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

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра « Прикладная математика и информатика»
(наименование кафедры)
01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки, специальности)
Системное программирование и компьютерные технологии
(направленность (профиль)/специализация

БАКАЛАВРСКАЯ РАБОТА

на тему **Моделирование процессов обнаружения контура зуба на 3D модели челюсти**

Студент	Д.А. Насонов	(личная подпись)
Руководитель	Т.Г. Султанов	(личная подпись)
Консультанты	Н.В. Андрюхина	(личная подпись)

Допустить к защите

Заведующий кафедрой к.т.н., доцент, А.В. Очеповский_______________
(ученая степень, звание, И.О. Фамилия) (личная подпись)
«_____»______________________20_____г.

Тольятти 2019

АННОТАЦИЯ

Бакалаврскую работу выполнил студент Тольяттинского государственного университета, Насонов Дмитрий Александрович. Название работы: «Моделирование процессов обнаружения контура зуба на 3D модели челюсти». Актуальность работы заключается в том, что при разработке стоматологических элайнеров, врачам приходится сталкиваться с множеством различных проблем. Выделение зуба на STL модели челюсти является основной проблемой при моделировании модели элайнера. Для решения данной проблемы, было решено разработать и сравнить два алгоритма: ручного и автоматического выделения зуба на основе определения контура зубной коронки на челюсти.

Объект исследования: процесс выделения контура зубной коронки.

Предмет исследования: алгоритм Дийкстры, метод парного сравнения, алгоритм Саати.

Цель: моделирование и реализация алгоритмов выделения контура зубной коронки на STL модели челюсти для определения наиболее удобного на практике и упрощения процесса разработки элайнера.

Первая глава работы посвящена постановке цели разработки алгоритмов обнаружения контура зуба на 3D модели челюсти.

Во второй главе описывается процесс моделирования алгоритмов выделения зубной коронки.

Третья глава посвящена настройке окружения для разработки алгоритмов.

Четвертая глава описывает процесс реализации и оптимизации алгоритмов.

Пятая глава посвящена тестированию алгоритмов и определению наиболее эффективного из них

Бакалаврская работа выполнена на 47 страницах, состоит из введения, пяти глав, заключения, списка литературы, состоящего из 11 литературных источников и 38 рисунков и 0 таблиц.

ABSTRACT

This bachelor's graduation work is made by the student of Togliatti state university, Nasonov Dmitrii Alexandrovich. The work's title is "The modeling of processes for the contour detection of dental crowns for 3D dental model segmentation".

The modern way of fixing the direction of the teeth growth is by using plastic or silicon aligners. Due to the method's popularity, dentists are now in need of the proper software that will ease the process of creating such aligners. The biggest time consuming problem is to separate dental crown models from the jaw model. To solve this problem, we are going to simulate, implement and compare two algorithms of detecting a dental crown's contour on the STL model of the jaw.

The object of this work is a process of dental crown detection.

The subjects of this work are Djikstra and Saati algorithms and pairwise comparison method.

The goal of the graduation work is to model and implement dental crown contour detection algorithms, to determine the best one for using it in practice.

This work includes five chapters.

The first chapter shows the purposes of creating dental crown contour detection algorithms.

Second chapter displays a process of modeling mentioned algorithms.

The third one shows a process of implementation and optimization for mentioned algorithms.

The last chapter focused on testing, analyzing and determining the most effective algorithm.

The graduation thesis is 47 pages, which consist of an introduction, three chapters, conclusion, references from 11 literature sources, 38 figures and 0 tables.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ ... 2
Глава 1 ПОСТАНОВКА ЦЕЛИ РАЗРАБОТКИ АЛГОРИТМА ОБНАРУЖЕНИЯ КОНТУРА ЗУБА НА 3D МОДЕЛИ ЧЕЛЮСТИ ... 3
Глава 2 РАЗРАБОТКА АЛГОРИТМА АВТОМАТИЧЕСКОГО ВЫДЕЛЕНИЯ КОНТУРА ЗУБА ... 9
_____2.1 Разработка алгоритма ... 9
_____2.2 Оптимизация алгоритма ... 17
Глава 3 РЕАЛИЗАЦИЯ АЛГОРИТМОВ ВЫДЕЛЕНИЯ КОНТУРА ЗУБА ... 19
_____3.1 Подготовка сторонних библиотек ... 19
_____3.2 Реализация алгоритма ручного выделения ... 22
_____3.3 Реализация алгоритма автоматического выделения ... 27
Глава 4 АНАЛИЗ РАБОТЫ АЛГОРИТМОВ ... 33
ЗАКЛЮЧЕНИЕ ... 41
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ ... 43

ВВЕДЕНИЕ

Современная медицина за последние десятилетия поднялась на недосягаемый ранее уровень. Значительно улучшилась техническая оснащенность медицинских учреждений, появилась возможность диагностировать заболевание на самой ранней стадии, обеспечить быстрейшее выздоровление и восстановление работоспособности обратившегося за помощью человека.

В связи с появлением и обширным распространением технологий по сканированию 3D объектов, стало возможно дешево и быстро изготавливать различные макеты, так необходимые в медицине.

Ортодонты так же нашли способ применения технологии 3D сканирования: Клиники по всему миру, начали использовать новый способ выравнивания зубов и исправления прикуса. Суть технологии заключается в том, что по отсканированной полости рта пациента создается специальная пластиковая или силиконовая каппа, называемая элайнером. Он накладывается поверх зубов и создает давление на зубной ряд, тем самым изменяя направление роста первых.

Недавно открытый ортодонтами способ выпрямления зубов набирает популярность, а с ней приходит и необходимость в новом программном обеспечении. С помощью слепков челюсти и специальных сканеров, получается 3D модель челюстей, с которой и должна работать программа.

К сожалению, на данный момент работа врачей-ортодонтов занимает много времени, так как алгоритмы выделения контура зуба выполняются вручную и неудобны в использовании.

Целю данной работы станет разработка алгоритма, автоматически выбирающего ключевые точки на коронке зуба для его последующего преобразования в отдельный объект.

Глава 1 ПОСТАНОВКА ЦЕЛИ РАЗРАБОТКИ АЛГОРИТМА ОБНАРУЖЕНИЯ КОНТУРА ЗУБА НА 3D МОДЕЛИ ЧЕЛЮСТИ

Не скоординированное направление роста зубов (неправильный прикус) может вызывать физиологические, физические и социальные аспекты, влияющие на качество жизни человека, так как здоровье полости рта является основой общего состояния здоровья. На сегодняшний день, внимание сосредоточено на наименее агрессивных процедурах ортодонтального лечения, позволяющих пациентам достигать ожидаемых результатов. Современная технология лечения неправильного прикуса с помощью пластиковых элайнеров появилась в США в конце девяностых годов благодаря внедрению компьютерных технологий в стоматологические дисциплины. На данный момент, в процессе создания элайнеров очень сложен, и включает в себя большой процент ошибки из за человеческого фактора.

Компьютерные алгоритмы, используемые при диагностике дефектов полости рта, нуждаются в цифровых моделях челюсти пациента. Существует три наиболее часто используемых пути получения таких моделей:

- Компьютерная томография (КТ) и конусно-лучевая компьютерная томография (КЛКТ);
- Прямое внутриротовое сканирование зубного ряда;
- Лазерное/оптическое сканирование гипсовых моделей.

В последние несколько лет, КЛКТ получило большую популярность в ортодонтической практике ввиду возможности трехмерного сканирования челюсти и зубов. В отличие от обыкновенных КТ сканеров, КЛКТ подходит для использования в клинической стоматологии, где стоимость и доза радиации является серьезным ограничением. Реконструкция цифровой модели челюсти на основе КЛКТ включает в себя анализ множества срезов, на которых область зуба может быть выделена путем изучением интенсивности серого цвета пикселя [2].

Внутриротовые сканеры были введены в стоматологическую практику для замены процедуры снятия слепка челюсти сравнительно недавно. Такой сканер очень прост в использовании и генерирует стереолитографические (STL) файлы, которые применяются для создания цифровых моделей. Специальное программное обеспечение, поставляемое совместно с внутриротовыми сканерами, используется для планирования зубного лечения [3].

Несмотря на это, многие зубные клиники не могут позволить себе использование дорогого оборудования или попросту не хотят полагаться на рабочий процесс предоставляемый производителями сканеров. Тем не менее, планирование зубного лечения могло бы воспользоваться положительными чертами 3D реконструкции. Несмотря на то, что таким способом можно сканировать только видимые поверхности, у него есть и много плюсов, таких как легкость в использовании, само-калибровка и автоматическая коррекция искажений изображения [4]. В данной работе, мы сосредоточимся на цифровых моделях, сгенерированных 3D сканерами по гипсовым моделям челюсти или внутриротовыми сканерами в STL формате.

Коррекция прикуса производится при помощи изменения расположения зубов. Для этого необходимо извлечь модели отдельных зубов из исходной модели. Выделение индивидуальных моделей зубов в модели челюсти является важным этапом перед выполнением перемещения и вращения моделей зубов.

В последнее время были подробно изучены методы сегментации 3D моделей [5]. В том числе были предложены подходы, предназначенные для решения проблемы выделения зубов. Среди них есть как автоматизированный, на основе проставления узловых точек и сечения плоскостью [6], а также полностью автоматический алгоритм, тоже основанный на методике сечения плоскостью [7].

Эти приемы генерируют горизонтальную плоскость, параллельную оклюзионной плоскости. Эта плоскость может перемещаться пользователем

вверх и вниз. Далее, для каждой точки цифровой модели челюсти определяется, по какую сторону плоскости она лежит. Следующим шагом программа генерирует вертикальные плоскости, разделяющие зубы. К модели зуба относятся все точки лежащие между двумя определенными вертикальными плоскостями. Различие между этими двумя подходами заключается в способе определения секущей горизонтальной плоскости. В отличие от алгоритма [6], использующего контрольные точки, выбираемые пользователем, для определения расположение плоскости, алгоритм [7] полностью автоматический. Главный недостаток этого метода заключается в том, что контур зубной коронки не может быть получен проверкой пересечения с плоскостью.

Кроме того можно выделить ряд исследований о выделении моделей зубов в целях идентификации человека. В автоматическом алгоритме сегментации для зубной биометрии [8] зубная область вычисляется при помощи информации о геометрической кривизне. Результаты предварительного анализа результатов работы данного алгоритма выглядят хорошо. Однако, полная реализация контурной модели является довольно сложной задачей, так как выбор коэффициентов модели произволен и получается эмпирическим путем.

На данный момент, для создания моделей капп изменения положения зубов, необходимо пройти несколько этапов обработки STL модели челюсти пациента:

1. Загрузка модели. Пользователь должен получить модель челюсти в STL формате и сохранить ее на диске. При помощи программы, он выбирает место на диске, где находится файл с моделью и загружает ее в программу.

2. Реконструкция модели. При помощи специальных алгоритмов, модель очищается от дефектов, возникших в процессе сканирования челюсти пациента.

3. Выделение и отсечение зубов. Пользователь определяет границы зубов, с которыми необходимо будет работать при помощи инструментов

выделения и выявления границ зуба. Коррекция прикуса производится при помощи изменения расположения зубов. Для этого необходимо извлечь модели отдельных зубов из исходной модели. Выделение индивидуальных моделей зубов в модели челюсти является важным этапом перед выполнением перемещения и вращения моделей зубов.

4. Составление модели желаемого прикуса. Далее, при помощи инструментов вращения и перемещения зубов, врач определяет новое состояние челюсти, определяющее правильный прикус.

5. Анимирование движения зуба. К сожалению, часто бывает так, что одной каппы не достаточно для того, чтобы сместить зуб на желаемое расстояние. Один элайнер редко может сместить зуб даже на один миллиметр, тогда как на практике перемещения могут достигать половины сантиметра. Для решения этой проблемы, процесс выравнивания положения зубов делится на этапы, в результате которых генерируются несколько капп на протяжении всей процедуры лечения. Каждая отдельная каппа смещает зуб на небольшое расстояние, после чего меняется на следующую, в плоть до достижения конечного результата. Программа запоминает как начальное, так и конечное положение каждого отдельного зуба, и предлагает вариации вышеописанных этапов будущего положения зубов пациента. Врач выбирает наиболее оптимальный вариант или определяет свой.

6. Генерация моделей элайнеров. Перед печатью, программа генерирует модели капп для всех этапов коррекции прикуса, которые внимательно осматриваются ортодонтами-технологами, и при необходимости вносятся необходимые правки.

Не сложно заметить, что весь процесс занимает много времени и сил, а так же имеет шанс на человеческую ошибку практически на каждом этапе производства каппы, от снятия слепка, до печати конечных моделей элайнеров. Поэтому главной задачей данной работы, является сокращение количества действий, которых необходимо совершить врачу, в процессе создания модели элайнера. Это позволит уменьшить шанс на ошибку во

время работы, и позволит специалисту сосредоточить внимание на детали, которые программа не может учесть.

В последнее время были подробно изучены методы сегментации 3D моделей. В том числе были предложены подходы, предназначенные для решения проблемы выделения зубов. Среди них есть как автоматизированный, на основе проставления узловых точек и сечения плоскостью, а также полностью автоматический алгоритм, тоже основанный на методике сечения плоскостью.

Эти приемы генерируют горизонтальную плоскость, параллельную оклюзионной плоскости. Эта плоскость может перемещаться пользователем вверх и вниз. Далее, для каждой точки цифровой модели челюсти определяется, по какую сторону плоскости она лежит. Следующим шагом программа генерирует вертикальные плоскости, разделяющие зубы. К модели зуба относятся все точки лежащие между двумя определенными вертикальными плоскостями. Различие между этими двумя подходами заключается в способе определения секущей горизонтальной плоскости. В отличие от алгоритма ручного выделения, использующего контрольные точки, выбираемые пользователем, для определения расположения плоскости, алгоритм произвольного выбора контура по двум контрольным точкам, полностью автоматический. Главный недостаток этого метода заключается в том, что контур зубной коронки не может быть получен проверкой пересечения с плоскостью.

Кроме того можно выделить ряд исследований о выделении моделей зубов в целях идентификации человека. В автоматическом алгоритме сегментации для зубной биометрии зубная область вычисляется при помощи информации о геометрической кривизне. Результаты предварительного анализа результатов работы данного алгоритма выглядят хорошо. Однако полная реализация контурной модели является довольно сложной задачей, так как выбор коэффициентов модели произволен и получается эмпирическим путем.

Наш подход к извлечению моделей зубов как отдельных объектов основан на определении контура зубной коронки. Определение контура зубной коронки может быть реализовано двумя способами:

- ручное выделение по точкам;
- автоматизированное выделение.

