



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

разработка ПО, которое:

- 1) реализует алгоритм деформации (сокращения и растяжения) человеческого бицепса с помощью геометрической модели на узлах, сохраняющей объем при деформации;**
- 2) предоставляет возможности загрузки модели из конфигурационного файла, управления ее состоянием (сокращение и растяжение) и положением (вращение, перемещение и масштабирование)**

Студент ИУ7-53Б
(Группа)

П. Г. Пересторонин
(И.О.Фамилия)

Руководитель курсового проекта

А. А. Оленев
(И.О.Фамилия)

2020 г.

Оглавление

Введение	3
1 Аналитическая часть	5
1.1 Методы визуализации мышц	5
1.1.1 Геометрические методы	5
1.1.2 Физические методы	6
1.1.3 Методы, основанные на данных	7
1.2 Существующие программные обеспечения	7
1.3 Модели мышцы и каркаса	9
1.3.1 Модель мышцы	9
1.3.2 Расчёт формул деформации мышцы	11
1.3.3 Модель каркаса мышцы	15
1.4 Анализ алгоритмов удаления невидимых линий и поверхностей	16
1.4.1 Алгоритм обратной трассировки лучей	16
1.4.2 Алгоритм, использующий Z-буфер	17
1.4.3 Алгоритм Робертса	18
1.5 Анализ методов закрашивания	20
1.5.1 Простая закраска	20
1.5.2 Закраска по Гуро	21
1.5.3 Закраска по Фонгу	22
2 Конструкторская часть	24
2.1 Требования к программному обеспечению	24
2.2 Разработка алгоритмов	25
2.2.1 Алгоритм деформации мышцы	25
2.2.2 Алгоритм триангуляции мышцы	26
2.2.3 Алгоритм синтеза изображения	27
3 Технологическая часть	29
3.1 Средства реализации	29
3.2 Реализация алгоритмов	30

4 Исследовательская часть	37
4.1 Результаты работы программного обеспечения	37
4.2 Постановка эксперимента	39
4.2.1 Цель эксперимента	39
4.2.2 Данные реального бицепса	40
4.2.3 Замеры реального бицепса	41
4.2.4 Замеры разработанной модели	44
4.2.5 Сравнение замеров	47
Заключение	49
Литература	50

Введение

Задача визуализации мышц решается в областях, где применяется компьютерная графика: разработка игр, профессиональное программное обеспечение, которое используется биологами и врачами, монтаж в кино и других. Такая задача решается геометрическими методами, физическими, а также методами, которые основаны на данных[1].

Метод решения задачи визуализации мышц выбирается исходя из требований и специфики предметной области, где эта задача ставится. Универсального метода решения подобной задачи нет: геометрические методы сохраняют инварианты за счёт решения соотношений без приближений, при этом редко находят применение в динамических средах; физические методы визуализации, наоборот, лучше визуализируют динамические среды, при этом аппроксимируя получающиеся во время решения алгебраические системы уравнений для ускорения вычислений или решая эти системы численными методами, что также приводит к потере точности. Методы, основанные на данных, применяются в случаях, когда требуется визуализировать мышцы существующего в нужном виде объекта.

Цель работы - разработать программное обеспечение, которое предоставляет возможности загрузки параметров модели геометрической модели бицепса на узлах из конфигурационного файла, изменения этих параметров в интерактивном режиме, управления состоянием модели (сокращение и растяжение), а также положением (вращение, перемещение и масштабирование).

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- формально описать структуру моделей мышцы и каркаса;
- рассчитать формулы деформации геометрической модели с сохранением объема;

- выбрать алгоритмы трехмёрной графики, визуализирующие модель;
- реализовать алгоритмы для визуализации описанных выше объектов.

1 Аналитическая часть

1.1 Методы визуализации мышц

Как уже было сказано во введении, все методы визуализации мышц можно разделить на 3 группы:

- геометрические;
- физические;
- методы, основанные на данных.

1.1.1 Геометрические методы

Геометрические методы описывают мышцу как поверхность, задаваемую математической функцией или множеством математических функций. Данные методы визуализации мышц использовались в ранних системах, потому что физическим методам было недостаточно вычислительных мощностей [1]. Геометрические методы позволяют решить задачу без интерполяции и аппроксимации, вследствие чего получается действительное решение, а не приближенное. Такие методы используются для визуализации сокращения мышц, которые могут использоваться как основа для визуализации деформации кожи и анимации лица. Данные методы позволяют моделировать также и веретенообразные мышцы, используя и комбинируя функции фигур вращения. Учитывая факт ограниченности данных методов из-за использования математических функций в своей основе, а также тот факт, что сокращение мышц определяется возможными сокращениями скелета, существуют задачи, в которых данные методы не позволяют достичь достаточной степени реализма с точки зрения физиологии и биомеханики [2].

Наиболее распространённые геометрические методы:

- FFD (Free Form Deformation)[3];
- методы, использующие параметрические и полигональные поверхности[4];
- методы, использующие поверхности, заданные неявными функциями[5][6].

1.1.2 Физические методы

В то время, как геометрический подход к решению задачи визуализации мышц доказал свою применимость в статических графических системах, лежащий в его основе подход описания моделей через математические функции не позволяет работать с динамическими системами, то есть системами, положение которых меняется со временем по законам физики[1]. Исследователи подошли к решению проблемы построения динамической системы с точки зрения физики.

Существуют физические методы решения задачи визуализации мышц, которые используются совместно с геометрическими, однако основная идея таких методов заключается в том, что в них весь объект делится на части и учитывается взаимодействие этих частей между собой в каждый момент времени.

Наиболее распространённые физические методы:

- MSS (Mass-Spring System)[7];
- метод конечных элементов (FEM)[8][9];
- метод конечных объемов (FVM)[10][11].

Примером модели мышц на основе физических методов может послужить VIPER[12] (англ. Volume Invariant Position-based Elastic Rods) - версия модели позиционных эластичных стержней[13] (англ. Position-based

Elastic Rods), сохраняющая объем. С помощью данных моделей мышца представляется в виде набора стержней, состоящих из узлов, имеющих в случае обычной модели 2 параметра для каждого узла: позиция и степень скручивания, а в случае модели, сохраняющей объем - 3 параметра: позиция, степень скручивания и масштаб. Данные модели используются для представления динамической системы.

1.1.3 Методы, основанные на данных

Эти методы используют данные, которые собираются с поверхности интересующего объекта с помощью системы захвата движения, которая считывает данные с исследуемого объекта при помощи так называемых маркеров[14]. Подобные методы новее методов, перечисленных выше, и нередко позволяют достичь более реальных результатов для определённого объекта, нежели их конкуренты. Однако данный подход решает более частную задачу, что сокращает область его применимости.

1.2 Существующие программные обеспечения

На рисунках 1.1 и 1.2 показано, как выглядят мышцы при использовании модели на основе VIPER, физического метода, описанного в пункте 1.1.2.

Ещё одним популярным продуктом в сфере визуализации мышц является X-Muscle System[15] - расширение для программы Blender[16], которое поддерживает создание мышцы в интерактивном режиме. В данном случае мышца состоит из *канонических костей* — геометрических примитивов, которые имеют 3 параметра: позиция, направление и ориентация. На рисунках 1.3 и 1.4 показаны примеры мышц, на основе модели X-Muscle



Рис. 1.1: Мышцы на основе VIPER, прозрачные стержни.

16 ms / 62 fps

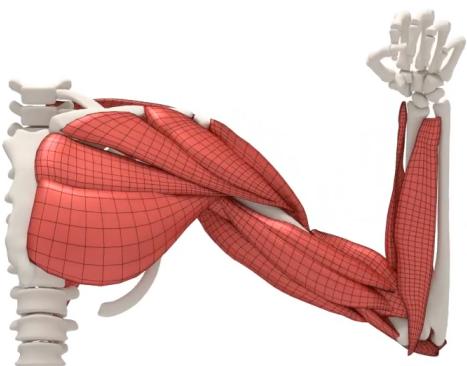


Рис. 1.2: Мышцы на основе VIPER, вид с сеткой.

System.

