Пример визуализации STL формата приведен на рис. 1: сфера, как и любой другой объект, может быть триангулирована множеством способов и основная задача при выборе представления – соблюдение баланса между точностью и сложность представления.

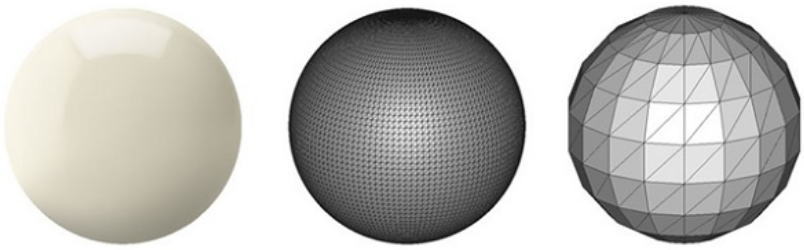

Рис. 1: Сфера и способы ее триангулированного представления с разной степенью точности

Стоит отметить, что формат не хранит информации о масштабах модели и, таким образом, они произвольны. Также стандартом не предусмотрен способ хранения информации о цвете модели и материале изготовления. Тем не менее, форматом предусмотрен зарезервированный атрибут, в значение которого разными программами обработки STL записывают как данные о цвете, так и о материале.

Механизмами обработки стереолитографии являются 3D-принтеры, способные представить объемную модель набором слоев. Очевидно, что при печати принтеру необходим 2D-контур и наиболее выгодным с точки зрения машинного представления является хранение набора замкнутых контуров. Тем не менее, толщина слоя зависит от используемого материала и точности печати и, таким образом, при разных значениях этих величин должны быть использованы разные контуры. Вместо этого оказалось проще использовать более наглядное представление (в виде набора треугольников) и выполнять разделение на слои непосредственно при печати.

Несмотря на то, что в 1987 году формат изначально был задуман для применения в 3D-печати, популярность он получил лишь в современном мире с развитием и удешевлением технологии 3D-печати. Так, 3D-печать сегодня используется в потребительско секторе и сфере быстрого прототипирования (макетирования), где важна

первоначальная заготовка и дальнейшая разработка производится уточнением первоначальной модели. Также 3D-печать все активнее используется в медицине и в частности – при проектировании протезов и имплантов.

Формат STL предусматривает 2 способа представления моделей: в **двоичном формате** и в текстовом **формате ASCII**. На Листинге 1 представлена структура STL-модели в формате ASCII. $n_i, n_j, n_k, v1_x, \ldots, v3_z$ – значения вершин, представленные вещественными числами.

Листинг 1: Текстовый формат представления STL модели

```
solid name
facet normal ni nj nk
    outer loop
        vertex v1x v1y v1z
        vertex v2x v2y v2z
        vertex v3x v3y v3z
    endloop
endfacet
endsolid name
```

На Листинге 2 представлена аналогичная структура, но в двоичном формате.

Листинг 2: Двоичный формат представления STL модели

```
UINT8[80] - заголовок
UINT32 - общее число треугольников

foreach triangle
REAL32[3] - вектор нормали
REAL32[3] - вершина 1
REAL32[3] - вершина 2
REAL32[3] - вершина 3
UINT16 - зарезервированный атрибут цвет(, текстура)
end
```

Из листингов выше становится понятен главный недостаток STL формата: объем представления модели стремительно растет с увеличением точности. При этом одни и те же вершины дублируются между многими треугольниками. Таим образом приходится соблюдать компромисс между точностью представления и объемом получившейся модели.

Прежде чем приступить к разработке, необходимо сформировать список требований к разрабатываемому ПО. Для этого требуется изучить представленные на рынке на данный момент решения. Затем, следует изучить возможности библиотеки, на которой будет базироваться основная функциональность программного комплекса. После этого список требований может быть сформирован и дальнейшая работа над проектом будет проходить с учетом поставленных требований.

1.2 Изучение приложений-аналогов

На данный момент (2018 год) технология 3D-печати набирает все бОльшую популярность, что способствует развитию как самой технологии, так и сопутствующей инфраструктуры. Благодаря высокому спросу появляются высокоскоростные 3D-принтеры, превосходящие вышедшие 1-2 года назад модели в скорости и качестве во множество раз. Появляется новая технология 3D-печати с помощью металла. Вместе со спросом на технологию растет и спрос на программное обеспечение, позволяющее использовать технологию все более эффективно. При этом в среде программного обеспечения существует разделение на профессиональное и непрофессиональное («любительское») ПО.

1.2.1 Непрофессиональное программное обеспечение

Непрофессиональное ПО способствует популяризации 3D-печати и распространению технологии «в массы». То есть, популярность технологии способствует упрощению способов ее использования, что в свою очередь ведет к еще большей популярности. Изучение непрофессионального ПО не является целью данной работы и присутствует в ознакомительных целях.

SketchUp (Trimble Inc, https://www.sketchup.com/) является программой для многопрофильного 3D моделирования. С ее помощью могут быть смоделированы архитектура и дизайн интерьера, ландшафтный дизайн, механизмы машиностроения, а также объекты, используемые в компьютерных играх и кинематографии. Инфраструктура вокруг программы достаточно развита: существует большая общедоступная библиотека 3D моделей со множеством готовых объектов (3D Warehouse), существует интеграция с картографическим сервисом Google Earth для экспорта моделей зданий на карту, а также существует большое количество плагинов (дополнений).

В библиотеке дополнений SketchUp имеется плагин SketchUp STL. Этот плагин позволяет экспортировать модели, находящиеся в сцене в формат STL, предварительно сгруппировав их. Также поддерживается импорт внешних моделей в SketchUp-сцену.

Функциональность SketchUp по-умолчанию невелика, но, благодаря дополнениям, она существенно расширяется. Это является одной из причин, по которой это ПО не используется для серьезных проектов, связанных с 3D-моделированием и коммерческой 3D-печатью: расширения написаны сообществом и могут содержать ошибки или конфликтовать друг с другом. Тем не менее, SketchUp подходит для моделирования в ознакомительных и развлекательных целях, т.е. в тех ситуациях, когда нет требований к точности проектируемой модели.

С технической токи зрения ПО представляет из себя классическую настольную программу и не требовательно к конфигурации оборудования. Графический интерфейс программы прост и интуитивно понятен. Скриншот интерфейса программы представлен на рис. 2.

1.2.2 Профессиональное программное обеспечение

Профессиональное программное обеспечение для 3D-печати используется в ходе рабочей деятельности во многих сферах. Это стимулирует развитие технологии в глубину, т.е. повышения качества изготовляемых моделей и повышение требова-

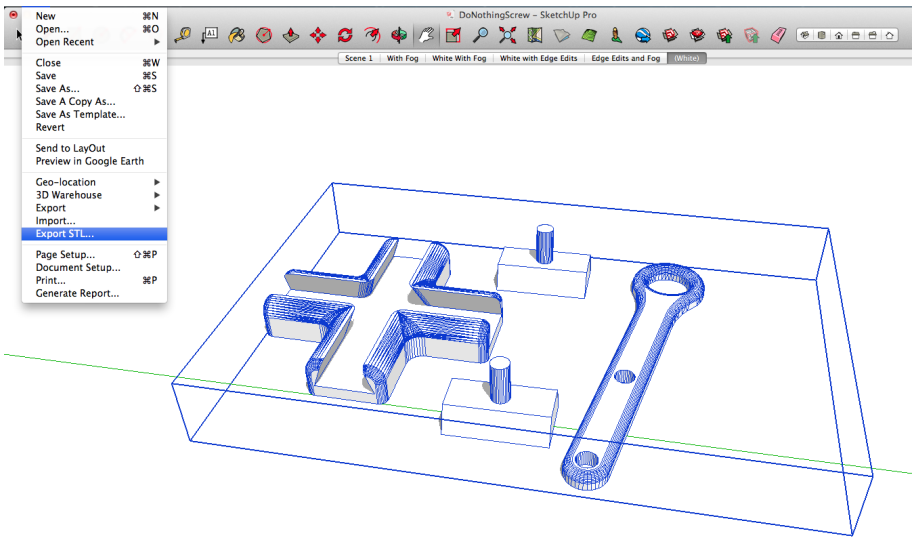

Рис. 2: Графический интерфейс программы SketchUP

ний к функциональности ПО. Сферы профессиональной деятельности, в которых возможно использование 3D-печати, могут быть разделены на несколько групп:

- архитектура и машиностроение. К моделям ставятся высокие требования ввиду возможных нагрузок и условий их использования. Из-за этого 3D-печать используется в минимальных объемах. Специализированного программного обеспечения для 3D-печати в этих сферах почти нет. Тем не менее, большинство CAD-программ (AutoCAD, SolidEdge, Inventor) поддерживают конвертацию в STL;

- потребительский сектор. Требования к моделям эстетические. Специализированным ПО являются программы среднего уровня сложности, такие как FreeCad, MeshMixer;

- медицина. Как правило, медицина персонализированная и, следовательно, требования к моделям ставятся более высокие, чем в машиностроительном моделировании. Программное обеспечение используется специализированное, спроектированное с учетом медицинской/научной специфики. Пример подобного ПО - 3D Slicer.

Существует так же универсальное ПО для 3D моделирования, которое поддерживает работу с STL. Пример такого графического пакета – **Blender** (Blender Foundation, http://blender.org/). В первую очередь это графический пакет для работы с 3D-графикой, визуальными эффектами и анимацией в кинематографе, видео играх и связанных областях. Развитие кинематографа и видеоигр привело к поддержке 3D-печати в этом графическом пакете. Более того, в Blender даже имеется встроенный игровой «движок». Обилие функций делает редактор универсальным, но, таким образом, поддержка STL является незаметной частью редактора и теряется на фоне других функций. Графический интерфейс из-за большого объема функциональности

перегружен и сложен в освоении. Все это делает редактор узкоспециализированным и создает высокий порог вхождения. Окно редактора представлено на рис. 3.

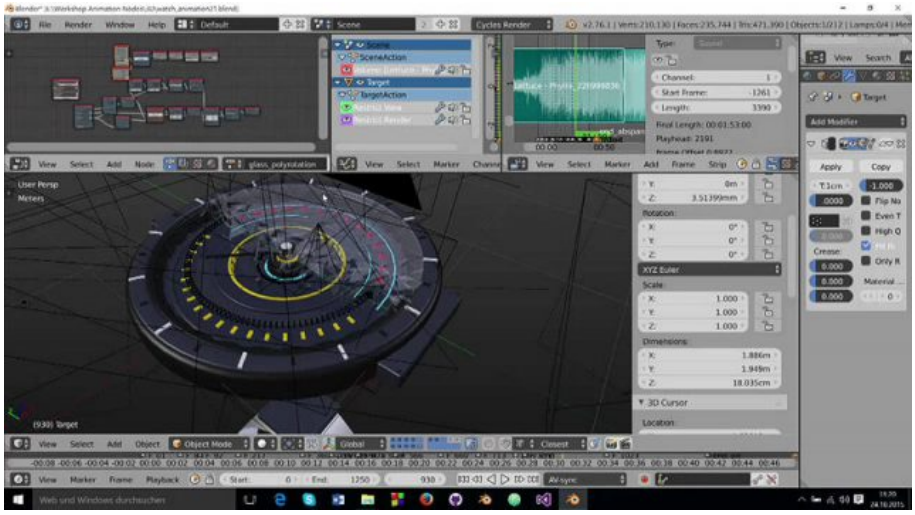

Рис. 3: Сложный и зашумленный интерфейс Blender'а

Blender, из-за наличия большого количества функций, очень требователен к оборудованию пользователя и, очевидно, представлен настольным приложением.

3D Slicer (The Slicer Community, http://www.slicer.org/) является примером специализированной программы для научной визуализации и анализа изображений, поддерживающей работу с STL. Эта программа поддерживает импорт файлов в формате DICOM, «умеет» самостоятельно разбирать DICOM файлы и организовывать и сохранять их во встроенную базу данных. В 3D Slicer имеется Volume Rendering модуль, отвечающий за преобразование снимков (например, МРТ) в объемную модель. Подобная функция представлена на рис. 4 и 5. 3D Slicer поддерживает подключение дополнений и является по сути расширяемой платформой. С технической стороны это приложение также является настольной программой. Исходный код программы находится в свободном доступе.

FreeCad (https://www.freecadweb.org/) является кроссплатформаенной настольной программой для работы с большим количеством свободных форматов: STL, SVG, OBJ, DAE и других. FreeCad является программным обеспечением с открытым исходным кодом, а отличительной его чертой является поддержка функциональности редактора с помощью библиотеки на языке Python. Целевой аудиторией являются разработчики, студенты и учителя специальностей, связанных с проектированием и опытные пользователи CAD программ. FreeCad обладает более простым интерфейсом, чем 3D Slicer и Blender и бОльшим количеством функций, чем SketchUP 3D. Пример окна редактора представлен на рис. 6.

После рассмотрения программ со схожей функциональностью можно сделать следующий вывод: большинство редакторов представлены настольными графическими пакетами. Среди достоинств программ можно отметить большое количество встроенных функций. Тем не менее, это в общем случае является и недостатком: большинство графических пакетов являются широкоспециализированными и работа с STL-моделями в них – лишь дополнение к общей функциональности. Это существенно

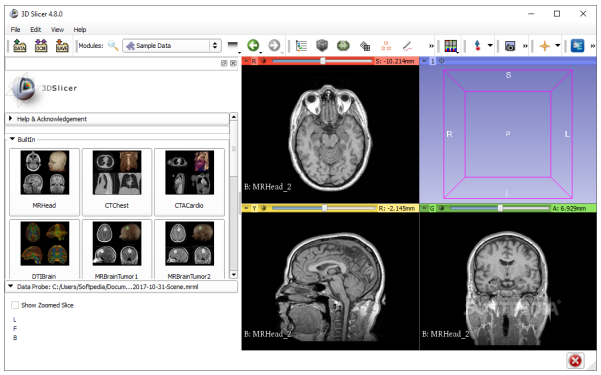

Рис. 4: Изучение результатов МРТ с помощью 3D Slicer

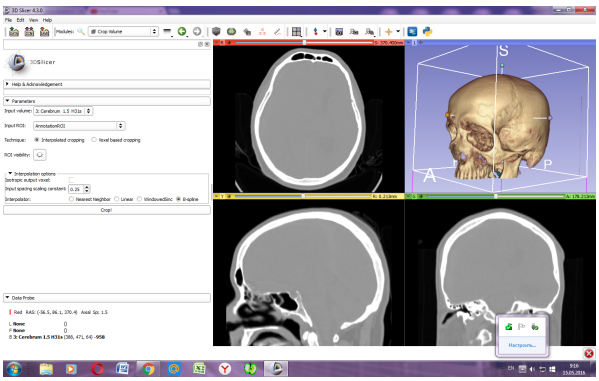

Рис. 5: Получение 3D-модели в 3D Slicer из серии снимков

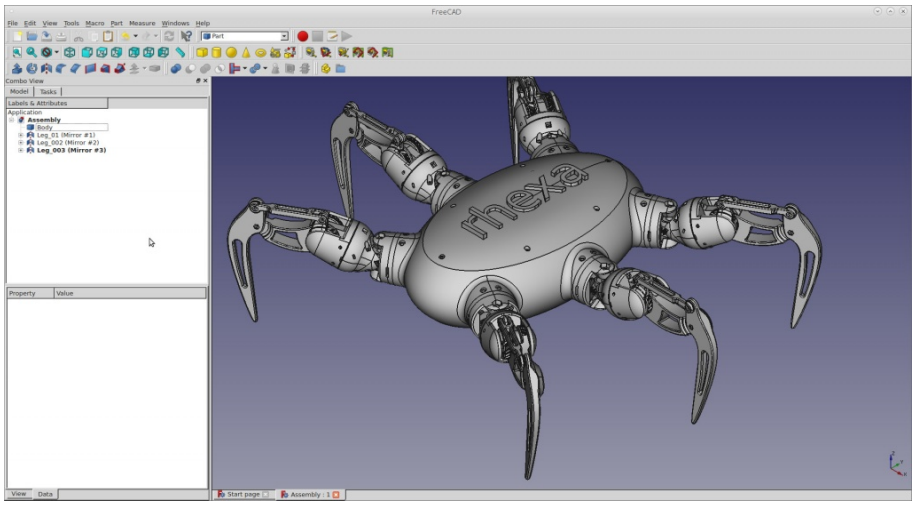

Рис. 6: Минималистичный графический интерфейс пользователя программы FreeCad

усложняет использование программного обеспечения, т.к. требует от пользователя навыков работы с конкретным графическим пакетом, несмотря на то, что пользователю необходима лишь незначительная часть функций пакета. Эту проблему следует учесть при проектировании интерфейса разрабатываемого в данной работе графического редактора.

Далее следует перейти к изучению библиотеки выполнения операций, которая будет являться ключевым элементом при редактировании моделей.

1.3 Изучение библиотеки выполнения операций

Geometry Kernel – библиотека выполнения операций над STL моделями является частью библиотеки работы с фигурами, основанными на сетках (mesh) и выполняет ключевую задачу бизнес логики будущего программного обеспечения – обработку моделей. Эта обработка может быть выполнена с использованием аппарата булевых функций, таких как:

- объединение;

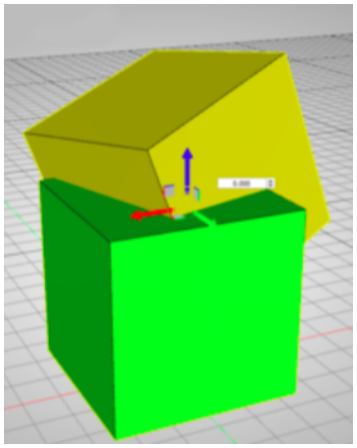

Рис. 7: Изначальное положение

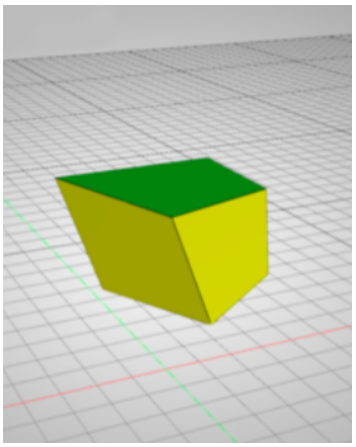

Рис. 8: Операция пересечения

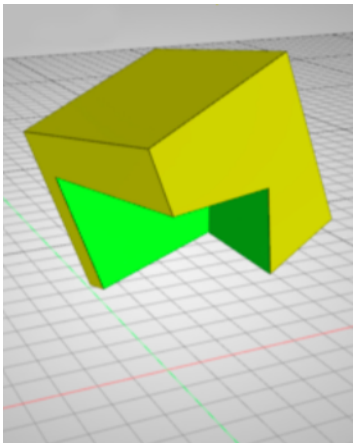

Рис. 9: Операция дополнения

- пересечение;
- дополнение (вычитание в обе стороны).

Пусть даны 2 куба, произвольно размещенные в пространстве с условием пересечения. Их взаиморасположение изображено на рис. 7. Тогда посредством булевых операций, примененных к кубам могут быть получены следующие фигуры: единое целое (визуально фигура остается точно такое же, как и на рис. 7), пересечение, изображенное на рис. 8, а также дополнение (вычитание зеленого куба из желтого), представленное на рис. 9.

В общем случае преобразуемые фигуры имеют неправильную форму и произвольную сложность. В примере выше были использованы фигуры правильной формы для простоты демонстрации, а также простоты вычисления, т.к. обработка моделей затрачивает много процессорного времени.

Библиотека написана на языке Rust, является внешней библиотекой по отношению к данной работе и будет использована в ней как «черный ящик». Также, после предварительного тестирования библиотеки, а также исходя из выполняемых ею функций можно говорить о ресурсоемких (CPU-intensive) операциях, что следует учесть в дальнейшей работе.

1.4 Формирование требований к программному обеспечению

Таким образом, после обзора предметной области, изучения аналогов, а также предварительного анализа библиотеки и выявления особенностей ее работы, необходимо сформулировать список требований к разрабатываемому программному обеспечению:

- поддержка операций, имеющихся в библиотеке обработки STL моделей;
- кроссплатформенность;
- удобство работы (асинхронность работы);

- удобный интерфейс;
- невысокие требования к программному и аппаратному обеспечению;
- поддерживаемость и возможность внесения изменений;
- учет специфики представления формата STL (объема файлов).

Функциональность разрабатываемого редактора не ограничена представленными в библиотеке функциями и новые функции могут быть получены с помощью комбинирования других функций.

Основной же целью данной работы является автоматизация процесса редактирования STL-моделей.

2 РАЗРАБОТКА ГРАФИЧЕСКОГО РЕДАКТОРА

2.1 Выбор типа приложения

2.1.1 Настольные приложения и веб-приложения

Как было установлено ранее, большинство имеющихся на рынке на данный момент приложений со схожей функциональностью, являются т.н. «настольными» (desktop) приложениями. Впрочем, в настоящее время (2018 год), идет переход в сторону т.н. распределенных (distributed) приложений. Одной из реализаций этого типа являются веб-приложения (web-application). Веб-приложение является своеобразным развитием идеи веб-сайта, когда кроме статической информации появляется и некоторый бизнес-процесс.

В рамках данной работы бизнес-процессом является автоматизация выполнения операций над STL моделями. Компонент, реализующий бизнес-процесс – черный ящик в системе (внешняя библиотека).

Чтобы выбрать итоговый тип приложения (веб-приложение или настольное приложение), необходимо произвести сравнение подходов. Для начала необходимо выделить особенности каждого типа.