На данный момент, реализовано лишь ручное выделение, суть его заключается в том, что пользователь ставит контрольные точки на модели челюсти в местах, где видна граница десны с зубной коронкой. Проблема этого способа заключается в том, что данный процесс весьма время-затратен и требует большого количества внимания. В результате врачу необходимо выделить все тридцать два зуба, для дальнейшей генерации поведения зубов.

Второй способ сильно сокращает время, затраченное на выделение каждого зуба, так как большая часть выделения происходит автоматически, а пользователю лишь необходимо определить границы коронки и отредактировать ключевые точки, которые алгоритм выбрал не достаточно точно.

Глава 2 РАЗРАБОТКА АЛГОРИТМА АВТОМАТИЧЕСКОГО ВЫДЕЛЕНИЯ КОНТУРА ЗУБА

2.1 Разработка алгоритма

Проблема определения контура зуба может быть кратко сформулирована следующим образом. Имеется множество точек $P = \ x, y, z$ исходной модели челюсти. Необходимо выбрать подмножество точек $P_c \in P$, которое определяет границу между зубной коронкой и челюстью.

Для решения данной проблемы использован нечеткий многокритериальный анализ с применением попарного сравнения. Многокритериальный анализ - это важная задача принятия решений, разработанная для сложных проблем. Стоит отметить, что намного сложнее определить важность конкретного параметра, чем определить лучший из двух при проведении попарных сравнений.

Попарное сравнение используется во избежание прямого указания значения весовых коэффициентов или оценок. Пусть $P = \{p_0, p_1, \dots, p_n\}$ множество точек, являющееся субъектом многокритериального анализа, P_{start} и P_{end} контрольные точки, определенные пользователем, $C = \{c_0, c_1, c_2, c_3\}$ множество критериев оценки точек.

c_0– расстояние до контрольной точки P_{start} или P_{end};

c_1– значение координаты y;

c_2– угол между завихренностью и вектором($P_{start}P_{end}$);

c_3 - значение градиента.

На рисунках 1-4 можно заметить, что точки, принадлежащие контуру, расположены в местах, где угол между завихренностью и вектором ($P_{start}P_{end}$) меняется и становится близок к 90 градусам. Градиент по координате Y так же как и завихренность может указать на расположение точек контура. На рисунках 5-7 видно, что точки контура расположены в местах, где градиент меняется от максимального значения и становится

близким к минимальному. А на рисунке 8, видно, что точки контура расположены там, где направление завихренности резко изменяется. Желтой линией, на рисунке, помечена примерная граница контура, красными стрелками – направление завихренности. Оба параметра дают похожие результаты, но в то же время они могут использоваться совместно для определения точек, которые сложны для распознавания, такие как области возле контрольных точек.

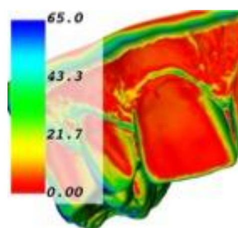

Рис. 1. Отклонение от угла между завихренностью и $(P_{start}P_{end})$ от 90 градусов

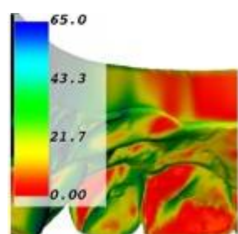

Рис. 2. Отклонение от угла между завихренностью и $(P_{start}P_{end})$ от 90 градусов

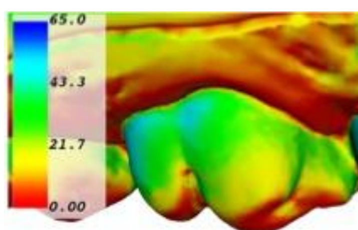

Рис. 3. Отклонение от угла между завихренностью и $(P_{start}P_{end})$ от 90 градусов

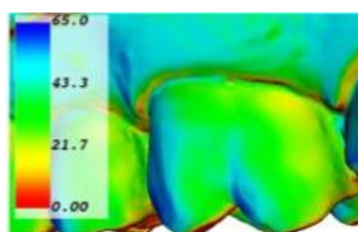

Рис. 4. Отклонение от угла между завихренностью и $(P_{start}P_{end})$ от 90 градусов

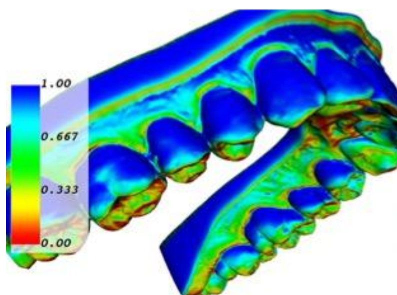

Рис. 5. Градиент по координате Y вид сбоку

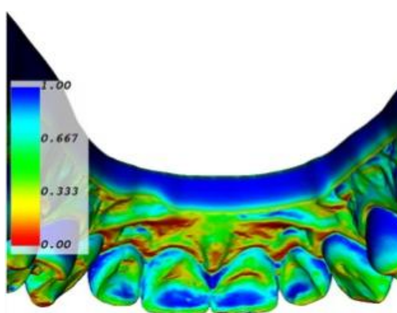

Рис. 6. Градиент по координате Y вид изнутри

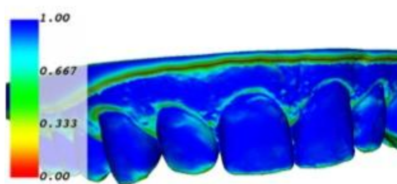

Рис. 7. Градиент по координате Y вид спереди

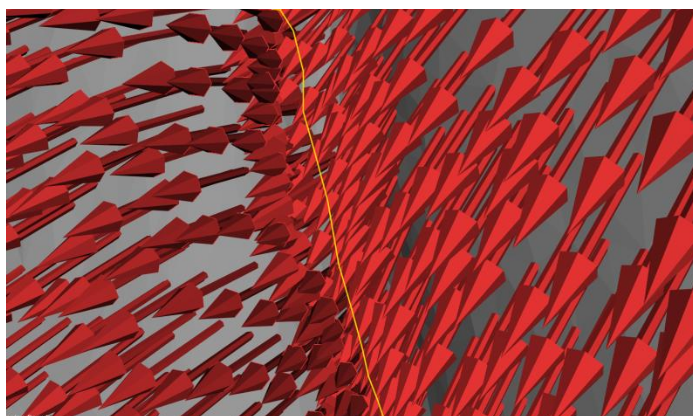

Рисунок 8 – Резкое изменение направления завихренности в точках, принадлежащих искомому контуру

Для упорядочивания элементов из множества Р в соответствии с критериями множества С предложено использовать следующие шаги:

- зададим нечеткое множество C, определенное на универсальном множестве Р с функцией степени принадлежности $m^l(p_i)$ для каждого критерия С;
- определим функции степени принадлежности нечеткого множества на основе попарных сравнений элементов Р;
- применим концентрацию $w_l, l = 1..m$ к функциям степени принадлежности для того чтобы учесть вариативные коэффициенты важности критериев;
- ранжируем все элементы Р на основе пересечения нечетких множеств-критериев, опираясь на схему Беллмана-Заде [9].

Для определения функции степени принадлежности необходимо сформировать матрицы попарных сравнений для каждого критерия. Общее число матриц совпадает с количеством параметров. Для параметра c_l, матрица попарных сравнений будет выглядеть следующим образом:

$$A\ c_l = \begin{matrix} a_{11}^l & a_{12}^l & \dots & a_{1n}^l \\ a_{21}^l & a_{22}^l & \dots & a_{2n}^l \\ \dots & \dots & \dots & \dots \\ a_{n1}^l & a_{n2}^l & \dots & a_{nn}^l \end{matrix} \quad (1)$$

где $a_{ij}^l = 1$ если $i = j$, $a_{ij}^l = p_i^l / p_j^l$ для c_3 (для максимизации критерия) и $a_{ij}^l = p_j^l / p_i^l$для остальных критериев (для минимизации). Матрица попарного сравнения критериев является диагональной, симметричной и транзитивной.

Таким образом, для определения функций степени принадлежности, необходимых для формирования нечеткого множества, может быть применена следующая формула [10]:

$$m^l\ p_i\ = 1/(a_{1i}^l + a_{1i}^l + \cdots + a_{ni}^l) \quad (2)$$

Коэффициенты относительной важности критериев $w_l, l = 1..m$ таким же образом могут быть рассчитаны по формуле функции степени принадлежности (2) и соответственно должна быть сформирована матрица попарного сравнения критериев (1). Так как в нашем случае нет количественной характеристики важности критерия, мы можем применить девятибалльную шкалу Саати [11]. В таком случае элемент a_{ij}^l матрицы А равен: 1, если критерий c_1 не важнее критерия c_2; 3, если критерий c_1 не сильно важнее критерия c_2; 5, если критерий c_1 умеренно важнее критерия c_2; 7, если критерий c_1 значительно важнее критерия c_2 ; 9, если критерий c_1 критически важен по сравнению с критерием c_2.

Нечеткое множество D, необходимое для анализа точек, определяется как пересечение:

$$D = \frac{min_1^m\ m^l\ p_1\quad ^w}{p_1}, \frac{min_1^m\ m^l\ p_2\quad ^w}{p_2}, \dots, \frac{min_1^m\ m^l\ p_n\quad ^w}{p_n} \tag{3}$$

где $m^l\ p_i$ функция степени принадлежности элемента p_i нечеткому множеству критериев c_l. Степень w производит концентрацию нечеткого множества c_l в соответствии с коэффициентом относительной важности критерия $c_l \in C$. Анализируя полученное нечеткое множество D, лучшим вариантом по всем критериям будет тот, степень принадлежности которого наибольшая.

Полагаясь на результаты исследования реализованного алгоритма, были сформированы два набора коэффициентов относительной важности. Один из них используется для постепенного спуска от контрольной точкиP_{start}, другой, для быстрого приближения к точке P_{end}.

Значения попарного сравнения критериев для первого случая следующие: критерий c_0 не важнее критерия c_2 ;критерий c_0 не важнее критерия c_3 ;критерий c_1 значительно важнее параметра c_0 .Значения попарного сравнения критериев для второго случая следующие критерий c_0 не сильно важнее критерия c_2 ;критерий c_0 умеренно важнее критерия

c_3 ;критерий c_0 значительно важнее параметра c_1 .В итоге, мы получили следующие матрицы попарного сравнения:

$$A\ C_1 = \begin{matrix} 1 & 0.14 & 1 & 1 \\ 7 & 1 & 7 & 7 \\ 1 & 0.14 & 1 & 1 \\ 1 & 0.14 & 1 & 1 \end{matrix} \quad (4)$$

$$A\ C_2 = \begin{matrix} 1 & 7 & 3 & 5 \\ 0.14 & 1 & 0.43 & 0.71 \\ 0.33 & 2.33 & 1 & 1.7 \\ 0.2 & 1.4 & 0.6 & 1 \end{matrix} \quad (5)$$

По формуле (2) и матрицам $A(C_1)$ и $A(C_2)$ получаем следующие коэффициенты важности: $w_0 = 0.1, w_1 = 0.7, w_2 = 0.1, w_3 = 0.1$ и $w_0 = 0.6, w_1 = 0.08, w_2 = 0.2, w_3 = 0.12$.

Итеративный анализ каждый выбранной точки множества P при помощи выбранных матриц $A\ C_1$ и $A\ C_2$ определит множество $P_c \in P$ точек, образующих контур зубной коронки.

Для соединения точек контура в замкнутый контур, необходимо определиться с алгоритмом, определяющим точки контура, лежащие между контрольными точками, выбранными на основе полученных коэффициентов.

Все полученные контрольные точки находятся на вершинах полигонов модели челюсти, а это значит что точки контура, находящиеся между ними располагаются так же на вершинах полигонов. Тогда путь контура будет кратчайшим расстоянием между контрольными точками на графе, в котором узлы – вершины полигонов, а пути – их ребра. Это значит, что задачу нахождения множества точек контура соединяющего контрольные точки можно свести к задаче поиска кратчайшего маршрута в неориентированном (двунаправленном) графе.

На данный момент, наиболее эффективным алгоритмами поиска кратчайшего маршрута на графе являются алгоритмы Дейкстры и Беллмана-Форда. Для выбора наиболее эффективного алгоритма, сравним их асимптотики, особенности и эффективность на графах небольшой глубины.

Алгоритм Дейкстры является оптимизацией алгоритма обхода в ширину, и имеет асимптотику V^2, при небольшой глубине. Особенностью алгоритма является невозможность применения к графам с отрицательными весами путей. Алгоритм так же потребляет достаточно много памяти, а так же сложнее в реализации.

Алгоритм Беллмана-Форда же имеет асимптотику $V * E$, но позволяет находить маршрут в графе с отрицательными путями.

Сравнив преимущества алгоритмов, было решено использовать алгоритм Дейкстры. Выбор обусловлен его большой скоростью работы по сравнению с алгоритмом Беллмана-Форда, а так же отсутствием в модели челюсти ребер полигонов с отрицательной длиной. Проигрыш в размере потребляемой памяти является критичным за счет того, что в конкретной реализации, граф будет относительно небольшой величины, ведь его задача в соединении контрольных точек, что и так располагаются достаточно близко друг к другу.

На рисунке 9, представлена диаграмма процессов разработанного алгоритма.