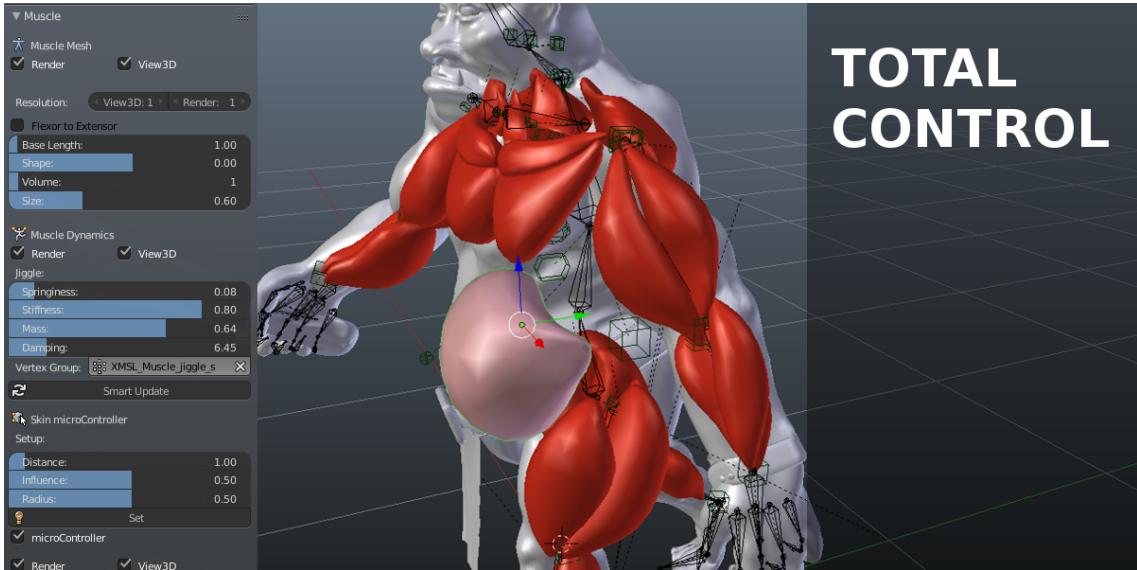


Рис. 1.3: Мышцы X-Muscle System. Проектирование.



Рис. 1.4: Мышцы X-Muscle System с текстурами.

1.3 Модели мышцы и каркаса

1.3.1 Модель мышцы

Бицепс является веретенообразной мышцой[2], что допускает использование геометрических методов. Учитывая, что программное обеспечение, получаемое в результате данной работы, предусматривает статическую систему, можно сделать вывод, что ограничений по использованию геомет-

рических методов нет[1].

Модель мышцы в данной работе представляется с помощью группы усечённых конусов, полученной путем вращения группы отрезков, находящихся между узлами модели. При такой модели радиусы основания конуса описываются значениями радиусов в точках начала и конца отрезка, которые далее будут называться *радиусом узла*. Такая модель позволяет аппроксимировать математические функции в продольном сечении с задаваемой точностью, а также задавать параметр в каждой точке получающейся табличной функции. Модель с 4 узлами представлена на рисунке 1.5, получена данная модель путем вращения группы отрезков, представленных на рисунке 1.6 (отрезки: r_1r_2 , r_2r_3 , r_3r_4). Чем большая точность требуется, тем большее количество узлов потребуется для аппроксимации мышцы.

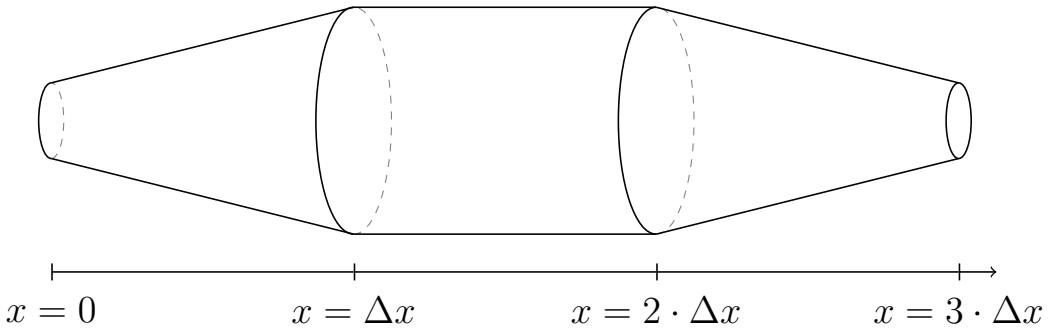


Рис. 1.5: Модель на 4 узлах.

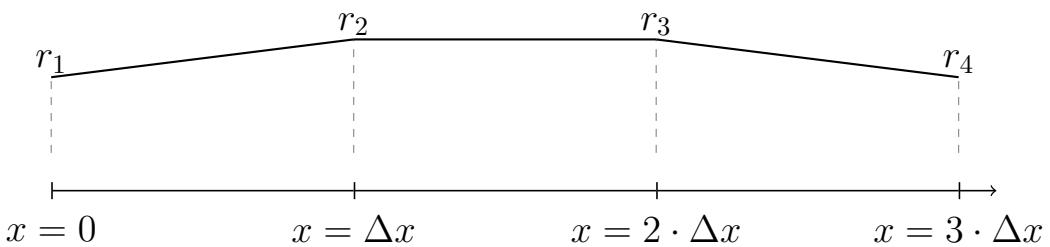


Рис. 1.6: Фигура, путём вращения которой получаются усечённые конусы, составляющие мышцу.

Расстояние между узлами постоянное. Данный факт не вносит ограничений в представление модели, так как, описывая узел пропорционально соседям этого узла, получается представление, в котором визуально расстояние между этими двумя соседними узлами удваивается. Сделав обратное действие можно вводить новые узлы, уменьшая при этом расстояние между узлами и увеличивая тем самым точность.

Учитывая описанное выше, для исчерпывающего описания состояния и положения мышцы требуется массив радиусов оснований конусов в узлах и расстояние между узлами.

Такой набор не хранит информации о поведении узлов при деформациях, поэтому для полного описания свойств мышцы также для каждого узла требуется добавить коэффициент приращения, который означает относительный прирост данного узла.

Таким образом мышцу в рассматриваемой системе можно описать следующим образом:

- массив радиусов узлов;
- расстояние между узлами;
- массив коэффициентов прироста радиусов узлов.

1.3.2 Расчёт формул деформации мышцы

Как уже было сказано выше, общий объем модели составляется из объемов составляющих ее усечённых конусов, объем которых вычисляется как фигура вращения отрезка.

Отрезок задаётся в виде прямой, заданной уравнением $y = ax + b$, которая ограничена по x : $x \in [0; \Delta x]$, где Δx - расстояние между узлами. Во всех последующих вычислениях подразумевается, что пара соседних узлов с индексами i и $(i + 1)$ сдвигается на расстояние $(1 - i) \cdot \Delta x$ по оси X (во всех последующих расчетах интересует только величина Δx , поэтому все последующие расчёты справделивы и для исходного положения составных частей модели). Площадь усечённого конуса в таком случае рассчитывается по формуле (1.1):

$$\mathcal{V} = \pi \cdot \int_0^{\Delta x} f^2(x) dx, \quad \text{где } f(x) = ax + b \quad (1.1)$$

Далее можно подставить значение функции и рассчитать интеграл (1.2):

$$\mathcal{V} = \pi \cdot \int_0^{\Delta x} (a^2 x^2 + 2abx + b^2) dx = \pi \cdot \left(\frac{a^2 x^3}{3} + abx^2 + b^2 x \right) \Big|_0^{\Delta x} \quad (1.2)$$

Из чего следует (1.3):

$$\mathcal{V} = \pi \cdot \left(\frac{a^2 \Delta x^3}{3} + ab \Delta x^2 + b^2 \Delta x \right) \quad (1.3)$$

Коэффициенты a (угла наклона прямой) и b (точки пересечения прямой с Oy) можно найти исходя из двух имеющихся точек отрезка. Так, для отрезка между i -ым и $(i+1)$ -ым узлами, получается (1.4) и (1.5):

$$a = \frac{y_{i+1} - y_i}{\Delta x} \quad (1.4)$$

$$b = y_i \quad (1.5)$$

Пусть n - кол-во узлов. Тогда $R = \{r_1, r_2, \dots, r_n\}$ - массив радиусов узлов, $M = \{m_1, m_2, \dots, m_n\}$ - массив коэффициентов прироста радиусов узлов. Отсюда (учитывая формулы (1.3), (1.4) и (1.5)) получается, что общий объем, для удобства последующих вычислений поделённый на π , можно вычислить по формуле (1.6):

$$V = \frac{\mathcal{V}}{\pi} = \sum_{i=1}^{n-1} \left(\frac{(r_{i+1} - r_i)^2 \cdot \Delta x}{3} + 2r_i(r_{i+1} - r_i) \cdot \Delta x + r_i^2 \Delta x \right) \quad (1.6)$$