Особенности настольных приложений:

- с точки зрения пользователя, настольное приложение, как правило, выглядит более органично. Под органичностью здесь имеется ввиду внешний вид приложения, созданный в соответствии со стандартами UI/UX (User Interface, User eXperience - пользовательский интерфейс и опыт взаимодействия), принятыми в операционной системе пользователя;
- отличительной чертой и главным достоинством десктопных приложений является тесное (непосредственное) взаимодействие с системой пользователя. Так, приложение может потреблять ресурсы, максимизируя эффективность и скорость работы. Ресурсами в данном случае являются память, как оперативная, так и на жестком диске, вычислительные ресурсы процессора, прямой доступ к

внешним устройствам пользователя. С одной стороны это плюс, т.к. при эффективном использовании ресурсов производительность приложения будет максимальна. С другой стороны это может быть и минусом: из-за неэффективного управления ресурсами (например, бесконтрольного их потребления) может снизиться быстродействие всей системы в целом;

- из пункта выше следует и недостаток. Так, настольные приложения имеют минимальные/рекомендуемые системные требования – требования, предъявляемые к оборудованию и системе пользователя для корректного функционирования приложения. Большинство сравнительно простых утилитарных программ не предъявляет требований к системе. Однако, некоторые категории ПО предъявляют жесткие требования. Примерами могут являться CAD программы (AutoCAD, Inventor, Siemens Solid Edge, Blender), программы для обработки видео/изображений (Photoshop) или компьютерные игры. Очевидно, набор требований к системе не только ограничивает охват аудитории, но также может вести к негативному пользовательскому опыту, который уменьшает эту аудиторию. Пример - программа, которая способна запуститься в системе пользователя, но не способна эффективно работать в ней и приводит к зависанию и сбоям все системы;

- настольное приложение по определению способно запуститься без подключения к сети интернет. Все необходимые компоненты уже установлены в системе пользователя. В некоторых случаях, впрочем, это не так. Отсюда следует еще одна особенность настольных приложений;

- в последнее время часть настольных приложений реализуется по принципу «тонкого» клиента: на компьютере пользователя устанавливается минимально необходимый набор компонентов (чаще всего – компоненты для отображения интерфейса), в то время как вся логика выполняется удаленно. В таком случае настольное приложение перестает чем-либо отличаться от веб-приложения, кроме самого клиента доступа – в одном случае это веб-браузер, а в другом – некоторый нативный компонент системы, способный скрыть логику обращений по сети. Такие приложения, впрочем, в зависимости от объема сетевого взаимодействия могут быть выделены в отдельный тип – централизованные сетевые приложения. БОльшая часть работы в таком случае выполняется на компьютере пользователя, а отдельные части по мере необходимости загружаются из сети. Примерами таких приложений являются Intellij IDEA и Eclipse. Загружаемые данные – это всевозможные плагины и дополнения к основной функциональности, доступной оффлайн;

- отдельный пункт с точки зрения разработчика – развитие и поддержка ПО. Так как десктопное приложение работает изолированно, своевременное обновление проблематично. Это является минусом при развитии приложения – большинство операционных систем, за исключением Windows 10, оставляют за пользователем право распоряжения программным обеспечением. Пользователь, в свою очередь, может по разным причинам не установить обновление своевременно, или не установить вовсе. Это приводит к сегментированности «модель-

ного ряда» версий приложения. При этом, необходимо соблюдать обратную совместимость версий, и поддерживать старые технически решения. Все это ведет к трудностям в поддержке ПО. Так, исправление критичного дефекта не может быть устранено у всех пользователей одновременно из-за пользовательского контроля над этим процессом. Все это ведет к неудобству как с точки зрения конечного пользователя, так и разработчика;

- не последним по значимости пунктом является безопасность использования. При использовании веб-приложения данные передаются по сети. Даже если эта передача происходит в зашифрованном виде по защищенным каналам связи, существует риск потери или повреждения данных. В повсеместно используемом TCP/IP стеке протоколов гарантом надежной доставки является TCP (Transmission Control Protocol, протокол управления передачей). Однако, уровень, отвечающий за защиту от внешнего доступа и воздействия изначально отсутствует и реализован отдельно (HTTPS). Несмотря на то, что HTTPS является де-фатко стандартом и приходит на смену незащищенному HTTP, модель работы, когда пользовательские данные никогда не покидают устройство, неоспоримо выглядит более безопасной. Этот пункт особенно важен при работе с т.н. sensitive data – персональными данными пользователя.

Далее необходимо рассмотреть **особенности веб-приложений**:

- главным недостатком веб-приложений является необходимость доступа к сети интернет. Даже если приложение может работать оффлайн, т.е. работать, не требуя постоянного подключения к сети, получение самого приложения требует наличия доступа к сети. В некоторых случаях предъявляются отдельные требования к скорости соединения. Примерами таких веб-сервисов являются стриминговые сервисы спортивных мероприятий, например, Twitch. Для использования сервиса с наилучшим качеством изображения требуется высокоскоростное подключение к сети интернет. Также, развитие языков программирования параллельно с развитием бразуеров ведет к отличиям в выполнении одного и того же кода в разных версиях браузеров. Примером такого поведения может служить обработка CSS-атрибутов бразуром Internet Explorer 8. Для решения подобных проблем существуют всевозможные транспиляторы (Babel), автоматически генерирующие код, соответствующий устаревшим стандартам. В настоящее время (2018 год) требования к соединению и браузеру удовлетворить проще, чем системные требования десктопных приложений;

- веб-приложение является кроссплатформенным по определению. Это одно из главных преимуществ по сравнению с настольными приложениями. Также, настольное приложение в большинстве случаев может быть использовано только на конкретных операционных системах, в число которых зачастую не входят мобильные операционные системы (iOS, Android). Большинство веб-приложений в то же время могут быть запущены в любой операционной системе, в которой присутствует веб-браузер, поскольку именно он является средой выполнения приложения. В последнее время все бОльшую популярность набирает

WebAssembly - технология выполнения клиентского кода в браузере с эффективностью, сопоставимой с выполнением нативного кода. В этом случае код, написанный на разных языках (Rust, C++) компилируется в web assembly код и выполняется совместно с остальным клиентским кодом на JavaScript. Такое разделение ведет с одной стороны к оптимизации трудоемкой части посредством описания ее на более строгих языках, чем JavaScript, а с другой - для не трудоемких задач остается незаменимой выразительность и простота JavaScript;

- недостатком является то, что взаимодействие с пользователем происходит через браузер и пользовательский опыт работы с приложением частично зависит от работы с веб-браузером. Вышеописанная технология WebAssembly решает лишь техническую задачу эффективного выполнения кода приложения. Доступ к внешним устройствам жестко лимитирован и может быть запрещен политикой безопасности бразуера. Свободного доступа к памяти также нет;

- достоинством веб-приложения, как с точки зрения пользователя, так и с точки зрения разработчика является отсутствие необходимости установки каких-либо компонентов системы. Для пользователя это является удобством. Для разработчика это отчасти решает проблему несогласованности зависимостей на компьютере пользователя (например, отсутствие DLL файла), так как подстраиваться надо лишь под используемые браузером стандарты. В это же время появляется другая проблема – эффективная сборка поставки приложения для развертывания в браузере пользователя;

- поддержка и развитие проекта не являются проблемой, т.к. приложение может быть изменено «на лету» и от пользователя может потребоваться лишь перезагрузить страницу. Таким образом, все пользователи получают одинаковый опыт использования приложения, а критические дефекты могут быть быстро исправлены. С точки зрения разработки возможно полноценное внедрение CI/CD (Continuous Integration – непрерывная интеграция, т.е. интеграция любого зафиксированного изменения (commit) в систему и проверка его работоспособности. Continuous Delivery – процесс непрерывного обновления используемой конечным пользователем версии продукта при каждом изменении продукта), которое повышает как скорость разработки, так и скорость доставки продукта конечному клиенту;

- возможен сбор детализированной информации об использовании приложения пользователями и на основе этих данных возможны дальнейшие изменения для улучшения пользовательского опыта. Настольные приложения точно так же могут собирать подобную статистику, однако отправка статистики на обработку затруднена, ввиду возможной изоляции приложения;

- поскольку есть некоторая общая среда выполнения бизнес-логики, при использовании правильной архитектуры возможно как вертикальное, так и горизонтальное масштабирование. А вместе с этим становится возможна и совместная работа пользователей над общим процессом.

На основании описанных выше особенностей функционирования и устройства веб-приложений и настольных приложений, можно сделать вывод о схожей трудоемкости разработки, которая лишь проявляется в разных местах и на разных этапах разработки. Однозначный вывод на данный момент сделать нельзя.

2.1.2 Сравнение графической инфраструктуры

Поскольку разрабатываемое приложение – графический редактор, то значительная часть разработки будет происходить с использованием графического API и библиотек. Необходимо, чтобы инфраструктура языка поддерживала в полной мере весь этот аппарат. Поскольку библиотека обработки STL моделей разработана на Rust, настольное приложение также должно быть написано на Rust для достижения максимальной производительности и эффективности использования библиотеки.

В качестве графического API был выбран OpenGL (Open Graphics Library) – кросс-платформенное API для рендеринга векторной графики в 2D и 3D. API берет свое начало в 1992 году и в данный момент разрабатывается некоммерческим консорциумом Khronos Group. OpenGL является де-факто стандартом при работе с графикой и используется при разработке CAD-программ, визуализации данных, а также в видео играх и работе с дополненной реальностью (AR, Augmented Reality) и виртуальной реальностью (VR, Virtual Reality). API изначально разработано на языке C и на данный момент имеет адаптеры для большинства языков и платформ. Работа с OpenGL возможна как из Rust (настольное приложение), так и из JavaScript (OpenGL ES в случае веб-приложения).

В Rust имеются 2 основных библиотеки работы с OpenGL: Glium и GFX. Далее следует краткий обзор каждого решения и последующее их сравнение.

Glium (https://github.com/glium/glium), как следует из описания проекта, является промежуточным слоем между разрабатываемым приложением и OpenGL. Glium делает ставку на безопасность использования, скрывая небезопасные части OpenGL API. Производится также некоторая оптимизация вызовов функций, что, по заверению авторов, дает возможность использовать OpenGL в соответствии с современными практиками разработки. В Glium заявлена поддержка шейдерной графики. Поддержка стандарта OpenGL ES также заявлена, но не гарантируется.

Среди ограничений приводится производительность, которая в теории должна быть меньше, чем вызов функций OpenGL напрямую (raw call). Немаловажным фактом является использование компилятора по максимуму. Переполнения буфера (stack overflow), внутренние ошибки компилятора и время компиляции около 1 часа завялены, как нормальное поведение при использовании данной библиотеки.

Самым существенным фактом, на основании которого можно сделать вывод о библиотеке является то, что с августа 2016 года разработка Glium оригинальным автором приостановлена. Автором обозначены следующие причины приостановки работы:

- ситуация с ошибками в драйверах OpenGL является «катастрофической». При старте разработки предполагалось обезопасить пользователя от этих проблем, но в определенный момент их количество стало чрезвычайно велико и библиотека перестала быть безопасной;

- пересечение с функциональностью Vulkan (прим. Vulkan – высокопроизводительное низкоуровневое API для обработки 3D графики, является развитием OpenGL. Разработчик – Khronos Group). Среди ранних целей была обозначена поддержка низкоуровневых операций, по аналогии с Vulkan, в то время как OpenGL является более высокоуровневым API. Впоследствии эта функциональность оказалась никому не нужна, но отказаться от нее было уже поздно;
- множество незавершенных областей API. Такие базовые блоки OpenGL, как текстуры и буферы так и не получили завершенную версию API;
- проблемы с привлечением контрибьюторов (соавторов). Так, через несколько месяцев после объявления о привлечении контрибьюторов откликов не последовало. Проблемой стало излишнее усложнение внутреннего устройства библиотеки, разобраться в которой стало слишком сложно. Сообщество могло помочь лишь с документированием имеющейся функциональности, но не развитием проекта.

Дальнейшее развитие предполагается автором за счет сообщества. На момент написания этой работы библиотека имеет версию v0.21.0 и 1578 звезд на GitHub, которые свидетельствуют о ее невысокой популярности. В списке контрибьюторов (развивающее библиотеку сообщество) значатся 94 человека. Glium распространяется под лицензией Apache-2.0.

GFX (https://github.com/gfx-rs/gfx) является низкоуровневой кроссплатформенной библиотекой, абстрагирующей графику. Состоит из трех компонентов:

- gfx-hal (Hardware abstraction layer) – небезопасное Vulkan API, перенаправляющее вызовы графическому бэкенду;
- gfx-backend-* – набор графических бэкендов. Включает в себя следующие платформы: Vulkan, DirectX 12, Metal, OpenGL 2.1+ / ES2+;
- gfx-warden – фреймворк для интеграционного тестирования бэкендов.

Главный компонент GFX, HAL (Hardware abstarction Layer), является тонким низкоуровневым слоем абстракции. Главная задача HAL – обеспечение кроссплатформенности за счет транслирования вызовов графическому бэкенду. API этого слоя имеет много общего с Vulkan API. Схематично слой представлен на рис. 10

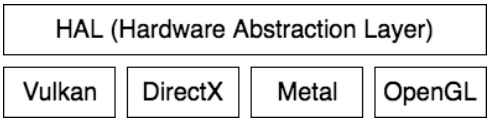

Рис. 10: Устройство HAL

GFX имеет 1724 звезды на GitHub, что также говорит о сравнительно невысокой популярности и 142 контрибьютора. Распространяется под лицензией Apache-2.0.

При детальном сравнении Glium и GFX можно выделить следующие пункты:

- Glium и GFX были выпущены в 2014 году с разницей в несколько месяцев. Несмотря на общую цель, внутреннее устройство библиотек разительно отличается;
- обе библиотеки поддерживают OpenGL. Поддержка OpenGL ES3 заявлена и не гарантируется у обоих библиотек;
- GFX предоставляет низкоуровневый API и минимальными накладными расходами, т.н. «zero-verhead». Glium больше фокусируется на безопасности, жертвуя производительностью: присутствует как CPU-overhead, так и GPU-overhead;
- «кривая обучения» у GFX выше, чем у Glium;
- Glium стремится к выявлению 100% проблем времени выполнения (runtime). У GFX заявлено лишь выявление «большей части» проблем;
- функциональность графических API реализована на примерно одинаково высоком уровне. Однако, поддержка библиотекой GFX тесселяции и геометрических шейдеров все еще на стадии разработки/тестирования.

На основании приведенных выше сравнений можно сделать вывод о еще не сформированной полноценной графической инфраструктуре Rust. Использование GFX тесно связано с использованием Vulkan, так как наибольшую эффективность библиотека имеет в сочетании именно с этим целевым бэкендом. Использование же Glium, который изначально ориентирован на OpenGL может привести к проблемам в самый неподходящий момент, когда окажется, что необходимая функциональность отсутствует, а перевод имеющейся функциональности на GFX слишком дорог.

Далее следует рассмотреть графическую инфраструктуру, которая может быть использована в веб-приложении.

WebGL (Web Graphics Library, подробное рассмотрение приведено в главе) – JavaScript API для рендеринга 2D и 3D графики внутри любого совместимого браузуера без использования каких-либо дополнительных плагинов и расширений. Это возможно благодаря тесной интеграции стандарта с браузерами. WebGL, в свою очередь, согласуется со стандартом OpenGL ES (OpenGl for Embedded Systems) и может быть использован прямо внутри элемента <canvas>, предоставляемого HTML5. Благодаря этому, приложения могут быть запушены практически на любом устройстве, т.к. WebGL поддерживается всеми популярными веб-браузерами, как настольными, так и мобильными: Google Chrome, Firefox, Opera, Safari, Vivaldi.

Строгое соответствие стандартам OpenGL ES поддерживается консорциумом, а «экосистема» развивается и увеличивается благодаря многочисленным сообществам фреймворков, основанных на WebGL. Среди них наиболее популярные – Three.js, BabylonJS – высокоуровневые абстракции, предоставляющие большое количество функций «из коробки», которые могут быть использованы и в рамках данной работы. Игровые «движки» Unity и Unreal Engine полноценно поддерживают WebGL, а спектр технологий в 2015 году пополнился вирутальной реальностью (VR) благодаря

выходу A-Frame. Этот веб-фреймворк с открытым исходным кодом, разрабатываемый Mozilla и сообществом и базирующийся на Three.js, позволяет описывать сцены виртуальной реальности с помощью HTML.

Краткая статистика популярности библиотек и фреймворков над WebGL (в качестве сравнительных признаков используется количество контрибьюторов и звезды GitHub, отражающие количество пользователей, интересующихся проектом):

- Библиотека BabylonJS имеет около 7 тысяч звезд на GitHub и сообщество из 185 человек, развивающих проект. Лицензия – Apache-2.0;
- Фреймворк A-Frame имеет более 8 тысяч звезд и более 240 контрибьюторов. Лицензия – MIT;
- Библиотека Three.js имеет более 42 тысяч звезд на GitHub, а количество контрибьюторов приближается к 1000 человек. Распространяется под лицензией MIT.

Первый релиз WebGL состоялся в 2011 году и на данный момент развитие WebGL в среде веб-разработки ушло далеко вперед по сравнению с развиваемыми одним лишь сообществом библиотеками Glium и GFX.

Таким образом, в качестве типа разрабатываемого приложения был выбран веб-сервис. Помимо лучшей по сравнению с Rust графической инфраструктуры, предоставляемой WebGL-совместимым веб-браузером, разработка компонентов веб-сервиса может производиться независимо друг от друга. Развитие инфраструктуры в среде веб-программирования идет несравнимо быстрее развития системного программирования, что, благодаря внедрению новых технологий существенно повышает скорость разработки. Итоговое решение должно стать кросспалтформенным, поддерживаемым и масштабируемым.

2.2 Веб-сервис

Веб-сервисом является программное обеспечение, с клиент-серверной архитектурой, доступное через по сети интернет и использующее стандартизированную систему сообщений, например в формате JSON или XML. Клиент-серверная архитектура предполагает наличие двух компонентов – клиента (фронтенд) и сервера (бэкенд). С точки зрения разработки это означает необходимость разрабатывать две программы – пользовательское приложение и серверную часть. Порядок разработки во многом зависит от конкретного проекта. Далее будут рассмотрены два подхода к разработке систем, ориентированных на работу с конечным пользователем и любой из подходов может быть применен не только при разработке веб-сервиса.

Существует множество подходов (методологий) к проектированию и разработке программного обеспечения. Одним из подходов является разработка «сверху-вниз» и «снизу-вверх».

В более традиционном подходе **«снизу-вверх»** проектирование начинается с наиболее низкого компонента системы. Как правило это база данных. Исходя из поставленных задач проектируется набор сущностей базы данных и отношений между

ними. После этого начинается разработка бэкенда, т.е. сущности базы данных проецируются на бэкенд и создается операционный слой, выполняющий манипуляции над этими сущностями в соответствии с поставленными ранее задачами. Когда разработка бэкенда завершена приступают к разработке фронтенда – клиентского приложения. Разрабатывается как протокол взаимодействия, так и графический интерфейс пользователя. Очевидно, что при таком подходе графический интерфейс становится своеобразным дополнением и уходит на второй план, поскольку интерфейс проектируется с учетом ограничений, которые накладывает разработанный бэкенд. А поскольку бэкенд с основной бизнес-логикой уже сформирован, его корректировка становится сложной и невыгодной: исправление и развитие ведет к усложнению всей архитектуры.

В итоге заказчик получает приложение, формально удовлетворяющее бизнес-требованиям. Конечный пользователь при этом, как правило, в расчет не берется. Получившееся приложение выполняет задачи, но эффективность использования чрезвычайно мала. Работа с таким приложением точки зрения пользователей неудобна, а пользовательский опыт (UX, User eXperience) в основном негативен.

Подход «снизу-вверх» дешевле с точки зрения разработки, т.к. бизнес-требования будут однозначно удовлетворены, а UI/UX не так важен. Такой подход характерен в корпоративной разработке, т.н. enterprise, разработке для государственного сектора а также крупных компаний, не специализирующихся на IT.

Подход **«снизу-вверх»** наоборот, в первую очередь предполагает проектирование пользовательской части приложения. Для этого приглашается эксперт в доменной области, представляющий конечного пользователя.

В это же время проектируется и утверждается протокол взаимодействия клиентской и серверной частей приложения. Далее разрабатывается внутреннее устройство клиентской части, а в качестве бэкенда используется т.н. «заглушка» или mock (ложный объект, лишь внешне имитирующий поведение реального объекта. Принимает входные параметры, но никак их не обрабатывает. Настроен лишь на выдачу заранее определенных результатов) на время, пока бэкенд формирует базу данных и бизнес-логику. В итоге получается процесс разработки, при котором фронтенд и бэкенд могу разрабатываться параллельно, поскольку между ними есть договоренность – протокол. Оба компонента системы должны его соблюдать. При этом фронтенд должен разрабатываться с учетом двух ограничений:

1. Внешний вид пользовательской части приложения изначально определен и согласован с экспертом;

2. Протокол взаимодействия с бэкендом также определен.

В определенный момент использование бэкендом «заглушек» меняется на использование реальных данных. Сложность разработки при подходе «сверху-вниз» возрастает, поскольку учитывается не только оптимальность представления объектов доменной области с точки зрения разработчика бэкенда и базы данных, но и с учетом потребностей фронтенда, который учитывает пользовательский опыт. В итоге получается приложение, удовлетворяющее не только интересам бизнеса, но и потребностям конечных пользователей.

2.3 Разработка клиентской части (фронтенд)

Разработку приложения по методологии «сверху-вниз» начинают с разработки клиентского приложения. Поскольку в рамках данной работы разрабатывается прототип графического редктора, основной упор сделан на реализации функциональности обработки моделей, имеющейся в редакторах, рассмотренных при обзоре программ-аналогов. Однако, при дальнейшем развитии проекта, в качестве консультанта может быть приглашен эксперт доменной области (протезирование с использованием моделей для 3D-печати).

2.3.1 Проектирование пользовательского интерфейса

Исходя из вышеприведенных рассуждений о методологии разработки, необходимо сначала спроектировать пользовательский интерфейс. Поскольку в рамках данной работы нет возможности привлечь эксперта доменной области (протезирование с использованием моделей для 3D печати), необходимо ориентироваться на аналоги.

БОльшая часть аналогов представлена настольными приложениями. На рис 3 и 2 представлены аналогичные программы с той лишь разницей, что это настольные приложения. Характерными чертами настольных приложений практически любой платформы являются наличие строки меню, содержащей привычные элементы – Файл, Правка, Вид, Справка и т.д. и строки состояния, отражающей текущее состояние программы. Характерной чертой графических редакторов является наличие бокового меню справа или слева от рабочей области, с помощью элементов которого, как правило, осуществляются манипуляции над объектами рабочей области. Рабочая область при этом должна занимать не менее 80% всего рабочего пространства программа. Таким образом, целевой интерфейс может быть условно разбит на несколько частей. Подобное разбиение представлено на рис 11.

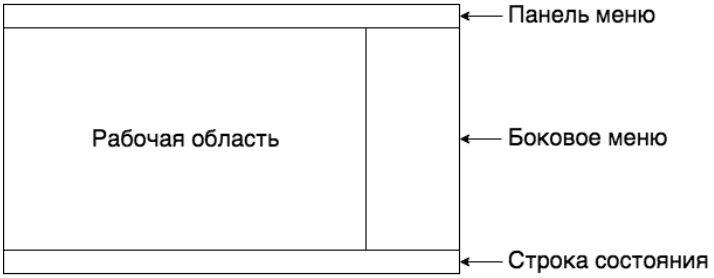

Рис. 11: Компоненты пользовательского интерфейса

Поскольку веб-приложения разрабатываются не под одну конкретную систему, стандарты разработки графических интерфейсов веб-приложений отсутствуют. Есть общие идеи и принципы, которые подходят и к настольным и к веб-приложениям. Однако, т.к. веб-приложения не обременены элементами UI операционных систем, таких, как строка состояния и строка меню, которые не могут быть отключены (и должны присутствовать в каждом приложении для формирования единого внешнего вида системы), эти элементы при разработке UI веб-приложения игнорируются в большинстве случаев.

В рамках данной работы эти части *необходимо* обеспечить в разрабатываемой программе, чтобы обеспечить комфорт и привычность использования, а также ин-

туитивную понятность интерфейса на уровне настольного приложения.

Разработка клиентских приложений в настоящее время (2018 год) имеет множество особенностей и не имеет практически ничего общего с разработкой веб-сайтов несколько лет назад, хотя внешний вид веб-сайтов и веб-приложений в значительной мере сходится. Так, техники применяемые в настоящее время для разработки веб-приложений могут быть использованы для создания веб-сайтов. Обратное, однако, неверно, даже если речь идет лишь об интерфейсе – техники построения веб-приложений в прошлом не подходят для современной разработки.

Все это (быстрая смена технологий) происходит благодаря быстрому развитию сети интернет и переходу все большего числа компаний к работе в сети. Развитие и рост популярности мобильных устройств (смартфонов и планшетных компьютеров) также способствует не только развитию экосистем самих мобильных устройств (мобильных приложений), но и развитию веб-приложений, поскольку все подобные устройства оснащены выходом в сеть интернет. Для все бОльшего количества пользователей мобильное устройство становится главным способом использования сети интернет и заменяет некогда повсеместное использование компьютеров.