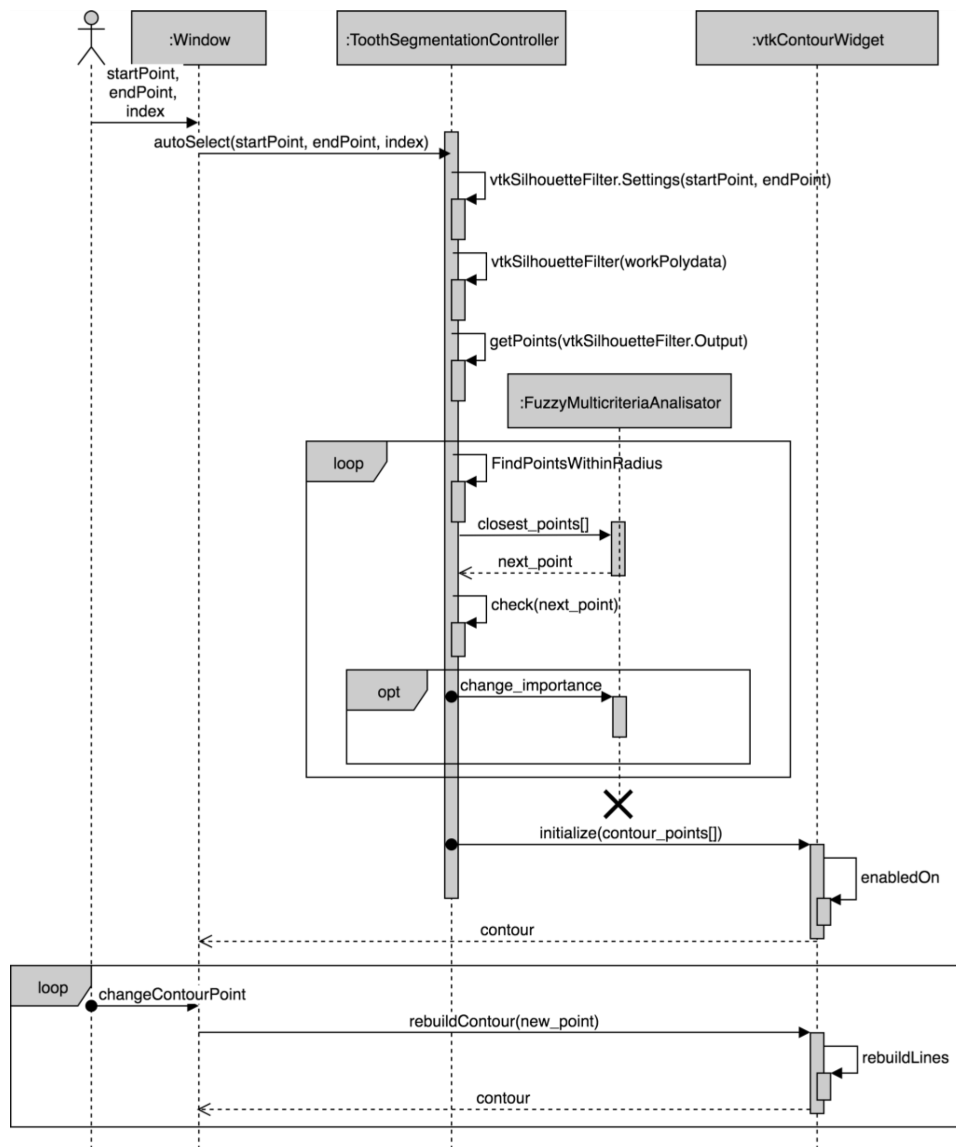

Рисунок 9 – Диаграмма процессов алгоритма автоматического выделения контура зуба

2.2 Оптимизация алгоритма

Анализ всего множества точек может быть очень ресурсозатратным. В целях уменьшения числа точек для проведения многокритериального анализа, использован фильтр VTK vtkPolyDataSilhouette, который извлекает подмножество границ полигональной сетки для генерации контура (силуэта) соответствующего 3D объекта. Для получения достаточного множества точек, определяющих контур, vtkPolyDataSilhouette должен быть инициализирован для каждого зуба независимо. Определение позиции и фокальной точки камеры позволяют настроить фильтр индивидуально для каждого зуба. Пример применения фильтров vtkPolyDataSilhouette для зуба 12 представлен на рисунках 10-12. Направление камеры для каждого случая обозначено стрелкой. Использование vtkPolyDataSilhouette значительно снижает число точек для анализа. В то же время, главным недостатком данного подхода является недостаток точек в местах, где граница определена нечетко.

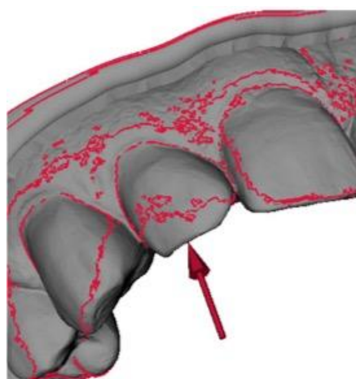

Рис. 10. Применение фильтра vtkPolyDataSilhouette

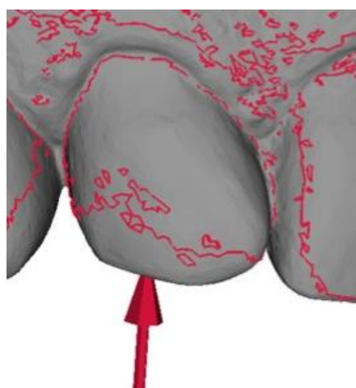

Рис. 11. Применение фильтра vtkPolyDataSilhouette

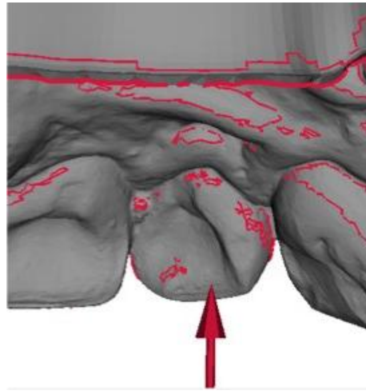

Рис. 12. Применение фильтра vtkPolyDataSilhouette

Глава 3 РЕАЛИЗАЦИЯ АЛГОРИТМОВ ВЫДЕЛЕНИЯ КОНТУРА ЗУБА

3.1 Подготовка сторонних библиотек

Для дальнейшей работы, понадобится создать окружение для написания алгоритмов. Для этого необходимо определить набор сторонних библиотек и инструменты, которые будут требоваться для реализации и тестирования алгоритмов.

Выбор языка программирования пал на современные языки объектно ориентированные языки Java и C++ а так же функциональный язык Python. У каждого языка есть свои плюсы.

Java значительно упрощает процесс разработки и порог вхождения персонала для дальнейшей доработки и совершенствования программы, но при этом сильно проигрывает в производительности. В силу того, что нам придется работать с 3D графикой, низкая производительность оказывается неприемлимой.

Python, обеспечивает более простую и интуитивно понятную работу с алгоритмизацией и математической части, но обладает еще меньшей производительностью чем Java, в силу динамической типизации, плохо интегрируем со сторонними библиотеками, а так же очень труден в поддержке.

C++ же в отличие от конкурентов обладает высокой производительностью, хорошей интегрируемостью с библиотеками, а так же средней простотой при алгоритмизации. К сожалению порог вхождения в язык высокий, а так же стоимость поддержки кода написанного на C++ оставляет желать лучшего.

Как итог, было решено остановиться на языке C++ в силу его скорости работы с графикой, наличия большого количества готовых библиотек и тонкостью настройки

Для реализации графической части алгоритмов, отрисовки моделей и построения контуров, было решено использовать открытую библиотеку VTK. VTK или Visualization Toolkit — открытая, кроссплатформенная библиотека для трёхмерного моделирования, обработки изображений и прикладной визуализации. Данная библиотека была выбрана благодаря ее хорошей интеграции с языком программирования C++, широкий набор готовых решений для создания 3D окружения и моделей, а так же благодаря поддержке библиотекой технологии smartPointer, добавленной в C++14, позволяющий значительно упростить процесс разработки.

В качестве библиотеки для дизайна и построения интерфейса, а так же работы с многопоточностью будет использоваться инструментарий Qt. Qt — кроссплатформенный фреймворк для разработки программного обеспечения на языке программирования C++. Включает в себя все основные классы, которые могут потребоваться при разработке прикладного программного обеспечения, начиная от элементов графического интерфейса и заканчивая классами для работы с сетью, базами данных и XML. Является полностью объектно-ориентированным, расширяемым и поддерживающим технику компонентного программирования. Такой фреймворк идеально подойдет для разработки графического интерфейса программы, дополнения алгоритма многопоточностью и структурирования программного кода.

Теперь, когда нам удалось определиться с основными используемыми во время разработки сторонними библиотеками, необходимо будет собрать их вместе и установить.

Поскольку было решено использовать язык программирования C++, для сборки библиотек нам потребуется дистрибутив Cmake. С его помощью будет проще всего настроить и сконфигурировать все выбранные библиотеки перед установкой.

Сначала загрузим все библиотеки и дистрибутив Cmake с официальных сайтов. Разработка будет проходить под управлением операционной системы

Windows 10 и в связи с этим проще будет воспользоваться графической версией Cmake.

Во время конфигурации библиотеки VTK, необходимо указать что мы собираемся использовать инструментарий Qt, для того чтобы Cmake сгенерировал файл с конфигурацией vtkWidget для Qt. Так же понадобится указать пути исполняемых файлов ядра и компилятора Qt как показано на рисунке 13.

Рисунок 13 – Конфигурация Qt для cmake.

После успешной генерации, необходимо скомпилировать библиотеки по инструкциям MakeFile и перенести сгенерированный файл libQVTKWidgetPlugin.dll в папку модуля Qt – Qt Designer и прописать установленные библиотеки VTK и Qt в системную переменную Path, а так же указать переменные QT_DIR и VTK_DIR чтобы компилятор qmake знал где можно найти исходные коды библиотек.

На этом сборка сторонних библиотек закончена, далее приступим к реализации алгоритмов.

3.2 Реализация алгоритма ручного выделения

Как упоминалось выше, алгоритм ручного выделения является базовым при выделении контура зуба. Необходимо понять тонкости его реализации при помощи библиотеки VTK и его математической части.

Ранее, ручное выделение выполнялось посредством отсечения лишних полигонов от модели специальными плоскостями, что добавляло большое количество затраченного времени на выделение зубов на челюсти. Новая идея заключается в том, что пользователю необходимо будет выбрать контрольные точки на модели, которые по его мнению находятся на контуре зубной коронки. Эти точки должны соединяться друг с другом автоматически, образуя максимально приближенный контур.

В первую очередь, необходимо реализовать механизм произвольной расстановки точек на модели. Сами модели в VTK описаны посредством так называемых актеров(actors), в нашем случае взятых и STL файлов содержащих модели челюсти. Не будем вдаваться в подробности загрузки модели, и обозначим рабочего актера челюсти как workActor, как показано на рисунке 14

```
vtkSmartPointer<vtkPoints> points = vtkSmartPointer<vtkPoints>::New() ;
```

Рисунок 14 – Инициализация рабочего актера челюсти.

В данном поле будет храниться модель, с которой производится работа в данный момент.

Для реализации механизма расстановки точек, воспользуемся классом vtkPolygonalSurfacePointPlacer в VTK. Он идеально подойдет для поставленных задач, так как предназначен для установки точек на полигональной поверхности 3D модели. Инициализируем его как показано на рисунке 15.

```
vtkSmartPointer<vtkPolygonalSurfacePointPlacerDebug> pointPlacer =
        vtkSmartPointer<vtkPolygonalSurfacePointPlacerDebug>::New();
pointPlacer->AddProp(workActor);
```

Рисунок 15 – Инициализация объекта для расстановки точек.

При запуске проекта, на загруженной модели челюсти теперь можно ставить точки. На рисунке 16 можно увидеть результат.

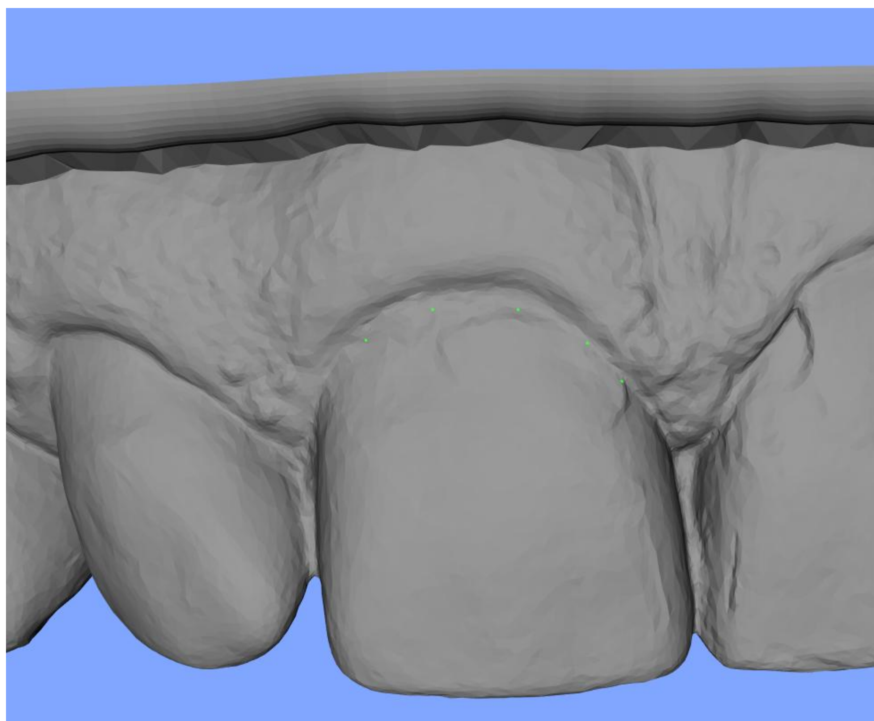

Рисунок 16 – Точки на модели челюсти

Далее, для соединения точек контуром, используем выше упомянутый vtkContourWidget. Для создания контура необходимо инициализировать 3 поля: представление(representation), виджет контура(contourWidget) и виджет точек(pointPlacer). Последний был инициализирован выше.

Инициализация vtkContourWidget показана на рисунке 17, а representation на рисунке 18. В метод SetInteractor() передан объект глобального интерактора, описывающего глобальное поведение взаимодействия с актерами, поэтому заострять на нем внимание мы не будем.

```
auto contourWidget = vtkSmartPointer<vtkContourWidget>::New();
contourWidget->Modified();
contourWidget->SetInteractor(rwi);
```

Рисунок 17 – Инициализация contourWidget

```
auto representation =
        vtkSmartPointer<vtkOrientedGlyphContourRepresentation>::New();
representation =
        vtkOrientedGlyphContourRepresentation::SafeDownCast(
                contourWidget->GetRepresentation());
representation->GetLinesProperty()->SetColor(1.0, 0.2, 0.0);
representation->GetLinesProperty()->SetLineWidth(3.0);
```

Рисунок 18 – Инициализация representation

После запуска программы, на рисунках 19-20, можно заметить что контур между точками строится автоматически с заданными параметрами, но

выглядит он совсем не так как хотелось бы. Контур проваливается под полигоны модели челюсти, и совершенно не учитывает направление поверхности при постройке маршрута.