После выноса за знак суммы Δx получается итоговая формула вычисления объема (1.7):

$$V = \Delta x \cdot \sum_{i=1}^{n-1} \left(\frac{(r_{i+1} - r_i)^2}{3} + 2r_i(r_{i+1} - r_1) + r_i^2 \right) \quad (1.7)$$

Которая представима как функцию от расстояния между узлами Δx и массива радиусов узлов (1.8):

$$V(\Delta x, R) = \Delta x \cdot F(R), \quad (1.8)$$

где $F(R)$ - функция (1.9):

$$F(R) = \sum_{i=1}^{n-1} \left(\frac{(r_{i+1} - r_i)^2}{3} + 2r_i(r_{i+1} - r_1) + r_i^2 \right) \quad (1.9)$$

Длина модели L является суммой расстояний между узлами и для n узлов находится следующим образом (1.10):

$$L = (n - 1) \cdot \Delta x \quad (1.10)$$

Откуда выводится зависимость расстояния между узлами от длины (1.11):

$$\Delta x = \frac{L}{n - 1} \quad (1.11)$$

При деформации модели (сокращении или растяжении) по заданию требуется, чтобы объем оставался постоянным. Из выражения (1.8) можно сделать вывод, что для равенства $V = V'$, где V - объем до деформации, а V' - после, требуется выполнение равенства (1.12):

$$\Delta x \cdot F(R) = \Delta x' \cdot F(R') \quad (1.12)$$

где R' - массив радиусов узлов после деформации.

$\Delta x'$, учитывая (1.11), находится из (1.13):

$$\Delta x' = \frac{L'}{n - 1} = \frac{L + \delta x}{n - 1} = \Delta x + \frac{\delta x}{n - 1} \quad (1.13)$$

где δx - изменение длины, а $L' = L + \delta x$ - новое значение длины модели.

$F(R')$ можно найти, как (1.14):

$$F(R') = \frac{F(R) \cdot \Delta x}{\Delta x'} = \frac{\Delta x}{\Delta x'} \cdot F(R) \quad (1.14)$$

Пусть δy - элементарное приращение радиуса. Тогда новое значение для радиуса каждого узла находится из соотношения:

$$r'_i = r_i + m_i \cdot \delta y \quad (1.15)$$

где m_i (как было описано выше) - коэффициент прироста радиуса.

Таким образом выражается через (1.15) радиусы массива R' и найти $F(R')$ как (1.16):

$$\begin{aligned} F(R') &= \sum_{i=1}^{n-1} \left(\frac{((r_{i+1} + m_{i+1} \cdot \delta y) - (r_i + m_i \cdot \delta y))^2}{3} + \right. \\ &\quad \left. + 2(r_i + m_i \cdot \delta y) \cdot (r_{i+1} + m_{i+1} \cdot \delta y - r_i + m_i \cdot \delta y) + (r_i + m_i \cdot \delta y)^2 \right) \end{aligned} \quad (1.16)$$

После приведения подобных слагаемых относительно δy получается (1.17):

$$F(R') = A\delta y^2 + B\delta y + C, \quad (1.17)$$

где A, B, C рассчитываются из (1.18), (1.19), (1.20) соответственно.

$$A = \sum_{i=1}^{n-1} \left(\frac{1}{3}m_{i+1}^2 - \frac{5}{3}m_{i+1}m_i + \frac{7}{3}m_i^2 \right) \quad (1.18)$$

$$B = \sum_{i=1}^{n-1} (m_{i+1}r_i + m_ir_{i+1}) \quad (1.19)$$

$$C = \sum_{i=1}^{n-1} \left(\frac{(r_{i+1} - r_i)^2}{3} + r_{n+1}r_n \right) \quad (1.20)$$

Квадратное уравнение (1.17) имеет решения (1.21):

$$\delta y_{1,2} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \quad (1.21)$$

Среди решений выбирается большее, потому что меньшее решение находит сохранение объема при отрицательных радиусах, что не является верным в рамках поставленной задачи. Таким образом δy - больший корень среди корней из (1.21), который равен (1.22):

$$\delta y = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \quad (1.22)$$

Таким образом при деформации модели на величину δx новые значения радиусов узлов имеют вид (1.15), где величина δy находится из выражения (1.22).

1.3.3 Модель каркаса мышцы

Каркас - модель, симулирующая в данной работе скелет руки двумя соединёнными концами стержнями, к которым крепится мышца. Каркас служит для улучшения визуализации мышцы и повышения реалистичности изображения. Положение каркаса может быть однозначно определено положением и состоянием модели мышцы, однако точки крепления не зависят от нее. В связи с этим для представления каркаса требуются следующие параметры:

- расстояние от нескрепленного конца первого стержня до точки крепления мышцы;
- расстояние от точки крепления мышцы к первому стержню до точки соединения стержней;
- расстояние от точки соединения стержней до точки крепления мышцы.

цы ко второму стержню;

- расстояние от точки крепления мышцы ко второму стержню до нескрепленного конца второго стержня;

Углы наклона стержней каркаса находятся из теоремы косинусов, которая для треугольника с длинами сторон A , B , C и угла α , лежащего напротив стороны с длиной A , иметь вид (1.23):

$$A^2 = B^2 + C^2 - 2BC \cos(\alpha) \quad (1.23)$$

Откуда можно выразить угол α как (1.24):

$$\alpha = \arccos\left(\frac{B^2 + C^2 - A^2}{2BC}\right) \quad (1.24)$$

1.4 Анализ алгоритмов удаления невидимых линий и поверхностей

При выборе алгоритма удаления невидимых линий и поверхностей учитывается особенность поставленной задачи - работа программы будет выполняться в реальном режиме при взаимодействии с пользователем. Этот факт предъявляет к алгоритму требование по скорости работы. Для выбора наиболее подходящего алгоритма следует рассмотреть уже имеющиеся алгоритмы удаления невидимых линий и поверхностей.

1.4.1 Алгоритм обратной трассировки лучей

Алгоритм работает в пространстве изображения[17].

Идея: для определения цвета пикселя экрана через него из точки наблюдения проводится луч, ищется пересечение первым пересекаемым объектом сцены и определяется освещенность точки пересечения. Эта освещенность складывается из отраженной и преломленной энергий, полученных от источников света, а также отраженной и преломленной энергий, идущих от других объектов сцены. После определения освещенности найденной точки учитывается ослабление света при прохождении через прозрачный материал и в результате получается цвет точки экрана.

Плюсы:

- изображение, которое строится с учётом явлений дисперсии лучей, преломления, а также внутреннего отражения;
- возможность использования в параллельных вычислительных системах.

Минусы:

- трудоёмкие вычисления[18];

1.4.2 Алгоритм, использующий Z-буфер

Алгоритм работает в пространстве изображения[19].

Идея: имеется 2 буфера - буфер кадра, который используется для запоминания цвета каждого пикселя изображения, а также z-буфер - отдельный буфер глубины, используемый для запоминания координаты z (глубины) каждого видимого пикселя изображения. В процессе работы глубина или значение z каждого нового пикселя, который нужно занести в буфер кадра, сравнивается с глубиной того пикселя, который уже занесен в z-буфер. Если это сравнение показывает, что новый пиксель расположен выше пикселя, находящегося в буфере кадра ($z > 0$), то новый пиксель заносится в

цвет рассматриваемого пикселя заносится в буфер кадра, а координата z - в z -буфер. По сути, алгоритм является поиском по x и y наибольшего значения функции $z(x, y)$.

Плюсы:

- возможность обработки произвольных поверхностей, аппроксимируемых полигонами;
- отсутствие требования сортировки объектов по глубине.

Минусы:

- отсутствие возможности работы с прозрачными и просвечивающими объектами (в классической версии).

1.4.3 Алгоритм Робертса

Алгоритм работает в объектном пространстве[20].

Идея: алгоритм прежде всего удаляет из каждого тела те ребра или грани, которые экранируются самим телом. Затем каждое из видимых ребер каждого тела сравнивается с каждым из оставшихся тел для определения того, какая его часть или части, если таковые есть, экранируются этими телами.

Плюсы:

- реализации алгоритма, использующие предварительную приоритетную сортировку вдоль оси z и простые габаритные или минимаксные тесты, демонстрируют почти линейную зависимость от числа объектов[20].