Это накладывает на веб-приложения существенные ограничения – мобильные устройства заметно уступают в производительности компьютерам, а размер экрана делает интерфейсы, изначально разработанные для использования с помощью компьютера, непригодными при использовании с мобильного устройства. Сам способ соединения с сетью также меняется – от надежного проводного (оптоволоконного) соединения идет переход к соединению посредством мобильного интернета. Этот процесс перехода еще не завершился и качество связи и область покрытия существенно разняться от центров городов к окраинам. Современная беспроводная связь посредством стандартов 4G (LTE) хоть и уступает в скорости пиковым скоростям проводного соединения, для веб-приложений это не является существенным, т.к. размер приложения не превышает килобайты данных – таком случае разница между скоростью в несколько мегабит и несколько сот мегабит не играет роли. Однако, такая скорость доступна далеко не везде.

При использовании более распространенных сетей 3-го поколения, 3G, с точки зрения пользователя заметны отличия. Загрузка данных занимает в таком случае уже не «мгновение», а секунду или несколько секунд. Таким образом, велика вероятность, что пользователь не станет ждать 5 секунд загрузки приложения, а перейдет на соседний ресурс. Чтобы учесть и побороть все эти проблемы и ограничения, была создана новая архитектура с использованием фреймворков.

В последнее время подобных фреймворков, и библиотек, облегчающих фронтенд-разработку появляется все большего. Фреймворк можно рассматривать, как некоторый каркас приложения, который спроектирован с учетом особенностей его функционирования и задач, решение которых этот фреймворк призван облегчить. В него встраивается бизнес логика и получается готовое приложение. При этом время на разработку системы сокращается, т.к. внутреннее устройство отлажено и функционирует независимо. Фреймворк отличается от библиотеки: библиотека – набор компонентов, каждый из которых может быть использован независимо (использование публично доступной функции A не требует использования публичной функции B) и

использован при, например, реализации бизнес-логики. Фреймворк же в свою очередь является конструкцией, в которую встраиваются компоненты (бизнес-логика). Он может работать самостоятельно, но для корректной работы требуется соблюдать набор правил. Другими словами, фреймворк – слой абстракции.

Современные фреймворки, такие как React от Facebook, Angular от Google, и разрабатываемый сообществом Vue доминируют на рынке веб-приложений. Основная же задача, которую они решают – синхронизация внутреннего состояния приложения (state). В традиционном подходе каждое взаимодействие пользователя с DOM (Document Object Model, модель документа, которым является веб-страница) должно быть зафиксировано и обработано специальным обработчиком (handler), который должен соответствующим образом модифицировать внутреннее состояние приложения и его визуально представление.

Примером может служить список постов (сообщений). При клике по какому-либо посту, сообщение этого поста показывается полностью: изначально состояние приложения содержит список элементов (постов), активного (выбранного) элемента нет. Приложению необходимо лишь отобразить список доступных элементов. Когда пользователь кликом по элементу выбирает активный элемент, обработчик в приложении должен определить, по какому элементу кликнул пользователь. Далее, приложение должно отметить пост, по которому кликнул пользователь как активный. На этом этапе происходит изменение DOM – необходимо по-другому отобразить выбранный элемент. Далее приложение должно запомнить, какой пост выбран во внутреннем состоянии и достать полную информацию этого поста и также отобразить. Это еще одно изменение визуального представления. Когда пользователь снимает выбор, требуется точно такая же синхронизация внутреннего состояния и интерфейса. Все это требует большого количества повторяющегося кода. И чтобы избавиться от этих манипуляций в крупных проектах, где внутреннее состояние приложения может быть чрезвычайно большим, а синхронизация действий весьма нетривиальна (например, Facebook), были разработаны фреймворки.

Самый популярный фреймворк и де-факто современный стандарт фронтенд разработки – React. «Под капотом» React манипулирует сразу двумя DOM'ами (DOM и Shadow DOM) и, таким образом, удается избегать перезагрузки всей страницы – обновляется минимально необходимое для визуального обновления число React-компонентов. Благодаря этому достигается большая эффективность работы, а разработчику нет необходимости решать проблему синхронизации внутреннего состояния самостоятельно. [3]

Листинг 3: Простейшая «dumb» компонента, не хранящая состояние

```
const DumbComponent = () => {
  return (
    <div>
      <Button kind="primary" onClick={() => console.log("clicked!")}>
        <Intrails/>
      </Button>
    </div>
  );
};
```

«Плата» за это – необходимость строго следовать правилам фреймворка. Так, при разработке приложения на React используется компонентный подход, выделяющий части интерфейса отдельно от остальных. Был разработан отдельный формат файлов для представления компонентов, jsx, являющийся сочетанием HTML и JavaScript. Пример React-компонента – на Листинге 3. Доступ к внутреннему состоянию приложения также осуществляется согласно набору правил. Впоследствии был разработан отдельный фреймворк, реализующий конечный автомат состояний, Redux, но вскоре был признан сильно усложняющим систему и от него было решено отказаться.

Все это отражает быстрое развитие технологий и быструю смену одних технологий другими. Параллельно с развитием фреймворков развивается и язык JavaScript. Современный стандарт, ES6, не в полной мере поддерживается всеми браузерами и был разработан ряд решений по адаптации современного JavaScript кода и фреймворков под бразуеры, не успевающие обновляться. Таким образом, появилась потребность в транспиляции – компиляции JavaScript ES6 в более ранние версии JavaScript.

Объем проектов рос и сборка приложения (bundle) стала занимать слишком большой объем. Потребовалось сжатие и появилась минификация – процесс сжатия объема кода. Простейший пример минификации – переименование константы в однобуквенный вид, например, const isUserAddressValid = true; в const x = true;.

Компонентный подход потребовал сборки приложений (т.к. клиенту должны отправиться всего несколько статических файлов) подобно настольным приложениям и появился webpack – менеджер сборки итогового бандла, выполняющий сборку компонентов в финальный вид, транспиляцию, минификацию, аглификацию (uglify) и прочие задания.

Подобные, подходы стали возможны благодаря появлению Node.js – новой платформы выполнения JavaScript, основанной на V8 – движке бразуера Chrome от Google, зарекомендовавшего себя крайне положительно в качестве среды выполнения Javascript. Node использует событийно-ориентированную архитектуру и асинхронную неблокирующую модель I/O операций. Скорое появление менеджера зависимостей, npm (Node Package Manager), дало огромный толчок к развитию веб-технологий и сделало экосистему npm крупнейшей библиотекой open-source проектов в мире.

Вышеописанный подход к разработке с использованием любого из фреймворков требует большого количества дополнительной работы по обеспечению всей инфраструктуры проекта. В данной работе бОльшая часть проекта – отображение моделей в рабочей области интерфейса. Коммуникация между фронтендом и бэкендом не затрагивает внутреннего состояния приложения. Количество функций, реализуемых на фронетнде и доступных пользователю невелико и, следовательно, синхронизация внутреннего состояния приложения и интерфейса не является сложной задачей. Дополнительно к этому, транспиляция, минификация и сборка затрудняют отладку кода. Внешний вид приложения по большей части статический – меняется в основном внутреннее состояние рабочей области и это состояние может быть эффективно синхронизировано с рабочей областью самостоятельно.

Таким образом, графический интерфейс пользователя будет реализован статиче-

ски. Описание интерфейса представляет из себя один файл, в который в качестве скриптов будет загружаться клиентская логика, реализованная на JavaScript и таблицы стилей, реализованные с использованием SASS. HTML-представление графического интерфейса частично представлено на Листинге 6.

Отдельного упоминания заслуживает **SASS**. Несмотря на бурное развитие технологий вокруг JavaScript, веб-браузеры по-прежнему используют классические таблицы стилей CSS (Cascade Style Sheet). Однако сам формат css устарел и разработка на нем неэффективна. На смену CSS с форматом css пришел SASS с форматом scss. Это формат, допускающий использование именованных констант для описания стилей атрибутов. Допускаются т.н миксины (mixin) – компоненты, с множественным наследованием стилей. Процесс предобработки (препроцессинг) SASS также может быть объединен с созданием бандла, но поскольку было принято решение не использовать webpack, перевод scss в css может быть осуществлен вручную с помощью компилятора Dart-Sass, написанного на языке Dart. Благодаря IDE Dart-Sass может следить за файлом и при каждом его изменении автоматически корректировать соответствующим образом выходной файл style.css.

Листинг 4: Входной scss файл

```
$color-light-blue: #1a6bc1;
$font-courier: "Courier New", Courier, monospace;
$border-menu-item: 1px solid $color-light-blue;

@mixin centered-text {
  display: flex;
  justify-content: center;
  align-items: center;
}

.menu {
  width: 100%;
  display: flex;
  background-color: $color-dark-black;

  #logo {
    height: 22px;
    color: $color-light-blue;
    @include centered-text;
  }
}
```

Листинг 5: Выходной css файл

```
body .menu #logo {
  height: 22px;
  color: #1a6bc1;
  display: flex;
```

```
  justify-content: center;
  align-items: center;
}
```

На Листинге 4 представлен фрагмент SASS файла в формате scss. На Листинге 5 – получаемый в итоге css файл, который будет отправлен в браузер пользователя.

Далее следует перейти к рассмотрению пунктов меню и кнопок доступных пользователю. Чтобы отделить представление интерфейса от логики, которая необходима для взаимодействия с моделями, нужно создать отдельные компоненты системы, реализующие только способ взаимодействия пользователя с интерфейсом. Если соблюдать это правило, то при изменениях графического интерфейса нужно будет лишь повторно использовать метод, вызывающий добавление или сохранение моделей (menu.addModel() и menu.saveModel()).

Листинг 6: Структура главной страницы

```
<!doctype html>
<html>
  <head>
      <meta charset="utf-8">
      <link rel="stylesheet" href="css/style.css">
  </head>
  <body>
    <script src="js/interface/menu.js"></script>
    <header>
      <div class="menu">
        <div class="dropdown">
          <div id="menu-file"
               onclick="menu.toggleFile()" class="dropbtn"> Файл</div>
          <div class="dropdown-content" id="file-dropdown">
            <div class="dropdown-content--item"
                 onClick="menu.addModel()"> Добавить модель</div>
            <div class="dropdown-content--item"
                 onClick="menu.saveModel()"> Сохранить модель</div>
          </div>
        </div>
      </div>
    </header>
    <div class="editor">
      <div class="view">
        <canvas id="glCanvas"></canvas>
      </div>
    </div>
    <script src="js/main.js"></script>
```

```
  </body>
</html>
```

Строка меню частично представлена в тэге header на Листинге 6. Поскольку фреймворки при проектировании интерфейса не были использованы, синхронизация отдельных частей меню, таких, как список выпадающих кнопок, должна быть реализована самостоятельно. Эта реализация представлена на рис. 12.

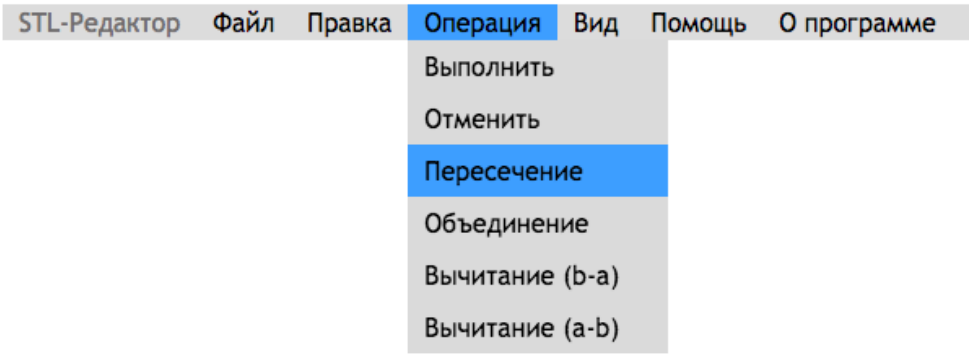

Рис. 12: Выпадающее меню, реализованное по аналогии с классическими «нативными» приложениями

На рис. 12 представлен список операций доступный пользователю и производимый непосредственно над моделями. Другие опрерации представлены в других пунктах.

Пункт меню «**Файл**» предоставляет команды для добавления и сохранения (скачивания) моделей:

- добавить модель;
- сохранить модель.

Пункт меню «**Правка**» содержит команды для манупуляции выбранной моделью, а также ее редактиирования без участия других моделей:

- зафиксировать активную модель;
- удалить активную модель;
- разделить перпендикулярно OX;
- разделить перпендикулярно OY;
- разделить перпендикулярно OZ.

Разделение моделей плоскостями перпендикулярно осям координат не реализовано в стандартной библиотеке и является видоизмененной операцией дополнения, где один из объектов – модель, а другой объект – симплекс, реализованный тетраэдром, «накрывающий» всю сцену и невидимый при этом пользователю.

Пункт меню «**Операция**» предоставляет доступ к операциям из библиотеки Geometry Kernel, а также непосредственно команды для их проведения или отмены:

- выполнить;
- отменить;
- пересечение;
- объединение;
- вычитание (b-a);
- вычитание (a-b).

Пункт меню «**Вид**» позволяет кооректировать внешний вид сцены, скрывая и отображая некоторые ее элементы:

- перезагрузить сцену;
- отобразить/скрыть все модели;
- показать/скрыть оси координат;
- показать/скрыть сетку;
- показать/скрыть лог.

Пункты меню «**Справка**» и «**О программе**» предоставляют информацию справочного характера.

Боковое меню, т.н. «сайдбар» реализован с расчетом на следующий порядок работы, выделенный в качестве основного при работе с приложением: добавление модели в сцену -> выбор активной модели -> ее редактирование -> фиксация как результат редактирования (фиксация является приемом оптимизации производительности, т.к. манипуляция над объектом до фиксации «вирутальна» и реальные координаты модели не пересчитываются при каждом изменении, т.к. иначе это привело бы к потере производительности) -> выбор операции и ее применение. Компоненты бокового меню, реализующие этот порядок выполнения представлен на рис. 13.

Среди набора функций сайдбара следует отметить такие функции, как выбор файла и добавление модели, выбор активной модели, к которой будут применены «Настройки активной модели»: выбор оси координат, а также значений масштаба (STL формат не предполагает наличие масштаба в информации о модели) и угла поворота. Любая модель может также быть скрыта – для этого используется чекбокс «Видимость» (при этом все модели, сетка координат, оси координат могут быть скрыты по отдельности с помощью соответствующих функций верхнего меню из раздела «Вид»). Кнопка «Зафиксировать положение» выполняет перерасчет координат модели, после чего модель может быть экспортирована в виде STL файла или передана на бэкенд для проведения операции. Самый нижний блок отображает выбранные для операции модели, которые вместе с идентификатором операции будут отправлены на бэкенд при нажатии кнопки «Выполнить операцию».

В заключение необходимо продумать реализацию и функциональность строки состояния приложения. Т.к. приложение по своей сути является прототипом, логично выводить в строку состояния информацию о функционировании приложения и

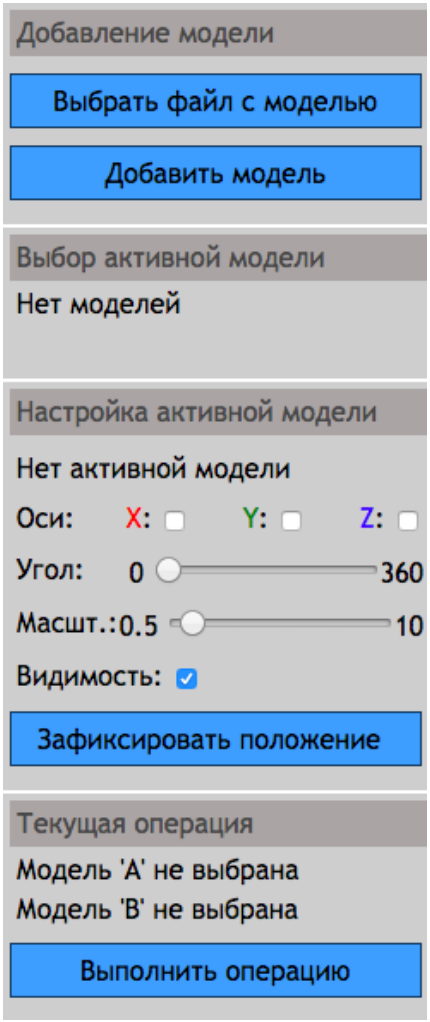

Рис. 13: Боковое меню с упором на основной поток управления, идущий в направлении сверху вниз

лог действий пользователя, а не просто строку формата «OK» или «ERROR». Вся необходимая для отладки приложения информация отправляется в системный лог бразуера (console.log), а информация, имеющая пригодный для чтения вид выводится в консоль приложения. В итоге соблюдается баланс между полнотой доступной (и необходимой) пользователю информации и возможностью в случае нештатной ситуации отследить не только действия пользователя, что привели к ней, но и собрать техническую информацию о случившейся ошибке из консоли разработчика.

Собрав описанные выше компоненты интерфейса (строка меню, строка состояния, боковое меню и рабочая область) воедино, получаем прототип интерфейса пользователя, привычный и интуитивно понятный. Скриншот интерфейса представлен на рис. 14.

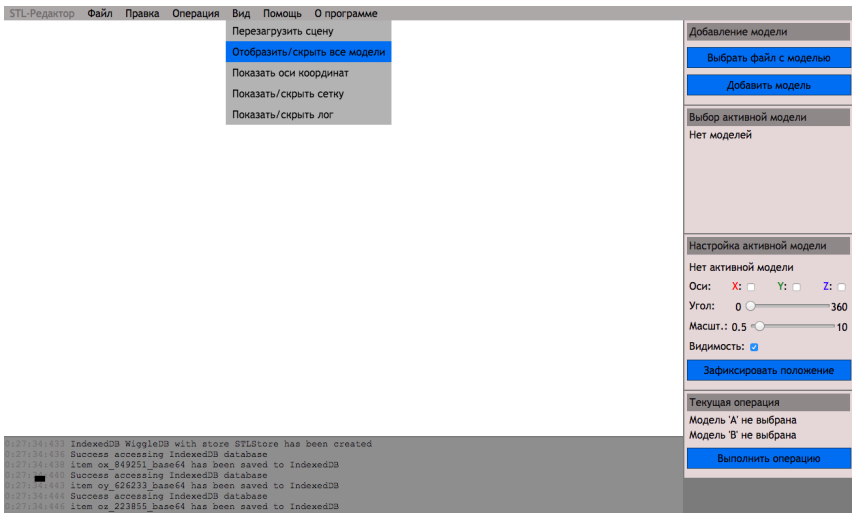

Рис. 14: Скриншот интерфейса без рабочей области

2.3.2 Графическая библиотека WebGL

БОльшую часть интерфейса пользователя составляет рабочая область редактора. На данный момент там расположен HTML5 элемент <canvas>. Этот элемент несет в себе контекст WebGL и именно с ним будет происходить бОльшая часть работы. [8]

Прежде чем приступить к разработке, необходимо узнать, как работает WebGL и шейдерная графика в целом. От этого во многом зависит эффективность работы будущего приложения.

WebGL является реализацией стандарта OpenGL ES (OpenGL for Embedded Systems, GLES, OpenGL для встраиваемых систем). GLES является в некотором роде упрощенной версией или подмножеством OpenGL. Приложения, написанные на GLES в большинстве своем работают с OpenGL, но не наоборот. Однако, не следует считать GLES простым в использовании из-за того, что это лишь подмножество. [5]

На практике «упрощение» означает прямо противоположное. GLES разрабатывался с учетом мобильных платформ и их низкой производительности и из-за этого авторам пришлось отказаться от некоторых требовательных к ресурсам частей API, оставив возможность работать с ними, используя более низкий уровень. В OpenGL многие вещи можно сделать разными способами. Эта многозначность убрана в WebGL. В настоящее время работа с текущей версией WebGL требует применения шейдеров, вершинных буферов и прочих низкоуровневых компонентов, которые можно использовать и в OpenGL, но там существуют их высокоуровневые аналоги. Высокоуровневая семантика glBegin-glEnd, которая в «полной версии» OpenGL позволяет оборачивать геометрические примитивы – отсутствует, т.к. рендеринг полигонов также не поддерживается.

Тем не менее, есть 2 вещи, которые реализованы в GLES, но не реализованы в «основной версии»:

1. Уменьшено количество функций API, непригодных на мобильных платформах. Пример – рендеринг полигонов и списки отображения. Добавлены функции специально для мобильных платформ;

2. GLES имеет интерфейс доступа к менеджеру окна, в котором отображается графика – EGS. Это своеобразный аналог GLUT с коллбеками displayFunc() и reshapeFunc() и функцией init(), которые отсутствуют в OpenGL.

Тем не менее, WebGL все также работает на GPU и все «упрощения» никак не отражаются на производительности.

Изучение WebGL следует начать с исследования конвейера обработки графики (rendering pipeline). Схема работы конвейера от получения трех вершин на вход до набора пикселей на экране на выходе представлена на рис. 15. Доступными разработчику элементами конвейера являются лишь вершинный и фрагментный шейдеры.

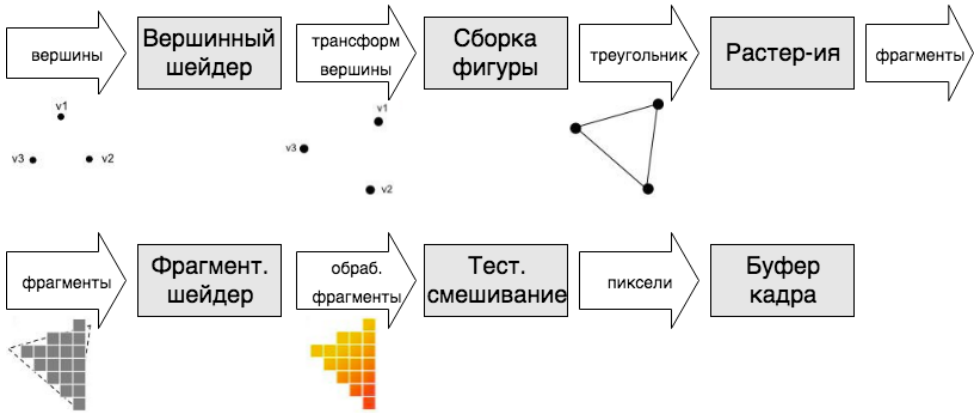

Рис. 15: Графический конвейер WebGL

Шейдер (shader) – программа, исполняемая на GPU и изначально специализирующаяся на шейдинге (shading) – вычислении и применении световых и цветовых эффектов к изображению. В данный момент шейдер в широком смысле означает программу, применяющую эффекты, не обязательно относящиеся к наложению света. Есть 3 типа шейдеров:

1. вершинный;
2. фрагментный;
3. геометрический.

Вершинный шейдер, как следует из схемы на рис 15 осуществляет обработку позиций вершин. Фрагментный шейдер отвечает за растеризацию фрагментов, то есть вычисление цветов их пикселей. Будучи сопряженными эти 2 шейдера образуют т.н. **шейдерную программу** – программу, написанную на строго типизированном C-подобном языке – GLSL (GL Shader Language). GLSL имеет несколько типов данных (векторные vec2, vec3, vec4 и матричные mat2, mat3, mat4) и поддерживает множество распространенных математических операций над ними. [1] Пример вершинного шейдера, используемого в данной работе приведен на Листинге 7.