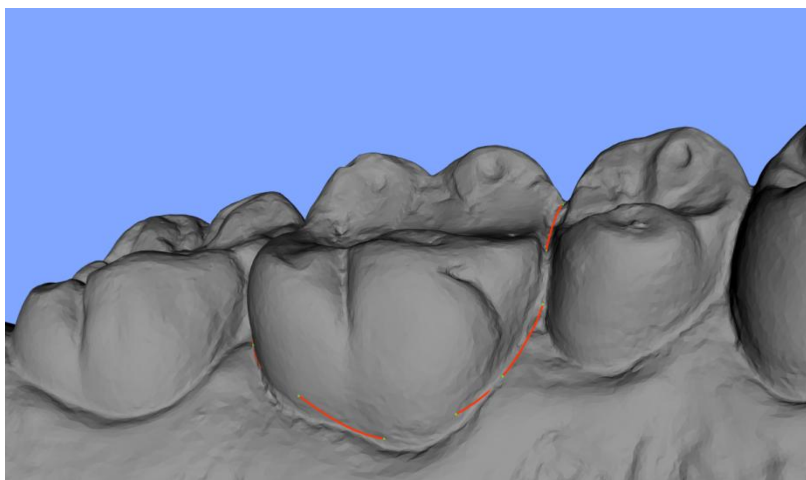

Рисунок 19 – Контур на маляре без интерполяции

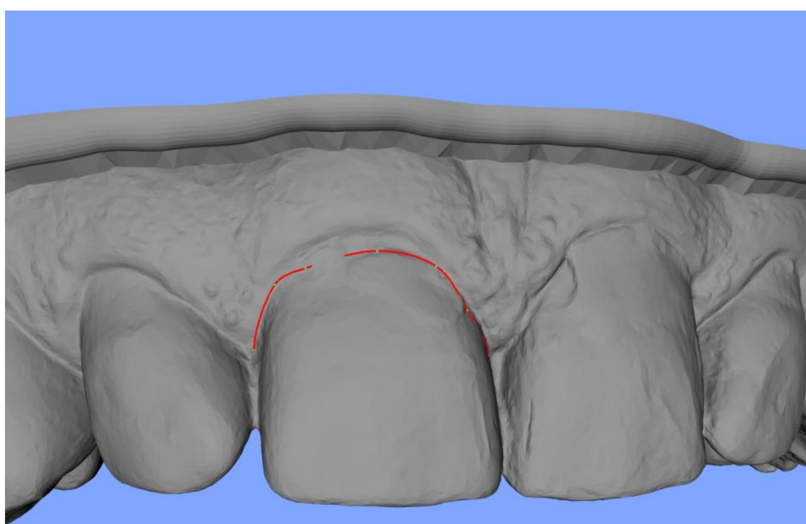

Рисунок 20 – Контур на резце без интерполяции

Дело в том, что функция контура автоматически не интерполируется и строится со стандартной аппроксимацией. Vtk предлагает явно передать представлению необходимый интерполятор. Для работы с 3D моделями в vtk предусмотрен лишь один интерполятор: vtkPolygonalSurfaceContourLineInterpolator. Он интерполирует путь от одной точки до другой при помощи алгоритма Дейкстры нахождения кратчайшего пути на графе. Графом в данном случае являются ребра полигонов, лежащие между точками. Инициализация интерполятора и передача его объекта представлению показана на рисунке 21, а результативный контур на рисунках 22-23.

```
vtkSmartPointer<vtkPolygonalSurfaceContourLineInterpolator> interpolator =
        vtkSmartPointer<vtkPolygonalSurfaceContourLineInterpolator>::New();
interpolator->GetPolys()->AddItem(workPolydata);
representation->SetLineInterpolator(interpolator);
```

Рисунок 21 – Инициализация интерполятора

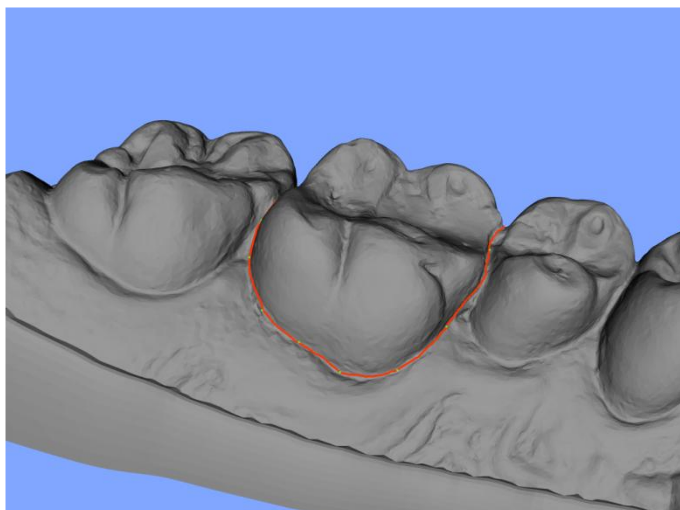

Рисунок 22 – Контур на маляре с применением интерполяции

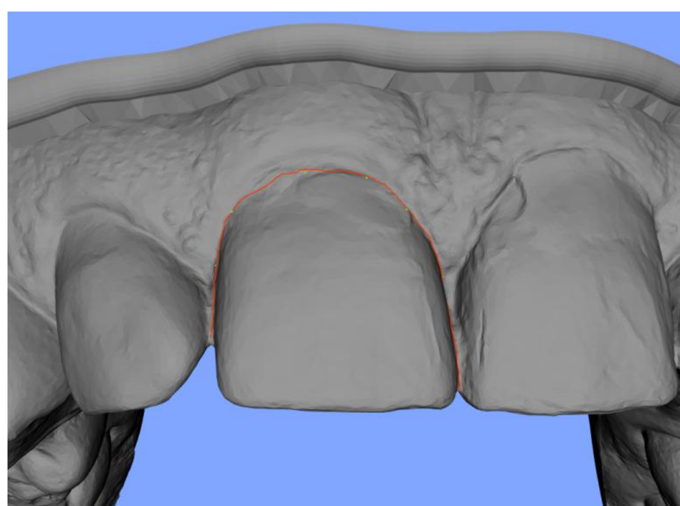

Рисунок 23 – Контур на резце с применением интерполяции

Как видно, после применения интерполяции, контур стал практически идеальным. Его путь движется ровно по полигонам заданного актера, выискивая оптимальный маршрут. Применение алгоритма Дейкстры в данной ситуации очень хорошо вписывается в рамки решение конкретной задачи выделения контура зуба. Все дело в том, что в большем количестве случаев, зуб пациента имеет четкий переход между десной и зубом, некоторое углубление относительно общей поверхности. При помещении контрольной точки в окрестность данного углубления, кратчайший путь до следующей точки, находящейся в этом углублении, будет пролегать по всему

углублению. К сожалению, расстояние между контрольными точками должно быть не меньше половины полу-дуги посадочного места зуба, иначе контур построится не правильно. Для наиболее правильного построения контура, рекомендуется располагать четыре или более точек на каждой полу-дуге посадочного места. Пример неправильной расстановки контрольных точек показан на рисунке 24.

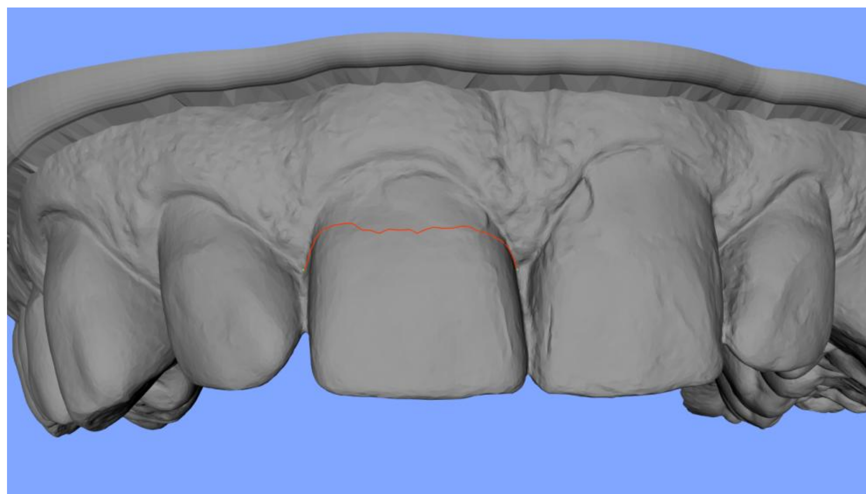

Рисунок 24 – Контур при двух точках, расположенных на полу-дуге посадочного места зуба.

В результате, данный алгоритм позволяет очень точно определять границы контура зуба, при минимальном количестве контрольных точек, с учетом оптимизации алгоритмом Дейкстры. К сожалению, такой способ выделения зуба, все равно является довольно медленным. Для выделения всех – тридцати двух зубов на челюсти пациента будет затрачено немалое количество времени и концентрации врача.

3.3 Реализация алгоритма автоматического выделения

Полный исходный код реализации алгоритма представлен в приложении А, так как реализация достаточна большая, и не является непосредственным объектом исследованием данной работы. Здесь будут рассмотрены ключевые моменты реализации.

Для создания реализации алгоритма автоматического выделения будет использоваться отличный от описанного выше метод выделения контура.

Сначала, необходимо получить данные о точках, нормалях, градиентах и завихренностях модели, так как они будут использоваться на протяжении всей реализации. Vtk по умолчанию хранит данные об актере и позволяет получить все необходимые данные о нем. Создание и настройка объектов содержащих данные о нормалях, градиенте и завихренностях изображены соответственно на рисунках 25, 26, 27.

```
vtkSmartPointer<vtkPolyDataNormals> normalsAll =
            vtkSmartPointer<vtkPolyDataNormals>::New();
normalsAll->SetInputData(workPolydata);
normalsAll->ConsistencyOn();
normalsAll->SplittingOff();
normalsAll->ComputeCellNormalsOn();
normalsAll->ComputePointNormalsOn();
normalsAll->Update();
workPolydata=normalsAll->GetOutput();
```

Рисунок 25 – Реализация объекта для хранения нормалей

```
vtkSmartPointer<vtkGradientFilter> gradientFilter =
        vtkSmartPointer<vtkGradientFilter>::New();
gradientFilter->SetInputData(workPolydata);
gradientFilter->
        SetInputScalars(vtkDataObject::FIELD_ASSOCIATION_POINTS, fieldName);
gradientFilter->SetComputeVorticity(1);
gradientFilter->SetComputeDivergence(1);
gradientFilter->SetResultArrayName("gradients");
gradientFilter->Update();

vtkSmartPointer<vtkDoubleArray> gradients =
        vtkDoubleArray::SafeDownCast(
            vtkDataSet::SafeDownCast(
                gradientFilter->GetOutput())->GetPointData()->GetArray("gradients")
            ) ;
gradients->SetNumberOfTuples(gradientFilter->GetOutput()->GetNumberOfPoints());
gradients->SetName("gradients");
```

Рисунок 26 – Реализация объекта для хранения градиентов

```
vtkDoubleArray* vorticityCellArray = vtkArrayDownCast<vtkDoubleArray>(
            vtkDataSet::SafeDownCast(
                gradientFilter->GetOutput())->GetPointData()->GetArray("Vorticity"));
```

Рисунок 27 - Реализация объекта для хранения завихренностей

Так же, понадобится дерево точек модели, которое так же можно получить при помощи vtk, как показано на рисунке 28.

```
vtkSmartPointer<vtkKdTreePointLocator> pointAllTree =
        vtkSmartPointer<vtkKdTreePointLocator>::New();
pointAllTree->SetDataSet(workPolydata);
pointAllTree->BuildLocator();
```

Рисунок 28 - Реализация объекта для хранения точек модели

Следует так же рассмотреть фрагмент реализации, представленный на рисунке 29. На нем показано, как при помощи такого инструмента как vtkSilhouette, можно ограничить количество рассматриваемых точек.

```
vtkSmartPointer<vtkPoints> outPts = vtkSmartPointer<vtkPoints>::New();
iDStart = outPts->InsertNextPoint(startPoint);

silhouette->GetOutput()->GetLines()->InitTraversal();
vtkSmartPointer<vtkIdList> idList = vtkSmartPointer<vtkIdList>::New();
while(silhouette->GetOutput()->GetLines()->GetNextCell(idList))
{
    for(vtkIdType pointId = 0; pointId < idList->GetNumberOfIds(); pointId++)
    {
        outPts->InsertNextPoint( silhouette->GetOutput()->GetPoint (idList->GetId(pointId)));
    }
}
```

Рисунок 29 – Фильтрация точек при помощи vtkSilhouette

Начало процесса итерации между точками будет выглядеть так, как показано на рисунке 30. Как видно на рисунке, обозначим так же координаты контрольных точек, выбранных пользователем.

```
std::map<int, vtkSmartPointer<vtkPoints>>::iterator it =
        autoContourPoints.begin();
while (it != autoContourPoints.end()) {

    Contour *contour = new Contour();

    double startPoint [3], endPoint[3];
    (*it).second->GetPoint(0, startPoint);
    (*it).second->GetPoint(1, endPoint);
    double c [3] = {(startPoint[0]+endPoint[0])/2,
                    (startPoint[1]+endPoint[1])/2,
                    (startPoint[2]+endPoint[2])/2};
```

Рисунок 30 – Реализация начала итерации алгоритма

На каждой итерации, стартовая точка будет меняться текущей, поэтому необходимо инициализировать список, в котором будут храниться точки контура на текущей итерации, обозначить стартовую точку текущей, и определить вектор от текущей точки, до ранее определенной конечной точки, как показано на рисунке 31.

```
vtkSmartPointer<vtkPoints> contourPs =
        vtkSmartPointer<vtkPoints>::New() ;
vtkSmartPointer<vtkIdList> closestPoints =
        vtkSmartPointer<vtkIdList>::New();

contourPs->InsertNextPoint(startPoint);
double nextPoint [3] ={ startPoint[0], startPoint[1], startPoint[2]};
double start2end_vector [3] = {startPoint[0]-endPoint[0],
                               startPoint[1]-endPoint[1],
                               startPoint[2]-endPoint[2]};
```

Рисунок 31 – Реализация инициализации следующей точки и итеративного вектора

Реализация алгоритма построения контура была реализована ранее и использует в себе алгоритм Дейкстры. Для алгоритма попарных сравнений же, было решено использовать модификацию алгоритма Саати и девятибалльную систему оценок коэффициентов.

Для реализации такого алгоритма было принято решение о создании самостоятельного класса ImportanceCoefficient. Его задача заключается в хранении информации о коэффициентах на текущей итерации, работе с ней и пересчете коэффициентов на разных этапах построения контура.

