Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра вычислительной физики
Направление «Физика»

Лебедева Анна Валерьевна

Реализация методов построения трёхмерных моделей человеческого тела и их применение для решения задач радиационной терапии

Implementation of methods for constructing three-dimensional models of the human body and their application for solving radiation therapy problems

Магистерская диссертация

Научный руководитель

к.ф.-м.н., доцент _____ Немнюгин С. А.

Рецензент

к.ф.-м.н., с.н.с. ОИЯИ_____ Мерц С. П.

Санкт-Петербург
2017

Оглавление

Введение

По данным статистики, онкологические заболевания являются одной из основных причин смертности в мире [1]. В современной медицине наиболее важную роль играет своевременное выявление рака, поскольку это заболевание не поддаётся лечению на поздних стадиях. На протяжении последних десятилетий технология медицинской интроскопии (medical imaging) постоянно усовершенствуется, использование компьютерных технологий улучшает контрастность получаемых изображений и качество данных, тем самым увеличивая эффективность назначаемого лечения.

Самая распространённая технология интроскопии (неинвазивное исследование внутренней структуры объекта) – это томография. Результаты обследований компьютерной (КТ) и магнитно-резонансной (МРТ) томографии, представляют собой серию снимков множества сечений тела пациента, которые характеризуют особенности его анатомии и физиологии [2]. Тем самым, создаётся массив плоских статических изображений, данные для построения которых хранятся в специально разработанном стандарте медицинских файлов DICOM 3.0 (Digital Imaging and Communications in Medicine) [3].

В большинстве случаев в медицинской диагностике проводится зрительный анализ изображений отдельных сечений – графическое представление данных среза исследуемой области человеческого тела. Однако для более тщательного планирования лечения, а также возможности численного анализа и моделирования процессов в теле человека необходимо использовать объёмные модели, содержащие информацию о реальных параметрах (плотность, состав) тканей, геометрии.

Актуальность работы обусловлена следующим:

Методы визуализации органов человека на основе DICOM-файлов получают все большую популярность в различных областях медицины, благодаря возможности создания точных и реалистичных визуальных представлений 3D объектов по медицинским данным [2].

На сегодняшний день существуют коммерческие проекты обработки и визуализации DICOM, такие как Инобитек, RadiAnt и др., однако они решают исключительно медицинские задачи, и их исходный код не доступен. Их использование для исследовательских задач нецелесообразно, в связи с отсутствием информации об используемых алгоритмах и приближениях. Неинформированность в этих вопросах допустима в случае индивидуального

использования, при необходимости же проведения дальнейших исследований на основе полученных данных, она приведёт к неизбежным затруднениям. Проведение расчётов, связанных с планированием лечения, в том числе с моделированием для радиационной терапии, требует учитывать все приближения, в том числе и те, которые вносятся при построении объёмной модели.

Кроме того, коммерческие продукты бесплатны для скачивания, однако их использование в медицинских и/или исследовательских учреждениях требует оплаты лицензионной версии, стоимость которой существенна.

Таким образом, целью данной работы является разработка системы для построения трёхмерных моделей человеческого тела.

Для достижения поставленной цели решались следующие задачи:

- разработка и программная реализация алгоритмов анализа (парсинга) DICOM-файлов
- проектирование и реализация графического интерфейса
- разработка программного модуля реконструкции трёхмерной воксельной модели на основе анализа изображений, полученных в результате томографии.

Разработанные при выполнении данной работы программные модули в дальнейшем могут стать частью так называемых систем планирования лечения.

Глава 1. Обзор литературы

1.1 Наиболее популярные методы неинвазивной диагностики

Компьютерная томография – особый вид рентгенологического исследования, непрямое измерение ослабления или затухания рентгеновских лучей из различных положений вокруг пациента. По большей части в компьютерной томографии используются снимки аксиальных (поперечных) срезов. Для каждого снимка рентгеновская трубка поворачивается вокруг пациента, а толщина сечения задаётся заранее в параметрах программы.

Современные медицинские учреждения оборудованы компьютерными томографами с многорядным расположением детекторов. В мультиспиральной компьютерной томографии напротив трубки располагается несколько рядов детекторов вместо одного. За счёт этого сокращается время проведения диагностики, а соответственно объем получаемого рентгеновского излучения.

Рентгеновская трубка описывает винтовую траекторию вокруг пациента (см. рис.1) [4].

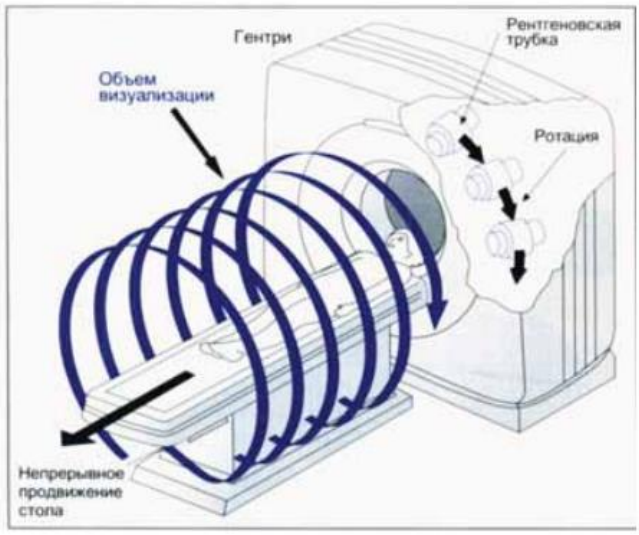

Рис.1. Траектория движения рентгеновской трубки в мультиспиральном КТ

Магнитно-резонансная томография (МРТ) – это технология получения томографических медицинских изображений, исследование внутренних органов и тканей, основанное на явлении ядерного магнитного резонанса (ЯМР). Ядерный магнитный резонанс – это физическое явление резонансного поглощения или излучения фотонов ядрами с ненулевым спином во внешнем магнитном поле определённой частоты, которая называется частотой ЯМР.

Поле, необходимое для процесса сканирования, создаётся специальным магнитом, градиентные катушки внутри него создают градиент в направлениях X, Y, Z, вид и амплитуда

каждого из трёх градиентных полей задаётся с помощью компьютера. Радиочастотная катушка создаёт магнитное поле для поворота спинов на 90 или 180º и она же регистрирует сигнал, получаемый от спинов.

Под контролем компьютера источник радиочастотных-импульсов и программатор соответственно генерируют импульс нужной частоты и придают им форму sinc импульсов с квадратным распределением частот. Градиентный усилитель увеличивает мощность градиентных импульсов до уровня, достаточного для управления градиентными катушками [5].

МРТ позволяет получить сечения в различных плоскостях, задаваемых медицинским персоналом в высоком качестве. Пациент помещается на стол томографа, катушка располагается вокруг исследуемой анатомической области. Эта часть тела будет находиться в изоцентре магнита во время исследования (см. рис.2).

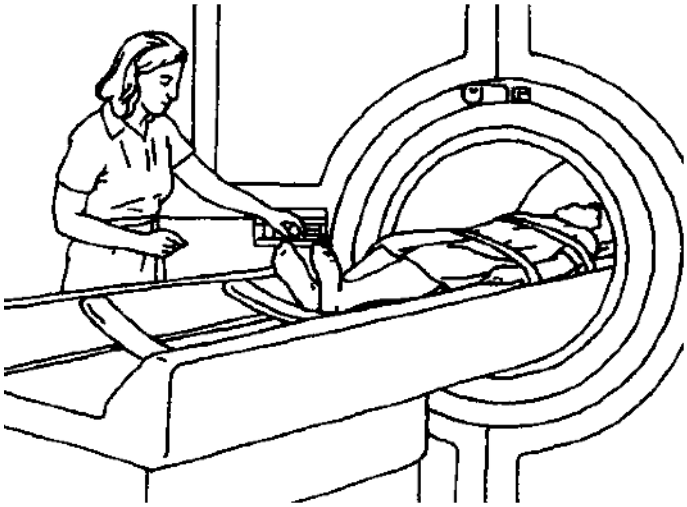

Рис. 2. Проведение МРТ обследования

КТ наиболее эффективна для исследования костных повреждений, травм головы, брюшной полости и малого таза. Костные ткани на МРТ видны, однако более качественные данные получаются при исследовании мягких тканей (хрящевая, мышечная ткань, связки, головной и спинной мозг).

1.2. DICOM

DICOM (англ. Digital Imaging and COmmunications in Medicine) –это международный стандарт создания, хранения, пересылки медицинских изображений и соответствующей дополнительной информации о пациентах. DICOM используется практически во всех областях медицины: радиологии, кардиологии и радиотерапии (рентген, КТ, МРТ, УЗИ и др.).

Фактически при обследовании пациента создаётся файл DICOMDIR и файлы DICOM, которые необходимо хранить совместно. DICOMDIR хранит в себе не изображения, а ссылки и метаданные, благодаря которым сохраняется иерархическая структура файлов обследования (см. рис. 3).

Рис.3. Формирование иерархии файлов в файле DICOMDIR

Файловый уровень стандарта DICOM 3.0 редакции 2016 года описывает параметры оборудования, медицинское учреждение и данные медицинского персонала, атрибуты и всю необходимую информацию о пациенте. Он также содержит изображение или серию изображений и их параметры, полученные при проведении томографии [3].

Все параметры и атрибуты сохраняются в так называемых тэгах (см. таблицу 1).

Таблица 1. Список тэгов, используемых в данной работе

Тэг	Описание	Значение (практическое)
(0010,0010)	Patient's Name	GALINA
(0010,0020)	Patient ID	89197580
(0010,0040)	Patient's Sex	F
(0018,0050)	Slice Thickness	3
(0018,0088)	Spacing Between Slices	4
(0020,0032)	Image Position (Patient)	-133.5079\-116.4593\-88.0799
(0028,0010)	Rows	768
(0028,0011)	Columns	768
(0028,1050)	Window Center	4988
(0028,1051)	Window Width	10023

При извлечении данных, стоит отметить, что есть определённые отличия между данными, которые записываются в тэги, для компьютерной и магнитно-резонансной томографии. Компьютерная томография (тэг CT) сохраняет в файлах рентгеновскую плотность, из которой, используя шкалу Хаунсфилда, можно получить плотность биологических тканей.