Листинг 7: Вершинный шейдер

```
attribute vec3 aPosition;
```

```
attribute vec3 aColor;
attribute vec3 aNormal;

uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProjection;
uniform mat4 uWorldInverseTranspose;

varying vec3 fragColor;
varying vec3 v_normal;

void main() {
    mat4 mvp = uProjection * uView * uModel;
    gl_Position = mvp * vec4(aPosition, 1);
    fragColor = aColor;
    v_normal = mat3(uWorldInverseTranspose) * aNormal;
}
```

Подобная программа компилируется, линкуется и передается на исполнение GPU. Соответственно, все передаваемые по конвейеру данные проходят через 2 функции – функцию вычисления позиции и функцию вычисления цвета (утрированно). Существует несколько способов передачи данных шейдеру:

- буферы и атрибуты;
- uniform-переменные;
- varying-переменные;
- текстуры.

Буферы – массивы данных, загруженных в графический процессор. **Атрибуты** – определяют то, каким образом разбираются массивы загруженных данных: сколько элементов из массива надо взять, какой тип этих элементов, какое смещение от начала массива. **Uniform-переменные** – своего рода глобальные переменные, доступные из любой части шейдерной программы. **Varying-переменные** – переменные, которые передаются из вершинного буфера во фрагментный. **Текстуры** – массивы данных с произвольным доступом к ним из программы шейдера. Стоит отметить, что «данными» здесь могут быть не только значения вершин, но и любая другая информация.

На Листинге 7 присутствует атрибут aPosition, представляющий из себя вектор, дополняемый до 4х элементов (о преобразовании координат будет сказано далее). Внутри стандартной функции main() его значение перемножается с матрицей преобразования координат и присваивается системной переменной gl_Position. Видеокарта сохраняет эти значения позиций у себя. Чтобы непосредственно вывести пиксели на этих позициях, видеокарта применяет растеризацию – получение фактических координат и вычисление для них значений цветов. Для этого вызывается похожий фрагментный шейдер, который принимает значение нормали и цвета, обрабатывает их

и полученные значения цветов присваиваются системной переменной gl_FragColor. После этого фигура может быть отображена на экране. Большинство 3D-движков генерируют шейдеры «на лету», используя шаблоны, конкатенацию и прочие приемы автоматизированной генерации. [2]

Далее следует рассмотреть буферы. Как было сказано ранее, это массивы данных, загруженных в видеокарту. Чтобы передать данные атрибуту внутри шейдера, необходимо выполнить поиск этого атрибута, а также создать буфер. Затем следует создать биндинг (связь) этого массива, чтобы обозначить, что это именно *массив данных* для буфера. Далее необходимо передать данные в буфер, сопроводив их информацией о типе этих данных. На этом процесс инициализации завершается.

Далее все действия осуществляются в процессе работы – в цикле отображения (render loop). Обозначается использование текущей шейдерной программы и «включается» ее атрибут. Затем надо указать, как именно должны быть выбраны данные из буфера (количество, тип элементов, нужна ли нормализация, смещение в массиве данных). После этого вызывается главная функция gl.drawArrays(), которая и запускает графический конвейер.

Листинг 8: Объем кода, без учета шейдеров, необходимый для вывода одного треугольника

```
// получение позиции буфера внутри шейдера
const positionAttributeLocation = gl.getAttribLocation(program, "aPosition");
// создание буфера
const positionBuffer = gl.createBuffer();
// связывание positionBuffer с ARRAY_BUFFER
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// передача данных о позициях вершин в буфер, связанный с ARRAY_BUFFER
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([< вершины >]), gl.STATIC_DRAW);

// цикл отображения (render loop)
// использование текущей шейдерной программы
gl.useProgram(program);
// включение атрибута для шейдера
gl.enableVertexAttribArray(positionAttributeLocation);
// повторное связывание вершинного буфера с ARRAY_BUFFER
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// извлечение по 3 компоненты типа Float ьез нормализации за раз со смещением 0
gl.vertexAttribPointer(positionAttributeLocation, 3, gl.FLOAT, false, 0, 0);

// отрисовка труегольника по м3 точкам из буфера по смещению 0
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Процесс, начиная с подготовки вершин до вывода пикселей на экран показан на Листинге 8. При этом на листинге отсутствуют шейдеры, а также код, необходимый для их компиляции и линковки. Также не приведен код настройки рабочей области (viewport). Отсюда можно сделать вывод о том, что WebGL API достаточно многословен (verbose) и для работы требуется большой объем повторяющегося кода.

Представленные выше Листинги позволяют отобразить на экране 2D объект. Трехмерные объекты, в силу наличия объема, не могут быть так же «просто» отображены на 2D экране. Для преобразования вершин в 3D пространстве в пиксели 2D экрана, необходимы преобразования между системами координат. Отличительной чертой WebGL является необходимость самостоятельной реализации подобных матричных преобразований.

Но прежде чем рассматривать преобразования, необходимо ввести **однородные координаты** – систему координат из проективной геометрии. Декартовы координаты не подходят в силу следующих соображений:

- декартовы координаты не позволяют отличить точку от вектора в пространстве. Действительно, $(1, 3, 7)$ может быть и направлением и точкой;
- применительно к компьютерной графике, д.к. не позволяют описать бесконечно удаленную точку, в то время, как введение «бесконечности» могло бы сильно упростить многие математические концепции. Пример – бесконечно удаленный источник света;
- невозможно использовать механизм работы с матрицами для задания перспективных преобразований (проекций).

Решение всех этих проблем возможно путем введения однородных координат.

Кортеж (x, y, z) из трех элементов до настоящего времени представлял вершину в трехмерном пространстве. Теперь, после ввода w, вектор выглядит так: (x, y, z, w). Необходимо ввести 2 правила:

- при $w = 1$ вектор $(x, y, z, 1)$ – позиция в пространстве;
- при $w = 0$ вектор $(x, y, z, 0)$ задает направление.

Теперь, когда однородные координаты введены, следует вернуться к рассмотрению матричных преобразований. Все рассматриваемые далее преобразования – аффинные. Преобразование называется **аффинным**, если оно взаимно однозначно и образом любой прямой является прямая. Преобразование называется **взаимно однозначным**, если оно разные точки переводит в разные, и при этом в каждую точку переходит какая-то точка.

Преобразование движения представлено ниже. Его матрица наиболее проста и выглядит следующим образом:

$$T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Здесь значения T_x, T_y, T_z добавляются к текущей позиции.

Преобразование растяжения (сжатия) представлено матрицей масштабирования:

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Здесь коэффициенты S_x, S_y, S_z отвечают за растяжение по соответствующим координатам.

Преобразование вращения представлено матрицей поворота:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_y = \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_z = \begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Здесь ϕ – угол поворота вокруг соответствующей оси.

Теперь, когда известен способ преобразования векторов, необходимо объединить все описанные выше преобразования воедино. Объединение преобразований состоит в перемножении матриц. Стоит также помнить, что порядок преобразований играет роль. Пусть под преобразованиями подразумеваются «шаг и поворот налево». Тогда результат шага и последующего поворота налево будет отличаться от поворота налево с последующим шагом.

Порядок выполнения преобразований должен быть следующим: **растяжение, поворот, перенос**. Попробуем выполнить операции в другом порядке на упрощенном примере:

1. перемещаем модель в координаты $(10, 0, 0)$. Ее центр сейчас удален на 10 единиц от центра мира

2. увеличиваем (масштабируем) модель в 2 раза. Каждая координата умножается на 2 относительно центра. В итоге имеем увеличенную в 2 раза модель, но размещенную в 20 единицах от центра.

Правильный порядок преобразования:

1. модель находится в центре системы координат. Масштабируем модель в 2 раза. Ее центр по-прежнему в точке $(0, 0, 0)$;

2. осуществляем перенос. Никаких побочных эффектов при этом не наблюдается.

Для удобства использования аффинных преобразований вводят 3 матрицы: **модельную** (model), **видовую** (view) и матрицу **проекции** (projection). Будучи объединенными, эти матрицы задают т.н. **MVP** матрицу (Model View Projection).

Любая модель представляет из себя набор вершин. Эти вершины имеют координаты (X, Y, Z), определенные относительно центра объекта. Таким образом, если вершина имеет координаты $(0, 0, 0)$, то она находится в центре объекта. Чтобы, например, подвинуть объект, необходимо выполнить преобразование движения. Чтобы

уменьшить объект, необходимо выполнить преобразование сжатия. Чтобы развернуть модель, необходимо выполнить преобразование вращения.

Следовательно, для манипуляции объектом в пространстве необходимо выполнять все три преобразования, или, как было замечено выше, они могут быть заменены одной матрицей преобразования, которая выполняет их в следующем порядке: растяжение, поворот, перенос. Это может быть записано следующим образом:

$$M = T \cdot R \cdot S,$$

где T – матрица переноса, R – матрица поворота, S – матрица масштабирования. Отсюда следует определение: **модельная матрица** – матрица M, содержащая все трансформации, примененные к объекту. На Листинге 9 представлено вычисление модельной матрицы с помощью векторов трансформаций. Такое вычисление происходит на каждой итерации цикла отображения (render loop).

Листинг 9: Вычисление модельной матрицы

```
const makeModelMatrix = (scaleVec, translationVec, rotationVec) => {
    const modelMatrix = mat4.create();
    // масштабирвание
    mat4.scale(modelMatrix, modelMatrix, scaleVec);
    // поворот
    mat4.rotateX(modelMatrix, modelMatrix, rotationVec[0]);
    mat4.rotateY(modelMatrix, modelMatrix, -rotationVec[1]);
    mat4.rotateZ(modelMatrix, modelMatrix, rotationVec[2]);
    // перенос
    mat4.translate(modelMatrix, modelMatrix, translationVec);
    return modelMatrix;
};
```

Таким образом, был осуществлен переход из **модельного пространства** (Model space), где все вершины определены относительно центра объекта к **мировому пространству** (World space), где вершины определены относительно центра мира.

Следующая на очереди – **видовая матрица**. Видовая матрица отвечает за представление сцены с точки зрения наблюдателя и используется при реализации камеры на сцене. Смысл проводимых ею преобразований проще понять из примера: чтобы сделать снимок, например, горы, но с другого ракурса, можно передвинуть камеру, но можно передвинуть и гору. В то время, как это неприменимо в реальном мире, именно этот прием применяется в компьютерной графике.

Изначально камера расположена в центре мира (мировых координат), ее верх сонаправлен с вектором $(0, 1, 0)$ и она смотрит против направления оси Z. Объекты пространства расположены на отрезке $[0, -\infty]$ по Z. Чтобы переместить камеру, например, на 5 единиц *влево* – потребуется переместить мировое пространство на 5 единиц *вправо*. Подобные преобразования обеспечиваются встроенной функцией lookAtPos, а вычисление видовой матрицы в рамках данной работы будет производится с помощью функции, представленной на Листинге 10 для камеры camera, которой манипулирует пользователь с помощью мыши и клавиатуры.

Листинг 10: Вычисление видовой матрицы

```
const makeViewMatrix = (camera) => {
    const identityMatrix = mat4.create();
    const eye = camera.position;
    const lookAtPosition = camera.lookAtPos;
    return mat4.lookAtPos(identityMatrix, eye, lookAtPosition, camera.top);
};
```

Таким образом, был осуществлен переход из **мирового пространства** (World space), где вершины определены относительно центра мира к **пространству камеры** (Camera space), где вершины определены относительно камеры.

Последней из рассматриваемых матриц является **матрица проекции**. По-умолчанию объекты одного размера в OpenGL отображаются в одинаковым размере, независимо от того, где они находятся относительно камеры. Тогда объекты с координатами $(1, 1, 13)$ и $(1, 1, 37)$ имеют одинаковый размер для смотрящего через камеру, так как расстояние по оси Z не учитывается. Очевидно, что это не подходит для изображения реалистичной сцены: согласно построению перспективы, *из двух объектов с одинаковыми координатами x и y тот объект, чье значение координаты z больше, должен быть размещен ближе к центру экрана (изображения)*. Это называется **перспективной проекцией**.

Особенность OpenGL: *отображению подлежат лишь объекты, что находятся внутри пространства отсечения (clipspace)*. Если объект находится в пространстве частично, к нему применятся алгоритмы отсечения, которые оставляют лишь видимую его часть. Для 2D графики пространство отсечения – квадрат с длиной стороны, равной 2 и центром в точке $(0, 0)$. Для 3D графики – куб со стороной 2 и центром в точке $(0, 0, 0)$.

Чтобы разместить объекты внутри пространства отсечения, необходимо применить к ним преобразование с помощью матрицы перспективной проекции:

$$P = \begin{bmatrix} \frac{2-near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2-near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \cdot far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Значения $top, bottom, right, left$ вычисляются следующим образом:

$$top = near \cdot \tan(\frac{\pi}{180} \cdot FOV/2)$$

$$bottom = -top$$

$$right = top \cdot aspect$$

$$left = -right$$

Здесь FOV (Field Of View) обозначает область видимости, $aspect$ – соотношение сторон, а $near$ и far – ближнюю и дальнюю плоскости, между которыми располо-

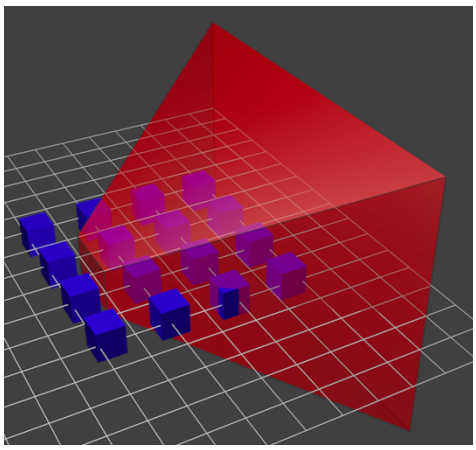

Рис. 16: Пространство до проекционного преобразования

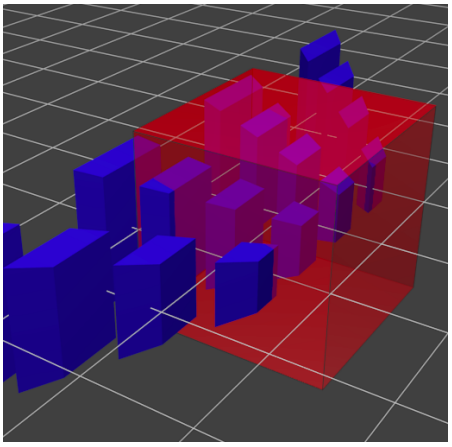

Рис. 17: Пространство после проекционного преобразования

жены видимые объекты. Объекты ближе $near$ и дальше far не отображаются. Все эти значения задают т.н. **усеченную пирамиду отображения** (frustum), внутри которой находятся видимые объекты, а снаружи – невидимые. Вычисление матрицы проекции (и, соответственно, размещение усеченной пирамиды) в рамках данной работы представлено на Листинге 11.

Листинг 11: Вычисление матрицы проекции

```
const makeProjectionMatrix = () => {
    const identityMatrix = mat4.create();
    const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
    const zNear = 1;
    const zFar = 5000;
    const fowRadians = Math.PI / 3; // field of view
    return mat4.perspective(identityMatrix, fowRadians, aspect, zNear, zFar);
};
```

Более наглядно проекционное преобразование представлено на рис. 16 и рис. 17. На рис. 16 представлено пространство камеры (Camera space) и получаемая из матрицы P усеченная пирамида отображения. На рис. 17 представлен результат перемножения с матрицей проекции: усеченная пирамида трансформируется в куб со стороной 2, а объекты деформируются и изменяются в размере, чтобы изобразить перспективу.

Более формально преобразование представлено на рис. 18.

После всех преобразований выполняется еще одна автоматическая трансформация, которая преобразует куб пространства отсечения в 2D изображение и адаптирует его к размеру экрана, на котором отображается сцена.

Стоит отметить, что *все* описанные выше преобразования происходят на каждой итерации бесконечного цикла отображения. Последовательность отражена на рис. 19. OpenGL без использования шейдерной графики проводит эти преобразования автоматически. В WebGL, из-за использования шейдеров, они должны быть заданы

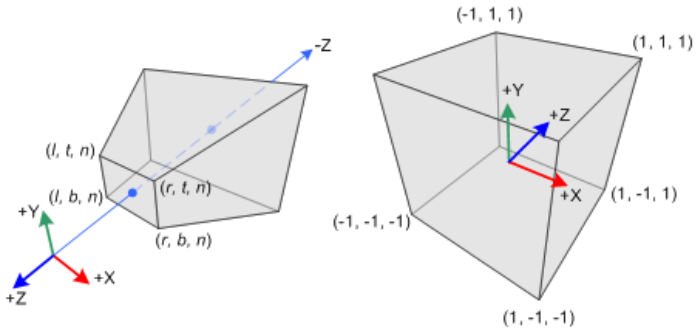

Рис. 18: Проекционное преобразование усеченной пирамиды в куб отсечения

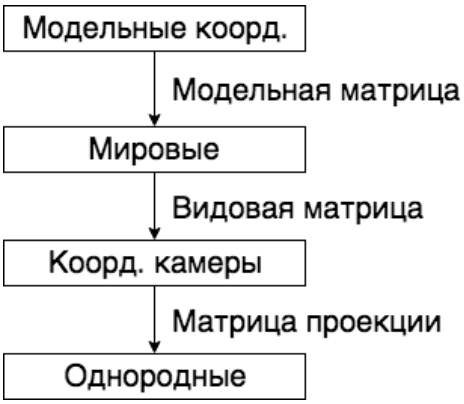

Рис. 19: Последовательность матричных преобразований

разработчиком и эффективность работы приложения во многом зависит от корректности этих операций.

На данный момент на сцене может быть отображен 3D-объект и над ним может производиться манипуляция благодаря описанному выше и реализованному аппарату математических преобразований. Использование перспективной матрицы преобразований добавляет реалистичности отображаемой графике, однако не реализована одна важная деталь – освещение. Так, сложные трехмерные объекты, будучи окрашенными в один цвет визуально перестают быть трехмерными, т.к. части объектов, расположенные на переднем плане визуально ничем не отличаются от частей на заднем плане и объемная фигура предстает своей двухмерной одноцветной проекцией на плоскость экрана. Объемной графику делает свет и тень и для наилучшего визуального восприятия изображения эти элементы должны быть добавлены на сцену.

В большинстве графических редакторов используется направленное освещение. **Направленное освещение** – освещение, равномерно исходящее от бесконечно удаленного источника света, такое, что направление его лучей может считаться параллельным.

Процедура вычисления направленного освещения является достаточно простой. Зная нормаль к лицевой стороне треугольника, составляющего объект и направление света, исходящего от источника, может быть вычислено скалярное произведение. Это

даст косинус угла между этими двумя векторами. Тогда будем считать, что значение косинуса равно 1, когда эти 2 вектора противоположны (лучи света перпендикулярны поверхности) и -1, если значения сонаправлены (лучи света перпендикулярны поверхности и направлены от нее). Схематично это представлено на рис. 20. Умножив значения компонентов цвета на получившееся в ходе вычислений значение получаем новое значение компонентов с учетом количества света, попадающего на область поверхность.

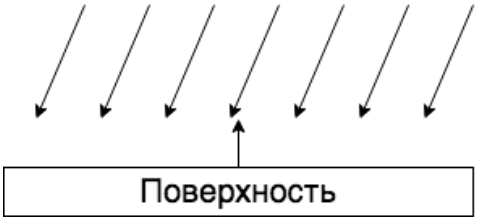

Рис. 20: Вычисление интенсивности света из скалярного произведения векторов

Вычисление значений компонентов цвета производится во фрагментном шейдере, представленном на Листинге 12. Значение величины без учета спецэффектов устанавливается в системную переменную gl_FragColor, а ее поле rgb, отвечающее за отдельные компоненты цветов модифицируется с использованием скалярного произведения векторов описанным выше способом (с незначительной оптимизацией).

Листинг 12: Фрагментный шейдер