Сначала, зададим размерность матрицы и обозначим столбцы при помощи перечислений, а так же обозначим поле для хранения итоговых коэффициентов coefficients как показано на рисунке 32.

```
class ImportanceCoefficient
{
    const static int n_criteria = 4;

        enum {
            DISTANCE = 0,
            Y_VALUE = 1,
            VORTICITY = 2,
            GRADIENT = 3,
        };

        float coefficients [n_criteria];
```

Рисунок 32 – Объявление полей класса ImportanceCoefficient

Далее, реализуем метод calculate_coefficients(), задача которого состоит в пересчете коэффициентов с использованием алгоритма Саати. Реализация метода представлена на рисунке 33.

```
void ImportanceCoefficient::calculate_coefficients()
{
    float pair_comp [n_criteria][n_criteria];

    if (this->y_value)
    pair_comp[DISTANCE][Y_VALUE] = this->y_value;
    else
        pair_comp[DISTANCE][Y_VALUE] = 1.0 / (this->y_value_distance*1.0);

    pair_comp[DISTANCE][VORTICITY] = this->vorticity;
    pair_comp[DISTANCE][GRADIENT] = this->gradient;

    for (int i = 1; i<(n_criteria-1); i++) {
        for (int j = i + 1; j < n_criteria; j++) {
            pair_comp[i][j] = pair_comp[i - 1][j] / pair_comp[i - 1][i];
        }
    }

    pair_comp[0][0] = 1.0;

    for (int i = 1; i<n_criteria; i++) {
        pair_comp[i][i] = 1.0;
        for (int j = 0; j < i; j++) {
            pair_comp[i][j] = 1.0d / pair_comp[j][i];
        }
    }

    for (int j = 0; j<n_criteria; j++) {
        float sum = 0.0;
        for (auto &i : pair_comp) {
            sum+= i[j];
        }
        coefficients[j] = 1.0d / sum;
    }
}
```

Рисунок 33 – Реализация алгоритма Саати

Напоследок, обозначим элементы шкалы Саати при помощи перечислений, зададим приоритет коэффициентов по умолчанию и реализуем конструктор, чтобы при инициализации объекта, автоматически производился пересчет коэффициентов заданных изначально. Реализация представлена на рисунке 34.

```
typedef enum {
    NO_ADVANTAGE         = 1 ,
    WEAK_ADVANTAGE       = 3 ,
    MODERATE_ADVANTAGE   = 5 ,
    STRONG_ADVANTAGE     = 7 ,
    EXTREME_ADVANTAGE    = 9 ,
    NOT_SPECIFIED = 0
} saaty_scale_t;


saaty_scale_t vorticity = NO_ADVANTAGE;
saaty_scale_t gradient = NO_ADVANTAGE;
saaty_scale_t y_value = NOT_SPECIFIED;

saaty_scale_t y_value_distance = STRONG_ADVANTAGE;
saaty_scale_t y_value_gradient = MODERATE_ADVANTAGE;
saaty_scale_t y_value_vorticity = MODERATE_ADVANTAGE;

ImportanceCoefficient() {
    calculate_coefficients();
}
```

Рисунок 34 – Инициализация приоритета коэффициентов по умолчанию

Процесс нахождения точек контура делится на 2 части, спуск от стартовой граничной точки, и быстрое приближение к конечной. На первом этапе будет использоваться матрица попарного сравнения $A\ C_1$, из формулы (4), так как во время спуска наиболее важен критерий c_1 - значение координаты y, а на втором соответственно матрица $A\ C_2$ из формулы (5), так как при приближении к конечной точке критерии расстояния от стартовой граничной точки до конечной и градиента функции в данной точке набирают большую ценность. Меняться значения матриц будут при превышении угла значения в 55 градусов.

Реализация алгоритма, меняющая набор коэффициентов при переходе к приближению к конечной точке показана на рисунке 35.

```
double angle = get_angle(startPoint, c, nextPoint);
if (angle > 55){
    coefficient->set_y_value(ImportanceCoefficient::STRONG_ADVANTAGE);
    coefficient->set_vorticity(ImportanceCoefficient::MODERATE_ADVANTAGE);
    coefficient->set_gradient(ImportanceCoefficient::STRONG_ADVANTAGE);
    coefficient->calculate_coefficients();
}
```

Рисунок 35 – Реализация пересчета коэффициентов.

После того, как выбраны коэффициенты, используемые для поиска следующей точки, следует запустить алгоритм определения следующей точки контура, реализация которого представлена в приложении В.

После определения одной половины контура, полученные точки контура отражаются на противоположную сторону, в целях сокращения времени работы алгоритма и избежать повторения. На рисунке 36 показана непосредственная реализация экранирования точек.

```
for(vtkIdType i = contourPs_rotate; i > (contourPs_rotate-4); i--) {
    double current_point[3] ;
    contourPs->GetPoint(i, current_point);
    double current_vector [3] = {current_point[0] - endPoint[0],
                              ,current_point[1] - endPoint[1],
                              current_point[2]- endPoint[2]};
    double result_vector[3];
    vtkQuaternion<double> q = vtkQuaternion<double>();
    q.SetRotationAngleAndAxis(angle, vector[0],vector[1],vector[2]);
    q.Normalize();
    double norm_q [4];
    q.Get(norm_q);
    RotateVectorByNormalizedQuaternion(current_vector,
                                       norm_q ,
                                       result_vector);
    double mirror_point [3] = {endPoint[0]+result_vector[0],
                            endPoint[1]+result_vector[1],
                            endPoint[2]+result_vector[2]};

    vtkSmartPointer<vtkKdTreePointLocator> pointTree =
            vtkSmartPointer<vtkKdTreePointLocator>::New();
    pointTree->SetDataSet(contourPD);
    pointTree->BuildLocator();
    vtkIdType point_ind = pointTree->FindClosestPoint(mirror_point);
    contourPD->GetPoint(point_ind, nextPoint);
    contourPs->InsertNextPoint(nextPoint);
}
```

Рисунок 36 – Реализация экранирования полученных точек

Третья и последняя часть алгоритма же, аналогична первой и необходима чтобы достроить концы полученного контура до конечной точки и замкнуть контур.

Глава 4 АНАЛИЗ РАБОТЫ АЛГОРИТМОВ

Тестирование алгоритма будет проводиться на системе со следующими характеристиками:

- Процессор: AMD FX-8300 Eight-Core Processor (8 CPUs), ~3.3GHz
- Память: 8192MB RAM
- Операционная система: Windows 10 Pro 64-bit (10.0, Build 17134) (17134.rs4_release.180410-1804)
- Версия DirectX: DirectX 12
- Графическая карта: AMD Radeon (TM) R9 380 Series

Стоит обратить внимание, что очень большое влияние на производительность алгоритма оказывает графический процессор, из за скорости отрисовки модели и контура, после вычисления координат контрольных точек.

Так же стоит отметить, что количество ядер процессора практически не влияет на работу алгоритма, так как при разработке алгоритм не распараллеливался. Однако нужно учитывать, что методы библиотеки vtk, могут использовать параллельные алгоритмы во время получения фильтром набора точек, или при отрисовке контура.

Версия DirectX так же влияет на скорость работы алгоритма по причине более оптимизированных способов отрисовки 3D графики. На других версиях DirectX, скорость работы алгоритма будет существенно отличаться

Для определения эффективности разработанных алгоритмов, необходимо провести анализ их работы. Ручной алгоритм тестироваться не будет, так как оценка его работы является субъективной.

В свою очередь, алгоритм автоматического выделения будет анализироваться по нескольким параметрам:

- Скорость работы,

- Точность обнаружения контура.

Все тесты будут совершаться на машине с одной конфигурацией для точности производимого эксперимента. Тестирование каждого параметра будет производиться отдельно для резцов и маляров, чтобы выявить эффективность алгоритма для конкретного вида зубов.

При анализе скорости работы алгоритма, будет замеряться общая скорость работы алгоритма. Замеры времени производятся при помощи входящего в библиотеку ctime метода clock(). Представленное время указано в милисекундах для дополнительной точности.

Анализ скорости работы был произведен на девяти моделях челюстей. Далее, вычислялось среднее время работы алгоритма для зубов одного из типов: Резец, клык, премоляр, моляр. К резцам относятся первые два зуба челюсти, к клыкам – третий, к премолярам – четвертый и пятый, к молярам – шестой, седьмой и восьмой (при наличии). Диаграмма 1, представляет скорость работы алгоритма при выделении резцов. На диаграмме четко выражено, что время работы алгоритма для резцов верхней челюсти сильно больше чем для резцов нижней челюсти. Это обусловлено тем, что алгоритм вынужден выбирать большее количество точек, а соответственно и сортировать больше точек при больших размерах модели зуба. На подсчет коэффициентов и выбор каждой точки тратится большое количество времени, соответственно при большей площади поверхности зуба и большем радиусе посадочного места для него, время работы алгоритма увеличивается на фиксированное количество времени, которое зависит от условий тестирования.

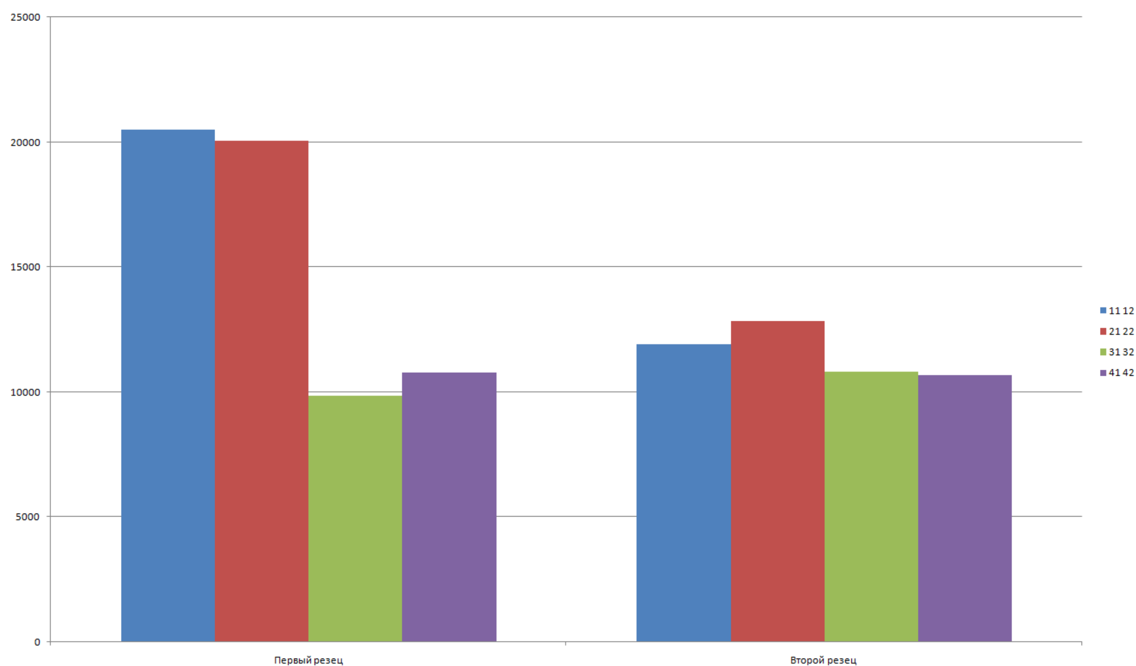

Диаграмма 1 – Среднее время работы алгоритма для резцов

На диаграмме 2, отражено среднее время работы алгоритма при выделении клыков.

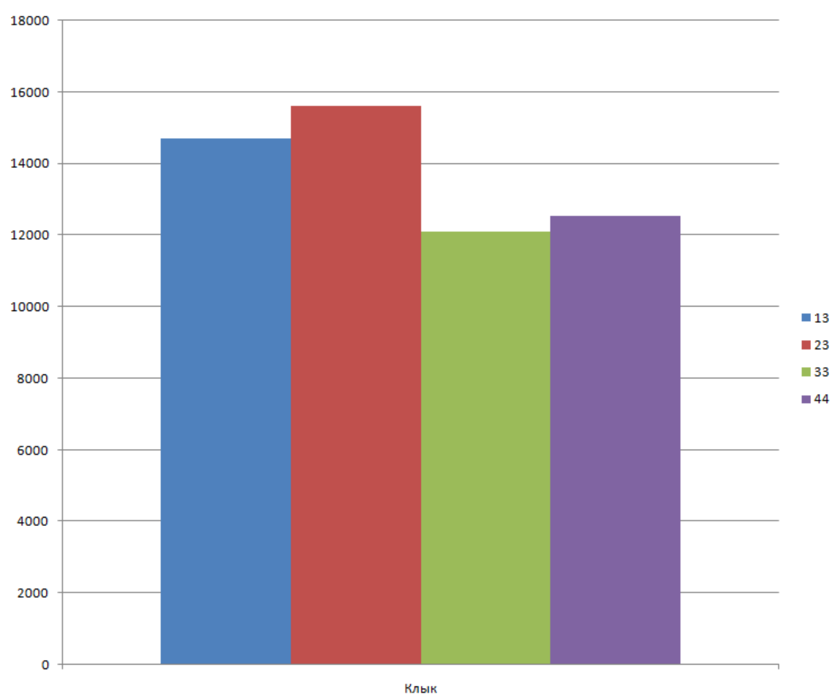

Диаграмма 2 - Среднее время работы алгоритма для клыков

На диаграмме видно, что как и с резцами, среднее время работы алгоритма на клыках верхней челюсти больше, однако уже не так значительно. Связанно это так же с большими размерами клыков на верхней челюсти.

Для премоляров же, результаты более гладкие. Время работы алгоритма немного больше на втором премоляре, чем на первом. Результаты тестирований отражены на диаграмме 3

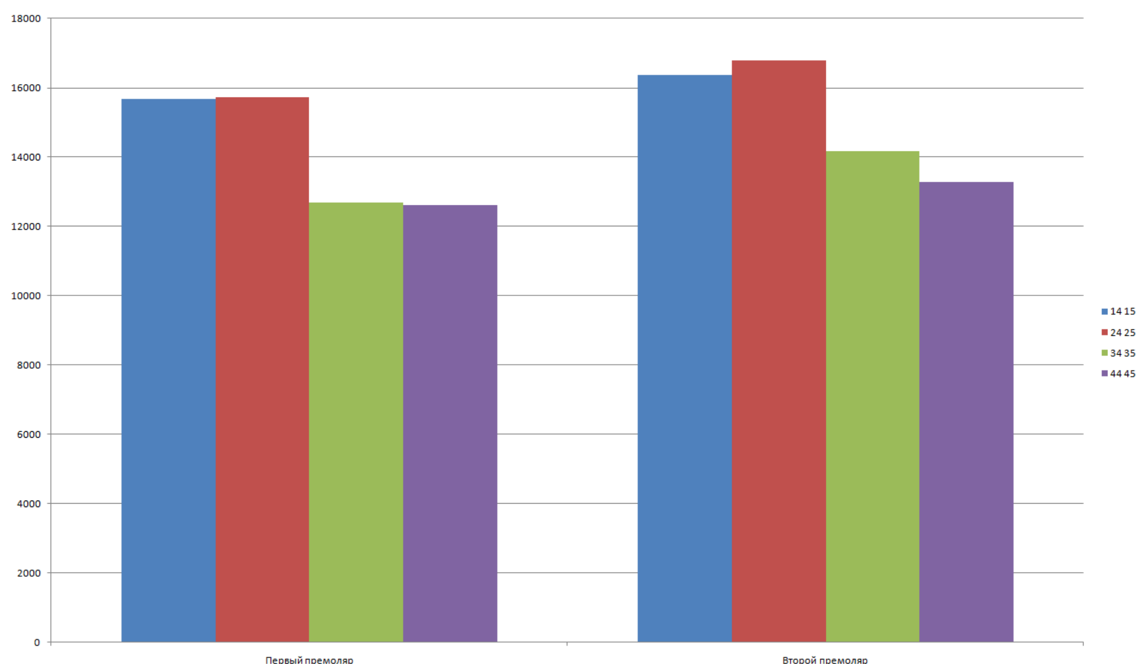

Диаграмма 3 - Среднее время работы алгоритма для премоляров

Результаты, отраженные на диаграмме 4, требуют пояснения. К молярам относятся также и восьмой зуб – зуб мудрости. Этот зуб вызывает множество проблем при небольших размерах челюсти, в следствии чего, в большинстве случаев удаляется уже во время прорези. У многих людей он так же остается ретинированным, или полуретинированным. На большей части исследуемых моделей верхних челюстей, зуб мудрости отсутствовал как минимум с одной стороны, а на нижней челюсти зубов мудрости не было вообще, из за этого диаграмма 4 представляет не полные данные о времени работы алгоритма во время обнаружения контура зуба мудрости.