Магнитно-резонансная томография (тэг MR) сохраняет интенсивность обратного сигнала. При этом аппарат МРТ записывает значения в нормализованном виде в зависимости от текущего исследования. Для получения значений плотности в этом случае необходимо

использовать значения дополнительных тэгов «ширина» и «центр окна» (width window и center). Таким образом, используются относительные значения каждого исследования, а для отрисовки используется только тот диапазон плотностей, которые «рекомендует» томограф для исследования той или иной области тела.

1.3. Воксельная графика

Для реальных медицинских исследований в основном используются двумерные изображения. Однако, зачастую, удобнее оказывается объёмная модель, наиболее реалистично отражающая картину человеческого тела. Поэтому следует ввести понятие воксельной графики и воксельного рендеринга.

Воксель – это «объёмный пиксель», то есть точечный элемент объёмной модели. В вокселе содержится значение цвета, или в случае моделирования – плотность. Координаты каждого вокселя вычисляются из его относительного расположения.

Воксельный или объёмный рендеринг – отображение проекции трёхмерной модели на экран.

В качестве плюсов использования вокселей стоит отметить сохранение всей информации о плотностях во время построения изображения. Это позволяет использовать данный метод при интерактивном исследовании данных. Кроме того, рендеринг на основе вокселей позволяет комбинировать изображения разных типов: непрозрачные, полупрозрачные поверхности, разрезы и проекции максимальных интенсивностей [2].

Фактически единственный минус воксельной графики – необходимость обработки больших объёмов данных. Этот недостаток, однако, может быть преодолён применением технологий высокопроизводительных вычислений. В практической части магистерской работы предложен метод отображения вокселей, позволяющий эффективно получать трёхмерное изображение.

1.4. Основные алгоритмы визуализации серии снимков

1.4.1. Алгоритм FTB и BTF

Алгоритмы, использующие z-буфер - одни из наиболее простых алгоритмов удаления невидимых поверхностей, которые работают в пространстве изображения. Для запоминания атрибутов каждого пикселя используется буфер кадра, а z-буфер используется для запоминания координаты z или глубины видимых пикселей, являясь буфером глубины.

На каждом шаге глубина каждого пикселя, который нужно занести в буфер кадра, сравнивается со значением глубины, которое уже занесено в z-буфер. Если этот пиксель

расположен впереди пикселя, находящегося в буфере кадра, то новый заносится в этот буфер, после чего z-буфер изменяет значение **z**. Если результат противоположный, то никаких действий не производится. Т.е. алгоритм можно описать, как поиск максимума функции z (x,y) по x и y. Главное преимущество алгоритма – его простота. Основной недостаток алгоритма - большой объем требуемой памяти [6].

Наиболее простой путь для построения визуального образа - это рассмотреть каждый воксель как 3D точку, которую необходимо преобразовать с помощью матрицы видового преобразования, после чего спроецировать в Z-буфер и отобразить на экран. Такой способ визуализации называют рендерингом в объектном пространстве (object-space rendering) или рендерингом прямого хода (forward rendering) [7].

Back-to-front (BTF) рендеринг, фактически совпадает с использованием Z-буфера и отличается только дополнительной предварительной сортировкой массива вокселей, позволяющей считывать его компоненты в порядке уменьшения или увеличения расстояния до наблюдателя. Для отсева невидимых вокселей в таком случае уже не нужен Z-буфер. При рисовании очередной воксель будет записываться поверх уже отрисованных вокселей или смешивается со значением фона. К этому типу следует отнести методы сплэттинга, сдвиговые алгоритмы и текстурный рендеринг.

Рендеринг FTB (front-to-back) практически совпадает с BTF, но в нем обход вокселей производится в порядке возрастания расстояния. К методу FTB относят методы трассировки [8].

Следует иметь в виду, что метод Z-буфера в чистом виде не способен обеспечивать рендеринг полупрозрачных материалов, так как воксели отображаются на экран в произвольном порядке.

Смешение света основывается на вычислениях, моделирующих его прохождение через разные материалы. В методах BTF и FTB объекты отображаются на экран в том же порядке, в котором по сцене проходят световые лучи, что позволяет реализовать полупрозрачность.

1.4.2. Алгоритм сплатинга (splatting)

В работе [9] предложен уже упомянутый выше способ реконструкции для методов прямого хода - алгоритм сплатинга (splatting). Каждый воксель после отображения на экран, размывается по нескольким соседним пикселям с использованием 2D таблицы прямого доступа (таблицы следа). В этой таблице хранятся "отпечатки" вокселей, которые при сплатинге смешиваются с массивом изображения.

1.4.3. Методы визуализации объёма обратного хода

В отличие от методов визуализации объёма прямого хода, в методах, использующих обратный ход, лучи испускаются «из глаза» в направлении каждого пикселя экрана. Прохождение луча через объёмные данные отслеживается до тех пор, пока он не пересечёт непрозрачный объект или не накопится критическое значение величины непрозрачности.

Простейший метод для формирования новой выборки - двигаться с некоторым шагом вдоль луча, находя воксели, ближайшие к дискретным точкам траектории, и выполняя интерполяцию нулевого порядка между ними.

Альтернативный способ состоит в том, чтобы использовать в качестве точек новой выборки точки пересечения луча и граней вокселей. В этом случае значение точки выборки находится путём интерполяции и затем смешивается с лучом. В более точном алгоритме для вычисления значений равномерно расположенных вдоль луча точек выборки используется интерполяция более высокого порядка.

1.5. Рендеринг и шейдеры

Визуализацию или рендеринг можно разделить на две группы: рендеринг по требованию и рендеринг реального времени.

Рендеринг по требованию (пре-рендеринг) ориентирован на получение максимального качества изображения. Алгоритмы расчёта в этом случае достаточно сложны и требуют затрат значительного количества времени для получения конечного изображения. Расчёты производятся максимально «честно» для достижения максимальной реалистичности. Сфера применения таких алгоритмов – это создание спецэффектов для фильмов и создание анимационных мультфильмов. Такой рендеринг не подходит для данной задачи, поскольку немаловажным условием является отображение трёхмерных объектов в режиме реального времени.

Рендеринг реального времени ориентирован на получение изображения за минимальное время. Данный метод рендеринга используется в приложениях, в которых требуется быстро получать конечную сцену и минимизировать задержки, например, в компьютерных играх и симуляторах. Этот вид рендеринга полностью удовлетворяет поставленным в данной работе задачам.

На сегодняшний день наиболее эффективно рендеринг производится с помощью специальных программ, предназначенных для исполнения на графическом процессоре (англ. *graphics processing unit*, *GPU*). Такие программы получили название шейдеры (англ. *shader* — затеняющая программа). Шейдеры составляются на одном из специализированных языков

программирования и компилируются в инструкции для GPU. В силу того, что различные графические процессоры имеют разные процессорные команды, шейдеры передаются в виде исходного кода, а компилируются непосредственно при запуске основной программы.

В данной работе был выбран язык программирования шейдеров GLSL и платформонезависимая библиотека OpenGL. Среда разработки DirectX не рассматривалась из-за жесткой привязки к ОС Windows. Также на выбор библиотеки OpenGL повлияло то, что она поддерживается кроссплатформенным инструментарием разработки ПО на языке программирования C++.

1.5.1 OpenGL и шейдеры в Qt

Доступ к API библиотеки OpenGL в Qt 5.8 осуществляется через класс QOpenGLFunctions. Для отображения результатов рендеринга существует специальный виджет QOpenGLWidget. В этот виджете определены стандартные для OpenGL методы инициализации сцены initializeGL, изменения размеров окна resizeGL и метод вывода сцены paintGL. Работа с шейдерами производится с помощью класса QOpenGLShaderProgram.

Перед тем, как осуществлять вызов функций OpenGL в Qt 5.8 необходимо провести их инициализацию. Данная процедура выполняется с помощью метода initializeGL класса QOpenGLFunctions. Обычно она вызывается в самом начале при инициализации сцены в методе initializeGL класса QOpenGLWidget. После чего производится установка различных параметров визуализации и компиляция шейдеров. В случае если компиляция пройдёт неудачно, то отобразить сцену согласно требованиям не получится.

Как уже упоминалось ранее, шейдерные программы передаются в виде исходного кода на GPU, а их компиляция производится непосредственно при запуске программы. Исходный код шейдерных программ можно хранить непосредственно в исходном коде C++. Это удобно для несложных и необъёмных программ, но для больших и сложных данный подход неудобен, поскольку увеличивает сложность чтения кода, а также усложняет его редактирование. Для загрузки и компиляции шейдера из исходного кода методу addShaderFromSourceCode класса QOpenGLShaderProgram необходимо передать два аргумента. Первый аргумент – это тип шейдера, а второй – это строка, в которой хранится текст шейдера.

Другой способ заключается в хранении исходного кода шейдерных программ в отдельном файле. Для языка GLSL файлы имеют расширение *.glsl для всех видов шейдеров. Загрузка шейдера из файла выполняется с помощью метода addShaderFromSourceFile. Первый аргумент – это тип шейдера, а второй – это путь к файлу с исходным кодом шейдера. Если компиляция каждого шейдера произошла успешно, производится связывание шейдеров

между собой. Это выполняется с помощью метода link. Далее с помощью метода bind связывают шейдерную программу с активным QOpenGLContext и делают её текущей.

1.5.2 Виды шейдеров

На сегодняшний день поддерживаются 6 видов шейдеров:

- вершинный шейдер (QOpenGLShader::Vertex);
- фрагментный шейдер (QOpenGLShader::Fragment);
- геометрический шейдер (QOpenGLShader::Geometry);
- управляющей тесселяцией шейдер (QOpenGLShader::TessellationControl);
- определяющий тесселяцию шейдер (QOpenGLShader::TessellationEvaluation);
- вычислительный шейдер (QOpenGLShader::Compute);

Вершинный шейдер (вертексный шейдер) получает на вход информацию об одной вершине полигона, обрабатывает её и передает на выход. В качестве примера можно рассмотреть прямоугольник, полученный с помощью вершинного шейдера из квадрата (см. рис. 4).