```
precision highp float;

varying vec3 fragColor;
varying vec3 v_normal;
uniform vec3 uReverseLightDirection;

void main() {
    vec3 normal = normalize(v_normal);
    float light = dot(normal, uReverseLightDirection);
    gl_FragColor = vec4(fragColor, 1.0);
    gl_FragColor.rgb *= light;
}
```

Для упрощения работы со светом необходимо лишь автоматизировать вычисление нормалей для каждой представленной в сцене модели. Стоит также отметить, что примитивы, такие, как линии (gl.LINES) и объекты на их основе должны использовать свои собственные шейдеры, не использующие освещение.

Отдельно стоит рассмотреть «связь» формата STL и шейдерной графики. Как было отмечено во время рассмотрения особенностей WebGL, набор используемых примитивов невелик. Ими являются точка, отрезок, набор точек, как ломаная прямая, треугольник и набор «связанных» треугольников. Для «применения света» к модели необходим расчет нормалей для составляющих модель элементов.

В свою очередь, особенность STL формата заключается в том, что модели представлены в нем в виде набора треугольников. Каждый треугольник при этом снабжен вычисленной заранее нормалью. То есть, созданы все условия для визуализации STL-модели при помощи графических примитивов. Тем не менее, при применении матричных преобразований модель изменяется и необходим последующий перерасчет нормалей, что, впрочем, не представляет сложности. В качестве примера на рис. 21 представлена модель кости кисти, состоящая из 1,33 *млн* вершин, что дает порядка 440000 треугольников. Размер модели составляет 22 МБ, что не является пределом в данной предметной области. Размер может превышать 100 МБ. Тем не менее, благодаря использованию шейдеров и буферов вершин, такое количество треугольников представляется не отдельными треугольниками (что создало бы существенную нагрузку на графическую подсистему), а набором вершин, которые компонуются в треугольники уже графическим процессором и, таким образом, составляют цельную модель, эффективно отображаемую даже на смартфонах.

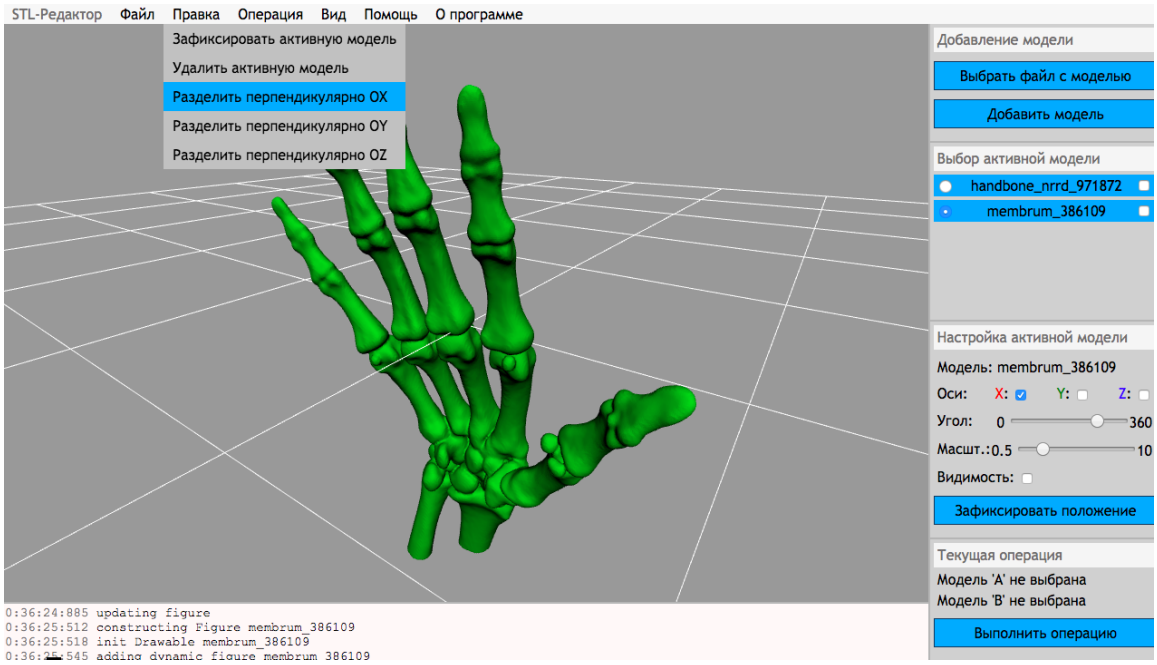

Рис. 21: Скриншот интерфейса с примером модели

На этом разработку функциональности для отображения произвольной трехмерной модели можно считать завершенной. Наряду с графическим интерфейсом пользователя теперь имеется и функционирующая рабочая область. Функции обработки STL моделей все еще не имеют реализаций, но, часть приложения, связанная с просмотром моделей завершена. Получившийся интерфейс пользователя, а также рабочая область с моделью в ней представлены на рис. 21.

2.3.3 Хранение данных

После того, как работа с интерфейсом завершена, встает вопрос о хранении данных. Основными данными, хранимыми в клиентском приложении являются сами

STL модели. В силу специфики формата STL, хранение моделей может потребовать значительного объема пространства.

Хранение моделей может быть с легкостью организовано на сервере. Таким образом, клиент после добавления модели в сцену отправляет ее на сервер. Организация работы бэкенда на данный момент не рассматривается, но в таком подходе прослеживается серьезная проблема: графический редактор предполагает редактирование объектов. Применительно к STL моделям, даже перенос в пространстве является редактированием модели, если это положение будет зафиксировано. Возникает проблема рассинхронизации реального положения модели и изначального, в котором модель была отправлена на сервер.

Эта проблема может быть решена принудительной синхронизацией, организованной самостоятельно, но это не только усложняет систему, но и замедляет ее (регулярная передача данных по сети). Еще на стадии проектирования становится очевидно, что синхронизация не является приемлемым решением.

Таким образом, стоит вернуться к вопросу хранения данных на стороне клиента.

Хранение данных в браузере пользователя используется повсеместно. Самый простой пример – хранение пользовательских данных и состояния приложения, на котором пользователь закончил работу с ним. Это может быть, как персонализированная настройка внешнего вида сайта, так и информация о прошлом входе в приложение или даже JWT-токен для доступа к защищенному ресурсу. В данной работе пользовательскими данными являются STL модели, что, ввиду их большего по сравнению с, например, JWT-токеном, объема, накладывает дополнительные ограничения.

На данный момент существует несколько способов реализации хранения. Чтобы выбрать оптимальный для решаемой в данной работе задачи способ, необходимо проанализировать все имеющиеся варианты.

Самым простым способом хранения данных являются **cookie**. Cookie – это небольшой объем данных, которые сервер отправляет веб-браузеру пользователя. При следующем обращении браузер может передать cookie вместе с запросом и, таким образом, сообщить серверу о состоянии пользователя по отношению к серверу, т.е., например, совершил ли пользователь вход в приложение. Это называется управлением сессией. Хранить данные на клиентской стороне и отправлять их каждый раз приходится, т.к. протокол HTTP не поддерживает состояние (stateless).

Cookie могут быть сохранены на стороне пользователя путем добавления заголовка Set-Cookie в HTTP ответ севера и представляются в виде набора пар $key = value$. В одном ответе севера могут возвращаться от одной до нескольких cookie. Пример такого ответа представлен на Листинге 13.

Листинг 13: Пример HTTP ответа, устанавливающего cookie

```
HTTP/1.0 200 OK
...
Set-Cookie: id=Asu; name=na; Expires=Wed, 30 Sep 2018 10:04:09 GMT;
Set-Cookie: SA=0; k1=r1_t0;
```

Причиной, по которой использование cookies на данный момент не рекомендуется – множественные технические проблемы и существенные проблемы с безопасностью. Среди технических проблем – возрастающая нагрузка на сеть, т.к. в запросы включается множество дополнительной информации. Проблемы с безопасностью заключаются в наличии множества способов «украсть» cookie. В настоящее время не обязательно знать логин и пароль от учетной записи для входа в нее: браузер, отправляя запрос серверу, на котором находится учетная запись, включает в запрос cookie, где указано, что пользователь уже совершил вход и проверка креденций не нужна. Таким образом, злоумышленник, завладев cookies пользователя для этого ресурса может зайти на него, не зная ни логина, ни пароля. Более того, злоумышленник может встроить вредоносный скрипт в страницу, подменив его, например, картинкой. Тогда в момент, когда пользователь совершает клик, серверу отправляется запрос от имени пользователя. В итоге аккаунт пользователя может быть с легкостью компрометирован.

Подобная атака называется межсайтовой подделкой запросов или CSRF (Cross-Site Request Forgery). Для ее предотвращения в активные элементы страницы (элементы, которые совершают действия, в том числе и от имени пользователя) сервер превентивно внедряет CSRF-токены – подписанные сервером «ключи». При клике по элементу с внедренным в него CSRF-токеном в запрос добавляется токен и сервер может удостовериться в том, что форма, отправившая запрос не подделана и пользователь действительно совершил запрос. Появляются все новые векторы атаки на cookies пользователя и вместе с ними появляются и способы защиты от этих атак.

Cookies активно используются для отслеживания действий пользователя в сети.

Также cookies не используются для хранения больших объемов данных. При желании такое хранения можно организовать (разделяя данные на части), но это вызовет ненужное усложнение системы.

Единственное преимущество этой устаревшей технологии – поддержка ее устаревшими бразуерами (например, Internet Explorer). В рамках текущей работы поддержка устаревших браузеров не заявлялась и, следовательно, технология cookies использована не будет. Против использования технологии говорят и список проблем, приведенных выше. Следовательно, необходимо найти другие варианты.

Альтернативой cookies служит **Web Storage API** – интерфейс для работы со встроенным в браузер хранилищем элементов типа $key = value$ (key-value store). Хранилище состоит из двух частей:

- сессионное хранилище (session storage) (хранилище, доступное домену в течение сессии);
- локальное хранилище (local storage).

Локальное хранилище работает постоянно и не очищается при выключении браузера, в отличие от сессионного хранилища. WebStorage предоставляет удобный и простой способ хранить и обрабатывать информацию – создаваемый в глобальном контексте объект Storage позволяется с легкостью сохранять, читать и удалять объекты по ключам. Это продемонстрировано на Листинге 14.

Листинг 14: Работа с локальным хранилищем в JavaScript

```
// сохранение данных
localStorage.setItem('operation', 'union');

// чтение данных
let operation = localStorage.getItem('operation'); // -> union

// удаление данных
localStorage.removeItem('operation');
operation = localStorage.getItem('operation'); // -> null
```

В дополнение к простому механизму записи и чтения, WebStorage предоставляет возможность обработки событий изменения данных. Событие StorageEvent происходит при любом изменении данных. Благодаря этому возможна синхронизация состояния приложения, изменяемого, например, сразу из нескольких вкладок браузера.

Безопасность хранения данных гарантируется на уровне браузера. Домену доступны только данные своего хранилища. Хранилища всех доменов отделены друг от друга.

В большинстве случаев WebStorage является оптимальным выбором. Однако, в рамках решаемой задачи WebStorage не выполняет одно из требований к механизму хранения – возможность хранить *большой* объем информации. Несмотря на то, что объем сессионного и локального хранилища многократно превышает объемы данных, которые способны хранить cookies (байты и килобайты информации), он (объем) также лимитирован. Браузеры Google Chrome, Firefox и Opera выделяют не более 10 МБ свободного пространства для домена. 10 мегабайт достаточно для хранения некоторых STL-моделей по отдельности, но т.к. согласно постановке задачи, количество просматриваемых и редактируемых моделей не ограничено, доступные 10 мегабайт не согласуются с этим требованием.

Тем не менее, локальное хранилище, благодаря простоте работы с ним, может быть использовано во время разработки, пока не найден более подходящий способ хранения данных.

Следующим на очереди способом хранения данных является **IndexedDB** (IDB). В отличие от предыдущих вариантов, где хранилища содержали лишь пары $key = value$, IDB представляет из себя полноценную базу данных. Интерфейс IDB предоставляет низкоуровневый доступ для работы с данными значительного размера: данными могут быть файлы и blob'ы (Binary Large OBject).

Среди особенностей IDB можно выделить:

- транзакционность. С этой стороны IDB является в некотором роде РСУБД (RDBMS, Relational DataBase Management System);
- документоориентированность. Данные представлены в виде документов в формате JSON, и проиндексированы ключами;
- асинхронность операций. Вместо возврата данных возвращаются коллбеки соответствующих событий (например, успешное сохранение, неудавшееся чтение).

IDB является развитием WebSQL Database, которая была признана устаревшей в 2010 году. Формально, именно WebSQL Database являлась реляционной базой данных, но из-за распространения NoSQL, концепция IDB была изменена в пользу новой модели. Таким образом, IDB является **системой индексированных таблиц**. Тем не менее, смена подхода делает IndexedDB уникальной: популярное современной NoSQL решение – документоориентированная MongoDB – не поддерживает полноценные транзакции (вместо этого используются атомарные операции), в то время, как «наследие» WebSQL добавляет в IDB такую возможность.

Безопасность IDB гарантируется поддержкой правил ограничения домена (same-origin policy): домены могут работать с доступной только им информацией.

Среди ограничений IDB можно выделить отсутствие оператора Like, сортировки интернационализированных данных и встроенных механизмов синхронизации с серверной базой данных.

IDB, как и изученные ранее решения, имеет ограничения на объем хранимой информации. Так, если объем жесткого диска составляет 500 ГБ, Quota Manager, инструмент работы с памятью Firefox, при отсутствии других ограничений выделит браузеру квоту в 250 ГБ. Далее ресурсы могут сохранять свои данные со следующими условиями: существует т.н. групповой лимит памяти, не позволяющий превысить 2ГБ объема или 20% глобальной квоты. В данном примере этот лимит составляет 2ГБ. Группой считаются ресурсы, принадлежащие одному домену. Например, example.com и blog.example.com принадлежат одной группе, но разным источникам (origins). Так как групп может быть очень много, клобальная квота может исчерпаться. Тогда Quota Manager среди всех групп ищет источник (origin), используя стартегию LRU (Least Recently Updated, наиболее давно обновленный) и по нахождении удаляет его полностью (чтобы избежать неконсистентности данных в случае частичного удаления). Эта процедура повторяется до тех пор, пока объем не придет в норму.

В рамках данной работы недостатки и ограничения не играют существенной роли, а преимущества делают IndexedDB идеальным выбором для хранения данных на стороне пользователя.

На Листинге 15 частично продемонстрировано выполнение операции (с использованием Indexed DB), переданной в функции обратного вызова callback. Порядок работы с IDB следующий:

1. открытие соединения с базой данных;
2. создание/открытие объекта-хранилища;
3. открытие транзакции;
4. ожидание окончания выполнения транзакции (DOM-событие);
5. работа с результатами в случае успеха.

Листинг 15: Выполнение произвольной операции над хранилищем в IDB

```
const execute = (callback) => {
    // открытие соединения с базой данных и создание хранилища STL моделей
```

```
    const open = indexedDB.open("EDITOR_DB", 1);
    open.onupgradeneeded = function() {
        const db = open.result;
        const store = db.createObjectStore("STL_STORAGE", {keyPath: "id"});
    };
    open.onsuccess = function (event) {
        const db = open.result;
        const transaction = db.transaction(DB_STORE, "readwrite");
        const store = transaction.objectStore(DB_STORE);

        // выполнение произвольной операции
        callback(error, db, store);
    }
}
```

На Листинге 16 представлена реализация операций чтения и записи модели с использованием функции execute из Листинга 15. Функция upsertFigure выполняет операцию вставки-обновления (update insert). Если ключа, figureId в хранилище нет, то произойдет вставка ключа и в значение, соответствующее ключу будут записаны данные о модели в виде объекта с полями id, name, modeldata в формате JSON. Если такой ключ уже существует, соответствующее ему значение будет заменено на новое.

Чтение бинарных данных модели (в формате Base64) осуществляется также асинхронно и результат чтения доступен в функции обратного вызова onsuccess. Фактически, при чтении создается объект, обозначающий намерение чтения (getModel) и, в зависимости от результата, вызывается соответствующая функция обратного вызова (onsuccess или onerror).

Листинг 16: Запись и чтение моделей в IDB

```
// сохранение модели
const upsertFigure = (figureId, figureName, figureData) => {
    // экспорт в бинарный файл
    const modelByteArray = STLExporter.exportToBinaryStl(figureData);
    const modelBlob = new Blob([modelByteArray], {type: "application/sla"});

    // конвертация в Base64 формат и обновление записи в базе данных
    B64Converter.convertToBase64(modelBlob, (err, result) => {
        execute((dbErr, db, store) => {
            store.put({
                id: figureId,
                modeldata: result,
                name: figureName
            });
        });
    });
}
```

```
// чтение бинарных данных модели по ключу "x"
execute((dbErr, db, store) => {
    const getModel = store.get("x");
    getModel.onsuccess = function () {
        const stlModelData = getModel.result.modeldata;
    }
    getModel.onerror = function () {
        // чтение не удалось
    }
});
```

После выбора хранилища данных (IDB) следует обозначить способ хранения данных в нем. Здесь тип приложения накладывает ограничения. Так как приложение сетевое и основная задача – редактирование STL-моделей, требуется частая отправка данных на сервер и обратно. Сетевое взаимодействие занимает много времени по сравнению с другими операциями. Это время будет увеличено еще больше, если перед операцией выполнять конвертацию используемой в WebGL модели в пригодный для передачи по сети формат. При выполнении взаимодействия необходимо свести к минимуму количество «ручной» подготовки данных (например, конвертации) и предоставить браузеру лишь необходимую для отправки информацию.

Следовательно, появляется необходимость хранить модели в подготовленном для отправки на сервер виде, чтобы «по клику» пользователя совершить минимум действий и не замедлять общий процесс.

Протокол, по которому происходит сетевой взаимодействие – HTTP – текстовый. Таким образом, все файлы должны быть преобразованы в текст. Как известно, существует большое число способов представления текста. Пример – шестибитные, семибитные, однобайтовые и двубайтовые кодировки. Немногие из них подходят для использования при текстовом взаимодействии, т.к. обе стороны должны однозначно понимать переданный текст. При передаче файлы должны быть представлены в виде набора байтов а затем переданы в текстовом виде по сети. Однако, существует проблема с истолкованием данных различными устройствами. В настоящее время сама проблема практически отсутствует, а способ ее решения стал де-факто стандартом кодирования.

Речь идет об алгоритме **Base64**. Так, принятые сегодня повсеместно однобайтовые и двубайтовые кодировки в свое время были не первыми кодировками. Уже долгое время существовали шести- и семибитные кодировки. И значительная часть инфраструктуры работала с 6 или 7 битами. Эта инфраструктура не использовала 7 или 8 биты и обнуляла их. Очевидно, что информация при этом безвозвратно терялась. Тогда, для обратной совместимости был разработан алгоритм Base64. [10]

Сейчас Base64 используется в том числе, чтобы кодировать строки URL. Такое кодирование не защищает от взлома (декодирования), но скрывает природу данных при попытке прочтения сторонним человеком.

В основе алгоритма – обратимое кодирование, которое гарантированно сохраняет данные при передаче между любыми сетями и устройствами. Кодируемые данные представляются в бинарном виде и каждые 3 восьмерки битов преобразуются в 4

Таблица 1: Представление строки "model"в бинарном виде

m	o	d	e	l
01101101	01101111	01100100	01100101	01101100

Таблица 2: Разбиение битов на группы по 6 элементов

011011	010110	111101	100100	011001	010110	1100 + 00
b	W	9	k	Z	W	w

шестерки. 6 битов могут быть однозначно представлены символами таблицы ASCII. В качестве примера можно выполнить кодирование строки "model":

1. строка "model"в бинарном виде представлена в Таблице 1;

2. получившийся набор восьмерок битов конкатенируется и разбивается на шестерки. К последним 4м битам дописываются нули, чтобы образовать шестерку. Разбиение представлено в Таблице 2;

3. в конец дописывается знак =, как признак лишних битов.

Таким образом, строка "model"при Base64 кодировании превращается в строку "bW9kZWw=". Очевидно, что размер строки увеличился в соотношении 4:3

Применительно к решаемой задаче, Base64 используется при хранении файлов внутри базы данных. Base64-кодирование при записи бинарных данных модели в IDB представлено на Листинге 16. В итоге для отправки файла на сервер, необходимо лишь получить его из базы данных. Значение, лежащее в БД уже закодировано и готово к транспортировке.

Плата за использование кодировки Base64 – возрастающий объем данных и применительно к специфике STL этот объем возрастает значительно: так 30 МБ превращаются в 40 МБ, а 100 МБ превращаются в 130+ МБ. Устранение этого недостатка будет рассмотрено в рамках оптимизации проекта.

На этом разработка клиентской части закончена и далее будет приведен протокол взаимодействия клиентской и серверной частей приложения.

2.4 Клиент-серверное взаимодействие

Тип разрабатываемого приложения – веб-приложение или, другими словами, веб-сервис. Веб-сервис по своей сути является сетевой службой. Сетевая служба – это пара клиент-сервер и протокол работы, обеспечивающий их взаимодействие. Сетевая служба не обязательно должна состоять из 2х элементов, но в рамках данной работы это так.

Протокол взаимодействия – набор правил и соглашений, действующий в рамках сетевой службы. Под правилами и соглашениями могут пониматься формат сообщений, порядок следования сообщений и определенные бизнес-логикой дополнительные правила.

Протокол взаимодействия является неотъемлемой и, вероятно, главной частью веб-сервиса. Без протокола клиент (как компонент архитектуры) не сможет отправить информацию на обработку на сервер. Сервер, в свою очередь может выполнить обработку информации, только имея входные данные. В итоге даже если есть 2 готовых компонента архитектуры, но механизм их «общения» не отлажен или реализуется не в полной мере, это ударяет по системе в целом.

Прежде чем приступить к проектированию протокола, стоит разобраться с техническим устройством сетевых протоколов, так как это понадобится в дальнейшей работе над проектом и в том числе при оптимизации работы.

Сетевая инфраструктура основана на декомпозиции набора сетевых протоколов на уровни согласно некоторым правилам. Так, компоненту i-го уровня разрешается взаимодействия только с компонентом (i-1) уровня. Компонент i-го уровня совершает некоторые преобразования над полученными от уровня выше данными и далее делегирует последующую их обработку компоненту уровня ниже: i-1. Соответственно, данные на конкретном уровне имеют какой-либо смысл лишь на этом уровне. Исходный же смысл данных понятен лишь на входе в цепь преобразований. Основываясь на таком подходе некоторый набор протоколов может быть упорядочен. Таким образом, появляется иерархия протоколов.

Теперь для взаимодействия двух узлов требуется, чтобы оба узла поддерживали совместимые наборы протоколов. В таком случае сообщения, отправленные одним узлом проходят обработку (кодирование) каждым компонентом (протоколом) с одну сторону, передаются по сети другому узлу и проходят обработку согласно набору протоколов, но уже в другую сторону (декодирование). Простейший пример – азбука Морзе. На вход подается сообщение в человекочитаемом виде. Согласно протоколу передачи данные должны быть закодированы, используя только 2 сигнала: короткий (точка) и длинный (тире). Представленные в таком виде данные могут быть переданы с помощью телеграфа от одного узла другому. Полученные на другом узле данные известным образом могут быть восстановлены в сообщение, пригодное для чтения.

Кодирование сообщения в набор точек и тире – работа 2го уровня протокола. Передача набора точек и тире с помощью сигнала телеграфа – работ 1 уровня.

В идеализированной компьютерной сети присутствует 7 уровней протоколов: физический, канальный, сетевой, транспортный, сеансовый, представительский и прикладной.

Разрабатываемый на данном этапе протокол будет работать поверх стандартного протокола HTTP версии 1.1. Таким образом, HTTP, протокол седьмого уровня, служит транспортом протокола взаимодействия веб-сервиса. Наивная реализация схемы взаимодействия изображена на рис. 22.

Вернувшись к разрабатываемому проекту следует выделить основные задачи стадии разработки, для выполнения которых необходимо сетевое взаимодействие:

- получение «заглушки» STL-модели от сервера;
- получение «заглушки» модели из списка вершин от сервера;
- выполнение операции по преобразованию STL модели;
- «разбор» модели для получения списка вершин и нормалей.

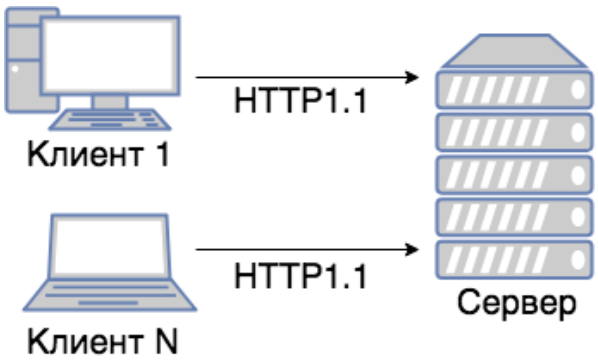

Рис. 22: Наивная схема взаимодействия

В настоящее время есть несколько подходов к построению архитектуры распределенных систем. Самые популярные подходы – REST и GraphQL.

REST (REpresentational State Transfer, передача состояния представления) – архитектурный стиль построения распределенных систем. Для этого стиля характерна слабая связь клиента и сервера. В идеале клиент может ничего не знать об устройстве сервера, кроме точки входа (entrypoint), и используемых медиатипов. Стиль REST предполагает наличие нижележащего протокола, от которого требуется:

- наличие конечных точек, т.н. эндпоинтов (endpoint), которые могут быть однозначно присвоены ресурсу;
- наличие методов, которые могут быть быть применены к эндпоинтам для совершения действий как над ресурсами, так и над передаваемыми данными;
- наличие гипермедиа, т.е. текста и ссылок на другие ресурсы.

Отдельно стоит отметить наличие HATEOAS (Hypermedia As The Engine Of Application State) – это устройство REST, при котором клиент и сервер (почти) независимы. А возвращаемые сервером ресурсы содержат связанные ресурсы и доступные переходы (ссылки). Пример HATEOAS с HTTP в качестве протокола: Отправляется POST запрос на создание учетной записи. Учетная запись создается и в ответе помимо информации о ней приходит ссылка на эндпоинт, по которому доступна учетная запись (GET /api/users/13), а также набор инструкций для смены статуса (POST /api/users/13/status) и удаления учетной записи (DELETE /api/users/13).

Хотя REST, как стиль, не зависит от протокола, наиболее популярен он с нижележащим протоколом HTTP. На месте HTTP, тем не менее, может быть и, например, FTP. REST не имеет стандартов и каждая сетевая служба по-своему строит REST архитектуру. Стандартизация REST следует из стандартов компонентов нижележащего HTTP, таких, как, например, безопасность и аутентификация в рамках HTTP.

Среди недостатков подхода REST отмечается невозможность управлять объемом данных, которые возвращаются с сервера. Данных может быть как в избытке, так и недостаточно, что приведет к новым запросам. Чтобы избежать этих проблем был разработан **GraphQL** (Graph Query Language) – язык описания взаимодействия клиента и сервера, когда клиент сам запрашивает нужный ему набор данных (например,

только год рождения пользователя, а не весь его профиль, как в REST). Вводятся запросы на чтение и т.н. мутацию, т.е. изменение ресурса.

Фронтенд из-за этого становится проще, а бэкенд усложняется в разы, так как надо поддерживать запросы разной сложности. Необходимо также следить за вложенностью запросов, так как она может стать причиной падения производительности. Кэширование заметно усложняется, т.к. одни и те же данные могут быть получены разными способами.

Схематичное сравнение REST и GraphQL изображено на рис. 23

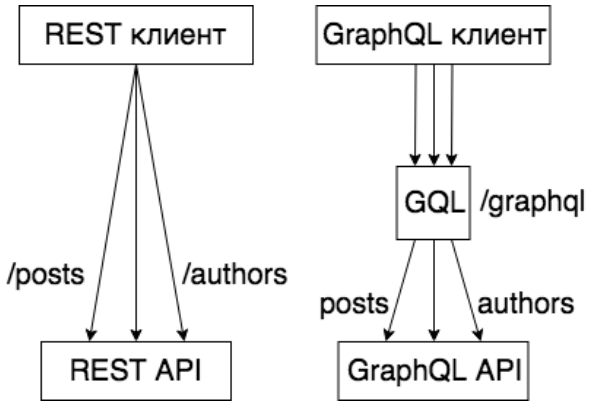

Рис. 23: Сравнение стилей REST и GraphQL

GraphQL постепенно набирает популярность и активно применяется в таких компаниях, как Twitter и Facebook. В рамках данной работы возможности GraphQL избыточны, а усложнение бэкенда, к которому ведет внедрение GraphQL не стоит упрощений фронтенда. Протокол взаимодействия прост, т.к. число решаемых задач невелико. Следовательно, предпочтение будет отдано REST'у.

Далее необходимо согласовать формат и содержание сообщений.

JSON и XML являются двумя популярными форматами (медиатипами), используемыми в сети интернет. **XML** многословен, сложен в описании, сложен в визуальном восприятии, а сериализация или десериализация может быть затруднена ввиду наличия множества схем, несовместимых друг с другом. Этот формат характерен для крупных устаревших корпоративных систем, переводить которые на новый формат слишком дорого. Пример таких систем – банковские системы или системы, применяемые в государственном секторе. Формат XML является устаревшим и на смену ему пришел лаконичный формат **JSON** (JavaScript Object Notation, нотация объектов JavaScript), удобный, как для восприятия человеком, так и для передачи данных по сети. В рамках данной работы будет использоваться именно он.

Наличие сразу двух способов получить mock-объект («заглушку» модели) связано с необходимостью проверки работоспособности сервера и механизмов обработки принимаемых клиентом данных, а поскольку разработка еще в процессе, неизвестно, какой из форматов представления модели лучше себя проявит.

После этого необходимо согласовать формат запроса и ответа для приведенных выше задач. Порядок следования сообщений неважен, т.к. считается, что за запросом всегда следует ответ, а сервер не хранит текущее состояние клиента (stateless). То

есть, порядок, в котором клиент выполняет действия не влияет на ответы сервера. Форматы запросов и ответов представлены на листингах ниже.

Получение STL-мока представлено на Листинге 17.

Листинг 17: HTTP запрос и ответ для получения STL-мока