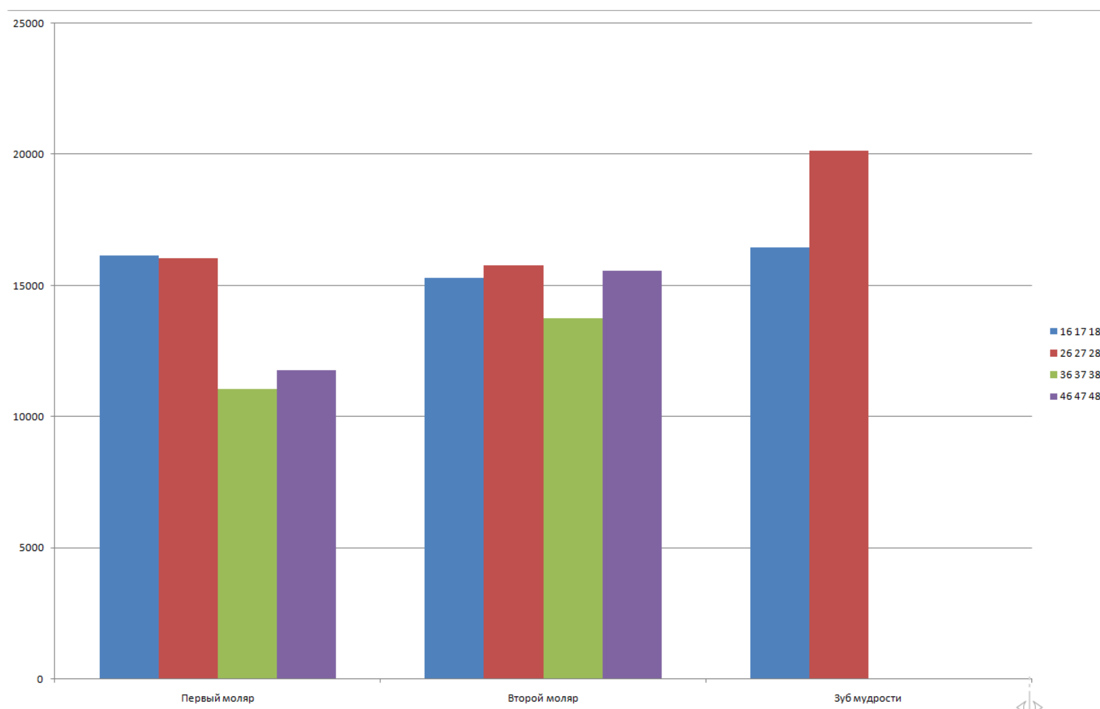

Диаграмма 4 - Среднее время работы алгоритма для моляров

На диаграмме видно, что среднее время работы для моляров на верхней челюсти такое же как и на нижней, учитывая погрешность. Время работы на зубах мудрости сильно разниться, но эти данные нельзя считать достоверными из за недостатка тестов.

Точность обнаружения контура будет определяться на основе процента приемлемо обнаруженных контуров. Приемлемым контуром будет считаться такой контур, в котором будет не более пяти точек, надлежащих к исправлению. Как и во время тестирования скорости работы алгоритма, тестирование точности будет делиться на группы резцов, клыков, премоляров и моляров. Тестирование так же будет проводиться на девяти различных моделях челюстей. На рисунках 37 и 38 показаны примеры приемлемых и неприемлемых контуров соответственно.

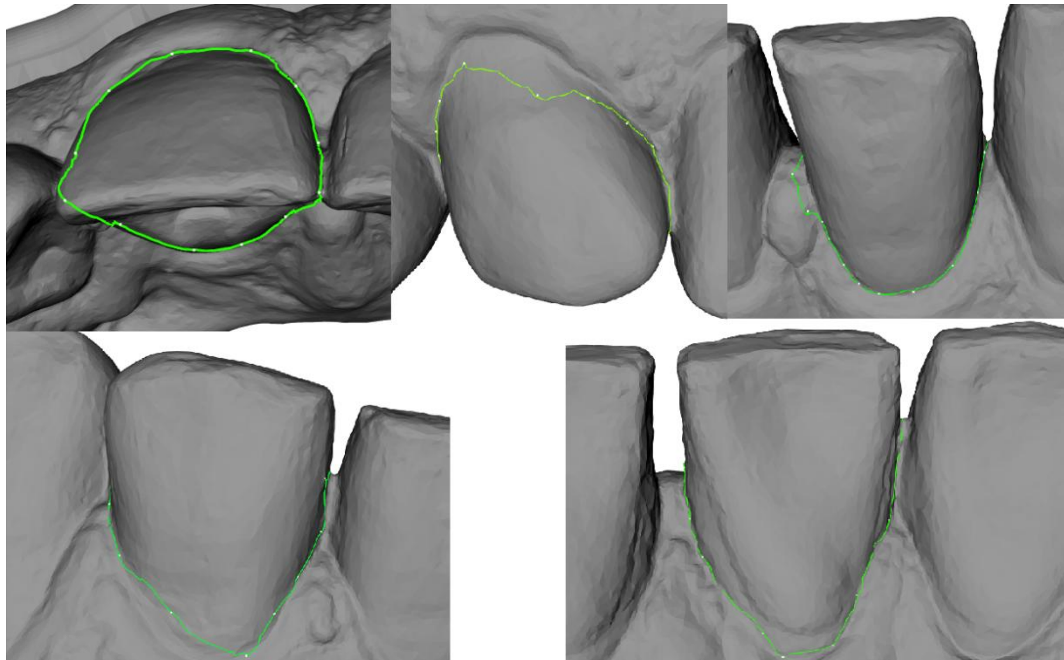

Рисунок 37 – Примеры приемлемых контуров

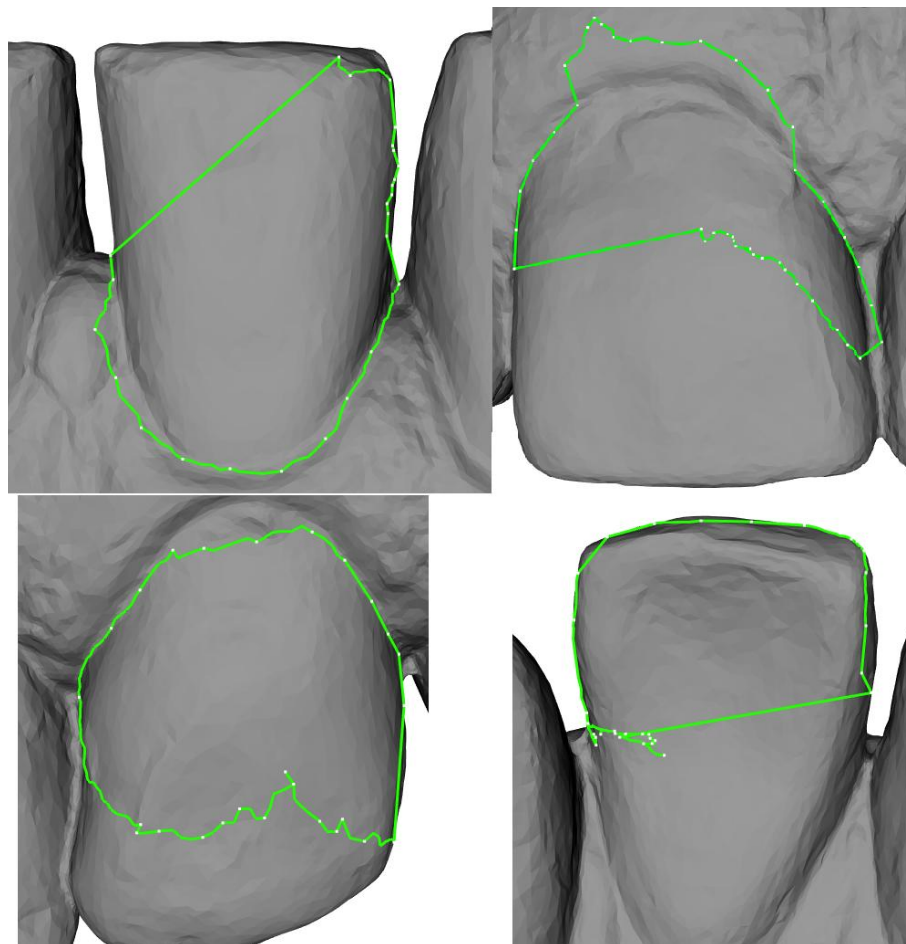

Рисунок 38 - Примеры неприемлемых контуров

На диаграммах 5 и 6 показаны результаты тестирования: Процент приемлемых контуров для верхней и нижней челюсти соответственно.

По диаграмме 5 четко видно, что лучше всего алгоритм работает с молярами и премолярами, не считая зубов мудрости, так как они часто располагаются на краю модели, что стирает контур одной из границ.

Так же, эксперимент выявил что клыки плохо сегментируются алгоритмом из за нечеткой границы между десной и зубной коронкой.

На резцах же, алгоритм показывает средний процент приемлемости контура, что делает его ненадежным при выделении резцов на практике.

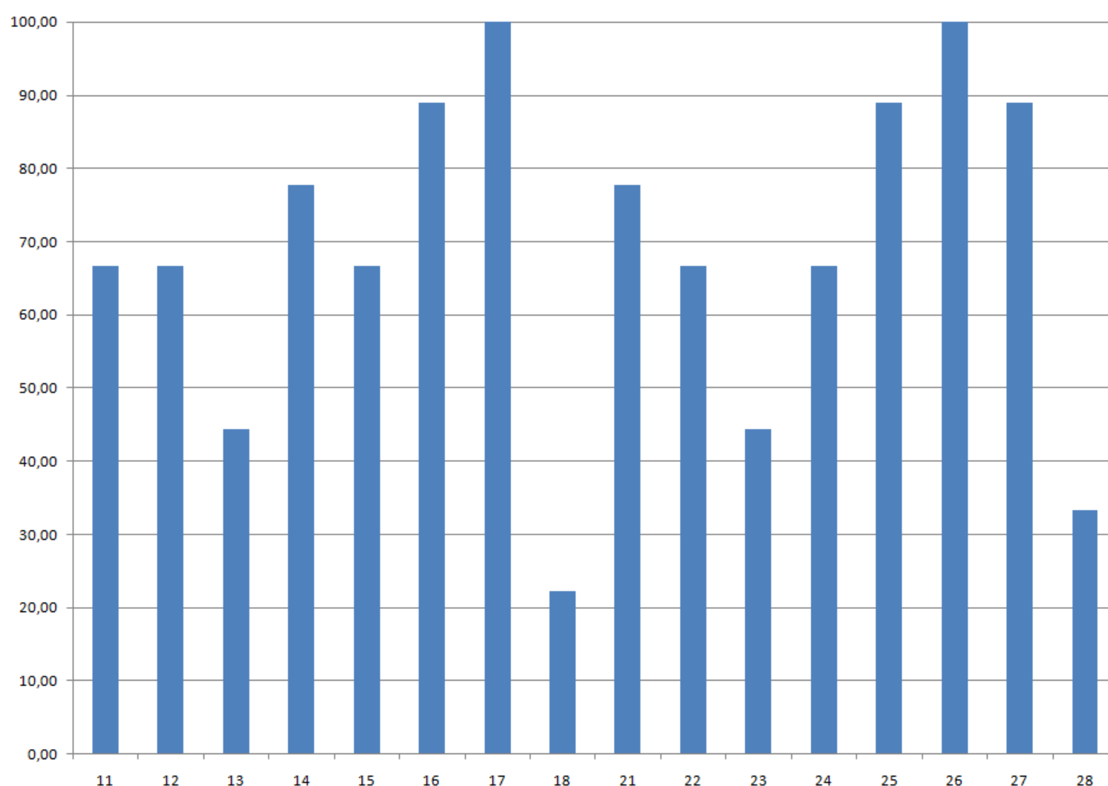

Диаграмма 5 – Процент обнаружений приемлемого контура для зубов верхней челюсти

Результаты, приведенные на диаграмме 6, показывают, что как и на верхней части челюсти, обнаружение контура моляров на нижней части показывает большой процент приемлемости.

Процент успешного обнаружения клыков так же оставляет желать лучшего.

Большое различие с верхней частью челюсти заключается в обнаружении контуров резцов нижней части. Процент приемлемости оказался очень малым. Обусловлено это нечеткой границей между десной и зубной коронкой, а так же, пологим спуском на внутренней стороне модели. Автоматическое обнаружение контура резцов на нижней челюсти требует подхода, отличающегося от подхода, исследуемого в данной работе.