Минусы:

- вычислительная трудоёмкость алгоритма теоретически растет, как квадрат числа объектов[20];
- отсутствие возможности работы с прозрачными и просвечивающими объектами.

Вывод

В таблице 1.1 представлено сравнение алгоритмов[21] удаления невидимых линий и поверхностей (по каждому параметру составлен рейтинг: 1 - лучший алгоритм, 3 - худший). Так как главным требованием к алгоритму является скорость работы, алгоритмы были оценены по следующим критериям:

- скорость работы (С);
- масштабируемость с ростом количества моделей (ММ);
- масштабируемость с увеличением размера экрана (МЭ);
- работа с фигурами вращения (ФВ).

Алгоритм	С	ММ	МЭ	ФВ
Z-буфера	1	2	1	1
Трассировка лучей	3	1	3	2
Робертса	2	3	1	3

Таблица 1.1: Сравнение алгоритмов удаления невидимых линий и поверхностей.

С учётом результатов в таблице 1.1 был выбран алгоритм **Z-буфера** удаления невидимых линий и поверхностей.

1.5 Анализ методов закрашивания

Методы закрашивания используются для затенения полигонов (или поверхностей, аппроксимированных полигонами) в условиях некоторой сцены, имеющей источники освещения. С учётом взаимного положения рассматриваемого полигона и источника света находится уровень освещённости по закону Ламберта (1.25):

$$I_\alpha = I_0 \cdot \cos(\alpha) \quad (1.25)$$

где I_α - уровень освещённости в рассматриваемой точке, I_0 - максимальный уровень освещённости, а α - угол между вектором нормали к плоскости и вектором, направленным от рассматриваемой точки к источнику освещения (в случае нормированных векторов может быть рассчитан как скалярное произведение данных векторов).

1.5.1 Простая закраска

Идея: вся грань закрашивается одним уровнем интенсивности, который зависит высчитывается по закону Ламберта[21]. При данной закраске все плоскости (в том числе и те, что аппроксимируют фигуры вращения), будут закрашены однотонно, что в случае с фигурами вращения будет давать ложные ребра.

Плюсы:

- используется для работы с многогранниками, обладающими преимущественно диффузным отражением.

Минусы:

- плохо подходит для фигур вращения: видны ребра.

1.5.2 Закраска по Гуро

Идея: билинейная интерполяция в каждой точке интенсивности освещения в вершинах[22].

Нормаль к вершине можно найти несколькими способами:

- интерполировать нормали прилегающих к вершине граней;
- использовать геометрические свойства фигуры (так, например, в случае со сферой ненормированный вектор нормали будет в точности соответствовать вектору от центра сферы до рассматриваемой точки).

После нахождения нормали ко всем вершинам находится интенсивность в каждой вершине по закону Ламберта (1.25). Затем алгоритм проходит сканирующими строками по рассматриваемому полигону для всех $y : y \in [y_{min}; y_{max}]$. Каждая сканирующая строка пересекает 2 ребра многоугольника, пусть для определённости это будут ребра через одноименные вершины: MN и KL . В точках пересечения высчитывается интенсивность путём интерполяции интенсивности в вершинах. Так, для точки пересечения с ребром MN интенсивность будет рассчитана как (1.26):

$$I_{MN} = \frac{l_1}{l_0} \cdot I_M + \frac{l_2}{l_0} \cdot I_N \quad (1.26)$$

где l_1 - расстояние от точки пересечения до вершины N , l_2 - расстояние от точки пересечения до вершины M , l_0 - длина ребра MN . Для точки пересечения сканирующей строки с ребром KL интенсивность высчитывается аналогично.

Далее, после нахождения точек пересечения, алгоритм двигается по Ox от левой точки пересечения X_{left} до правой точки пересечения X_{right} и в каждой точке \mathcal{X} интенсивность рассчитывается как (1.27):

$$I_{\mathcal{X}} = \frac{\mathcal{X} - X_{left}}{X_{right} - X_{left}} \cdot I_{X_{right}} + \frac{X_{right} - \mathcal{X}}{X_{right} - X_{left}} \cdot I_{X_{left}} \quad (1.27)$$

Плюсы:

- преимущественно используется с фигурами вращения с диффузным отражением, аппроксимированными полигонами.

Минусы:

- при закраске многогранников ребра могут стать незаметными.

1.5.3 Закраска по Фонгу

Идея: данный алгоритм работает похожим на алгоритм Гуро образом, однако ключевым отличием является то, что интерполируются не интенсивности в вершинах, а нормали[22]. Таким образом, закон Ламберта в данном алгоритме применяется в каждой точке, а не только в вершинах, что делает этот алгоритм гораздо более трудоёмким, однако с его помощью можно гораздо лучше изображаются блики.

Плюсы:

- преимущественно используется с фигурами вращения с зеркальным отражением, аппроксимированными полигонами.

Минусы:

- самый трудоёмкий алгоритм из рассмотренных[21].

Вывод

В таблице 1.2 представлено сравнение алгоритмов[21] закраски (по каждому параметру составлен рейтинг: 1 - лучший алгоритм, 3 - худший). Так

как требованиями к алгоритму являются высокая скорость работы, а также возможность закраски фигур вращения с диффузными свойствами отражения, алгоритмы были оценены по следующим критериям:

- скорость работы (С);
- работа с фигурами вращения (ФВ);
- работа с фигурами со свойствами диффузного отражения (ДО).

Алгоритм	С	ФВ	ДО
Простой	1	3	1
Гуро	2	1	1
Фонга	3	1	3

Таблица 1.2: Сравнение алгоритмов закраски.

С учётом результатов в таблице 1.2 был выбран алгоритм закраски **Гуро**.

Вывод

В данном разделе были формально описаны модели мышцы, каркаса и законы, по которым эти модели деформируются, были рассмотрены алгоритмы удаления невидимых линий и поверхностей, методы закрашивания поверхностей. В качестве алгоритма удаления невидимых линий и поверхностей был выбран алгоритм z-буфера, в качестве метода закрашивания был выбран алгоритм закраски Гуро.

2 Конструкторская часть

В данном разделе предсатвлены требования к программному обеспечению, а также схемы алгоритмов, выбранных для решения поставленной задачи.

2.1 Требования к программному обеспечению

Программа должна предоставлять доступ к функционалу:

- конфигурирование параметров мышцы через конфигурационный файл;
- конфигурирование параметров произвольного узла мышцы в интерактивном режиме;
- сокращение и растяжение модели;
- вращение, перемещение и масштабирование модели;
- изменение положения источника света.

К программе предъявляются следующие требования:

- время отклика программы должно быть менее 1 секунды для корректной работы в интерактивном режиме;
- программа должна корректно реагировать на любые действия пользователя.

2.2 Разработка алгоритмов

Для алгоритмов, разработанных автором работы, представлены схемы алгоритмов. Для алгоритма синтеза изображения, использующего в своей основе алгоритмы Z-буфера[19] и Гуро[22] представлена блок-схема.

2.2.1 Алгоритм деформации мышцы

На рисунке 2.1 представлена схема алгоритма деформации мышцы.