```
// запрос
GET /api/stl/stub

// ответ
{
  res: < STLмодель-, строка Base64 >
}
```

Получение «мока» из списка вершин и нормалей для каждого треугольника представлено на Листинге 18. Подобную возвращаемую модель будем называть сеткой (mesh).

Листинг 18: HTTP запрос и ответ для получения mesh-мока

```
// запрос
GET /api/mesh/stub

// ответ
{
  len: 4,
  triangles:
  [
    {
      a: [< 3 координаты вершины >]
      b: [< 3 координаты вершины >]
      c: [< 3 координаты вершины >]
      n: [< 3 координаты вершины >]
    },
    ... // для каждого треугольника
  ]
}
```

Запрос на выполнение операции над STL-моделями представлен на Листинге 19. В запрос входят сразу 2 модели и название операции, которую библиотека обработки STL-моделей может выполнить над моделями. На данный момент можно считать, что все операции производятся только с двумя моделями. Поле operation запроса принимает одно из следующих значений:

- union;
- intersection;
- difference_ab;

- difference_ba.

Также, для простоты на этапе разработки считаем, что в ответе возвращается не более одной модели. Если результатом операции являются несколько моделей, то возвращается лишь первая из них. В дальнейшем это будет исправлено.

Листинг 19: HTTP запрос и ответ для выполнения операции

```
// запрос
POST /api/stl/perform
{
  stl1: < STLмодель-, строка Base64 >,
  stl2: < STLмодель-, строка Base64 >,
  operation: < строка, обозначающая производимую операцию >
}
// ответ
{
  res: < STLмодель-, строка Base64 >,
}
```

Запрос на экспорт модели в виде набора вершин и нормалей (без деления на треугольники) из STL-модели представлен на Листинге 20

Листинг 20: HTTP запрос и ответ для экспорта вершин и нормалей из STL-модели

```
// запрос
POST /api/mesh/extract
{
  stl: < STLмодель-, строка Base64 >
}

// ответ
{
  vertices: [< упорядоченный список координат вершин >],
  normals: [< упорядоченный список координат радиусвекторов- нормалей >]
}
```

Нештатные ситуации не регламентированы протоколом. На ошибки запроса со стороны клиента возвращаются соответствующие статусы HTTP ответа. В остальных случаях при успешной обработке запроса статус ответа – 200 OK, если не оговорено иное.

Таким образом, протокол взаимодействия разработан и следует перейти к разработке севера, поддерживающего этот протокол.

2.5 Разработка серверной части (бэкенд)

Клиент-серверная архитектура приложения предполагает распределение задач и нагрузки между поставщиками услуг –серверами и запрашивающими эту услуги клиентами. Серверная часть двухзвенной архитектуры по-другому называется

бэкенд (back-end), по аналогии с фронтендом (front-end). Для эффективной работы приложения и положительного пользовательского опыта необходима налаженная совместная работа двух частей приложения. Однако, если из «удобного» веб-сервиса убрать «удобный» фронтенд, веб-сервис потеряет в уровне пользовательского опыта, но продолжит работать. Если подобный продукт единственный на рынке, и бэкенд, производящий его по сути уникален, потеря будет несущественной ввиду отсутствия аналогов и конкуренции. Если же убрать бэкенд, веб-сервис перестанет приносить какую-либо пользу, т.к. именно бэкенд отвечает за производство бизнес-ценности (business-value). Поэтому, в некоторых случаях, его можно считать более важной частью, чем фронтенд.

Если из данной работы, например, убрать фронтенд (оставить минимально пригодное для использования клиентское приложение), приложение все также будет являться редактором. Если же убрать бэкенд, приложение перестает выполнять функции редактирования и остается лишь программой для просмотра STL-моделей без возможности выполнения каких-либо действий над ними.

В общем случае бэкенд разработка включает в себя разработку бизнес-логики, взаимодействие с сервером базы данных, интеграцию с внешними сервисами, сборку и размещение бэкенда на сервере приложений, масштабирование, а также всяческие оптимизации производительности и настройка безопасности.

В рамках данной работы за бизнес-логику отвечает внешняя библиотека обработки STL-моделей – Geometry Kernel (https://github.com/AsadiR/geometry_kernel). Эта библиотека разработана на кафедре ИУ-9 МГТУ им. Баумана в рамках совместной работы факультетов, описанной ранее и является магистерской работой Е. Копьева. Взаимодействие с библиотекой должно происходить через публичные методы API, которые «выставляет» наружу библиотека. Таким образом, задача бэкенда – организовать доступ к методам библиотеки извне. Это является достаточно простой задачей. Между тем, кроме выполнения операций над STL-моделями были заявлены возможность конвертации моделей из одного формата в другой и на время разработки – возврат объектов-заглушек для проверки работоспособности взаимодействия.

На данный момент (июнь 2018) публичные методы библиотеки для совершения операций над STL-моделями принимают на вход файлы и возвращают дескриптор файла с результатом операции. Таким образом, взаимодействие происходит через файловую систему, что ведет к постоянным I/O операциям (операции ввода-вывода). На данный момент исключить их невозможно, но в будущем такая возможность может появиться и следует заложить основу для возможного будущего изменения интерфейса взаимодействия.

Таким образом, появляется потребность в проработке дизайна бэкенда, так как присутствует не только взаимодействие с библиотекой, но и собственные дополнительные функции.

2.5.1 Проектирование архитектуры серверной части

Неотъемлемой частью процесса разработки программного обеспечения является его проектирование:

- проектирование позволяет использовать подходящие абстракции, которые наиболее точно отражают предметную область;
- качественный дизайн информативен и прост одновременно. Он предотвращает избыточность и способствует повторному использованию компонентов;
- на этапе проектирования становятся видны плюсы и минусы будущего ПО;
- будущая поддерживаемость, масштабируемость и развитие проекта могут быть заложены заранее.

Существует большое количество подходов к проектированию архитектуры. Необходимо изучить некоторые из них и выбрать наиболее подходящий паттерн (шаблон) проектирования. Подходы можно разделить по группам паттернов проектирования, которые они используют

Паттерны доменной логики отдельно выделяют предметную область приложения. Среди них можно выделить:

- **сценарий транзакции** (Transaction script) – паттерн, организующий бизнес-логику в процедуры. Приложение при этом может быть представлено, как набор транзакций. Внутри транзакций выполняются разные операции – от «простого» чтения до записи с множеством проверок в процессе. Вся логика при этом может быть организована внути одной процедуры, а взаимодействие с базой данных – либо напрямую, либо через тонкую обертку;
- **доменная модель** (Domain Model). Так как в общем случае бизнес-логика может быть очень запутанной, создаются так называемые доменные модели, представляющие отдельные части бизнес-процесса. В этих моделях скрыта логика и всевозможные бизнес-проверки. Размер модели может при этом варьироваться от модели, символизирующей целую компанию до модели, представляющей номер строки заказа;
- **сервисный уровень** (Service layer) – паттерн, определяющий границу между приложением и слоем сервисов. Приложения как правило взаимодействуют с другими приложениями через интеграционные шлюзы. Среди внешних интерфейсов к данным могут также располагаться загрузчики данных, пользовательский интерфейс и прочее. Как правило все эти интерфейсы требуют доступа к самому приложению, и чтобы избежать дублирования логики взаимодействия приложения и внешнего интерфейса вводится сервисный уровень, который всю эту логику инкапсулирует. Сервисный уровень при этом не хранит состояние между вызовами.

Паттерны источников данных выделяют логику работы приложения с сущностями. Примерами являются:

- **шлюз доступа к данным таблицы** (Table data gateway) – паттерн, скрывающий всю логику работы (например, CRUD) с SQL вручную в одном классе, для быстрого доступа к ней со стороны администратора баз данных (DBA);

- **активная запись** (Active record) – паттерн, при котором объект управляет и данными и поведением. Большинство данных при этом постоянны и их надо хранить в БД. Тогда в этот объект встраивается логика по изменению данных внутри сущности.

Среди **паттернов объектно-реляционной логики** можно выделить паттерн **Единица работы** (Unit of work): объект UOW следит за всеми изменениями, выполняемыми над объектами в рамках одного бизнес-действия и управляет записью изменений в базу данных и решением проблем конкурентности. UOW является своеобразным буфером, накапливающим изменения и затем фиксирующим их в базе данных. Без использования UOW потребовалось бы производить множество обращений к БД, что существенно замедлило бы работу приложения, либо понадобилось бы постоянно держать открытой транзакцию, что ограничило бы возможность параллельных обращений. Этот паттерн активно используется в ORM фреймворках, например, Hibernate.

Паттерны веб-представления адаптированы для использования в веб-приложениях, т.к. учитывают наиболее популярные сценарии их работы. Среди паттернов можно выделить следующие:

- **модель-представление-контроллер** (MVC, Model View Controller) – один из самых популярных паттернов, лежащий в основе многих современных фреймворков, поставляющих в комплекте пользовательский интерфейс. Разделяет приложение на 3 функциональные роли: модель данных, пользовательский интерфейс, управляющая логика. Это способствует слабой связанности компонентов и позволяет вносить изменения в один компонент, не затрагивая при этом остальные. Так, например приложение может иметь сразу несколько взаимозаменяемых моделей данных. Примеры подобных фреймворков – Spring MVC и Zend Framework;

- **контроллер входа** (Front controller) – один контроллер обрабатывает все запросы к серверу. В сложных веб-приложениях есть множество однотипных действий, которые надо проводить над входящим запросом. Это может быть проверка аутентификации, валидаций запроса, логгирование. Этот паттерн скрывает всю логику в одном объекте-обработчике, и, таким образом, логика не дублируется.

В списке выше приведено краткое описание популярных паттернов, которые так или иначе могут быть применены к веб-приложению. Некоторые из них касаются только дизайна серверной части, некоторые описывают архитектуру всей системы в целом. Паттерны так же могут комбинироваться. [6]

Паттерны источников данных, в первую очередь, описывают работу с базой данных. В рамках данной работы база данных существует на данный момент только в клиентской части приложения. Паттерны веб-представления могли бы использоваться, но они предполагают активное взаимодействие с клиентской частью, а следовательно частые соединения. Объем STL-моделей не позволяет использовать их достаточно эффективно. Паттерны объектно реляционной логики можно использовать, если количество операций большое, а сложность операций при этом невысока.

При наличии таких задач, как разбор модели на геометрические примитивы и конвертация из одного формата в другой может быть выделенная предметная область – домен, состоящий из геометрических примитивов. Тогда вся логика конвертации и разбора будет использовать примитивы доменного уровня. Доменный уровень при этом будет независим от вышележащих компонентов.

Таким образом, паттерн «Доменная модель» может быть применен при разработке системы в данной работе. Доменная модель активно используется в своеобразном расширении паттерна до методологии разработки – **Domain Driven Design** (DDD). Термин DDD, как методология разработки был введен в обращение Эриком Эвансом. Методология предполагает активную работу с предметной областью, а также выстраивание дополнительных слоев с бизнес-логикой приложения и логикой, специфичной для конкретных используемых технологий поверх нее.

DDD не привязан к конкретной технологии, что является существенным плюсом, т.к. многие описанные выше паттерны или методологии требует наличия компонентов системы, которые присутствуют не во всех технологиях.

Минусом является сложность соблюдения DDD, т.к. этот подход существенно отличается от всех остальных, но т.к. предметная область данной работы не объемна, его реализация возможна.

Первым дело следует выделить **предметную область (доменную модель)**. Ею, как было замечено ранее, является в общем смысле «геометрия» (медицинское ее применение является лишь частным случаем и не составляет доменную модель). Среди доменных моделей могут быть выделены такие геометрические примитивы, как точка, треугольник и сетка (набор треугольников), т.е. модели, с которыми работает, как клиентское приложение (приложение способно отображать в трехмерном пространстве точки, прямые и треугольники по точкам, а также сетки из наборов треугольников), так и библиотека обработки моделей. Считается, что приложение всегда работает только с 3D графикой, а 2D представление является ее подмножеством, где одна из координат зафиксирована.

Будучи выделенными в отдельный слой, модели доменного уровня практически не изменяются:

- точка всегда является точкой в пространстве, т.е. упорядоченным набором из трех координат;
- вектор всегда будет вектором, то есть кортежем из 2х точек;
- треугольник – упорядоченный набор из трех вершин-точек и дополнительный вектор нормали, определяющий «лицевую» сторону треугольника.

Отдельным плюсом доменной модели в данной работе можно считать ее «неприкосновенность». Действительно, модель, представляющая, например, заказ в интернет-магазине или пациента в медицинском приложении рано или поздно будет изменена, т.к. потребности бизнеса могут измениться в любой момент. В свою очередь, модель, представляющая геометрический объект в декартовых координатах не может быть изменена. Изменение декартовой системы координат на, например, полярные координаты не рассматривается в силу очевидных причин.

Важно отметить, что модели ничего «не знают» об уровнях выше.

Далее следует определить **слой приложения** (Application layer). Этот слой содержит основную бизнес-логику приложения. Он оперирует моделями слоя ниже и полностью полагается на них. Об уровнях выше слой приложения также ничего не знает.

С технической стороны слой приложения представляет из себя набор сервисов – компонентов, выполняющих логику, но не хранящих состояние между выполнениями.

Применительно к данной работе, функция слоя приложения – преобразование одного формата представления модели в другой, то есть разбиение набора координат на точки в пространстве, набора точек на треугольники, добавление нормалей к треугольникам и т.д. Другая задача слоя приложения – выполнение операций над STL моделями. Библиотека обработки STL оперирует понятиями геометрии только «внутри». Внешняя ее часть, доступная разработчику оперирует лишь файлами.

Далее следует **слой порт-адаптеров**. Этот слой «адаптирует» уровень приложения под используемую технологию (порт). Пример: приложение оперирует учетными записями пациентов. Доменной моделью является учетная запись. Приложение «умеет» создавать, обновлять и удалять учетные записи. Технически это реализуется наличием интерфейса на слое приложения, т.е. декларации действия над учетными записями, с методами create(account), update(account), delete(account). Предположим, что создание учетной записи (create(account)) равносильно вставке учетной записи в таблицу базы данных. Реализация этого интерфейса в таком случае должна учитывать используемые в проекте технологии работы с базами данных. Если используется MongoDB, то последовательность действий будет одна:

1. сериализовать учетную запись в JSON;
2. открыть соединение с БД;
3. добавить документ в коллекцию Patients;
4. закрыть соединение.

Если же используется PostgreSQL (без ORM), то будет другая последовательность действий:

1. сформировать SQL запрос;
2. открыть соединение с БД;
3. открыть транзакцию;
4. выполнить запрос;
5. закрыть транзакцию;
6. закрыть соединение.

Это дает ощутимое преимущество: при, например, миграции базы данных нет необходимости менять логику работы с учетными записями. Сами учетные записи тоже не изменятся. Изменение на каком-либо уровне не затрагивает нижележащие уровни.

В данной работе на бэкенде нет необходимости «адаптироваться» к базе данных, но есть необходимость адаптироваться для выполнения запросов от клиентской части приложения. Таким образом, слой порт-адаптеров позволит использовать приложение извне, обрабатывая HTTP запросы и перенаправляя их на уровни ниже.

Адаптация к взаимодействию по HTTP в рамках REST архитектуры имеет особенность: ключевые компоненты слоя адаптеров должны реализовывать REST стиль. Значит, можно выделить:

- **ресурсы** – являются по сути реализацией конечных ресурсов, описанных в REST. Это эндпоинты, по которым можно взаимодействовать с сервисом;

- **команды** – отображения входящих запросов на структуры, используемые в языке реализации бэкенда. Команда – входной параметр ресурса. Запросы в данном случае приходят в формате JSON и *команды* нужны, чтобы разграничить часть с десериализацией JSON в структуры языка (десериализация как правило происходит автоматически) и обработку десериализованной команды в ресурсе;

- **модели** – возвращаемые бэкендом данные. По аналогии с *командами* модели являются отображением возвращаемых ресурсом результатов в JSON объекты, используемые при передаче данных. Моделями уровня порт-адаптер могут быть модели доменного уровня, так как правила иерархии доступа это позволяют, но при взаимодействии может потребоваться дополнительная информация, не входящая в доменную модель и появляющаяся лишь на этапе обработки на уровне приложения. Это может быть, например, размер модели, получившейся в результате операции или время выполнения операции, или у треугольников может быть добавлен цвет.

С точки зрения иерархии уровень порт-адаптеров формально не отличается от других уровней: можно использовать нижележащие уровни (приложение и модели), использовать вышележащие уровни запрещено. Отличием является лишь то, что этот уровень – верхний и «выше» него уже ничего нет.

В итоге внешний и последний уровень бэкенд архитектуры описан и становится видна структура приложения в соответствии с подходом Domain Driven Design. Схематично приложение представлено на рис. 24.

В общем смысле, как следует из рис. 24, дизайн DDD «навязывает» лишние конвертации объектов одного уровня в другой. С одной стороны это несколько усложняет разработку, т.к. соблюдение подхода требует четкой иерархии уровней

На данный момент имеется продуманный дизайн приложения, который будет способствовать простоте поддержки и дальнейшего развития (благодаря дизайну). Стоит также отметить то, что в ходе описания дизайна не была упомянута конкретная его программная реализация. При этом уже на стадии проектирования понятно, *как именно* сервис будет функционировать. Это является существенным плюсом, так как

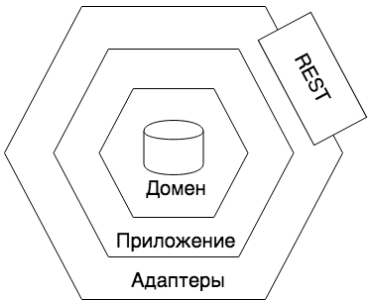

Рис. 24: Структура приложения в соответствии с Domain Driven Design

позволяет проектировать приложение с точки зрения бизнес-потребностей: решаемая задача важнее используемых технологий.

2.5.2 Программная реализация

По завершении проектирования дизайна необходимо приступить к его программной реализации. Как было отмечено ранее, языком реализации бэкенда выбран Rust для эффективного взаимодействия с библиотекой обработки STL-моделей. Разработка с использованием Rust имеет множество особенностей и отличается от разработки на большинстве других языков.

Rust является языком системного программирования, ориентированным на многопоточность, безопасность и высокую скорость работы. Rust активно развивается сообществом и компанией Mozilla, которая использует язык в ядре своего браузера Firefox.

Синтаксически язык имеет много общего с C++, но при этом дает больше гарантий безопасности, почти не уступая при этом C++ в скорости работы. Язык поддерживает множество парадигм: императивное программирование, функциональное, конкурентное. [4]

Можно выделить следующие особенности Rust:

- **гарантия безопасности памяти**: язык не имеет автоматической сборки мусора. Вместо этого используется модель RAII (Resource Acquisition Is Initialization) – получение ресурса есть инициализация, а освобождение есть удаление. С помощью проверки указателей во время компиляции решается проблема висячих указателей (dangling pointers) и неопределенного поведения (undefined behavior). Состояние гонки также исключено;

- **семантика заимствования**: у объекта только один владелец. Допустимо множество неизменяемых ссылок на объект и только одна изменяемая (мутабельная);

- **вывод типов и полиморфизм**: система типов поддерживает полиморфизм с помощью т.н. трейты (trait, подмешивание), вдохновленные языком Haskell. В противоположность другим языкам требуется явно указывать изменяемые переменные, т.к. все переменные по-умолчанию неизменяемы;

- использование **небезопасного кода** включается ключевым словом unsafe. Гарантом безопасности в таком случае должен выступить сам разработчик, а не компилятор. Эта возможность используется, например, при использовании FFI-функций (Foreign Function Interface), т.к. компилятор не может «проверить» функции другого языка.

При использовании Rust большинство ошибок решается на этапе разработке и можно считать, что успешная компиляция, с большой долей вероятности, означает успешное выполнение, т.к. компилятора языка Rust «строже» компиляторов других языков и заставляет разработчика задумываться о многих аспектах выполнения кода, которые в других языках скрыты и происходят автоматически. Все это повышает безопасность и стабильность работы программ. «Кривая обучения» при этом существенно круче и на начальных этапах обучения это уменьшает скорость разработки.

В рамках данной работы Rust используется в качестве бэкенд-языка и именно с его помощью будет разработана серверная часть приложения. Rust, как и большинство языков, имеет специализированные веб-фреймворки для реализации веб-сервисов. О фреймворках было рассказано ранее в данной работе.

В качестве веб-фреймворка был выбран Rocket. Он обладает следующими достоинствами:

- строгая типизация запроса и ответа;
- минимум boilerplate кода. Весь необходимый код порождается на этапе компиляции;
- расширяемость благодаря механизму дополнений.

«Платой» за использование Rocket является переход на Nightly версию Rust, стабильность которой не гарантируется – это особенность кодогенерации в Rocket. После того, как выбран фреймворк, следует приступить к программной реализации дизайна, описанного ранее. Таким образом, на данный момент имеется ряд требований, порожденных дизайном и ряд, правил, «навязываемых» фреймворком.

На Листинге 21 частично представлена функция main, запускающая приложение, а также конфигурация эндпоинтов.

Листинг 21: Конфигурирование фреймоврка и запуск