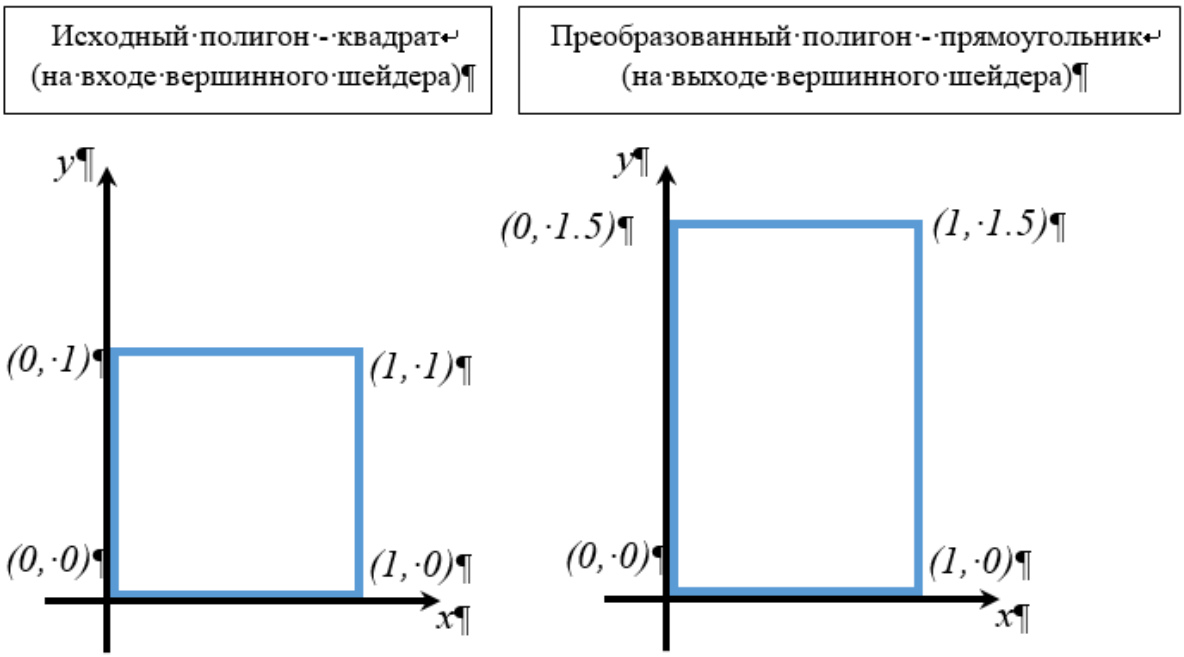

Рис. 4. Пример преобразования вершин в вершинном шейдере

Пример кода вершинного шейдера представлен на листинге 1. Если координата y больше или равна 1, то необходимо изменить её на 1.5. В результате квадрат трансформируется в прямоугольник.

Листинг 1. Вершинный шейдер. Преобразование квадрата в прямоугольник.

```
uniform mat4 mvp_matrix;
attribute vec4 a_position;


void main() {

    if (a_position.y >= 1.0)
       a_position.y = 1.5;

    gl_Position = mvp_matrix * a_position;

}
```

У каждого набора вершин существуют одинаковые (общие) параметры и индивидуальные. В языке GLSL для описания таких переменных используются соответственно uniform и attribute. Вообще переменная uniform общая не только для вершинного шейдера, но и для всех шейдеров, слинкованных вместе.

Общей для всех вершин является матрица MVP (ModelViewProjection). Матрица ModelViewProjection – это произведение трёх матриц - мировой, видовой и проекционной. В примере с прямоугольником эта матрица переводит координаты из (0,0), (1,0), (0,1.5) и (1,1.5) в координаты вершин внутри виджета QOpenGLWidget. Матрица MVP передается в шейдер в uniform переменной mvp_matrix.

Индивидуальными параметрами для каждой из вершин являются их координаты. Передаются координаты с помощью переменной a_position.

После вершинного шейдера вершины уже в качестве полигона передаются в геометрический шейдер. Если не указан пользовательский геометрический шейдер, то преобразования в геометрическом шейдере не происходят. Полигон собирается и передаётся во фрагментный шейдер.

Окраска фрагмента производится во фрагментном шейдере, а во встроенную переменную gl_FragColor записываются значения цвета. Классический пример фрагментного шейдера представляет собой:

```
void main() {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Встроенной переменной gl_FragColor передаётся вектор, координаты которого называют r, g, b и a. Диапазон изменения координат от 0.0 до 1.0. Последняя координата определяет прозрачность фрагмента.

Во фрагментном шейдере необходимо записывать переменную gl_FragColor, иначе значение цвета не будет отображено. Если требуется отбросить цвета, то используется

специальная команда discard. При выполнении операции discard в буфер глубины не заносится значение цвета.

1.5.3 Текстуры в шейдерах и Qt.

В поставленной задаче необходимо каждому вокселю передавать значение его цвета. В таком случае можно заранее вычислить положение вокселя и передать шейдеру через атрибут. Другой способ заключается в использовании текстуры, позволяющей заранее загрузить в память в GPU цвета вокселей, а во фрагментном шейдере извлечь цвет вокселя из этой текстуры. Воксели прорисовываются в трёхмерном пространстве, поэтому необходимо использовать трёхмерную текстуру.

В Qt 5.8 текстура создаётся с помощью класса QOpenGLTexture. При создании объекта этого класса в конструктор передаётся аргумент – тип текстуры: QOpenGLTexture::Target3D. С помощью метода setSize, устанавливается размер текстуры по трём координатам. После установки выделяется память с помощью метода allocateStorage.

После выделения памяти на GPU необходимо передать данные для текстуры с помощью метода setData. Важно отметить, что массив должен иметь размер соответствующий размеру, переданному в метод setSize. Иначе не все данные будут скопированы, или произойдёт обращение к участку памяти за пределами массива.

Текстуры имеют свою систему координат UVW, аналогичную координатам XYZ, но к координатам UVW можно обращаться только в диапазоне от 0 до 1.

Глава 2. Алгоритм визуализации

2.1. Анализ существующих алгоритмов визуализации

Применение алгоритмов, описание которых дано в Главе 1, в текущей задаче неэффективно, поскольку требуются дополнительные затраты на осуществление сортировки на CPU или на GPU с использованием технологии CUDA.

Более того, большую часть исследуемых изображений занимают «пустые области» - области, в которых нет искомого объекта. Таким образом, возникает неопределённость выбора шага для перемещения Z-буфера для достижения требуемого результата. Изначально плоскости снимков находятся на определённом расстоянии друг от друга, т.е. между ними расположена «пустота». Следовательно, если строить отображение сцены вдоль поверхности снимков, то Z-буфер будет доходить до противоположного конца сцены. В таком случае необходимо выполнить предобработку и пре-рендеринг для подготовки к отображению трёхмерной сцены. Кроме того, необходимо на подготовительном этапе заполнить промежутки между плоскостями и провести интерполяцию между слоями. В связи с этим для всех рассматриваемых ранее алгоритмов невозможно динамически изменять порог яркости отображения вокселя во время работы.

Для подтверждения вышесказанного было проведено тестирование алгоритма Z-буфера для исходных данных этой работы. Измерялось время обработки объёмной модели в зависимости от шага для z-буфера. Тестирование проводилось на персональном компьютере со следующими характеристиками:

CPU: Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz 2.49 GHz

ОЗУ: 8,00 Гб

Графическая карта: NVIDIA GeForce 840M

Результаты представлены на рис 5. Установлено, что время обработки одного кадра в среднем 200 мс, а это 5 кадров в секунду.

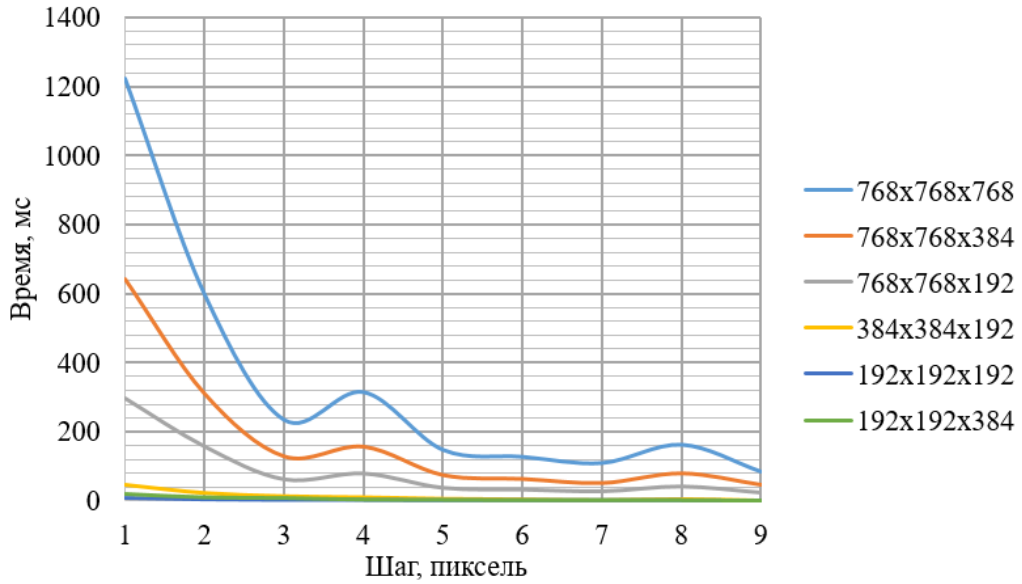

Рис. 5. Зависимость времени проверки на глубину в зависимости от шага для разных размеров воксельной модели.

2.2. Предложенная методика

Предложенный метод визуализации серии DICOM снимков состоит из нескольких этапов, которые показаны на рисунке 6.

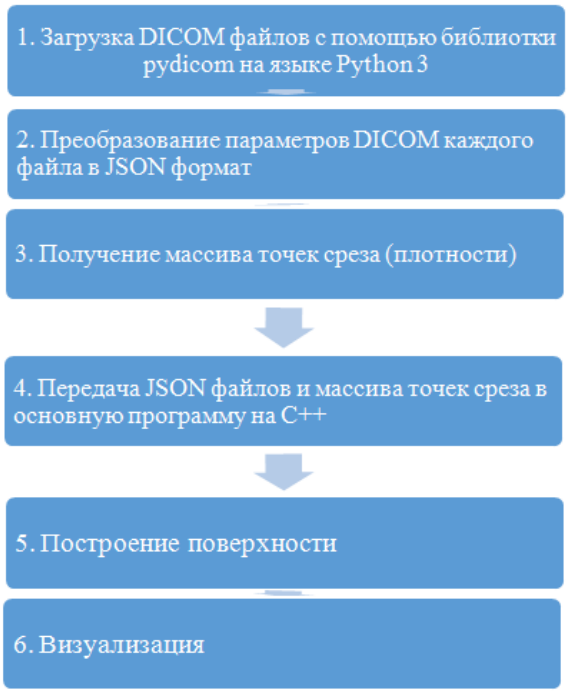

Рис. 6. Блок-схема предложенного метода визуализации

Для реализации метода, описываемого в данной главе, использовался язык программирования c++11 и библиотека Qt версии 5.8 и язык программирования python 3.4 (дополнительные пакеты: pydicom, simplejson).

Первые три этапа (см. рис. 6) реализованы на языке Python с использованием библиотеки pydicom. Эта библиотека позволяет считывать все данные, сохранённые в DICOM-файле. Вся информация сохраняется в специализированном типе данных python: словарь (dictionary) (см листинг 2). Для чтения DICOM-файла используется функция `dicom.read_file`.

Листинг 2. Файл dcmtodic.py. Функция преобразования DICOM-файла в словарь