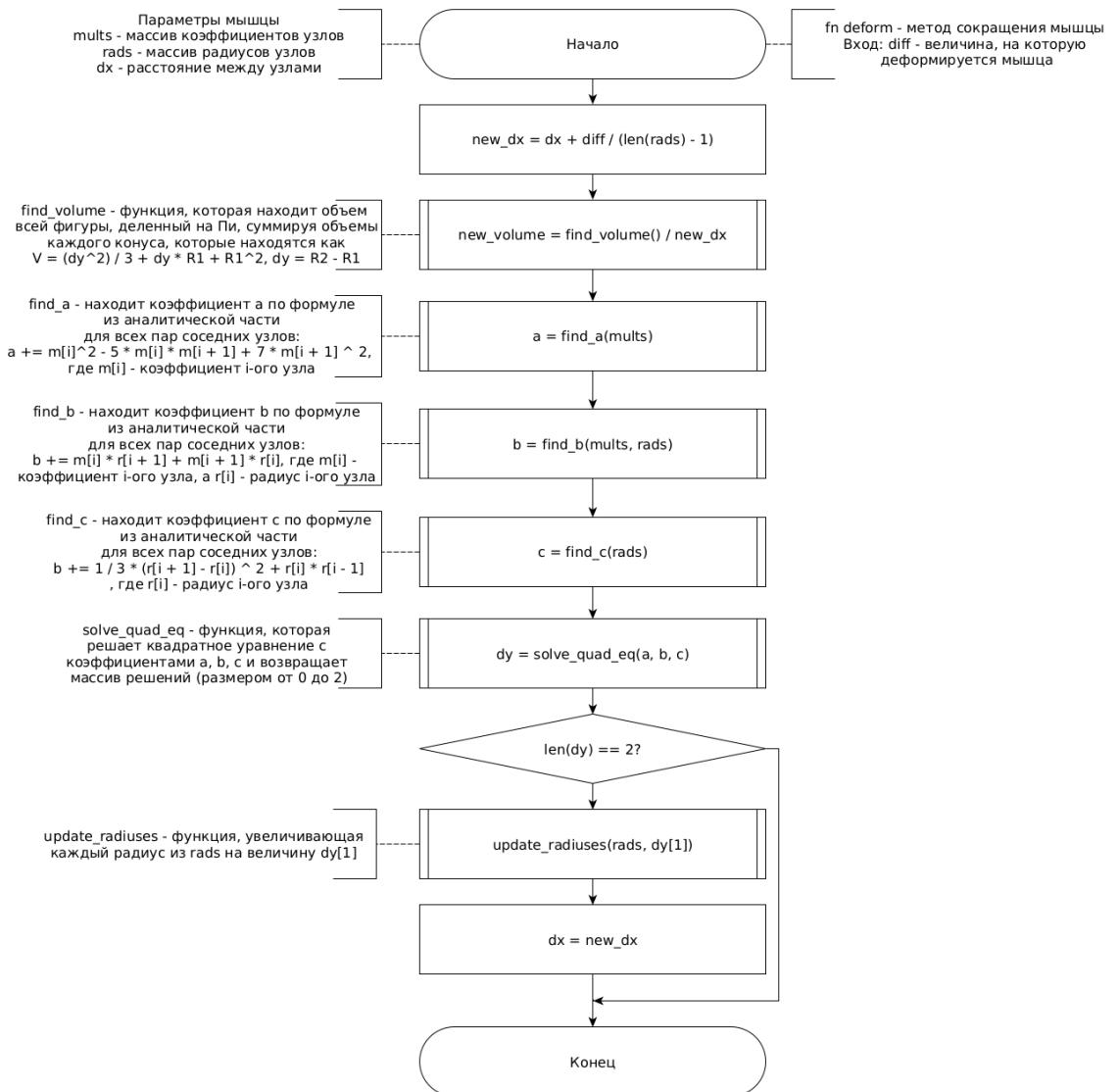


Рис. 2.1: Схема алгоритма деформации мышцы

2.2.2 Алгоритм триангуляции мышцы

На рисунке 2.2 представлена схема алгоритма триангуляции мышцы.

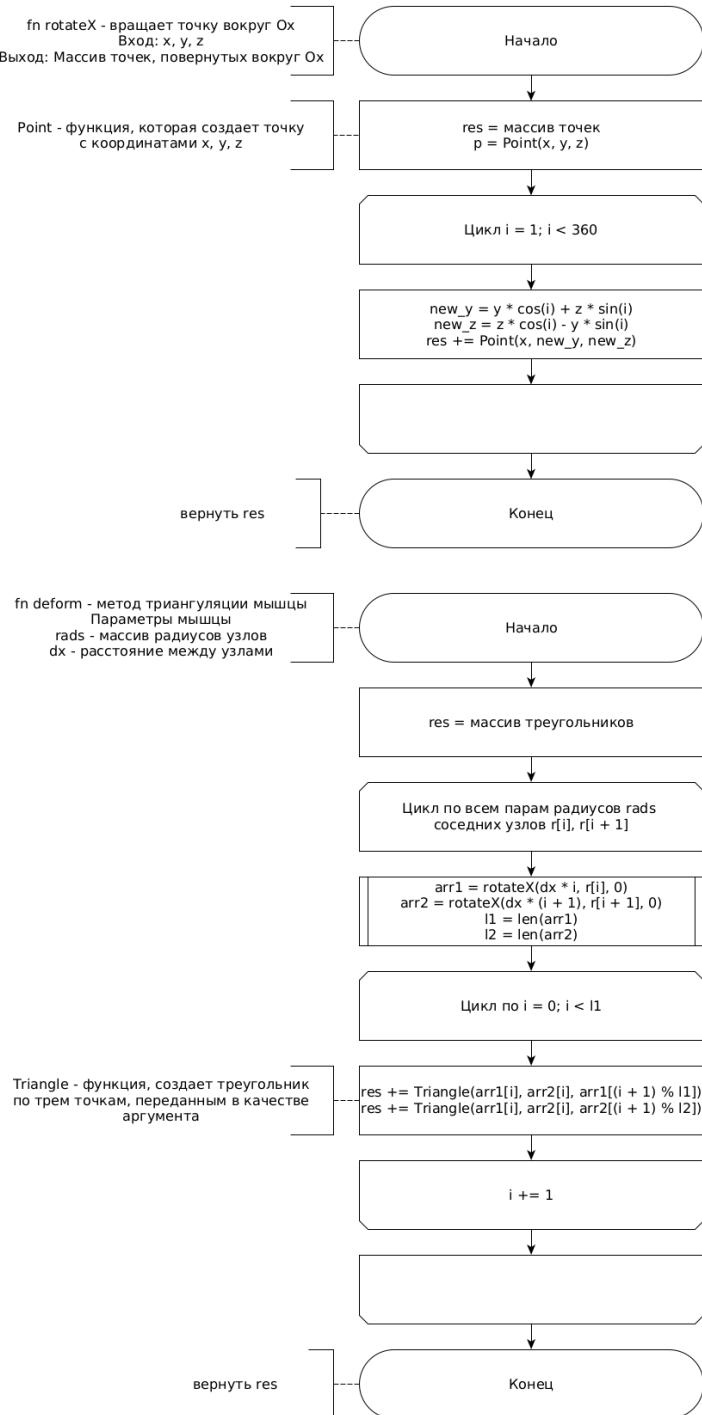


Рис. 2.2: Схема алгоритма триангуляции мышцы

2.2.3 Алгоритм синтеза изображения

На рисунке 2.3 представлена блок-схема алгоритма синтеза изображения.



Рис. 2.3: Блок-схема алгоритма синтеза изображения

Вывод

В данном разделе были представлены требования к программному обеспечению и разработаны схемы реализуемых алгоритмов.

3 Технологическая часть

В данном разделе представлены средства разработки программного обеспечения, детали реализации и тестирование функций.

3.1 Средства реализации

В качестве языка программирования для разработки программного обеспечения был выбран язык программирования Rust[23]. Данный выбор обусловлен тем, что данный язык предоставляет весь требуемый функционал для решения поставленной задачи, а также обладает связанным с ним пакетным менеджером Cargo[24], который содержит инструменты для тестирования разрабатываемого ПО[25].

Для создания пользовательского интерфейса программного обеспечения была использована библиотека gtk-rs[26]. Данная библиотека содержит в себе объекты, позволяющие напрямую работать с пикселями изображения, а также возможности создания панели управления с кнопками, что позволит в интерактивном режиме управлять изображением.

Для тестирования программного обеспечения были использованы инструменты пакетного менеджера Cargo[24], поставляемого вместе с компилятором языка при стандартном способе установке, описанном на официальном сайте языка[23].

В процессе разработки был использован инструмент RLS[27] (англ. *Rust Language Server*), позволяющий форматировать исходные коды, а также в процессе их написания обнаружить наличие синтаксических ошибок и некоторых логических, таких как, например, нарушение правила владения[28].

В качестве среды разработки был выбран текстовый редактор VIM[29], поддерживающий возможность установки плагинов[30], в том числе для работы с RLS[27].

3.2 Реализация алгоритмов

В листинге 3.1 представлена структура объекта мышцы, а также реализация методов деформации и триангуляции. В листинге 3.2 представлена реализация алгоритмов компьютерной графики: z -буфера и Гуро.