```
#[post("/perform", format = "application/json", data = "<cmd>")]
fn mesh_performer_wrapper(cmd: Json<PerformOnMeshCommand>) -> Json<MeshModel> {
    perform_bool_op_on_meshes(cmd)
}

fn main() {
    rocket::ignite()
        .mount("/api/mesh", routes![
            extract_mesh_wrapper,
            mesh_stub_wrapper,
```

```
            mesh_performer_wrapper
        ])
        .attach(CORS())
        .launch();
}
```

На Листинге 21 символ # означает атрибут, присваиваемый следующему за ним элементу. Синтаксис атрибутов позаимствован из языка C#. В данном случае функция становится эндпоинтом по указанному URL'у, принимающим POST-запросы в формате JSON и возвращающим данные также формате JSON. Типы данных PerformOnMeshCommand и MeshModel (Листинг 22) имеют описанную ранее структуру, согласованную при проектировании взаимодействия клиентской и серверной части.

Описанное ранее взаимодействие уровней приложения согласно DDD методологии представлено на примере структуры MeshModel (STL-модель, как набор треугольников) на Листинге 22. MeshModel имеет атрибуты Serializable, Deserializable, что делает ее JSON-сериализуемой и десериализуемой.

Листинг 22: Преобразование треугольников доменного уровня в сетку (mesh) уровня порт-адаптера

```
#[derive(Serialize, Deserialize)]
pub struct MeshModel {
    pub len: u32,
    pub triangles: Vec<TriangleModel>
}

impl MeshModel {
    pub fn from_triangles(triangles: Vec<Triangle>) -> Self {
        let models: Vec<TriangleModel> = triangles.iter()
            .map(|triangle| TriangleModel::from_triangle(*triangle))
            .collect();
        MeshModel::from_triangle_models(models)
    }

    pub fn from_triangle_models(triangle_models: Vec<TriangleModel>) -> Self {
        MeshModel{ len: triangle_models.len() as u32, triangles: triangle_models }
    }
}
```

На Листинге 23 представлена итоговая структура серверной части проекта. Доменный уровень представлен без изменений тремя элементами: точка пространства, треугольник и сетка (набор треугольников).

Уровень приложения представлен сервисом, выполняющим конвертацию из бинарного и текстового STL-формата и обратно и сервисом, взаимодействующим непосредственно с библиотекой, производящей операции. На уровне приложения без изменений используются доменные модели уровня ниже.

Уровень порт-адаптера представлен лишь частично ввиду его большего объема. Здесь также соблюдены поставленные ранее требования:

- модуль **command** содержит структуры для отображения входящих запросов (команд) в формате JSON на структуры языка Rust;
- модуль **model** содержит возвращаемые бэкендом модели: здесь также представлены структуры языка Rust, отображаемые впоследствии в тело ответа в формате JSON;
- модуль **resource** содержит эндпоинты, принимающие команды модуля command и возвращающие модели модуля model.

Листинг 23: Структура проекта в соответствии с DDD дизайном

```
src
|__ main.rs
|__ lib.rs
|__ application
| |__ operation_performer
| | |__ bool_op_performer.rs
| |__ stl_reader
|     |__ abstract_stl_reader.rs
|     |__ ascii_stl_reader.rs
|     |__ binary_stl_reader.rs
|__ domain
| |__ model
|     |__ mesh.rs
|     |__ point.rs
|     |__ triangle.rs
|__ port_adapter
  |__ command
  | |__ perform_on_mesh_command.rs
  |__ model
  | |__ mesh_model.rs
  |__ resource
      |__ stl_resource.rs
```

Запуск приложения осуществляется с помощью **Cargo** – пакетного менеджера языка Rust (по аналогии с Maven и Gradle – менеджерами в языке Java и NPM – менеджером в языке JavaScript в экосистеме Node.js), берущего на себя всю ответственность за включение внешних библиотек в проект. При работе с внешними библиотеками Rust (и, соответственно, Cargo) оперирует таким понятием, как crate, что фактически является библиотекой из регистра библиотек crates.io. Cargo используется также для сборки проекта, делегируя компиляцию исходного кода проекта rustc – компилятору Rust.

Применение Cargo способствует стандартизации структуры проекта:

- проект имеет файл cargo.toml, представленный в табличном формате TOML (Tom's Obvious Markup Language). В этому фале перечислены все зависимости проекта и указаны их версии, чтобы предотвратить появление ощибок, вызванных отуствием обратной совместимости;
- весь исходный код расположен в директории src. Язык Rust использует модульную структуру и все директории внутри src являются корневыми модулями, которые должны быть «зарегистрированы» в src/lib.rs;
- файл src/lib.rs используется в качестве описания текущего проекта, как crate'а и для объявления внешних библиотек в качестве используемых crate'ов;
- файл src/main.rs – файл, добавляемый Cargo к запускаемым отдельно (standalone) проектам. В нем должна быть объявлена функция main.

В большинстве случаев проекты используют внешние зависимости, а значит используют и Cargo. Работа с компилятором Rust напрямую в большинстве случаев не производится. Тем не менее, чтобы увидеть вызов rustc с помощью Cargo используется следующая команда:

```
$ cargo build -verbose
```

Запуск приложения выполняется следующей командой:

```
$ cargo run
```

После этого серверная часть приложения запускается и способна принимать входящие запросы.

На этом этап разработки может считаться завершенным. Дальнейшая работа будет проводиться с целью исследования недостатков «наивной» реализации, а также поиска способов их устранения.

2.6 Оценка наивного подхода

На данный момент считается, что есть рабочая версия проекта: клиентское приложение и REST API, предоставленный серверной частью. Далее необходимо оценить проведенную работу. Оценить можно как клиентскую часть, так и серверную часть. Ввиду невозможности привлечения эксперта предметной области, как было отмечено ранее, оценка работы будет произведена с технической стороны.

Клиентскую часть веб-сервиса представляет приложение, написанное на JavaScript без использования фреймворков. Использовались возможности современной версии языка, соответствующей стандарту ES6. Верстка страницы осуществлялась с помощью HTML5, а визуальная часть была разработана с использованием SASS. Полученное клиентское приложение фактически является статической страницей с внедренной в него шейдерной WebGL графикой. Сетевое взаимодействие осуществляется с помощью функций асинхронных запросов, предоставленных библиотекой AXIOS.

Серверная часть разработана с использованием фреймворка Rocket, написанного на языке Rust. Бэкенд написан в соответствии с паттерном Domain Model и с соблюдением методологии Domain Driven Design.

Для совместной работы клиентской и серверной части был разработан протокол, базирующийся не текстовых сообщениях в формате JSON и использующий в качестве «транспорта» протокол HTTP.

В общем смысле проект представляет из себя «классическое» веб-приложение. Однако, присутствует специфика, «порожденная» предметной областью – работа с именно STL-моделями. Устройство STL формата было рассмотрено ранее и оно эффективно сочетается с возможностями шейденой графики. Далее представлен ряд проблем выявленных на этапе разработки и тестирования.

Объем, требуемый для точного описания модели велик и это вызывает необходимость выбирать между точностью представления и объемом файла. Подобный недостаток неустраним и является спецификой формата описания;

Обработка моделей на бэкенде и передача моделей по сети являются «проблемными» частями: объем файла размером N байт после Base64-кодирования составляет $\frac{N}{3} \times 4$ байт, что увеличивает нагрузку на сеть. Base64-кодирование, если рассматривать ее как технику, способствующую увеличению размера файла и, следовательно, нагрузки, является неустранимым «недостатком». Усугубляет проблему так же и тот факт, что при проведении операции отправляется сразу несколько моделей. 2 модели размером в 100 МБ каждая в процессе передачи будут занимать порядка 270 МБ. Это *колоссальный* объем информации по меркам современных веб-приложений, большинство из которых оперирует запросами в килобайты и десятки килобайт. В конечном итоге это приводит к медленному выполнению операций и негативному пользовательскому опыту. Компенсирует эту проблему лишь асинхронность работы клиентского приложения: после отправки моделей на выполнение операции работа может продолжена, т.к. этот процесс неблокирующий. При получении результата выполнения, получившаяся модель будет добавлена в OpenGL сцену. Однако, неспособность используемых сетевых протоколов быстро передать файл является устранимым недостатком и решение проблемы будет предложено при рассмотрении возможностей оптимизации проекта. На данном этапе важно лишь выявить имеющиеся недостатки;

Взаимодействие с библиотекой выполнения операций Geometry Kernel. Так, большинство публичных методов библиотеки работает с файлами, т.е. взаимодействует с файловой системой. Это является недостатком в силу того, что протокол взаимодействия не может быть оптимизирован под библиотеку, т.к. библиотеке требуется файл для выполнения операции. Данный недостаток не может быть устранен в рамках данной работы, т.к. Geometry Kernel используется, как «черный ящик», модификация которого невозможна. Тем не менее, Geometry Kernel является библиотекой с открытым исходным кодом и дополнительная функциональность, не предоставленная автором, может быть добавлена посредством запроса на слияние (pull request) при условии согласия на это автора библиотеки;

Веб-фреймворк Rocket, отвечающий за работу сервера **минималистичен**. Это способствует низкому порогу вхождения и простоте использования. Тем не менее

минималистичность затрудняет тонкую настройку работы, т.к. множество функций не поставляется «из коробки» (в противовес, например, фреймворку Django). Так, в Rocket отсутствует асинхронность, поддержка HTTP/2, а количество плагинов для расширения функциональности невелико;

Последним, на что стоит обратить внимание являются **высокие аппаратные требования к оборудованию**, на котором выполняются операции на STL моделями. Эта особенность специфична для конфигурации компьютера, на котором производится разработка и тестирование системы: Intel Core i5 1.6 GHz, Intel HD Graphics 6000, 8Gb DDR3, macOS. После проведения тестирования, был сделан вывод о том, что подобная конфигурация не способна удовлетворить потребности библиотеки в вычислительных ресурсах. Проведение операции над двумя STL моделями, каждая из которых имеет объем порядка 1,5 МБ, занимает несколько минут. Библиотека выполняет ресурсоемкие (CPU-intensive) вычисления. В «реальных» условиях при недостаточно производительном оборудовании, на котором запущена серверная часть, низкая скорость работы может способствовать негативному пользовательскому опыту. Благодаря тому, что приложение состоит из двух компонентов, пользователю не требуется располагать существенными вычислительными мощностями для его использования. Так, оборудование бэкенда может быть заменено на существенно более мощное, способное обрабатывать большое количество запросов, выполняя для каждого запроса свою CPU-intensive операцию по обработке STL-моделей. В случае разработки настольного приложения это могло стать существенным недостатком.

В ходе разработки и последующего тестирования выявлены недостатки «наивной» реализации проекта. Далее будут предложены и применены варианты оптимизации проекта.

3 ОПТИМИЗАЦИЯ ВЗАИМОДЕЙСТВИЯ КОМПОНЕНТОВ

Оптимизация является неотъемлемой частью полного цикла разработки приложения наравне с поддержкой и развитием. В данной работе целью оптимизации является устранение ранее выявленных недостатков. Так как не все недостатки могут быть устранены в силу очевидных причин, процесс оптимизации затронет только самые существенные из них.

3.1 Оптимизация сетевого взаимодействия

Наиболее существенным из устранимых недостатков является сетевое взаимодействие. Подобная проблема свойственна всем распределенным приложениям. Отличия лишь в объемах передаваемой информации. Для одностраничных приложений, рассмотренных ранее на примере разработки приложения на React, объем передаваемой информации исчисляется десятками и сотнями килобайт. Протокол HTTP с легкостью справляется с таким незначительным объемом информации.

В разработанном приложении объем передаваемой информации исчисляется мегабайтами и десятками мегабайт. Даже при наличии высокоскоростного соединения с сетью интернет, отправка запроса величиной в десятки мегабайт и получение десят-

ков мегабайт обратно занимает некоторое время. Таким образом, проблему **низкой скорости передачи большого объема данных** можно разделить на 2 части и сформулировать 2 решения:

1. увеличение пропускной способности канала;

2. уменьшение передаваемого объема данных.

Таким образом, проблема сформулирована и декомпозирована на 2 подзадачи. Решение первой задачи сводится к использованию более современного протокола.

3.1.1 Протокол HTTP/2

Как было установлено ранее, HTTP является протоколом прикладного уровня, то есть используемым при пользовательском взаимодействии. При этом существует еще целый набор протоколов уровней ниже, на которые опирается HTTP и которым, соответственно делегирует работу. Поскольку в стеке TCP/IP не реализованы 5 и 6 уровни протоколов (сеансовый и представительский), ближайшим к HTTP протоколом является TCP. Это протокол транспортного уровня и считается, что прямого доступа к нему нет. Протоколы уровней ниже настроены, как правило, непосредственно в оборудовании, так что к ним тоже нет доступа. Таким образом, единственным доступным и заменимым протоколом является HTTP.

Используемый сейчас протокол HTTP версии 1.1 был представлен в 1999 году как развитие протокола HTTP (и позднее HTTP1.0), представленного в 1989 году Тимом Бернерсом-Ли и на данный момент является самым популярным сетевым протоколом прикладного уровня. HTTP/1 является протоколом, не сохраняющим состояние (stateless) – все запросы атомарны и не связаны с предыдущими запросами (для введения состояния между запросами используется технология Cookie, рассмотренная ранее как способ хранения информации). Передача информации при этом всегда начинается с запроса от клиента, а сервер лишь отвечает на запросы. Подобная имплементация является достаточно низкоуровневой и во избежание проблем любые оптимизации должны иметь обратную совместимость с HTTP/1. [7]

В итоге такой «жестко определенный» механизм работы с атомарностью и синхронностью запросов стал узким местом протокола: в момент получения браузером страницы происходит парсинг страницы на предмет дополнительных ресурсов, требуемых для рендеринга текущей страницы. Это могут быть как CSS, JavaScript, так и картинки и прочие ресурсы. Как только бразуер находит подобный ресурс, он останавливает загрузку страницы и обращается к серверу с целью получить найденный ресурс. До тех пор, пока сервер не вернет ответ, браузер останавливает вывод страницы. Так обходится и запрашивается каждый ресурс. В этом заключается синхронная модель загрузки и пока эта загрузка продолжается пользователь видит пустую страницу или загруженный на половину сайт.

Разработчиками из Mozilla было отмечено, что первая версия протокола не использует все возможности нижележащего протокола – TCP. Таким образом, была предложена идея конвейерной загрузки (HTTP pipelining) – отправки большего количества ресурсов за одно TCP соединение. Это существенно снижало время необходимое на загрузку, но к текущему моменту (2018 год) большинство браузеров не

поддерживают подобное решения по причине проблем с proxy-серверами и Head-of-Line блокировок пакетов. На рис. 25 слева представлена общепринятая схема работы, а справа предложенное (и отвергнутое) улучшение.

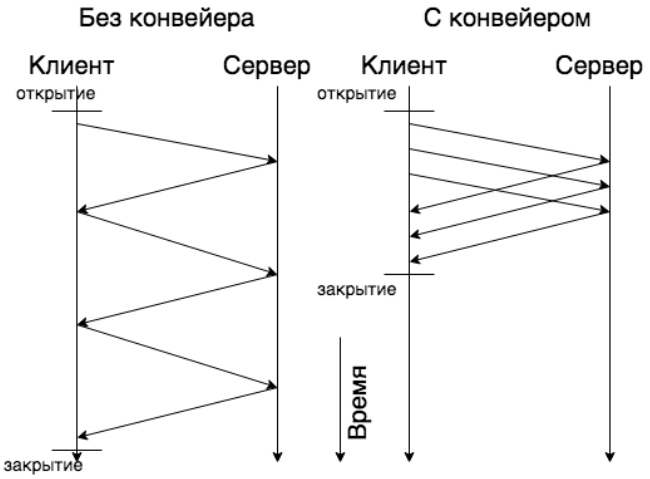

Рис. 25: Сравнение обычной и конвейерной загрузки ресурсов

Так как оптимизация, представленная на рис. 25 так и не была принята, каждый ресурс по-прежнему требует новое TCP соединение и выполнение сопутствующих действий, таких как рукопожатие, поиск DNS и т.д.

В 2009 году Google представил проект протокола нового поколения – **SPDY**, добавив его поддержку в Google Chrome и перейдя на его использование в своих сервисах. Следующим протокол принял Twitter, далее – Apache, nginx, Node.js.

SPDY ввел **мультиплексирование** – параллельную отправку запросов за одно TCP соединение. Соединение при это защищено, а данные сжаты. Введение SPDY на 25 наиболее популярных сайтах ускорило их работу на 27-60%. В дальнейшем 3я версия SPDY послужила основой проекта HTTP/2.

Особенностями HTTP/2 являются:

- сжатие HTTP заголовков;
- внедрение Server Push;
- мультиплексирование запросов за одно соединение;
- разрешение проблемы Head-of-Line блокировки пакетов.

Данные в HTTP/2 отправляются в бинарном формате, а сжатие заголовков осуществляется в помощью алгоритма HPACK. Это существенно уменьшает размер запроса. TLS защита данных требуется большинством браузеров для использования протокола. Технология Server Push позволяет передавать ресурсы браузеру пользователя еще до того, как он их запросил.

На данный момент большинство веб-браузеров поддерживает HTTP/2 лишь при наличии TLS несмотря на то, что использование TLS опционально согласно спецификации. Серверная часть требует бОльших изменений и поэтому лишь небольшое

число веб-фреймворков поддерживает новый стандарт. Rocket, используемый в данной работе поддержки HTTP/2 не имеет.

Тем не менее оптимизация пропускной способности канала, обозначенная ранее, может быть проведена. Так как канал соединяет клиентскую и серверную часть, протокол HTTP/2 должен поддерживаться обеими частями. При этом, поскольку бразуер управляет всем исходящими соединениями, вопрос использования HTTP/2 также решается им. Благодаря тому, что стандарт внедрен достаточно давно, его использование сервером является ожидаемым для браузера, и таким образом, если HTTP/2 может быть использован, он *будет использован*. Возможность использования протокола «обсуждается» при первичном установлении соединения или с помощью дополнительного OPTIONS запроса. Этот механизм называется согласованием схемы (scheme negotiation) и реализован в браузере и может варьироваться, разработчику клиентской части при этом нет необходимости производить какие-либо изменения.

Серверная часть требует существенной доработки путем введения прокси-сервера в схему взаимодействия из-за отсутствия нативной поддержки HTTP/2 Rocket'ом. Эта доработка будет рассмотрена далее в работе.

Таким образом, пропускная способность способность будет увеличена и первый пункт оптимизации может считаться завершенным для серверной части. Далее следует рассмотреть возможность уменьшения объема передаваемых данных – второй пункт оптимизации.

3.1.2 Сжатие данных

Для корректной передачи данных используется «кодирование» в формат Base64. Эта конвертация увеличивает размер передаваемых данных на треть, что вкупе с большим объемом данных существенно снижает скорость передачи. Отказ от формата Base64 не рассматривается ввиду его «стандартизованности» и простоты работы с ним. Следующим решением является сжатие данных. Перед внедрением сжатия необходимо изучить имеющиеся решения и затраты на их использование.

HTTP сжатие является техникой уменьшения объема данных при передаче для эффективного использования сети. Данные сжимаются до отправки и только в том случае, если клиент явно укажет поддержку сжатия. Клиенты, не способные принять сжатый контент, тем не менее, получат данные в изначальном виде. Клиентами, как и ранее, являются браузеры.

Наиболее популярными схемами сжатия являются **gzip** и **deflate**. При этом gzip использует алгоритм Deflate, а Deflate в свою очередь является комбинацией алгоритмов LZ77 и алгоритма Хаффмана. [9] Алгоритм LZ77 (назван в честь создателей – А. Лемпеля и Я. Зива, разработавших алгоритм в 1977 году) является алгоритмом «сжатия без потерь» и заменяет повторное вхождение данных на ссылку на первый экземпляр вхождения. Ссылка при этом характеризуется смещением и длиной. Например, следующий текст представлен 81 байтом:

ServerGrove, the PHP hosting company, provides hosting solutions for PHP projects

После применения алгоритма LZ77 (путем замены повторяющихся частей на ссыл-

ки) представляется 73 байтами:

ServerGrove, the PHP hosting company, p<3,32>ides<9,26>solutions for<5,52><3,35>jects

Алгоритм Хаффмана применяется к данным, представленным в бинарном виде: более частым символам назначаются более короткие коды. Алгоритм также разрешает проблему с разделением кодов (конец i-го и начало $(i+1)$-го кода).

Deflate является комбинацией этих двух алгоритмов. Несмотря на то, что Deflate (gzip) не является лучшим методом сжатия, он является оптимальным выбором в силу двух причин:

1. обеспечивается компромисс между степенью сжатия и скоростью процесса сжатия;

2. механизм сжатия является де-факто стандартом и внедрен во все современные браузеры.

Согласование схемы сжатия в рамках HTTP тривиально и выполняется в 2 этапа. Клиент отправляет серверу HTTP-запрос с заголовком Accept-Encoding, значение которого соответствует поддерживаемым схемам. В большинстве случаев это значение равно «gzip, deflate». Далее веб-сервер, найдя заголовок Accept-Encoding считывает схемы, которые доступны клиенту и, если среди них есть поддерживаемая сервером схема сжатия – применяет ее. В HTTP-ответе при этом указывается примененная схема сжатия добавлением заголовка Content-Encoding: gzip.

При этом, как и в случае выбора протокола взаимодействия, все действия со стороны клиента (согласование схемы, добавление заголовка и восстановление сжатых данных) совершаются бразуером автоматически, что упрощает работу разработчика клиентской части.

Таким образом, передаваемый объем данных будет уменьшен в случае, если сервер будет поддерживать сжатие данных. Сжатие данных не поддерживается Rocket'ом и, следовательно, появляется еще одно требование к вводимому прокси серверу – поддержка gzip. Процесс изменения схемы взаимодействия будет рассмотрен далее.

3.1.3 Реверс-прокси сервер

На данный момент клиентская часть оптимизирована в части сетевого взаимодействия: браузер стремится максимально эффективно утилизировать сеть и для этого использует наилучший из доступных протоколов и по возможности сжимает данные. Тем не менее, для использования и правильного функционирования этих технологий, серверная часть должна «уметь с ними работать». Следовательно, можно сформулировать 2 требования к серверной части приложения:

1. поддержка SPDY и HTTP/2;

2. сжатие данных gzip, deflate.

Как было отмечено ранее, веб-фреймворк Rocket, используемый на данный момент в серверной части, не удовлетворяет ни одному из предъявляемых требований. Следовательно, работу по «обеспечению» протокола и сжатия необходимо делегировать другому компоненту системы. Этим компонентом может являться реверс-прокси-сервер.

Прокси сервер (далее – прокси) – промежуточный слой между клиентом, выполняющим запросы к ресурсам сервера и самим севером. При этом прокси действует от собственного лица, скрывая клиента, при этом принимая запросы клиента, отправляя собственные запросы целевому серверу и возвращая ответы целевого сервера клиенту. Подобная схема взаимодействия решает несколько задач:

- кэширующий прокси-сервер ускоряет взаимодействие;
- безопасность клиента выше, т.к. он «скрыт» за прокси-сервером;
- возможность доступа к заблокированной для клиента информации (например, блокировка веб-сайтов внутри корпоративной сети компании или государственная цензура).

Прокси используется клиентом для доступа к ресурсам. **Реверс-прокси**, в свою очередь, является сервером, проксирующим входящие запросы на другие ресурсы или внутрь сети. Во втором случае реверс-прокси скрывает реальное устройство внутренней сети.

Реверс-прокси в частной сети, как правило, «выставлен наружу» и перенаправляет запросы клиентов на соответствующие бэкенды. Это вводит дополнительный уровень абстракции, позволяя эффективнее управлять трафиком между клиентами и серверами.

Основные задачи реверс-прокси:

- балансирование нагрузки – распределение запросов между группой серверов для в соответствии с некоторыми правилами. Это позволяет эффективно использовать сеть, при этом гарантируя, что ни один отдельный сервер не будет перегружен (что могло бы привести к падению производительности);
- ускорение работы путем сжатия входящих и исходящих данных, а также кэширование и обеспечение безопасности трафика путем шифрования;
- безопасность и анонимность: путем перехвата запросов к бэкенд серверам, реверс-прокси скрывает реальное устройство сети, выступая дополнительным уровнем защиты.

Можно заметить, что второй пункт списка задач, решаемых реверс-прокси содержит требования, предъявляемые к бэкенду в рамках данной работы. Следовательно, введение прокси оправдано.

Примером реверс-прокси является **Nginx**. Это решение используют в «боевых» условиях множество веб-сайтов и оно зарекомендовало себя, как быстрая и эффективная реализация прокси-сервера.

Nginx использует асинхронную событийно-ориентированную модель при обработке запросов, а среди достоинств стоит выделить следующие:

- поддержка 10000 одновременных соединений с расходом памяти в 2,5 МБ на 10 тысяч открытых соединений;
- кэширование запросов;
- балансировка нагрузки;
- поддержка TLS/SSL;
- поддержка веб-сокетов;
- совместимость с IPv6.

Таким образом, и поддержка gzip, и поддержка протокола HTTP/2 реализованы в Nginx «из коробки».

Nginx конфигурируется из файла nginx.conf и имеет статическую конфигурацию, что является одним из немногих недостатков, т.к. после изменения конфигурации требуется перезагрузка веб-сервера. Пример конфигурационного файла представлен на Листинге 24.

Листинг 24: Часть конфигурационного файла Nginx