```
def _dicomtodic(dcm):
    '''Преобразует DICOM формат в словарь'''
    dcmInfo = {}
    for key in dcm.dir():
        value = getattr(dcm, key, '')

        if not isinstance(value, list):
            value = str(value)
        dcmInfo[key] = value
    return dcmInfo
```

Для взаимодействия программы считывания DICOM на python с основной программой на c++ был использован текстовый формат передачи данных – JSON (см. листинг 3).

Листинг 3. Файл dicomdir.py. Функция преобразования в JSON-формат

```
def DICOMFileToJSON(fullfilename):
    """Преобразует словарь в JSON формат"""
    ds = dcmtodic.readDICOMFile(fullfilename)
    return str(json.dumps(ds))
```

После проведенных преобразований, используется функция getPixArray (Листинг 4), извлекающая массив точек среза, которые содержат значения плотности.

Листинг 4. Файл dcmtodic.py. Функция получения массива

```
def getPixArray(fullfilename):
    try:
        print(fullfilename)
        dcm = dicom.read_file(fullfilename)
        arr = np.array(dcm.pixel_array, dtype=int)
        return arr.tolist()
    except FileNotFoundError:
        print("FileNotFoundError")
    return ()
```

На четвертом этапе (см. рис. 6) используется функция-член getJSONFromDICOM класса PyDICOMJson, выполняющая преобразование JSON-файла в объект QJsonObject, с которым возможна работа в c++ (см. листинг 5).

Листинг 5. Файл pydicomjson.cpp. Функция-член: преобразование JSON-файла

```
QJsonObject PyDICOMJson::getJSONFromDICOM(const QString &file) {

    if(!(pDICOMFileToJSON && PyCallable_Check(pDICOMFileToJSON)))
        throw NotFoundDICOMFuncException("Not found getDICOMFiles");

    const char* str = file.toStdString().c_str();
    CPyObject pyfile = PyUnicode_FromString(str);

    CPyObject args = PyTuple_Pack(1, pyfile.getObject());
    CPyObject pValue = PyObject_CallObject(pDICOMFileToJSON, args);
    QString value(PyUnicode_AsUTF8(pValue.getObject()));

    return ObjectFromString(value);
}
```

Функция-член PyDICOMJson::getPixArray, которая отвечает за передачу массива точек, принимает в качестве аргумента название файла, из которого необходимо его извлечь, а также ширину и высоту среза (см. листинг 6).

Листинг 6. Передача массива точек c++

```
int* PyDICOMJson::getPixArray(const QString &file, size_t& width, size_t& height) {
    if(!(pgetPixArray && PyCallable_Check(pgetPixArray)))
        throw NotFoundDICOMFuncException("Not found getPixArray");

    const char* str = file.toStdString().c_str();
    CPyObject pyfile = PyUnicode_FromString(str);

    CPyObject args = PyTuple_Pack(1, pyfile.getObject());
    PyObject* pValue = PyObject_CallObject(pgetPixArray, args);

    if (nullptr == pValue)
        return Q_NULLPTR;

    int row = static_cast<int>(PyList_Size(pValue));
    int col = 0;
    if (row > 0) {
        PyObject* pCurRow = PyList_GetItem(pValue, row-1);
        col = static_cast<int>(PyList_Size(pCurRow));
    }

    height = static_cast<size_t>(row);
    width  = static_cast<size_t>(col);

    int size = row*col;
    int* arr = new int[size];
    int cnt = 0;

    for (int i = 0; i < row; ++i) {
        PyObject* pCurRow = PyList_GetItem(pValue, i);
        for (int j = 0; j < col; ++j) {
            PyObject* item = PyList_GetItem(pCurRow, j);
            arr[cnt] = PyLong_AsLong(item);
            ++cnt;
        }
    }

    Py_DECREF(pValue);

    return arr;
}
```

Текстовый формат передачи данных DICOM-файлов может иметь свои недостатки. Однако скорость передачи данных в программе играет немаловажную роль для пользователя.

Пятый этап метода (см. рис. 6) заключается в построении объёмной поверхности. В поставленной задаче требуется визуализация массива не менее 768x768x768 (~ 450 млн. шт.) вокселей. Для визуализации существуют различные алгоритмы, которые были рассмотрены ранее.

Для окрашивания вокселей так же используют различные методы. Благодаря появлению трёхмерных текстур можно провести окрашивание наиболее удобным и эффективным способом. Трёхмерная текстура загружается в GPU и хранится непосредственно там.

Оценим сложность построения одного куба. Для этого требуется построить 12 треугольников, т.е. использовать 36 вершин. Для их построения можно создать VBO с координатами граней куба и порядком обхода индексов по кубу, где первой координатой указывается координата точки треугольника, а второй координаты текстуры (см. листинг 7).

Листинг 7. Пример использования VBO для построения (пример библиотеки Qt)

```
VertexData vertices[] = {
    // Vertex data for face 0
    {QVector3D(-1.0f, -1.0f,  1.0f), QVector2D(0.0f, 0.0f)},  // v0
    {QVector3D( 1.0f, -1.0f,  1.0f), QVector2D(0.33f, 0.0f)}, // v1
    {QVector3D(-1.0f,  1.0f,  1.0f), QVector2D(0.0f, 0.5f)},  // v2
    {QVector3D( 1.0f,  1.0f,  1.0f), QVector2D(0.33f, 0.5f)}, // v3

    // Vertex data for face 1
    {QVector3D( 1.0f, -1.0f,  1.0f), QVector2D( 0.0f, 0.5f)}, // v4
    {QVector3D( 1.0f, -1.0f, -1.0f), QVector2D(0.33f, 0.5f)}, // v5
    {QVector3D( 1.0f,  1.0f,  1.0f), QVector2D(0.0f, 1.0f)},  // v6
    {QVector3D( 1.0f,  1.0f, -1.0f), QVector2D(0.33f, 1.0f)}, // v7

    // Vertex data for face 2
    {QVector3D( 1.0f, -1.0f, -1.0f), QVector2D(0.66f, 0.5f)}, // v8
    {QVector3D(-1.0f, -1.0f, -1.0f), QVector2D(1.0f, 0.5f)},  // v9
    {QVector3D( 1.0f,  1.0f, -1.0f), QVector2D(0.66f, 1.0f)}, // v10
    {QVector3D(-1.0f,  1.0f, -1.0f), QVector2D(1.0f, 1.0f)},  // v11

    // Vertex data for face 3
    {QVector3D(-1.0f, -1.0f, -1.0f), QVector2D(0.66f, 0.0f)}, // v12
    {QVector3D(-1.0f, -1.0f,  1.0f), QVector2D(1.0f, 0.0f)},  // v13
    {QVector3D(-1.0f,  1.0f, -1.0f), QVector2D(0.66f, 0.5f)}, // v14
    {QVector3D(-1.0f,  1.0f,  1.0f), QVector2D(1.0f, 0.5f)},  // v15

    // Vertex data for face 4
    {QVector3D(-1.0f, -1.0f, -1.0f), QVector2D(0.33f, 0.0f)}, // v16
    {QVector3D( 1.0f, -1.0f, -1.0f), QVector2D(0.66f, 0.0f)}, // v17
    {QVector3D(-1.0f, -1.0f,  1.0f), QVector2D(0.33f, 0.5f)}, // v18
    {QVector3D( 1.0f, -1.0f,  1.0f), QVector2D(0.66f, 0.5f)}, // v19

    // Vertex data for face 5
    {QVector3D(-1.0f,  1.0f,  1.0f), QVector2D(0.33f, 0.5f)}, // v20
    {QVector3D( 1.0f,  1.0f,  1.0f), QVector2D(0.66f, 0.5f)}, // v21
    {QVector3D(-1.0f,  1.0f, -1.0f), QVector2D(0.33f, 1.0f)}, // v22
    {QVector3D( 1.0f,  1.0f, -1.0f), QVector2D(0.66f, 1.0f)}  // v23
};
```

Другой вариант построения – использование геометрического шейдера. Вершинный шейдер передаёт в геометрический координату центра куба. После чего происходит прорисовка куба (см. Листинг 8).

Листинг 8. Пример использования геометрического шейдера для построения куба