Листинг 3.1: Реализация объекта мышцы.

```
1 use super::prelude::*;
2 use std::vec::Vec;
3
4 pub struct Muscle {
5     radiuses: Vec<f64>,
6     grow_mults: Vec<f64>,
7     dx: f64,
8     min_dx: f64,
9     max_dx: f64,
10 }
11
12 impl Muscle {
13     pub fn new(radiuses: Vec<f64>, grow_mults: Vec<f64>, len: f64) -> Self {
14         let dx = len / (radiuses.len() - 1) as f64;
15         Self {
16             radiuses,
17             grow_mults,
18             dx,
19             min_dx: dx * constants::MIN_PART,
20             max_dx: dx * constants::MAX_PART,
21         }
22     }
23
24     pub fn radiuses(&self) -> &[f64] {
25         &self.radiuses
26     }
27
28     fn get_angle(&self, i: usize) -> f64 {
29         let last = self.radiuses.len() - 1;
30         match i {
31             0 => {
32                 f64::atan((self.radiuses[1] - self.radiuses[0]) / self.dx)
33                     + std::f64::consts::PI / 2f64
34             }
35             val if val == last => {
36                 f64::atan((self.radiuses[last] - self.radiuses[last - 1]) / self.dx)
37                     + std::f64::consts::PI / 2f64
38             }
39         }
40     }
41 }
```

```

39     val => {
40         let a1 = (self.radiuses[val] - self.radiuses[val - 1]) / self.dx;
41         let a2 = (self.radiuses[val + 1] - self.radiuses[val]) / self.dx;
42         (f64::atan(a1) + f64::atan(a2)) / 2f64 + std::f64::consts::PI / 2f64
43     }
44 }
45 }
46
47 fn normal_ep(&self, i: usize) -> Point3d {
48     let angle = self.get_angle(i);
49     Point3d::new(
50         self.dx * i as f64 + f64::cos(angle),
51         self.radiuses[i] + f64::sin(angle),
52         0_f64,
53     )
54 }
55
56 fn find_intersections(&self, mut i1: usize, mut i2: usize) -> (Point3d, Point3d) {
57     if i1 > i2 {
58         std::mem::swap(&mut i1, &mut i2);
59     }
60     (
61         Point3d::new(self.dx * i1 as f64, self.radiuses[i1], 0_f64),
62         Point3d::new(self.dx * i2 as f64, self.radiuses[i2], 0_f64),
63     )
64 }
65
66 fn fill_pn_connectors(
67     &self,
68     points: &mut Vec<Vec<Point3d>>,
69     normal2points: &mut Vec<Vec<Point3d>>,
70 ) {
71     for i in 0..(self.radiuses.len() - 1) {
72         let (p1, p2) = self.find_intersections(i, i + 1);
73
74         let (mut new_points, mut new_norm2points) = rotate_intersections(
75             &[p1, p2],
76             &[self.normal_ep(i), self.normal_ep(i + 1)],
77             constants::MUSCLE_STEP,
78         );
79         cycle_extend(&mut new_points, 2);
80         cycle_extend(&mut new_norm2points, 2);
81
82         points.push(new_points);
83         normal2points.push(new_norm2points);
84     }
85 }
86

```

```

87 | fn fill_pn_spheres(
88 |     &self,
89 |     points: &mut Vec<Vec<Point3d>>,
90 |     normal2points: &mut Vec<Vec<Point3d>>,
91 | ) {
92 |     let index_arr = [0, self.radiuses.len() - 1];
93 |     for (center, rad) in index_arr
94 |         .iter()
95 |             .map(|&index| (self.dx * index as f64, self.radiuses[index]))
96 |             {
97 |                 add_uv_sphere(points, normal2points, center, rad);
98 |             }
99 | }
100 |
101 pub fn get_points_and_normals(&self) -> (Vec<Vec<Point3d>>, Vec<Vec<Point3d>>) {
102     let mut points = Vec::with_capacity(self.radiuses.len() *
103         (constants::SPHERE_PARTS) - 1);
104     let mut normal2points =
105         Vec::with_capacity(self.radiuses.len() * (constants::SPHERE_PARTS) - 1);
106     self.fill_pn_connectors(&mut points, &mut normal2points);
107     self.fill_pn_spheres(&mut points, &mut normal2points);
108
109     (points, normal2points)
110 }
111
112 // volume divided by pi
113 fn find_volume(&self) -> f64 {
114     let mut res = 0_f64;
115
116     for rads in self.radiuses.windows(2) {
117         let dy = rads[1] - rads[0];
118         res += dy * dy / 3_f64 + dy * rads[0] + rads[0] * rads[0];
119     }
120
121     res * self.dx
122 }
123
124 fn find_a(&self) -> f64 {
125     let mut res = 0_f64;
126
127     for mults in self.grow_mults.windows(2) {
128         res += mults[1] * mults[1] - 5_f64 * mults[0] * mults[1] + 7_f64 * mults[0] *
129             mults[0];
130     }
131
132     res / 3_f64
133 }
134

```

```

133     fn find_b(&self) -> f64 {
134         let mut res = 0_f64;
135
136         for (rads, mults) in self.radiuses.windows(2).zip(self.grow_mults.windows(2)) {
137             res += mults[1] * rads[0] + mults[0] * rads[1];
138         }
139
140         res
141     }
142
143     fn find_c(&self, g: f64) -> f64 {
144         let mut res = 0_f64;
145
146         for rads in self.radiuses.windows(2) {
147             res += 1_f64 / 3_f64 * f64::powi(rads[1] - rads[0], 2) + rads[0] * rads[1];
148         }
149
150         res - g
151     }
152
153     fn update_radiuses(&mut self, dy: f64) {
154         for (rad, mult) in self.radiuses.iter_mut().zip(self.grow_mults.iter()) {
155             *rad += mult * dy;
156         }
157     }
158
159     pub fn deform(&mut self, diff: f64) {
160         let new_dx = self.dx + diff / (self.radiuses.len() - 1) as f64;
161         if new_dx < self.min_dx || new_dx > self.max_dx {
162             return;
163         }
164
165         let g2 = self.find_volume() / new_dx;
166
167         let a = self.find_a();
168         let b = self.find_b();
169         let c = self.find_c(g2);
170
171         let dy = solve_quad_eq(a, b, c);
172         if let Some(dy) = dy.1 {
173             0_f64,

```

Листинг 3.2: Реализация алгоритмов компьютерной графики.

```

1 use super::prelude::*;
2
3 static mut Z_BUFFER: [[f64; constants::WIDTH]; constants::HEIGHT] =
4     [[f64::MIN; constants::WIDTH]; constants::HEIGHT];
5 static mut COLOR_BUFFER: [[u32; constants::WIDTH]; constants::HEIGHT] =

```

```

6     [[constants::DEFAULT_COLOR; constants::WIDTH]; constants::HEIGHT];
7
8 // INPUT: points with normals, transformation matrix, light_source, color of input
9 // figure.
10 // RESULT: flushes all visible parts of transformed figure in internal COLOR_BUFFER.
11 pub unsafe fn transform_and_add(
12     points_groups, normals_groups): &(Vec<Vec<Point3d>>, Vec<Vec<Point3d>>),
13     matrix: &Matrix4,
14     light_source: Point3d,
15     color: u32,
16 ) {
17     // for every triangulated group of input figure:
18     for (points, normals) in points_groups.iter().zip(normals_groups.iter()) {
19         let (p1, p2) = (
20             transform_and_normalize(points[0], normals[0], matrix),
21             transform_and_normalize(points[1], normals[1], matrix),
22         );
23         let mut current_window = [p1, p2, (Point3d::default(), Vec3d::default())];
24
25         // for every triangle (3 points + 3 normal points):
26         for (change_index, (&new_point, &new_normal)) in (2..)
27             .map(|elem| elem % 3)
28             .zip(points.iter().skip(2).zip(normals.iter().skip(2)))
29         {
30             // transform new point
31             current_window[change_index] = transform_and_normalize(new_point, new_normal,
32                                         matrix);
33             // check if any part of triangle visible and triangle isn't rotated to
34             // background
35             if check_pos_all(current_window.iter().map(|elem| elem.0))
36                 && check_normals_all(current_window.iter().map(|elem| elem.1))
37             {
38                 // divide triangle on points array and normal points array
39                 let points = [
40                     current_window[0].0,
41                     current_window[1].0,
42                     current_window[2].0,
43                 ];
44                 let normals = [
45                     current_window[0].1,
46                     current_window[1].1,
47                     current_window[2].1,
48                 ];
49                 // add transformed triangle polygon to buffer
50                 add_polygon(points, normals, &light_source, color);
51             }
52         }
53     }
54 }
```

```

51 }
52
53 unsafe fn add_polygon(
54     points: [Point3d; 3],
55     mut normals: [Vec3d; 3],
56     light_source: &Point3d,
57     color: u32,
58 ) {
59     // cast Y coordinate to integer (coordinates of the screen are integers)
60     let mut int_points = [
61         IntYPoint3d::from(points[0]),
62         IntYPoint3d::from(points[1]),
63         IntYPoint3d::from(points[2]),
64     ];
65     // sort points by Y coordinate
66     sort_by_y(&mut int_points, &mut normals);
67     // find brightnesses for all vertexes for furhter processing by Gouraud algorithm
68     let brightnesses = find_brightnesses(points, normals, light_source);
69     // divide triangle on 2 pairs of sections, which make up 2 triangles with
70     // parallel to X axis edge
71     let sections = divide_on_sections(int_points, brightnesses);
72     process_sections(sections, color);
73 }
74
75 // application of Gouraud and Z-buffer algorithms for 2 processed triangles
76 unsafe fn process_sections(mut sections: [Section; 4], color: u32) {
77     for pair in sections.chunks_mut(2) {
78         if pair[0].x_start > pair[1].x_start {
79             continue;
80         }
81
82         if pair[0].y_start < 0 {
83             let diff = (-pair[0].y_start) as f64;
84             for sec in pair.iter_mut() {
85                 sec.x_start += diff * sec.x_step;
86                 sec.br_start += diff * sec.br_step;
87                 sec.z_start += diff * sec.z_step;
88             }
89             pair[0].y_start = 0;
90         }
91
92         for y in (pair[0].y_start..=pair[0].y_end)
93             .filter(|&elem| elem < constants::HEIGHT as i16)
94             .map(|y| y as usize)
95         {
96             let x_from = f64::round(pair[0].x_start) as usize;
97             let x_to = f64::round(pair[1].x_start) as usize;
98             let diff_x = (x_to - x_from) as f64;