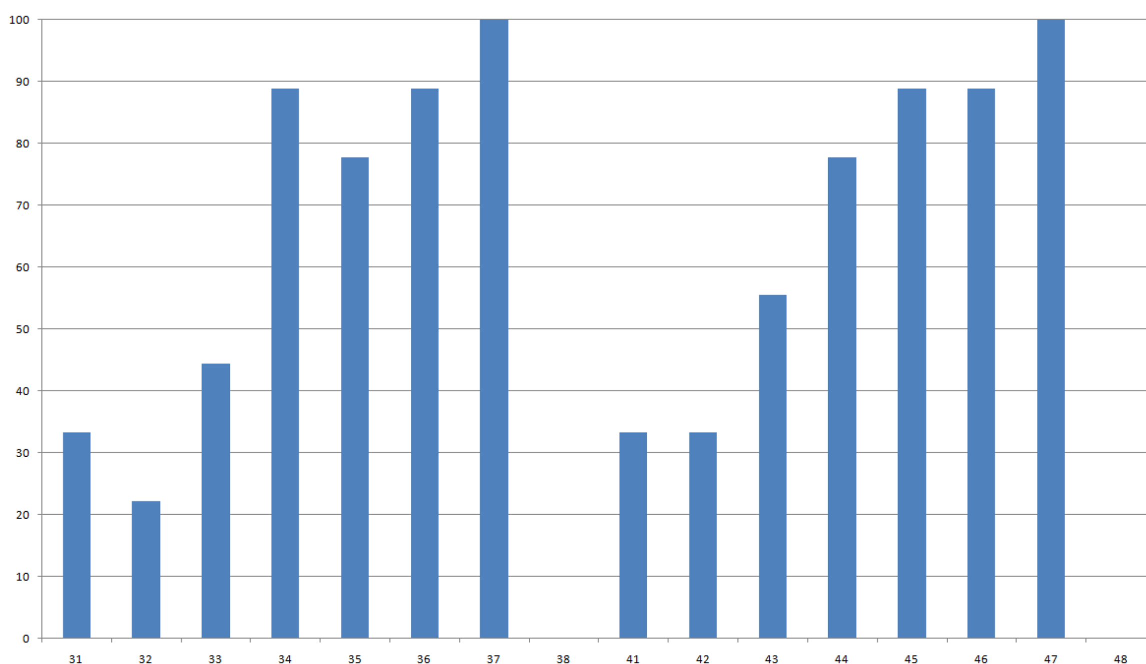

Диаграмма 6 - Процент обнаружений приемлемого контура для зубов нижней челюсти

В целом, алгоритм автоматического обнаружения контура зуба показал впечатляющие результаты. При проведении дополнительных экспериментов, можно добиться практически стопроцентной вероятности корректного выделения приемлемого контура моляров и премоляров, а так же значительно повысить эффективность нахождения алгоритмом контура резцов и клыков.

ЗАКЛЮЧЕНИЕ

В ходе данной работы, были разработаны два алгоритма обнаружения контура зубной коронки на 3D модели челюсти: Ручной и автоматический. Данные алгоритмы и их реализации не являются модернизацией или оптимизацией уже существующих алгоритмов: они являются новыми в сфере медицины.

В результате тестирования и анализа алгоритмов, были получены следующие результаты:

- Точность обнаружения моляров и премоляров при помощи алгоритма автоматического определения контура зубной коронки очень высока, и может достичь ста процентов при проведении дополнительных экспериментов для подбора коэффициентов;
- Точность обнаружения резцов сильно отличается для верхней и нижней челюсти. На верхней челюсти, алгоритм показывает удовлетворительные результаты, но нуждается в доработке и более тонком подборе коэффициентов для критериев оценки. На нижней челюсти, алгоритм совершенно не справляется, и к этим зубам требуется совершенно иной подход из за неявной границы между зубной коронкой и десной, на поиске которой и основан алгоритм представленный в данной работе;
- Точность обнаружения клыков мала, но в случае проведения дополнительного исследования может увеличиться, не меняя разработанный в данной работе алгоритм;
- Скорость работы алгоритма зависит от количества точек, выбранных алгоритмом в качестве контрольных, что означает возможность будущей оптимизации.
- В целом, время работы алгоритма на разных зубах достаточно высокое, однако при увеличении мощности системы, на которой он запускается, можно добиться много более низких значений времени работы.

Подводя итоги, можно сказать, что разработанные алгоритмы намного более удобные и легкие в использовании при определении контура зубной коронки, чем использующиеся на сегодняшний день.

Ручной алгоритм очень точен, но требует большого количества действий пользователя, что требует большого количества его концентрации.

Автоматический же алгоритм существенно облегчает процесс выделения контура, практически исключая из него человеческий фактор. При определенных условиях, и использовании мощного оборудования, алгоритм показывает скорость, сравнимую со скоростью выделения зуба вручную, а в определенных условиях и выигрывает его. К сожалению, без дополнительных экспериментов и доработки алгоритма, его точность страдает, и на данный момент может конкурировать с ручным алгоритмом только при выделении моляров и премоляров.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Letieri, A.S. Age and aesthetics perception related to different types of orthodontic devices. Is there a relationship? / A.S. Letieri, C.C. Avelar Fernandez, S.O. Aguiar Sales Lima // Journal of the World Federation of Orthodontists. – 2018. – V. 7, N 1. – P. 29-33.

2. 2. Barone, S. CT segmentation of dental shapes by anatomy driven reformation imaging and Bspline modelling / S. Barone, A. Paoli, A.V. Razionale // International Journal for Numerical Methods in Biomedical Engineering. – 2016. – V. 32, N 6.

3. Camardella, L.T. Accuracy and reproducibility of measurements on plaster models and digital models created using an intraoral scanner / L.T. Camardella, H. Breuning, O. de Vasconcellos Vilella // Journal of Orofacial Orthopedics. – 2017. – V. 78, N 3. – P. 211-220.

4. Lemos, L.S. Reliability of measurements made on scanned cast models using the 3Shape R700 scanner / L.S. Lemos, I.M. Rebello, C.J. Vogel, M.C. Barbosa // Dentomaxillofacial Radiology. – 2015. – V. 44, N 6.

5. Hackel, T. Contour Detection in Unstructured 3D Point Clouds / T. Hackel, J. D.Wegner, K. Schindler // 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*)*, Las Vegas, NV. – 2016. – V. 1. – P. 1610-1618.

6. Sinthanayothin, C. Orthodontics treatment simulation by teeth segmentation and setup / C. Sinthanayothin, W. Tharanont // 2008 5th International Conference on Electrical Engineering / Electronics, Computer, Telecommunications and Information Technology, Krabi. – 2008. – V. 1. – P. 81-84.

7. Wongwaen, N, Computerized algorithm for 3D teeth segmentation / N. Wongwaen, C. Sinthanayothin // 2010 International Conference on Electronics and Information Engineering, Kyoto. – 2010. – V. 1. – P. 277.

8. Mairaj Danish, Wolthusen Stephen D., Busch Christoph. Teeth Segmentation and Feature Extraction for Odontological Biometrics // Proceedings of the 2010 Sixth International Conference on Intelligent Information Hiding and Multimedia Signal Processing, Darmstadt. – 2010. – V. 1. – P. 323-328.

9. Bellman, R. Decision Making in a Fuzzy Environment / R. Bellman, L.A. Zadeh // Management Sciences. – 1970. – V. 17. – P. 141-164.

10. Rotshtein A. Modication of Saaty method for the construction of fuzzy set membership functions // Fuzzy Logic and Its Applications: International Conference, Zichron, Israel. – 1997. – V. 1. – P. 125-130.

11. Saaty, T.L. How to make a decision: The analytic hierarchy process // European Journal of Operational Research. – 1990. – V. 48, N 1. – P. 9-26.