```
#version 400
layout(points) in;
layout(triangle_strip, max_vertices = 14) out;
uniform mat4 proj_matrix;
uniform mat4 model_matrix;

const vec4 cubeVerts[8] = vec4[8](
    vec4(-0.5,  0.5,  0.5, 1.0),  // 0
    vec4( 0.5,  0.5,  0.5, 1.0),  // 1
    vec4( 0.5, -0.5,  0.5, 1.0),  // 2
    vec4(-0.5, -0.5,  0.5, 1.0),  // 3

    vec4(-0.5,  0.5, -0.5, 1.0),  // 4
    vec4( 0.5,  0.5, -0.5, 1.0),  // 5
    vec4( 0.5, -0.5, -0.5, 1.0),  // 6
    vec4(-0.5, -0.5, -0.5, 1.0)); // 7
const int  cubeIndices[14]  = int [14]
            ( 2, 3, 1, 0,
              4, 3, 7, 2,
              6, 1, 5, 4,
              6, 7);
void main() {
    mat4 mvp = proj_matrix*model_matrix;
    vec4 transVerts[8];

    for (int i=0; i < 8; ++i)
        transVerts[i] = mvp * (gl_in[0].gl_Position + cubeVerts[i]);

    for (int i = 0; i < 14; ++i) {
        int pt0 = cubeIndices[i];
        vec4 v0 = transVerts[pt0];
        gl_Position = v0;
        EmitVertex();
    }
    EndPrimitive();
}
```

И в первом, и во втором подходе необходимо построить 12 треугольников по 36 вершинам для получения куба.

Таким образом, классические алгоритмы построения 3D-модели требуют построения каждого вокселя по отдельности.

Работу классического алгоритма трассировки лучей следует рассмотреть подробнее.

- Сначала строятся ряды кубиков (не прорисовываются), как показано на рис. 7.

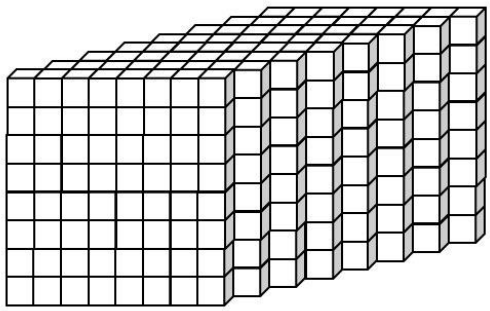

Рис. 7. Исходный массив

- окраска вокселей цветом с помощью 3D текстуры (см. рис. 8).

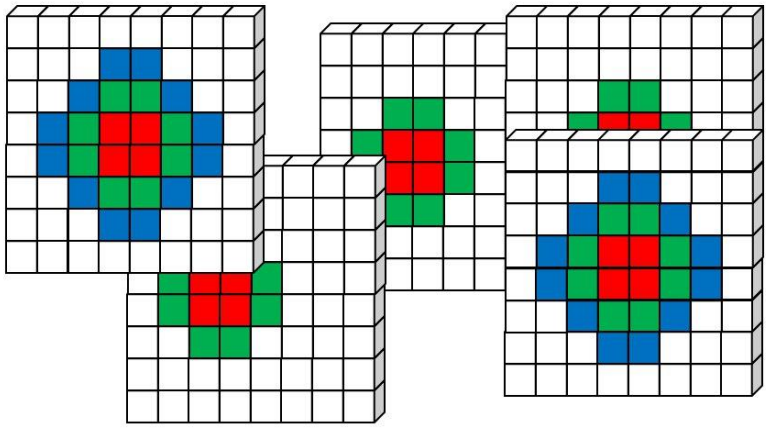

Рис. 8. Окрашенные слои вокселей

- удаление прозрачных или темных (яркость которых ниже пороговой) вокселей.

Завершающие шаги алгоритма показаны на рисунке 9.

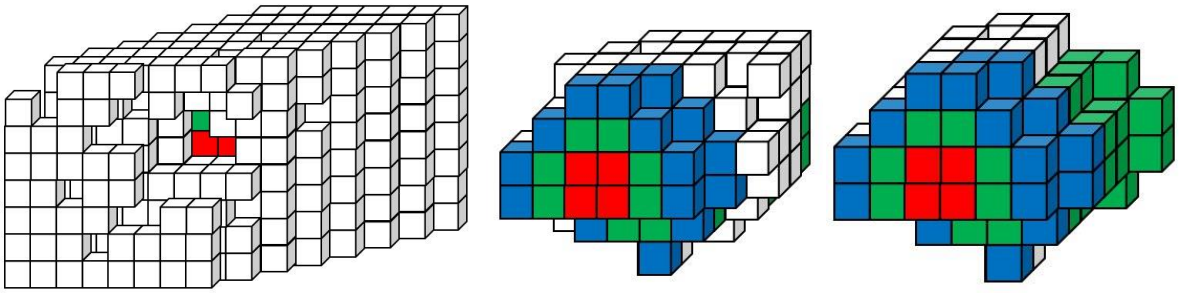

Рис. 9. Удаление прозрачных или темных вокселей. Завершающие шаги алгоритма трассировки лучей.

В этой работе предложен алгоритм, позволяющий сократить количество операций построения.

2.2.1. Оригинальный алгоритм построения 3D объекта

1. Для построения вокселей строятся поверхности – сначала вдоль оси X, затем – вдоль оси Y, и в конце вдоль оси Z, как на рисунке 10.

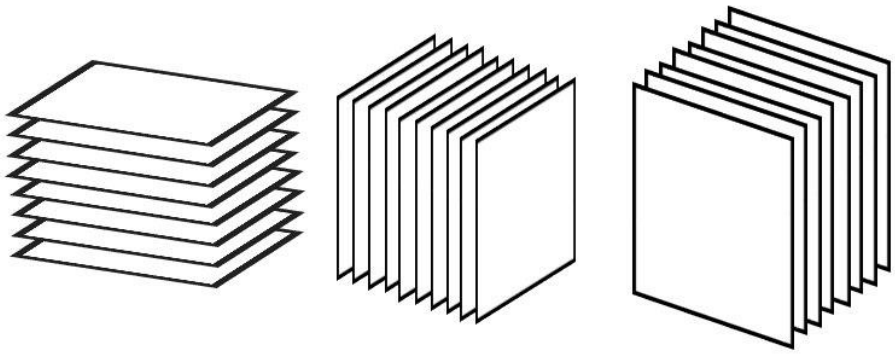

Рис. 10. Примеры построения плоскостей по отдельным осям

После чего необходимо совместить поверхности (см. рис. 11). В результате получаются аналогичные воксели, однако не требуется строить каждый куб по отдельности, необходимо построить определенное число параллельных плоскостей вдоль каждой из осей, т.е. провести гораздо меньше операций.

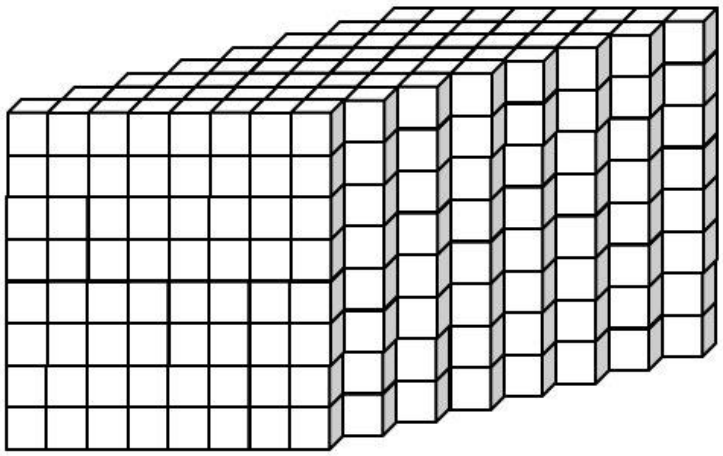

Рис.11. Массив вокселей

2. Для окрашивания плоскостей накладывается общая трёхмерная текстура. Интерполяция цветов выполняется аппаратно.

3. При отбрасывании невидимых вокселей нет необходимости использования алгоритма трассировки лучей. Используется специальная команда discard во фрагментном шейдере. В случае отбрасывания фрагмента, никакие расчёты в дальнейшем с ним производиться не будут, и фрагмент уже не попадёт в буфер кадра.

2.2.2. Реализация оригинального алгоритма построения вокселей

1. Формирование координат точек плоскостей

Для формирования массива вычисляется его размер:

int ArrSize = 3*4*(dimSize+1);

где 3 – количество осей (X, Y, Z), 4 – это количество точек, необходимое для построения квадрата в пространстве.

2. Копирование созданного массива на GPU с помощью класса QOpenGLBuffer:

```
arrayBuf.bind();
arrayBuf.allocate(vertices, ArrSize * sizeof(QVector3D));
```

3. Формирование массива индексов (см. листинг 9)

Для построения плоскостей недостаточно указать только координаты точек плоскости. Необходимо указать также порядок использования точек, для чего передаются массивы индексов. В данной задаче используется простая индексация, т.к. необходимо строить только плоскости.

Листинг 9. Формирование массива индексов

```
countIndeces = ArrSize;

GLushort* indices = new GLushort[countIndeces];

for (int i = 0; i < countIndeces; ++i)
    indices[i] = i;
```

4. Копирование массива индексов на GPU:

```
indexBuf.bind();
indexBuf.allocate(indices, countIndeces * sizeof(GLushort));
```

5. Прорисовка сцены производится с помощью функции:

```
glDrawElements(GL_QUADS, countIndeces, GL_UNSIGNED_SHORT, 0);
```

На рисунке 12 представлен результат построения плоскостей вдоль оси Y.

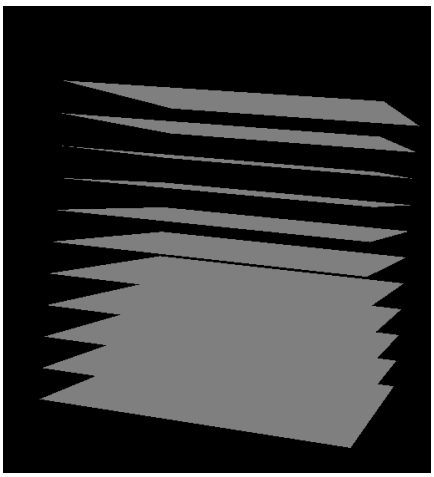

Рис. 12. Прорисовка плоскостей вдоль оси Y

6. Создание трёхмерной текстуры.

Для создания трёхмерной текстуры используется класс QOpenGLTexture. При создании этого объекта указывается, что будет создана трёхмерная текстура с помощью аргумента QOpenGLTexture::Target3D (см. листинг 10).

Листинг 10. Создание трёхмерной текстуры в OpenGL с использованием библиотеки Qt

```
QOpenGLTexture* tex = new QOpenGLTexture(QOpenGLTexture::Target3D);

tex->setFormat(QOpenGLTexture::RGBA8_UNorm);
tex->setSize(xDim, yDim, zDim);
tex->allocateStorage();

tex->setData(0, QOpenGLTexture::Red, QOpenGLTexture::UInt8, buf3d);
```

7. Отрисовка 3D-сцены

1. Выполнение на CPU

Благодаря простоте предложенного метода прорисовка плоскостей не требует выполнения сложных операций на CPU. Выполняются стандартные операции: очистка буфера цвета и буфера глубины; привязка текстуры, буфера вершин и буфера индексов к текущему контексту OpenGL; вычисление матрицы поворота и перемещения (модельно-видовой матрицы). В шейдер передаются переменные и вызывается функция прорисовки (см. листинг 11).

Листинг 11. Отрисовка 3D-сцены