```

```

99
100    let mut br = pair[0].br_start;
101    let br_diff = (pair[1].br_start - br) / diff_x;
102    let mut z = pair[0].z_start;
103    let z_diff = (pair[1].z_start - z) / diff_x;
104
105    for x in (x_from..=x_to).filter(|&x| x < constants::WIDTH) {
106        if z > Z_BUFFER[y][x] {
107            Z_BUFFER[y][x] = z;
108            put_color(x, y, color, br);
109        }
110
111        br += br_diff;
112        z += z_diff;
113    }
114
115    for sec in pair.iter_mut() {
116        sec.x_start += sec.x_step;
117        sec.br_start += sec.br_step;
118        sec.z_start += sec.z_step;
119    }
120}
121}
122}

```

Выход

В данном разделе были рассмотрены средства, с помощью которых было реализовано ПО, а также представлены листинги кода с реализацией объекта мышцы и алгоритмов компьютерной графики.

4 Исследовательская часть

В данном разделе будут приведены примеры работы разработанного программного обеспечения, а также будет поставлен эксперимент, в котором будут сравнены геометрические характеристики разработанной модели с геометрическими характеристиками бицепса реального человека.

4.1 Результаты работы программного обеспечения

В листинге 4.1 описана конфигурация модели мышцы с рисунков 4.1 и 4.2, на рисунках 4.1 и 4.2 представлен пример модели мышцы в полностью растянутом и сокращённом состояниях соответственно, на рисунке 4.3 представлено окно панели управления.

Листинг 4.1: Конфигурация для мышцы из примера.

```
1 ---
2 muscle_config:
3   radiuses: [5.0, 5.0, 10.0, 15.0, 18.0, 18.0, 15.0, 10.0, 5.0, 5.0]
4   grow_mults: [0.0, 1.0, 1.0, 2.0, 2.0, 2.0, 2.0, 1.0, 1.0, 0.0]
5   len: 250.0
6
7 carcass_config:
8   data: [[30.0, 230.0], [30.0, 120.0]]
9   thickness: 5.0
```

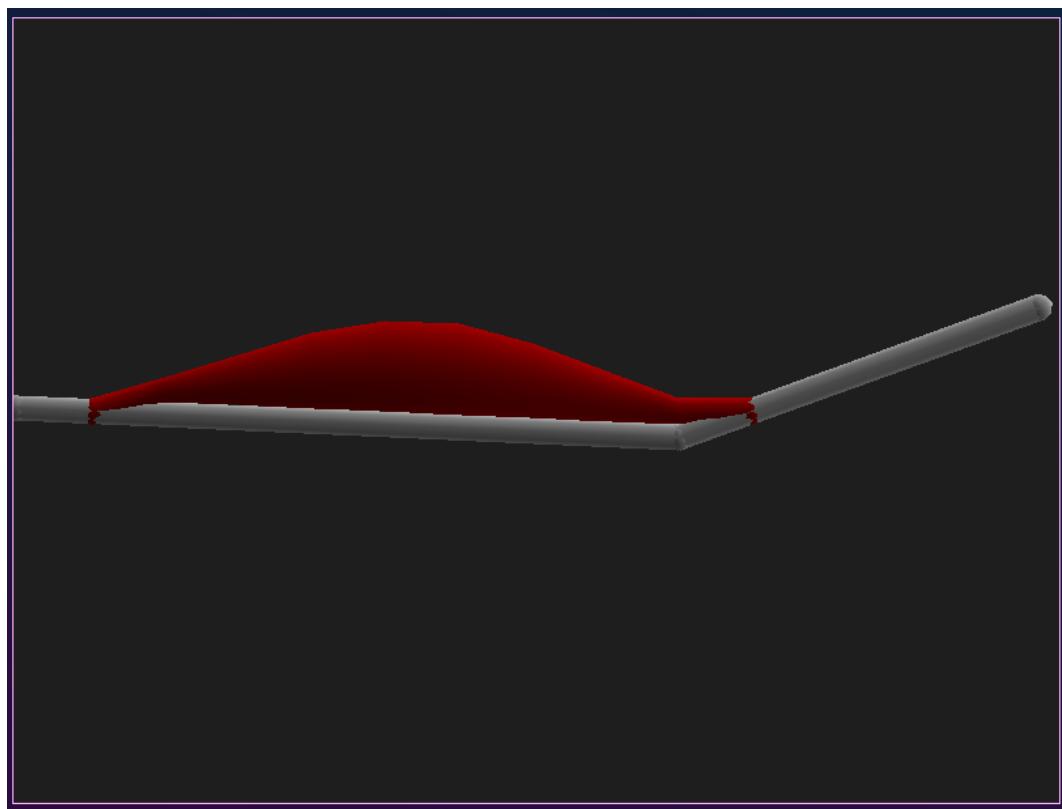


Рис. 4.1: Пример модели мышцы в полностью растянутом состоянии.

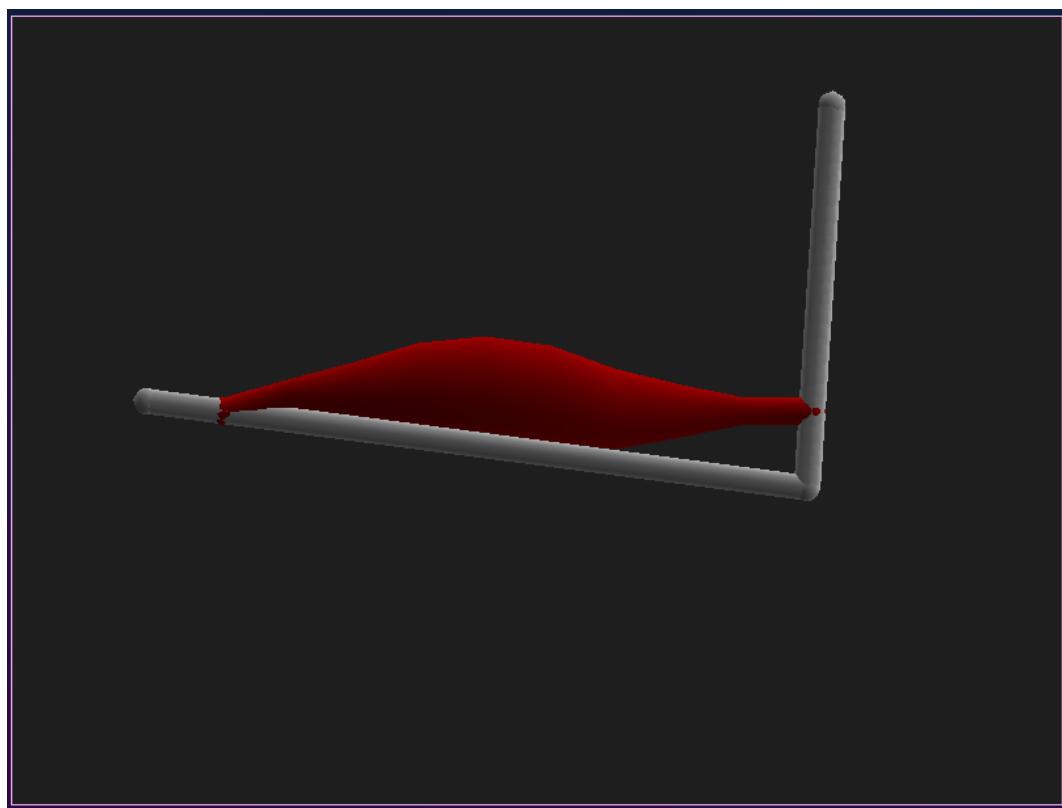


Рис. 4.2: Пример модели мышцы в полностью сокращённом состоянии.