ПРИЛОЖЕНИЕ А

Исходный код, алгоритма автоматического обнаружения контура зубной коронки на 3D модели челюсти

```
    if (autoContourPoints.empty())
        return;
    std::cout<<"autoBounds ... Start"<<std::endl;

    vtkSmartPointer<vtkPolyDataNormals> normalsAll =
                vtkSmartPointer<vtkPolyDataNormals>::New();
    normalsAll->SetInputData(workPolydata);
    normalsAll->ConsistencyOn();
    normalsAll->SplittingOff();
    normalsAll->ComputeCellNormalsOn();
    normalsAll->ComputePointNormalsOn();
    normalsAll->Update();
    workPolydata=normalsAll->GetOutput();

    const char fieldName[] = "LinearField";
    int offset = 1;
    const int numberOfComponents = 3;
    CreatePointData(workPolydata, numberOfComponents, offset, fieldName);

    vtkSmartPointer<vtkGradientFilter> gradientFilter =
            vtkSmartPointer<vtkGradientFilter>::New();
    gradientFilter->SetInputData(workPolydata);
    gradientFilter->
            SetInputScalars(vtkDataObject::FIELD_ASSOCIATION_POINTS,
fieldName);
    gradientFilter->SetComputeVorticity(1);
    gradientFilter->SetComputeDivergence(1);
    gradientFilter->SetResultArrayName("gradients");
    gradientFilter->Update();

    vtkSmartPointer<vtkDoubleArray> gradients =
            vtkDoubleArray::SafeDownCast(
                vtkDataSet::SafeDownCast(
                    gradientFilter->GetOutput())->GetPointData()-
>GetArray("gradients")
                ) ;
    gradients->SetNumberOfTuples(gradientFilter->GetOutput()-
>GetNumberOfPoints());
    gradients->SetName("gradients");

    vtkSmartPointer<vtkKdTreePointLocator> pointAllTree =
            vtkSmartPointer<vtkKdTreePointLocator>::New();
    pointAllTree->SetDataSet(workPolydata);
    pointAllTree->BuildLocator();

    std::map<int, vtkSmartPointer<vtkPoints>>::iterator it =
            autoContourPoints.begin();
    while (it != autoContourPoints.end()) {

        Contour *contour = new Contour();

        double startPoint [3], endPoint[3];
        (*it).second->GetPoint(0, startPoint);
        (*it).second->GetPoint(1, endPoint);
```

```
        double c [3] = {(startPoint[0]+endPoint[0])/2,
                        (startPoint[1]+endPoint[1])/2,
                        (startPoint[2]+endPoint[2])/2};

        //Find start and end points on mesh

        vtkIdType iDStart = pointAllTree->FindClosestPoint(startPoint);
        vtkIdType iDEnd = pointAllTree->FindClosestPoint(endPoint);

        vtkSmartPointer<vtkDijkstraGraphGeodesicPath>
DijkstraGraphGeodesicPath =
                vtkSmartPointer<vtkDijkstraGraphGeodesicPath>::New();
        DijkstraGraphGeodesicPath->SetInputData( workPolydata );
        DijkstraGraphGeodesicPath->SetStartVertex( iDStart );
        DijkstraGraphGeodesicPath->SetEndVertex( iDEnd );
        DijkstraGraphGeodesicPath->Update();

        vtkPolyData *centerLinePD = DijkstraGraphGeodesicPath->GetOutput();

        double *center_point= centerLinePD->GetCenter();

        // Find contour lines
        //std::cout<<"autoBounds ... silhouette"<<std::endl;
        vtkSmartPointer<vtkPolyDataSilhouette> silhouette =
                vtkSmartPointer<vtkPolyDataSilhouette>::New();
        silhouette->SetInputData(workPolydata);
        vtkSmartPointer<vtkCamera> camera =
vtkSmartPointer<vtkCamera>::New();

        double cp[3] = {center_point[0], center_point[1]-(10*normalTooth[1]),
center_point[2]};
        double fp[3] = {center_point[0], center_point[1]-(3*normalTooth[1]),
center_point[2]};

        camera->SetPosition(cp);
        camera->SetFocalPoint(fp);

        silhouette->SetCamera(camera);
        silhouette->Update();

        vtkSmartPointer<vtkPoints> outPts =
vtkSmartPointer<vtkPoints>::New();
        iDStart = outPts->InsertNextPoint(startPoint);

        silhouette->GetOutput()->GetLines()->InitTraversal();
        vtkSmartPointer<vtkIdList> idList =
vtkSmartPointer<vtkIdList>::New();
        while(silhouette->GetOutput()->GetLines()->GetNextCell(idList))
        {
            for(vtkIdType pointId = 0; pointId < idList->GetNumberOfIds();
pointId++)
            {
                outPts->InsertNextPoint( silhouette->GetOutput()->GetPoint
(idList->GetId(pointId)));
            }
        }

        iDEnd = outPts->InsertNextPoint(endPoint);


        vtkSmartPointer<vtkPolyData> contourPD =
                vtkSmartPointer<vtkPolyData>::New();
        contourPD->SetPoints(outPts);
```

```
        std::vector<double> vorticityCurrent ;
        vorticityCurrent.resize(3);
                vorticityCellArray->GetTuple(iDStart, &vorticityCurrent[0]);
        std::vector<double> gradientCurrent;
        gradientCurrent.resize(9);
                gradients->GetTuple(iDStart, &gradientCurrent[0]);


    vtkSmartPointer<vtkPoints> contourPs =
            vtkSmartPointer<vtkPoints>::New() ;
    vtkSmartPointer<vtkIdList> closestPoints =
            vtkSmartPointer<vtkIdList>::New();

    contourPs->InsertNextPoint(startPoint);
    double nextPoint [3] ={ startPoint[0], startPoint[1], startPoint[2]};
    double start2end_vector [3] = {startPoint[0]-endPoint[0],
                                   startPoint[1]-endPoint[1],
                                   startPoint[2]-endPoint[2]};

        cout<<"first part start"<<endl;

        double distance_stop = vtkMath::Distance2BetweenPoints(endPoint,
nextPoint);
        double prev_point [3] ={ endPoint[0], endPoint[1], endPoint[2]};
        int step = 0;
        ImportanceCoefficient * coefficient = new ImportanceCoefficient();

        bool f_coef = 1;
        while (distance_stop > 6.0 && step < 40){
            double angle = get_angle(startPoint, c, nextPoint);
            if (f_coef && angle > 55   ){
                f_coef = 0;
                coefficient-
>set_y_value(ImportanceCoefficient::STRONG_ADVANTAGE);
                coefficient-
>set_vorticity(ImportanceCoefficient::MODERATE_ADVANTAGE);
                coefficient-
>set_gradient(ImportanceCoefficient::STRONG_ADVANTAGE);
                coefficient->calculate_coefficients();
            }
            step++;
            vtkSmartPointer<vtkKdTreePointLocator> pointTree =
                    vtkSmartPointer<vtkKdTreePointLocator>::New();
            pointTree->SetDataSet(contourPD);
            pointTree->BuildLocator();

            pointTree->FindPointsWithinRadius (1.0, nextPoint,closestPoints);
            double h = 1.1;
            while (closestPoints->GetNumberOfIds() < 2){
                pointTree->FindPointsWithinRadius (h,
nextPoint,closestPoints);
                h+=0.1;
            }

            double nextPoints[closestPoints->GetNumberOfIds()][3];

            for(vtkIdType i = 0; i < closestPoints->GetNumberOfIds(); i++)
            {
                vtkIdType point_ind = closestPoints->GetId(i);
                contourPD->GetPoint(point_ind, nextPoints[i]);

            }
```

```
            std::vector<PointSpecification *> specs;

            for(vtkIdType i = 1; i < closestPoints->GetNumberOfIds(); i++) {
                PointSpecification *point_spec = new PointSpecification() ;
                point_spec->id = closestPoints->GetId(i);
                point_spec->y_value = nextPoints[i][1];

                vtkIdType point_ind = pointAllTree-
>FindClosestPoint(nextPoints[i]);

                double vVorticity[3];
                vorticityCellArray->GetTuple(point_ind, vVorticity);
                double vGradient[16];
                gradients->GetTuple(point_ind, vGradient);

                point_spec->dif_vorticity = fabs(90-
vtkMath::DegreesFromRadians(vtkMath::AngleBetweenVectors(vVorticity,start2end
_vector)));
                point_spec->dif_distance =
vtkMath::Distance2BetweenPoints(endPoint, nextPoints[i]);
                point_spec->dif_gradient = vGradient[1];

                specs.push_back(point_spec);

            }

            if (specs.empty()){
                continue;
            }

            int next_point_ind =0;
            this->point_evaluation (specs, &next_point_ind, coefficient);

            prev_point [0] = nextPoint[0];
            prev_point [1] = nextPoint[1];
            prev_point [2] = nextPoint[2];

            double* nextP = contourPD->GetPoint(next_point_ind);
                        nextPoint[0] =nextP[0];
                        nextPoint[1] = nextP[1];
                        nextPoint[2] = nextP[2];

            contourPs->InsertNextPoint(nextPoint);

            vtkIdType point_ind_next_all = pointAllTree-
>FindClosestPoint(nextPoint);

            vorticityCellArray->GetTuple(point_ind_next_all,
&vorticityCurrent[0]);
            gradients->GetTuple(point_ind_next_all, &gradientCurrent[0]);

            for(vtkIdType i = 0; i < closestPoints->GetNumberOfIds(); i++) {
                this->ReallyDeletePoint(contourPD->GetPoints(),
closestPoints->GetId(i));
            }

            distance_stop = vtkMath::Distance2BetweenPoints(endPoint,
nextPoint);

            std::cout<<"d stop "<<distance_stop<<std::endl;
        }
```

```
        free (coefficient);
        contourPs->InsertNextPoint(endPoint);
        nextPoint[0] = endPoint[0];
        nextPoint[1] = endPoint[1];
        nextPoint[2] = endPoint[2];

        cout<<"second part start"<<endl;

        int contourPs_rotate = contourPs->GetNumberOfPoints() - 2;

        double vector [3] = {endPoint[0]-startPoint[0],
                             endPoint[1]-startPoint[1],
                             endPoint[2]-startPoint[2]};
        vtkMath::Normalize(vector);

            double first_p[3];
            contourPs->GetPoint(1, first_p);
            double first_vector [3] = {first_p[0] - endPoint[0], first_p[1] -
endPoint[1], first_p[2]- endPoint[2]};
            double cross[3];
            vtkMath::Cross(vector, first_vector, cross);
            double angle = (cross[1] < 0)?vtkMath::RadiansFromDegrees(-90.0):
vtkMath::RadiansFromDegrees(90.0);


        for(vtkIdType i = contourPs_rotate; i > (contourPs_rotate-4); i--) {
            double current_point[3] ;

            contourPs->GetPoint(i, current_point);
            double current_vector [3] = {current_point[0] - endPoint[0],
current_point[1] - endPoint[1], current_point[2]- endPoint[2]};
            double result_vector[3];
            vtkQuaternion<double> q = vtkQuaternion<double>();
            q.SetRotationAngleAndAxis(angle, vector[0],vector[1],vector[2]);
            q.Normalize();
            double norm_q [4];
            q.Get(norm_q);
            RotateVectorByNormalizedQuaternion(current_vector, norm_q ,
result_vector);
            double mirror_point [3] = {endPoint[0]+result_vector[0],
endPoint[1]+result_vector[1],endPoint[2]+result_vector[2]};


            vtkSmartPointer<vtkKdTreePointLocator> pointTree =
                    vtkSmartPointer<vtkKdTreePointLocator>::New();
            pointTree->SetDataSet(contourPD);
            pointTree->BuildLocator();
            vtkIdType point_ind = pointTree->FindClosestPoint(mirror_point);
            contourPD->GetPoint(point_ind, nextPoint);
            contourPs->InsertNextPoint(nextPoint);

        }


        prev_point [0] = nextPoint[0];
        prev_point [1] = nextPoint[1];
        prev_point [2] = nextPoint[2];

        cout<<"third part start"<<endl;

        distance_stop = vtkMath::Distance2BetweenPoints(startPoint,
nextPoint);
```

```
        step = 0;
        coefficient = new ImportanceCoefficient();
                std::cout<<"COEFFICIENT START"<<std::endl;
                std::cout<<"distance coef = "<<coefficient-
>get_imp_distance()<<
                           " y-value coef = "<<coefficient->get_imp_y_value()
<<
                           " vorticity coef = "<<coefficient-
>get_imp_vorticity()<<
                           std::endl;
                f_coef = 1;
                while (distance_stop > 6.0 && step < 20){
                    double angle = get_angle(endPoint, c, nextPoint);
                    if (f_coef && angle > 55  ){
                        f_coef = 0;
                        coefficient-
>set_y_value(ImportanceCoefficient::STRONG_ADVANTAGE);
                        coefficient-
>set_vorticity(ImportanceCoefficient::NO_ADVANTAGE);
                        coefficient-
>set_gradient(ImportanceCoefficient::STRONG_ADVANTAGE);
                        coefficient->calculate_coefficients();
                        std::cout<<"distance coef = "<<coefficient-
>get_imp_distance()<<
                                   " y-value coef = "<<coefficient-
>get_imp_y_value() <<
                                   " vorticity coef = "<<coefficient-
>get_imp_vorticity()<<
                                   std::endl;
                        std::cout<<"COEFFICIENT WAS CHANGED"<<std::endl;
                    }
                    step++;
                    vtkSmartPointer<vtkKdTreePointLocator> pointTree =
                            vtkSmartPointer<vtkKdTreePointLocator>::New();
                    pointTree->SetDataSet(contourPD);
                    pointTree->BuildLocator();

                    pointTree->FindPointsWithinRadius (1.0,
nextPoint,closestPoints);
                    double h = 1.1;
                    while (closestPoints->GetNumberOfIds() < 2 && h <3){
                        pointTree->FindPointsWithinRadius (h,
nextPoint,closestPoints);
                        h+=0.1;
                    }

                    if (closestPoints->GetNumberOfIds() < 2) {
                        continue;
                    }
                    double nextPoints[closestPoints->GetNumberOfIds()][3];

                    for(vtkIdType i = 0; i < closestPoints->GetNumberOfIds();
i++)
                    {
                        vtkIdType point_ind = closestPoints->GetId(i);
                        contourPD->GetPoint(point_ind, nextPoints[i]);
                    }
                    std::vector<PointSpecification *> specs;

                    for(vtkIdType i = 1; i < closestPoints->GetNumberOfIds();
i++) {
                        PointSpecification *point_spec = new
PointSpecification() ;
```

```
                        point_spec->id = closestPoints->GetId(i);
                        point_spec->y_value = nextPoints[i][1];

                        vtkIdType point_ind = pointAllTree-
>FindClosestPoint(nextPoints[i]);

                        double vVorticity[3];
                        vorticityCellArray->GetTuple(point_ind, vVorticity);
                        double vGradient[16];
                        gradients->GetTuple(point_ind, vGradient);

                        point_spec->dif_vorticity = fabs(90-
vtkMath::DegreesFromRadians(vtkMath::AngleBetweenVectors(vVorticity,start2end
_vector)));
                        point_spec->dif_distance =
vtkMath::Distance2BetweenPoints(startPoint, nextPoints[i]);
                        point_spec->dif_gradient =  vGradient[1];



                        specs.push_back(point_spec);

                    }

                    if (specs.empty()){
                        continue;
                    }

                    int next_point_ind =0;
                    this->point_evaluation (specs, &next_point_ind,
coefficient);


                    prev_point [0] = nextPoint[0];
                    prev_point [1] = nextPoint[1];
                    prev_point [2] = nextPoint[2];

                    double* nextP = contourPD->GetPoint(next_point_ind);
                            nextPoint[0] =nextP[0];
                            nextPoint[1] = nextP[1];
                            nextPoint[2] = nextP[2];

                    contourPs->InsertNextPoint(nextPoint);

                    vtkIdType point_ind_next_all = pointAllTree-
>FindClosestPoint(nextPoint);

                    vorticityCellArray->GetTuple(point_ind_next_all,
&vorticityCurrent[0]);
                    gradients->GetTuple(point_ind_next_all,
&gradientCurrent[0]);
                    for(vtkIdType i = 0; i < closestPoints->GetNumberOfIds();
i++) {
                        this->ReallyDeletePoint(contourPD->GetPoints(),
closestPoints->GetId(i));
                    }

                    distance_stop =
vtkMath::Distance2BetweenPoints(startPoint, nextPoint);

                }

                free (coefficient);
```

```
                contourPs->InsertNextPoint(startPoint);

        vtkSmartPointer<vtkPolyData> myPoly =
vtkSmartPointer<vtkPolyData>::New();
        vtkSmartPointer<vtkPolyData> inPoly =
vtkSmartPointer<vtkPolyData>::New();
        inPoly->SetPoints(contourPs);
        pointsToLine (iDStart, inPoly, myPoly);


         vtkSmartPointer<vtkPolyData> outputPoly;
         outputPoly = myPoly;

         std::cout<<"autoBounds ... to contourWidget "<<(*it).first<<" #
points "<<outputPoly->GetNumberOfPoints()<<std::endl;

         vtkSmartPointer<vtkContourWidgetDebug> contourWidget =
vtkSmartPointer<vtkContourWidgetDebug>::New();
         contourWidget->Modified();
         contourWidget->SetInteractor(rwi);

         vtkSmartPointer<vtkOrientedGlyphContourRepresentationDebug>
representation =
vtkSmartPointer<vtkOrientedGlyphContourRepresentationDebug>::New();

         double color[3];
         double id;
         if (workJawId) {
             if ((*it).first <20)
                 id =  (19 - (*it).first) * 0.1d;
             else
                 id = (8 + (*it).first-20) * 0.1d;
         } else {

             if ((*it).first <40)
                 id =  (39 - (*it).first) * 0.1d;
             else
                 id = (8 + (*it).first-40 ) * 0.1d;

         }

         getLut()->GetColor(id, color);
         representation->GetLinesProperty()->SetColor(color);
         representation->GetLinesProperty()->SetLineWidth(3.0);
         representation->AlwaysOnTopOn();

         vtkSmartPointer<vtkPolygonalSurfacePointPlacerDebug> pointPlacer =
                 vtkSmartPointer<vtkPolygonalSurfacePointPlacerDebug>::New();
         pointPlacer->AddProp(workActor);

         pointPlacer->GetPolys()->AddItem(workPolydata);
         representation->SetPointPlacer(pointPlacer);
         vtkSmartPointer<vtkPolygonalSurfaceContourLineInterpolatorDebug>
interpolator =

vtkSmartPointer<vtkPolygonalSurfaceContourLineInterpolatorDebug>::New();
         interpolator->GetPolys()->AddItem(workPolydata);
         representation->SetLineInterpolator(interpolator);

         contourWidget->SetRepresentation(representation);
         contourWidget->On();
```

```
    contourWidget->Initialize(outputPoly, vtkContourWidgetDebug::Define);

    contourWidget->CloseLoop();
    contourWidget->SetWidgetState(vtkContourWidgetDebug::Manipulate);

    contour->setContourRepresentation(representation);
    contour->setContourWidget(contourWidget);
   contourWidget->EnabledOn();

    contours_map.emplace(std::make_pair( (*it).first, contour));
    workContour->push_back((*it).first);
    it++;

}
this->autoContourPoints.clear();

actorPoints->InitTraversal();
vtkIdType num = actorPoints->GetNumberOfItems();

for (vtkIdType i = 0; i < num; i++) {
    vtkSmartPointer<vtkActor> actor = actorPoints->GetNextActor();
    renderer->RemoveActor(actor);
}

actorPoints->RemoveAllItems();

std::cout<<"autoBounds ... End"<<std::endl;
renderWindow->Render();
```

ПРИЛОЖЕНИЕ В

Реализация алгоритма определения принадлежности точки контуру

```
void
ToothSegmentationController::point_evaluation(std::vector<PointSpecification*
> & points_spec, int *next_point_id, ImportanceCoefficient *coefficient) {

    unsigned long i = 0, j = 0;
    auto n_points = points_spec.size();

    float m_y_value[n_points][n_points], m_vorticity[n_points][n_points],
            m_distance[n_points][n_points], m_gradient[n_points][n_points];

    float dm_y_value[n_points], dm_vorticity[n_points],
dm_distance[n_points],
            dm_gradient[n_points];

    float intersection_cur;
    float intersection_max = -1.0f;

    for (i = 0; i < n_points; i++) {

        m_y_value[i][i] = 1.0;
        m_vorticity[i][i] = 1.0;
        m_distance[i][i] = 1.0;
        m_gradient[i][i] = 1.0;

        for (j = i + 1; j < n_points; j++) {

            m_y_value[i][j] = points_spec[i]->y_value / points_spec[j]-
>y_value; // max
            m_y_value[j][i] = points_spec[j]->y_value / points_spec[i]-
>y_value;


            m_vorticity[i][j] = points_spec[i]->dif_vorticity /
points_spec[j]->dif_vorticity; // max
            m_vorticity[j][i] = points_spec[j]->dif_vorticity /
points_spec[i]->dif_vorticity;

            m_distance[i][j] = points_spec[j]->dif_distance / points_spec[i]-
>dif_distance; // min
            m_distance[j][i] = points_spec[i]->dif_distance/ points_spec[j]-
>dif_distance;

            m_gradient[i][j] = points_spec[i]->dif_gradient / points_spec[j]-
>dif_gradient; // max
            m_gradient[j][i] = points_spec[j]->dif_gradient / points_spec[i]-
>dif_gradient;
        }
    }


    for (i = 0; i < n_points; i++) {

        dm_y_value[i] = .0;
        dm_vorticity[i] = .0;
        dm_distance[i] = .0;
        dm_gradient[i] = .0;
```

```
        for (j = 0; j < n_points; j++) {
            dm_y_value[i] += m_y_value[j][i];
            dm_vorticity[i] += m_vorticity[j][i];
            dm_distance[i] += m_distance[j][i];
            dm_gradient[i] += m_gradient[j][i];
        }


        dm_y_value[i] = powf((1.0f / dm_y_value[i]), coefficient-
>get_imp_y_value());
        dm_vorticity[i] = powf((1.0f / dm_vorticity[i]), coefficient-
>get_imp_vorticity());
        dm_distance[i] = powf((1.0f / dm_distance[i]), coefficient-
>get_imp_distance());
        dm_gradient[i] = powf((1.0f / dm_gradient[i]), coefficient-
>get_imp_gradient());


        intersection_cur = std::min(
                {dm_y_value[i], dm_vorticity[i], dm_distance[i],
dm_gradient[i], });\

        if (intersection_cur > intersection_max){
                intersection_max = intersection_cur;
                *next_point_id = points_spec[i]->id;
        }

    };
                                      }
```