```
void Widget3D::paintGL() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    texture->bind();
    QMatrix4x4 matrix;
    matrix.translate(0.0, 0.0, zoom);
    matrix.rotate(rotation);
    program.setUniformValue("mvp_matrix", projection * matrix);
    program.setUniformValue("thColor", thColor);
    geometries->drawCubeGeometry(&program);
}
```

```
void GeometryEngine::drawCubeGeometry(QOpenGLShaderProgram *program) {
    arrayBuf.bind();
    indexBuf.bind();

    int vertexLocation = program->attributeLocation("a_position");
    program->enableAttributeArray(vertexLocation);
    program->setAttributeBuffer(vertexLocation, GL_FLOAT, 0, 3, sizeof(QVector3D));

    glDrawElements(GL_QUADS, countIndeces, GL_UNSIGNED_SHORT, 0);
}
```

2. Выполнение на GPU

Для прорисовки используются два шейдера: вершинный и фрагментный.

- В вершинном шейдере координаты вершины умножаются на MVP (модельно-видовая проекционная матрица). Для извлечения данных из текстуры координаты от -1 до +1 по каждой из осей преобразуются в диапазон от 0 до 1 (см. листинг 12).

Листинг 12. Вершинный шейдер оригинального алгоритма

```
uniform mat4 mvp_matrix;
attribute vec4 a_position;
out vec3 v_texcoord;

void main() {
    gl_Position = mvp_matrix * a_position;

    v_texcoord = vec3(0.5*a_position.x+0.5,
                      0.5*a_position.y+0.5,
                      0.5*a_position.z+0.5);
}
```

- Во фрагментном шейдере с помощью функции texture3D извлекается текущий цвет вокселя (плотность). В случае если цвет вокселя ниже заданного пользователем порога, воксель отбрасывается с помощью функции discard (см. листинг 13).

Листинг 13. Фрагментный шейдер оригинального алгоритма

```
uniform sampler3D texture;
uniform float thColor;
in vec3 v_texcoord;

void main() {

    vec4 locColor = texture3D(texture, v_texcoord);

    if (locColor.r < thColor)
        discard;