```
worker_processes 2;
events {
    worker_connections  1024;
}

http {
    server {
        listen      443 ssl http2 default_server;

        ssl_certificate      cert/cert.crt;
        ssl_certificate_key  cert/cert.key;

        gzip on;
        gzip_min_length 100;
        gzip_http_version 1.1;
        gzip_types application/json;

        location / {
            proxy_pass http://localhost:8000;
            proxy_read_timeout 300s;
            proxy_http_version 1.1;
        }
    }
}
```

На Листинге 24 представлена часть конфигурации Nginx. Nginx по-умолчанию асинхронно выполняет запросы в одном потоке. Настройки worker_processes и connections

отвечают за количество потоков (которое равно количеству ядер) и количество входящих соединений, которые будут обработаны. Далее следует рассмотреть настройку gzip. Согласно настройкам, сжатие gzip будет включено для запросов в формате JSON и размером более 100 байт, что автоматически включает его для всех запросов. Проксирование включается внутри секции location. Настройка proxy_pass является целевым адресом проксирования, proxy_read_timeout устанавливает тайм-аут для соединений в 300 секунд (значение вычислено эмпирическим путем), а proxy_http_version устанавливает версию протокола «на выходе» из Nginx. Этим протоколом является HTTP/1, то есть старая версия.

Это связано с тем, что прокси-сервер и бэкенд, как правило, запущены на одном хосте. Это нивелирует выгоды от использования HTTP/2 между ними, т.к. сетевое взаимодействие фактически отсутствует. Более того, бэкенд может не поддерживать HTTP/2, и тогда ситуация аналогична имеющейся в данной работе. Таким образом, использование HTTP/1 во внутренней сети никак не влияет на работу всей системы.

Далее следует рассмотреть настройку HTTP/2: как было отмечено ранее, все современные браузеры поддерживают этот протокол только при условии использования SSL шифрования. Для включения SSL требуется пара приватный-публичный ключ и сертификатом. Пара ключей и самоподписанный сертификат могут быть созданы утилитой OpenSSL следующей командой:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout cert.key
                                -out cert.crt
```

После ее выполнения будет создан файл с ключом и подписанный им сертификат, действительный на протяжении 1 года. Путь до этих двух файлов указывается в соответствующих настройках конфигурационного файла. Настройка listen отвечает за принятие входящих HTTPS-запросов, приходящих на стандартный для HTTPS порт – 443. Аналогично понижению протокола во внутренней сети, шифрование во внутренней сети так же отсутствует: в настройке proxy_pass указан HTTP (не HTTPS).

Таким образом, Nginx является **SSL-терминатором** – звеном, проверяющим шифрование и терминирующим его, что является частым способом применения gateway прокси. Это основано на т.н. доверительных отношениях между узлами внутри сети.

На этом настройка HTTP/2 и gzip для бэкенда завершена.

Дополнительной оптимизацией может быть отказ от I/O операций и упрощение бэкенда путем введения следующей схемы: запрос на выполнение операции с помощью Nginx проксируется на дополнительный прокси. Этот прокси модифицирует запрос следующим образом: файл в формате Base64 достается из запроса, конвертируется в оригинальный вид и сохраняется в высокопроизводительное хранилище данных (Redis), а на его место в запросе вставляется идентификатор, соответствующий файлу в хранилище. Далее запрос проксируется на бэкенд, где по идентификаторам, извлеченным из запроса, бэкендом достается файл из хранилища, минуя файловую систему.

Приведенная выше архитектура изображена на рис. 26.

Несмотря на введение дополнительного узла (и, соответственно, «точки отказа») и усложнение логики обработки запросов, эта схема является оптимизацией при соблюдении следующих условий:

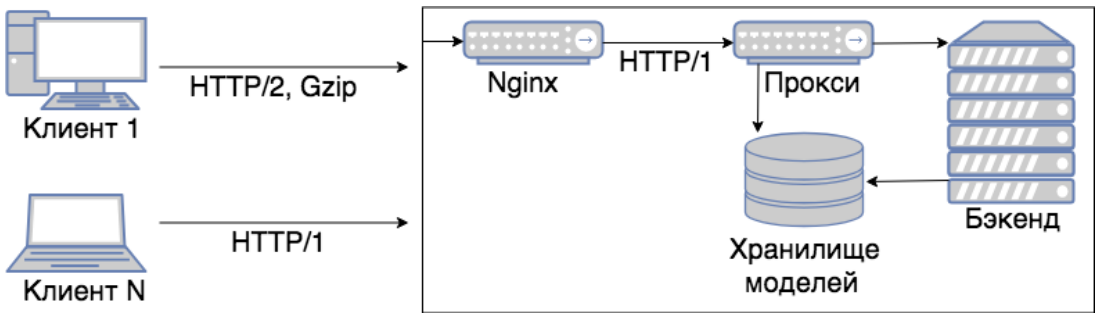

Рис. 26: Архитектура с учетом оптимизации

Второй **прокси-сервер высокопроизводителен**. Это может быть достигнуто использованием специализированных компонентов для проксирования. Одним из таких компонентов, помимо Nginx, является SingleHostReverseProxy – реализация прокси-сервера из стандартной библиотеки языка Go. В стандартной библиотеке языка есть множество функций для работы с сетью, а многопоточность с помощью горутин может быть организована проще, чем в большинстве других языков. Достоинством Go является высокая скорость выполнения, а наличие автоматического сборщика мусора облегчает разработку, не жертвуя при этом производительностью. Все это вкупе с простым синтаксисом делает Go наиболее подходящим языком реализации бэкенда, или, как минимум части, взаимодействующей непосредственно с сетью. На Листинге 25 частично представлена реализация описанного выше дополнительного прокси-сервера. Для этого необходим обработчик, принимающий запросы по определенному URL (в данному случае – «/»), добавляющий дополнительный заголовок ко всем запросам, проходящим через прокси и делегирующий непосредственную работу с запросом объекту (в данном случае это ProxyTransport), реализующему функцию RoundTrip(request). RoundTrip – полный цикл выполнения, начинающийся с получения запроса и заканчивающийся возвратом клиенту результата или ошибки. Во время выполнения RoundTrip имеется доступ как к запросу, так и, после выполнения проксирования, к ответу. В это время совершается модификация запроса описанным выше способом в функциях modifyRequest и modifyResponse. Благодаря высокой производительности Go подобная операция не снижает производительности всей схемы, при этом принимая на себя часть технических бэкенд-операций, не связанных с основной бизнес-логикой и, таким образом, разгружая бэкенд.

Хранилище высокопроизводительно, т.е. обладает большей скоростью работы, нежели чтение и запись файла из файловой системы. Примером такого хранилища является Redis In-Memory database – это база данных, полностью хранящаяся в памяти, что делает работу с ней чрезвычайно быстрой. При этом она опирается на хранение данных на диске и не может занимать больший объем, чем оперативная память. Благодаря использованию специализированных структур данных Redis может использоваться для хранения объектов разной степени сложности представления.

Листинг 25: Устройство простейшего реверс-прокси на языке Go

```
// создание прокси и начало прослушивания
var proxy = NewProxy("http://localhost:8000")
```

```
http.HandleFunc("/", proxy.handle)
http.ListenAndServe("http://localhost:5000", nil)

// обработчик запросов, приходящих по корневому URL
func (p *Proxy) handle(rsp http.ResponseWriter, rq *http.Request) {
  rsp.Header().Set("X-proxy", "GoProxy")
  p.proxy.Transport = NewProxyTransport()
  p.proxy.ServeHTTP(rsp, rq)
}

// функция передачи запроса на бэкенд и получения ответа
func (t *ProxyTransport) RoundTrip(request *http.Request) (*http.Response, error) {
  modifyRequest(request)

  // отправка запроса на бэкенд
  var response, roundTripErr = http.DefaultTransport.RoundTrip(request)

  modifyResponse(response)
  return response, roundTripErr
}
```

При оптимизации данного проекта Redis применен не был, и на рис. 26, представляющем итоговую архитектуру файловым хранилищем является файловая система.

Наличие большого количества сервисов (2 реверс-прокси и бэкенд) усложняет как разработку проекта (локальное развертывание), так и внедрение в работу с конечными пользователями. Действительно, каждый из сервисов должен быть собран, запущен и настроен соответствующим образом и этот процесс отличается для каждого компонента. Далее будет рассмотрена технология контейнеризации, которая призвана устранить обозначенные выше проблемы.

3.2 Контейнеризация

Контейнеризация – способ использования контейнеров Linux для развертывания приложений. Сама идея контейнеризации не нова, но ее реализация на платформе Docker полностью изменила подход к разработке, развертке и запуску приложений.

Контейнеризация основана на использовании **образов** – исполняемого пакета, который включает в себя все необходимое для запуска приложений – код, среду исполнения, библиотеки, переменные среды и файлы конфигурации. **Контейнер** – экземпляр (instance) образа в памяти во время выполнения. Контейнеры исполняются нативно в Linux и делят между собой ядро хоста, будучи запущенными в качестве отдельных процессов. Виртуальная машина, в отличие от контейнера запускает полноценную гостевую операционную систему с доступом к ресурсам хоста посредством гипервизора. БОльшая часть компонентов среды при этом может не использоваться приложением и в таком случае лишь создает лишнюю нагрузку на хост. На рис. 28 приведено расположение (устройство) контейнера относительно операционной системы. Можно говорить, что Docker или KVM реализуют доступ к механизмам ядра

операционной системы. На рис. 27 – устройство виртуальной машины.

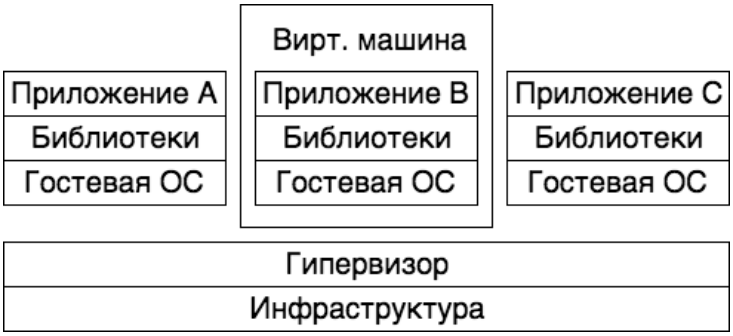

Рис. 27: Виртуальная машина и операционная система

Рис. 28: Контейнер и операционная система

Контейнеризованное ПО работает одинаково в любой среде, т.к. контейнер изолирует ПО от окружения. Это является одним из факторов популярности и повсеместного внедрения контейнеров.

Docker-приложение может быть **мультиконтейнерным**. Для работы с такими приложениями используется средство Docker Compose. В этом случае используется описание всего набора контейнеров с помощью docker-compose.yml файла. На Листинге 26 представлен docker-compose.yml файл, задающий набор из трех контейнеров – Nginx-прокси, Go-прокси и Rocket-бэкенд.

Листинг 26: Файл docker-compose

```
services:
  nginx:
    image: jinx
    ports:
      - "443:443"
    restart: always
    depends_on:
      - goproxy

  goproxy:
```

```
    image: gopro
    environment:
      - SELF_ADDR=0.0.0.0:5000
      - TARGET_ADDR=backend:8000
    ports:
      - "5000:5000"
    restart: always
    depends_on:
      - backend

backend:
  image: rocket
  environment:
    - ROCKET_ENV=production
  restart: always
  ports:
    - "8000:8000"
```

Каждый компонент системы представлен отдельным сервисом. Так, Nginx представлен сервисом, запускаемым из образа jinx, предварительно созданного на основе официального образа Nginx и конфигурации, заданной на Листинге 24 (в настройке proxy_pass при запуске в контейнере указывается название сервиса, т.е. gorproxy или backend, т.к. внутри контейнера действует своя система доменных имен). При любой нештатной ситуации в контейнере последует перезапуск nginx. При этом запуск послдует не ранее, чем сервисы goproxy и backend будут запущены. Nginx прослушивает порт 443 (HTTPS) на предмет входящих соединений и проксирует запросы приложению GoProxy, оформленному в виде сервиса goproxy, расположенному на порту 5000 и проксирующему модифицированные запросы далее на бэкенд – на порт 8000. При этом сервисы backend и goproxy должны иметь общий диск, с помощью которого должен производиться обмен STL-моделями. Описанная выше оптимизация внедрением Redis может быть применена и в таком случае в docker-compose.yml будет добавлен еще один сервис, имеющий образ Redis. На этом листинге также приведен пример установки переменных сред внутри контейнеров.

Таким образом, благодаря использованию Docker, бэкенд приложения может быть запущен одной командой, выполненной в корне проекта (должен присутствовать файл docker-compose.yml):

```
$ docker-compose up
```

На этом оптимизацию и разработку можно считать завершенными. Итоговая структура разработанного ПО с учетом оптимизаций изображена на рис. 26. Далее последует проверка и оценка примененных оптимизаций.

3.3 Оценка оптимизации

После внедрения описанных ранее изменений упростился процесс развертывания приложения, а так же повысилась скорость выполнения операций (с точки зрения

пользователя) за счет снижения объема передаваемых данных. Для проверки используется инструмент cURL со следующими параметрами:

```
$ curl -http2-prior-knowledge -X GET -H 'Accept-encoding: gzip, deflate'
    https://localhost/api/stl/stub -insecure -v -w "@curl-format.txt"
```

Эти параметры формируют GET запрос (с помощью протокола HTTP/2) к API бэкенда на получение заранее заданного объекта (stub) без проверки сертификата, с полным выводом и замером времени. В результате приходит ответ, представленный на Листинге 27.

Листинг 27: Результат запроса после внедрения HTTP/2 и gzip сжатия

```
< HTTP/2 200
< Content-Encoding: gzip
< Content-Length: 35297430
< Content-Type: application/json
< Date: Sat, 16 Jun 2018 00:04:26 GMT
< Server: nginx/1.13.9
< X-Proxy: GoProxy
time_namelookup:  0,004255
   time_connect:  0,005306
     time_total:  8,385913
```

На Листинге 28 представлен результат выполнения того же запроса, но с помощью протокола HTTP/1 и без gzip сжатия.

Листинг 28: Результат запроса при наивной схеме реализации серверной части

```
< HTTP/1.1 200 OK
< Content-Length: 63526390
< Content-Type: application/json
< Date: Sat, 16 Jun 2018 00:04:09 GMT
< Server: nginx/1.13.9
< X-Proxy: GoProxy
time_namelookup:  0,004175
   time_connect:  0,004863
     time_total:  11,47341
```

При сравнении Листингов 27 и 28 становится очевидно, что новая схема работает и показывает улучшенное на $\approx 30\%$ время, затраченное на передачу и сокращенный на $\approx 50\%$ объем передаваемых данных. Следовательно, изменения могут быть приняты и зафиксированы.

Итоговая структура ПО представлена на Листинге 26.

Выполненный проект доступен в GitHub репозитории по адресу github.com/avbelyaev/stl-editor.

3.4 Руководство пользователя

Клиентское приложение использует несколько внешних библиотек, которые должны быть подключены к проекту:

- GL-matrix, доступная по адресу glmatrix.net/ должны быть скомпилирована с помощью npm и размещена по следующему пути: /webclient/js/lib/gl-matrix.js;
- Axios, доступная по адресу github.com/axios/axios должна быть размещена по адресу
  /webclient/js/lib/axios.min.js.

При редактировании оформления клиентской части для обновления CSS стилей необходим препроцессор SASS (Dart-SASS), доступный по адресу sass-lang.com/install. Все эти библиотеки могут быть собраны и подключены либо локально, либо при сборке Docker-образа Nginx.

Серверная часть запускается с помощью Docker Compose. Для сборки необходимы Docker, не ниже версии 18 и Docker-Compose не ниже версии 1.21. Для запуска необходимы 3 описанных в docker-compose файле образа:

- jinx – образ Nginx;
- gopro – образ реверс-прокси сервера, реализованного на Go;
- rocket – образ бэкенда, использующий Nightly версию Rust.

Каждый из образов может быть собран отдельно выполнением следующей команды в соответствующей директории:

```
$ docker build -t <название_образа> .
```

Следует учесть, что Rust не имеет официального Docker-образа для Nightly версии и в Dockerfile бэкенда переход на Nightly версию Rust осуществляется самостоятельно, что занимает достаточно большое количество времени.

Для работы Nginx необходимы ключи и сертификат, расположенные в директории nginx/cert, которые могут быть сгенерированы описанной ранее командой:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout cert.key
                              -out cert.crt
```

Сборка также будет произведена, если образы не будут найден при попытке запуска командой:

```
$ docker-compose up
```

ЗАКЛЮЧЕНИЕ

В ходе работы были изучены имеющиеся аналоги а также специфика обработки STL-моделей, представленных в стереолитографическом формате STL. При разработке клиентского приложения были использованы приемы работы с современной шейдерной компьютерной графикой. При разработке серверной части было реализовано приложение, выполняющее бизнес-логику по обработке моделей, а также построена сопутствующая инфраструктура для оптимизации производительности.

В результате данной работы был спроектирован программный комплекс по обработке 3D-моделей и реализован в виде распределенной системы — приложения на основе клиент-серверной архитектуры.

После исследования недостатков первоначальной реализации был сформулирован ряд дополнительных требований, которые были выполнены при оптимизации системы, и была проведена оценка оптимизации.

Требования, предъявленные к разрабатываемому прототипу системы были выполнены. Благодаря тому, что разработанное ПО является веб-сервисом, поддержка и дальнейшее развитие проекта упрощаются. Так, разработка компонентов проекта может проводиться независимо. Благодаря контейнеризации, разработка, локальное тестирование и развертывание проекта значительно упрощены.

В перспективе планируется расширение функциональности разработанного программного обеспечения на основании консультаций со специалистом предметной области, в том числе:

- возможность точной манипуляции моделями путем задания величин поворота и перемещения;
- поддержка дополнительных функций библиотеки, таких, как разбиение моделей на компоненты связности;
- совместная работа нескольких пользователей над моделью.

Кроме того, с целью повышения производительности системы, может быть произведена ее оптимизация:

- замена бэкенда на более производительное решение;
- сохранение STL-моделей в высокопроизводительное хранилище данных в оперативной памяти (Redis) для исключения I/O операций;
- повышение отказоустойчивости и доступности сервиса за счет внедрения репликации и балансировки нагрузки;
- отправка модели на сервер при добавлении ее в сцену. Далее при каждой фиксации положения – отправка новых «параметров» модели на сервер и применение преобразований к модели.

Список литературы

1, Д. Вольф. OpenGL 4. Язык шейдеров. Книга рецептов — МСК: ДМК Пресс, 2015. — 98 с.

2, Tony Parisi. WebGL: Up and Running — O'Reilly, 2012. — 36 с.

3, А. Бэнкс, Е. Порселло. React и Redux. Функциональная веб-разработка — Питер, 2018. — 183 с.

4, Д. Блэнди, Д. Орендорф. Программирование на языке Rust — МСК: ДМК Пресс, 2018. — 12 с.

5, Д. Гинсбург, Б. Пурномо. OpenGL ES 3.0. Руководство разработчика — МСК: ДМК Пресс, 2015. — 206 с.

6, Catalog of Patterns of Enterprise Application Architecture [Электронный ресурс] — Режим доступа: https://www.martinfowler.com/eaaCatalog/index.html — Дата обращения: 22.03.2018

7, HTTP/2: Background, Performance Benefits and Implementations [Электронный ресурс] — Режим доступа: https://www.sitepoint.com/http2-background-performance-benefits-implementations/ — Дата обращения: 17.06.2018

8, Programming 3D Applications with HTML5 and WebGL — O'Reilly, 2018. — 47 с.

9, SGVsbG8gd29ybGQh или история base64 [Электронный ресурс] — Режим доступа: https://habr.com/post/88077/ — Дата обращения: 02.06.2018

10, Как работает сжатие GZIP [Электронный ресурс] — Режим доступа: https://habr.com/post/221849/ — Дата обращения: 24.05.2018