    gl_FragColor = vec4(locColor.r, locColor.r, locColor.r, 1.0);
}
```

2.2.3. Тестирование алгоритма на производительность

Для проверки эффективности оригинального алгоритма было проведено тестирование на некотором наборе DICOM-файлов. Было произведено измерение частоты кадров в секунду в зависимости от количества вокселей и от объема данных папки DICOM-файлов.

Тестирование проводилось на персональном компьютере со следующими характеристиками:

CPU: Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz 2.49 GHz

ОЗУ: 8,00 Гб

Графическая карта: NVIDIA GeForce 840M

Результаты тестирования можно представлены на рисунке 13.

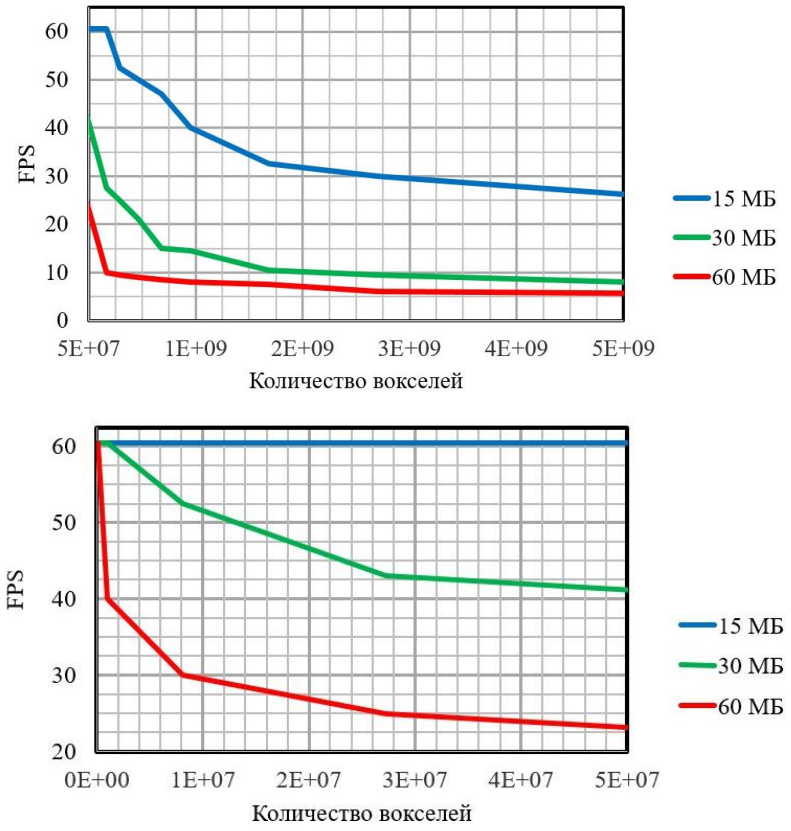

Рис. 13. График зависимости частоты кадра в секунду от количества вокселей для разных размеров текстур

Из графика следует, что при достаточно больших значениях количества вокселей значение частоты кадров в секунду позволяет пользователю взаимодействовать с трёхмерной моделью в режиме реального времени.

Глава 3. Результаты

В ходе проделанной работы был разработан программный модуль реконструкции трехмерной воксельной модели на основе анализа изображений, полученных в результате томографии. Был реализован графический интерфейс, позволяющий пользователю открывать папки с файлами dicom и открывать отдельные DICOM-файлы (см. рис. 14), а также восстанавливать 3D-модель из серии данных (см. рис. 15).

Для получения изображений требуется выбрать меню «Файл». Затем нажать «Открыть папку DICOM». (Либо просто выбрать иконку «Открыть папку»  .) При успешной загрузке во вкладке предварительного просмотра появится список файлов с полными именами и изображениями. При выборе любого из них в рабочую область разворачивается качественное изображение среза.

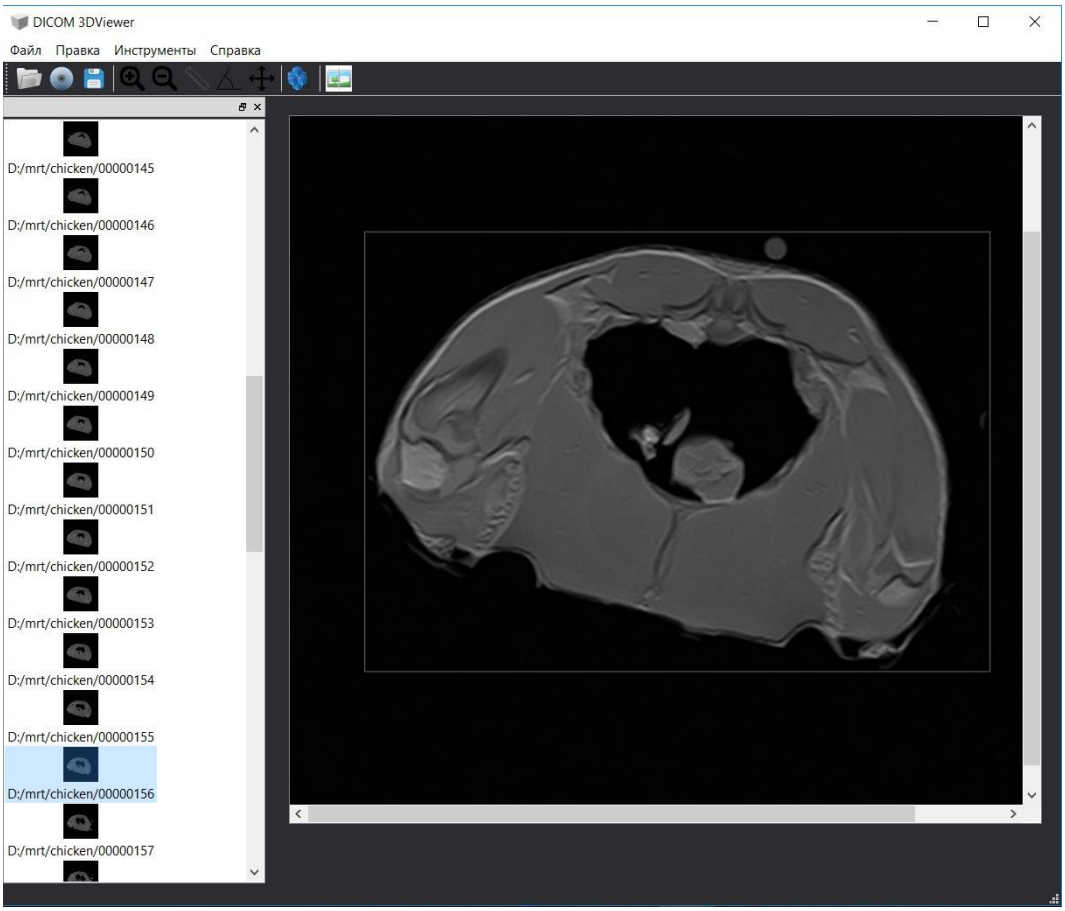

Рис. 14. Интерфейс программы в режиме просмотра снимков

Для построения трехмерного изображения следует нажать пункт меню «Правка», а затем выбрать «Панель 3D», либо просто нажать на иконку  . Для построения 3D откроется дополнительное окно и загрузится объект. Вращение и приближение осуществляется с

помощью клавиш мыши или с помощью стрелок на клавиатуре. Стоит отметить, что при приближении видны не только контуры, но и весь объем объекта.

В качестве дополнительной функциональности (обязательно используется медицинскими работниками) с помощью стрелок «вверх» и «вниз» на клавиатуре доступно изменение контрастности изображения.

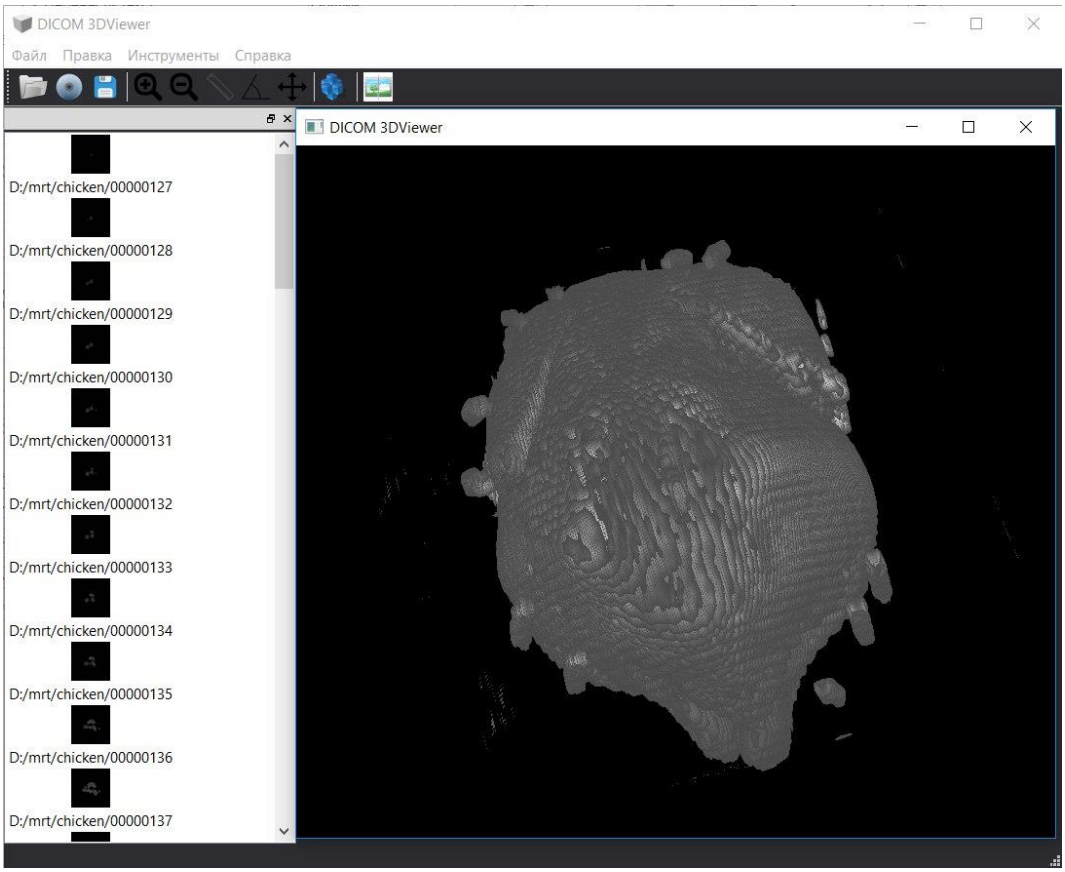

Рис.15. Интерфейс программы в режиме визуализации трехмерной модели

На рисунках 13 и 14 изображены срез МРТ курицы и объемная модель соответственно.

Кроме того, по экспериментальным данным было построено изображение МРТ головы, один из срезов изображён на рис. 16.

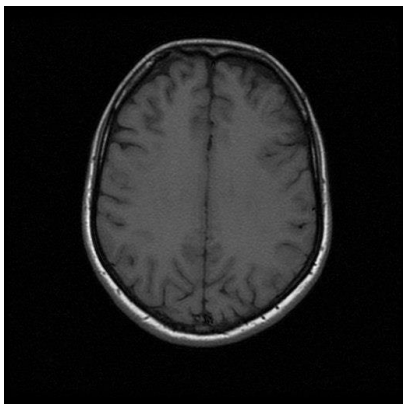

Рис. 16. МРТ головы

На рисунке 17 изображены различные положения реконструированной головы. Восстановление и визуализация были проведены с использованием оригинального алгоритма, представленного в главе 2.

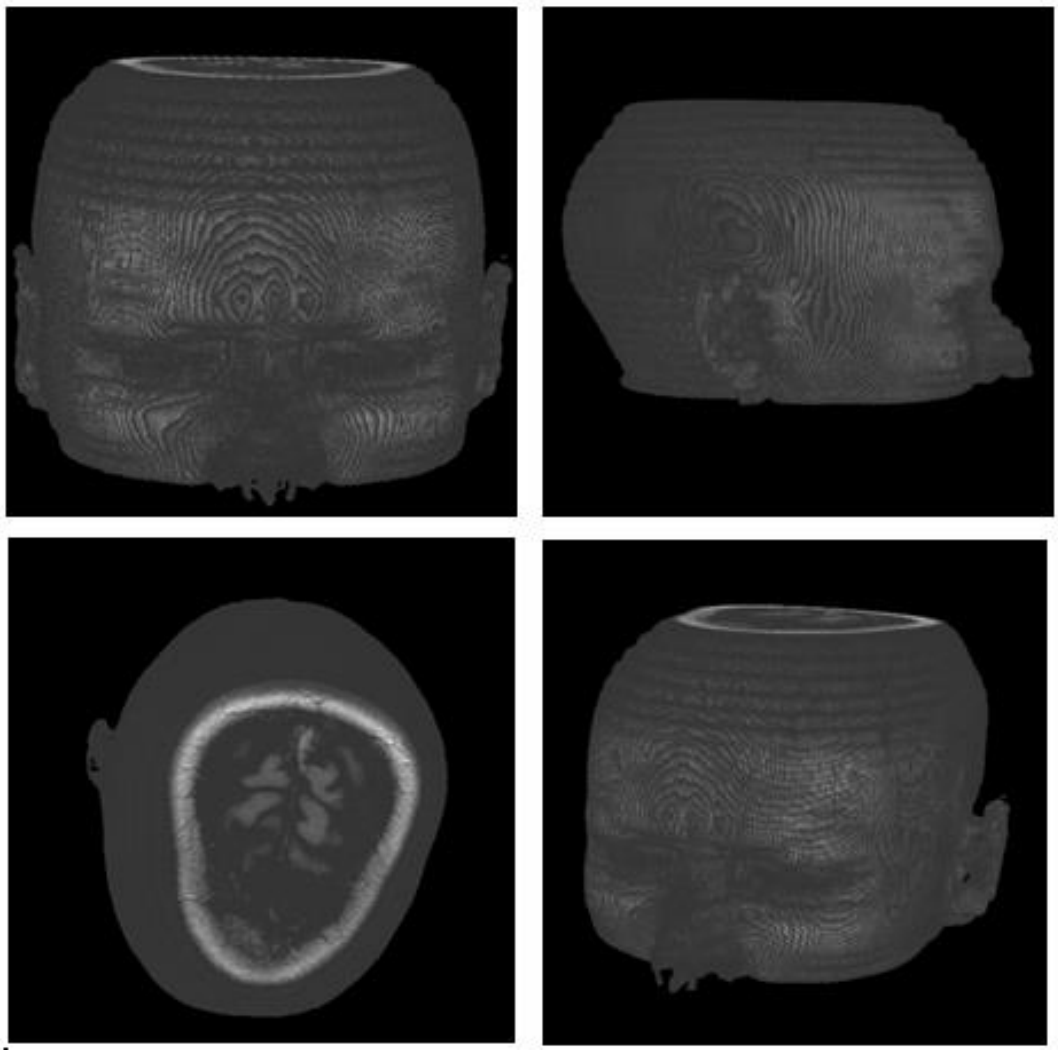

Рис. 17. Реконструкция головы

Исходные коды модулей отображения объема и виджета отображения трёхмерной сцены представлены в Приложении 1 и Приложении 2.

Выводы

– Проведён анализ существующих алгоритмов рендеринга;

– Разработан алгоритм восстановления трёхмерного изображения из серии двумерных медицинских снимков DICOM. Алгоритм позволяет отображать не только поверхность, но и весь объём трёхмерной модели, позволяет строить качественные изображения;

– Разработанный алгоритм был реализован в виде программного модуля;

– Проведено исследование эффективности разработанного алгоритма. В результате тестирования установлено, что работа алгоритма возможна в режиме реального времени;

– Разработан графический интерфейс взаимодействия пользователя с программным модулем.

Результаты работы докладывались на III международной конференции «Информационные технологии и нанотехнологии» (ИТНТ-2017), Самара: Самарский университет – 2017 и на 14 международной молодежной школе-конференции "Магнитный резонанс и его приложения - Spinus-2017", СПб – 2017. Также по результатам работы готовится публикация.

Планируется дальнейшее усовершенствование данного программного модуля и используемого алгоритма.

Литература

1. Заболеваемость и смертность от рака в мире. [Электронный ресурс] Режим доступа: http://www.oncoforum.ru/o-rake/statistika-raka/zabolevaemost-i-smertnost-ot-raka-v-mire.html
2. Поммерт, А., Пфлессер, Б., Визуализация объёма в медицине / А.Поммерт, Б.Пфлессер, М.Риемер, Т.Шиеманн, Р.Шуберт, В.Тиеде, К.Х.Хон // Открытые системы, № 5, 1996 г. [Электронный ресурс] Режим доступа: https://www.osp.ru/os/1996/05/178989, 25.05.2017.
3. Digital Imaging and Communications in Medicine [Электронный ресурс] Режим доступа: http://dicom.nema.org/
4. Хофер, М., Компьютерная томография - Базовое руководство // Изд-во: Медицинская литература. – 2008. - 203 с.
5. Джозеф П. Хорнак, Д., Основы МРТ [Электронный ресурс] Режим доступа: http://www.booksmed.com/luchevaya-diagnostika/2936-osnovy-mrt-dzhozef-hornak.html
6. Алгоритм, использующий z-буфер [Электронный ресурс] Режим доступа: http://compgraph.tpu.ru/zbuffer.htm
7. Ягель, Р., Рендеринг объемов в реальном времени // Открытые системы. СУБД, № 5, 1996 [Электронный ресурс] Режим доступа: https://www.osp.ru/os/1996/05/178968/
8. Скоков, А. А, Карих, В. П., Трехмерная визуализация результатов томографического контроля: материалы // III Междунар. Науч. Конф. (г. Казань, октябрь 2014 г.) – Казань: Бук, 2014. – 98 с.
9. Westover L., "Footprint Evaluation for Volume Rendering" // Computer Graphics. –1990. – Т 24. -№ 4. – С. 367-376.

Приложение 1. Класс, выполняющий прорисовку вокселей

Заголовочный файл geometryengine.h

```
#ifndef GEOMETRYENGINE_H
#define GEOMETRYENGINE_H

#include <QOpenGLFunctions>
#include <QOpenGLShaderProgram>
#include <QOpenGLBuffer>

#define TEXTURE_3D


class GeometryEngine : protected QOpenGLFunctions
{
public:
    GeometryEngine();
    virtual ~GeometryEngine();

    void drawCubeGeometry(QOpenGLShaderProgram *program);

private:
    void initCubeGeometry();

    QOpenGLBuffer arrayBuf;
    QOpenGLBuffer indexBuf;

    int countIndeces;
};

#endif // GEOMETRYENGINE_H
```

Файл исходных кодов geometryengine.c

```
#include "geometryengine.h"

#include <QVector2D>
#include <QVector3D>


struct VertexData
{
    QVector3D position;

#ifdef TEXTURE_3D
    QVector3D
#else
    QVector2D
#endif

    texCoord;
};

//! [0]
GeometryEngine::GeometryEngine()
    : indexBuf(QOpenGLBuffer::IndexBuffer)
{
    initializeOpenGLFunctions();

    arrayBuf.create();
```

```
    indexBuf.create();

    initCubeGeometry();
}

GeometryEngine::~GeometryEngine()
{
    arrayBuf.destroy();
    indexBuf.destroy();
}


void GeometryEngine::initCubeGeometry() {

    int dimSize = 250;


    int ArrSize = 3*4*(dimSize+1);

    QVector3D* vertices = new QVector3D[ArrSize];


    float pos = -1.0f;
    const float stepPos = 2.0f/dimSize;

    for (int i = 0; i < (dimSize+1); ++i) {
        // X
        vertices[0*4*(dimSize+1) + 4*i + 0] = QVector3D(  pos, -1.0f,  1.0f);
        vertices[0*4*(dimSize+1) + 4*i + 1] = QVector3D(  pos, -1.0f, -1.0f);
        vertices[0*4*(dimSize+1) + 4*i + 2] = QVector3D(  pos,  1.0f, -1.0f);
        vertices[0*4*(dimSize+1) + 4*i + 3] = QVector3D(  pos,  1.0f,  1.0f);
        // Y
        vertices[1*4*(dimSize+1) + 4*i + 0] = QVector3D(-1.0f,   pos,  1.0f);
        vertices[1*4*(dimSize+1) + 4*i + 1] = QVector3D(-1.0f,   pos, -1.0f);
        vertices[1*4*(dimSize+1) + 4*i + 2] = QVector3D( 1.0f,   pos, -1.0f);
        vertices[1*4*(dimSize+1) + 4*i + 3] = QVector3D( 1.0f,   pos,  1.0f);
        // Z
        vertices[2*4*(dimSize+1) + 4*i + 0] = QVector3D(-1.0f,  1.0f,   pos);
        vertices[2*4*(dimSize+1) + 4*i + 1] = QVector3D(-1.0f, -1.0f,   pos);
        vertices[2*4*(dimSize+1) + 4*i + 2] = QVector3D( 1.0f, -1.0f,   pos);
        vertices[2*4*(dimSize+1) + 4*i + 3] = QVector3D( 1.0f,  1.0f,   pos);

        pos += stepPos;
    }

    countIndeces = ArrSize;

    GLushort* indices = new GLushort[countIndeces];

    for (int i = 0; i < countIndeces; ++i)
        indices[i] = i;

    arrayBuf.bind();
    arrayBuf.allocate(vertices, ArrSize * sizeof(QVector3D));

    indexBuf.bind();
    indexBuf.allocate(indices, countIndeces * sizeof(GLushort));
```

```
    delete[] vertices;
    delete[] indices;

}


void GeometryEngine::drawCubeGeometry(QOpenGLShaderProgram *program) {
    arrayBuf.bind();
    indexBuf.bind();

    int vertexLocation = program->attributeLocation("a_position");
    program->enableAttributeArray(vertexLocation);
    program->setAttributeBuffer(vertexLocation, GL_FLOAT, 0, 3,
sizeof(QVector3D));

    glDrawElements(GL_QUADS, countIndeces, GL_UNSIGNED_SHORT, 0);
}
```

Приложение 2. Виджет, отображающий трехмерную модель

Заголовочный файл widget3d.h

```
#ifndef WIDGET3D_H
#define WIDGET3D_H

#include "geometryengine.h"

#include <QOpenGLWidget>
#include <QOpenGLFunctions>
#include <QMatrix4x4>
#include <QQuaternion>
#include <QVector2D>
#include <QBasicTimer>
#include <QOpenGLShaderProgram>
#include <QOpenGLTexture>

class GeometryEngine;

class Widget3D : public QOpenGLWidget, protected QOpenGLFunctions
{
    Q_OBJECT

public:
    explicit Widget3D(QWidget *parent, uchar* buf3d, int xDim, int yDim, int
zDim);
    ~Widget3D();


Q_SIGNALS:

    void initFinish();

protected:
    void mousePressEvent(QMouseEvent *e) Q_DECL_OVERRIDE;
    void mouseReleaseEvent(QMouseEvent *e) Q_DECL_OVERRIDE;
    void wheelEvent(QWheelEvent *e) Q_DECL_OVERRIDE;
    void keyReleaseEvent(QKeyEvent* e) Q_DECL_OVERRIDE;


    void timerEvent(QTimerEvent *e) Q_DECL_OVERRIDE;

    void initializeGL() Q_DECL_OVERRIDE;
    void resizeGL(int w, int h) Q_DECL_OVERRIDE;
    void paintGL() Q_DECL_OVERRIDE;

    void initShaders();
    void initTextures();

private:
    QBasicTimer timer;
    QOpenGLShaderProgram program;
    GeometryEngine *geometries;

    QOpenGLTexture *texture;

    QMatrix4x4 projection;

    QVector2D mousePressPosition;
    QVector3D rotationAxis;
    qreal angularSpeed;
```

```
    QQuaternion rotation;


    qreal zoom = -5.0;
    GLfloat thColor = 0.3;

    uchar* buf3d;
    int xDim, yDim, zDim;
};

#endif // WIDGET3D_H
```

Файл исходных кодов widget3d.c

```
#include "widget3d.h"

#include <QMouseEvent>

#include <QOpenGLPixelTransferOptions>
#include <math.h>

#include <QDebug>
#include <QDir>
#include <QFileInfoList>

#include <cstring>

Widget3D::Widget3D(QWidget *parent, uchar* buf3d, int xDim, int yDim, int zDim)
:
    QOpenGLWidget(parent),
    geometries(0),
    texture(0),
    angularSpeed(0),
    buf3d(buf3d),
    xDim(xDim),
    yDim(yDim),
    zDim(zDim)
{
}

Widget3D::~Widget3D() {
    makeCurrent();
    delete texture;
    delete geometries;
    doneCurrent();
}


void Widget3D::mousePressEvent(QMouseEvent *e) {
    mousePressPosition = QVector2D(e->localPos());

}

void Widget3D::mouseReleaseEvent(QMouseEvent *e)
{
    // Mouse release position - mouse press position
    QVector2D diff = QVector2D(e->localPos()) - mousePressPosition;

    // Rotation axis is perpendicular to the mouse position difference
    // vector
```

```
    QVector3D n = QVector3D(diff.y(), diff.x(), 0.0).normalized();

    // Accelerate angular speed relative to the length of the mouse sweep
    qreal acc = diff.length() / 100.0;

    // Calculate new rotation axis as weighted sum
    rotationAxis = (rotationAxis * angularSpeed + n * acc).normalized();

    // Increase angular speed
    angularSpeed += acc;
}

void Widget3D::wheelEvent(QWheelEvent *e)
{

    zoom += (e->delta()/250);
    repaint();
}

void Widget3D::keyReleaseEvent(QKeyEvent *e)
{
    if (Qt::Key_Up == e->key())
        thColor += 0.01;
    if (Qt::Key_Down == e->key())
        thColor -= 0.01;
    repaint();
}


void Widget3D::timerEvent(QTimerEvent *)
{
    // Decrease angular speed (friction)
    angularSpeed *= 0.99;

    // Stop rotation when speed goes below threshold
    if (angularSpeed < 0.01) {
        angularSpeed = 0.0;
    } else {
        // Update rotation
        rotation = QQuaternion::fromAxisAndAngle(rotationAxis, angularSpeed) *
rotation;

        // Request an update
        update();
    }
}



void Widget3D::initializeGL()
{
    initializeOpenGLFunctions();

    glClearColor(0, 0, 0, 1);

    initShaders();
    initTextures();

    // Enable depth buffer
    glEnable(GL_DEPTH_TEST);
```

```
    geometries = new GeometryEngine;

    // Use QBasicTimer because its faster than QTimer
    timer.start(12, this);

    initFinish();
}


void Widget3D::initShaders()
{
    // Compile vertex shader
    if (!program.addShaderFromSourceFile(QOpenGLShader::Vertex,
":/vshader.glsl"))
        close();

    // Compile fragment shader
    if (!program.addShaderFromSourceFile(QOpenGLShader::Fragment,
":/fshader.glsl"))
        close();

    // Link shader pipeline
    if (!program.link())
        close();

    // Bind shader pipeline for use
    if (!program.bind())
        close();
}

void Widget3D::initTextures() {

    int xyzSize = xDim*yDim*zDim;

    QOpenGLTexture* tex = new QOpenGLTexture(QOpenGLTexture::Target3D);

    tex->setFormat(QOpenGLTexture::RGBA8_UNorm);
    tex->setSize(xDim, yDim, zDim);
    tex->allocateStorage();

    tex->setData(0, QOpenGLTexture::Red, QOpenGLTexture::UInt8, buf3d);


    tex->setWrapMode(QOpenGLTexture::Repeat);

    tex->setMagnificationFilter(QOpenGLTexture::Linear);
    tex->setMinificationFilter(QOpenGLTexture::Linear);

    texture = tex;

    qDebug() << "Ready";
    qDebug() << "Size: " << xyzSize/(1024.0*1024.0) << " Mb";
    qDebug() << "Ready";

}

void Widget3D::resizeGL(int w, int h)
{
    // Calculate aspect ratio
    qreal aspect = qreal(w) / qreal(h ? h : 1);
```

```
    // Set near plane to 3.0, far plane to 7.0, field of view 45 degrees
    const qreal zNear = 1.0, zFar = 200.0, fov = 45.0;

    // Reset projection
    projection.setToIdentity();

    // Set perspective projection
    projection.perspective(fov, aspect, zNear, zFar);
}


void Widget3D::paintGL() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    texture->bind();
    QMatrix4x4 matrix;
    matrix.translate(0.0, 0.0, zoom);
    matrix.rotate(rotation);
    program.setUniformValue("mvp_matrix", projection * matrix);
    program.setUniformValue("thColor", thColor);
    geometries->drawCubeGeometry(&program);
}
```