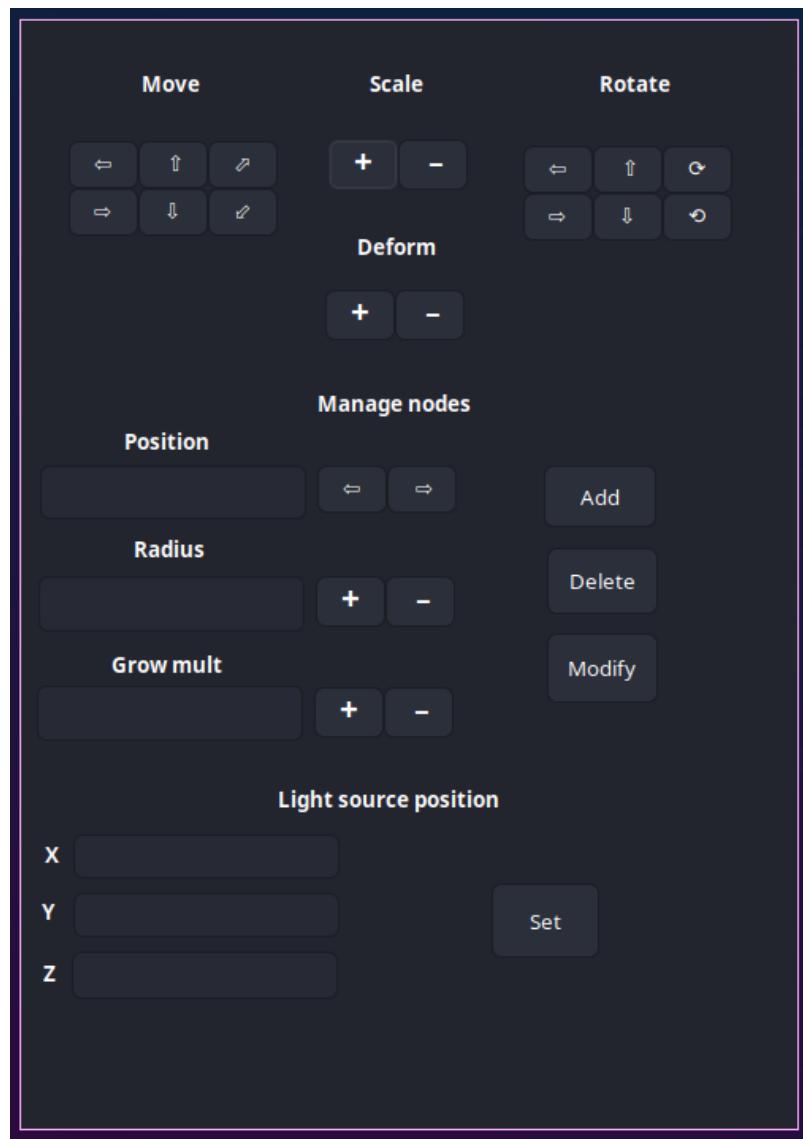


Рис. 4.3: Окно панели управления.

4.2 Постановка эксперимента

4.2.1 Цель эксперимента

Цель эксперимента - сравнение геометрических характеристик (радиусов узлов) разработанной модели и реального человеческого бицепса в различных положениях мышцы.

4.2.2 Данные реального бицепса

Данные реального бицепса были взяты из проекта OpenArm 2.0 [31]. Данный проект предоставляет необработанные и сегментированные ультразвуковые сечения плечевой части руки (в том числе бицепса) [32]. Анализ данных может быть произведен с помощью ПО ITK-SNAP [33]. На рисунке 4.4 приведён пример необработанных данных, а на рисунке 4.5 - пример сегментированных данных.

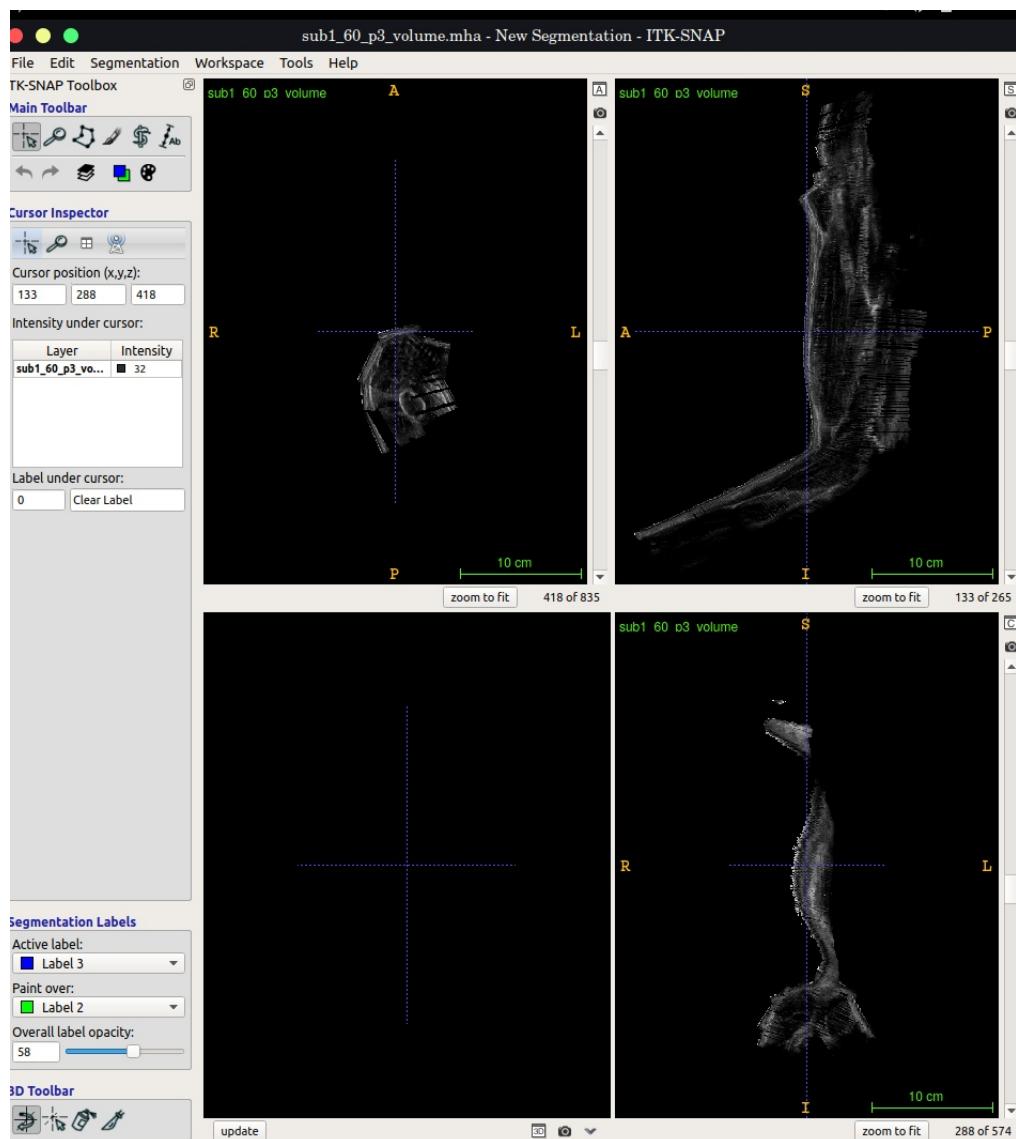


Рис. 4.4: Необработанные данные.

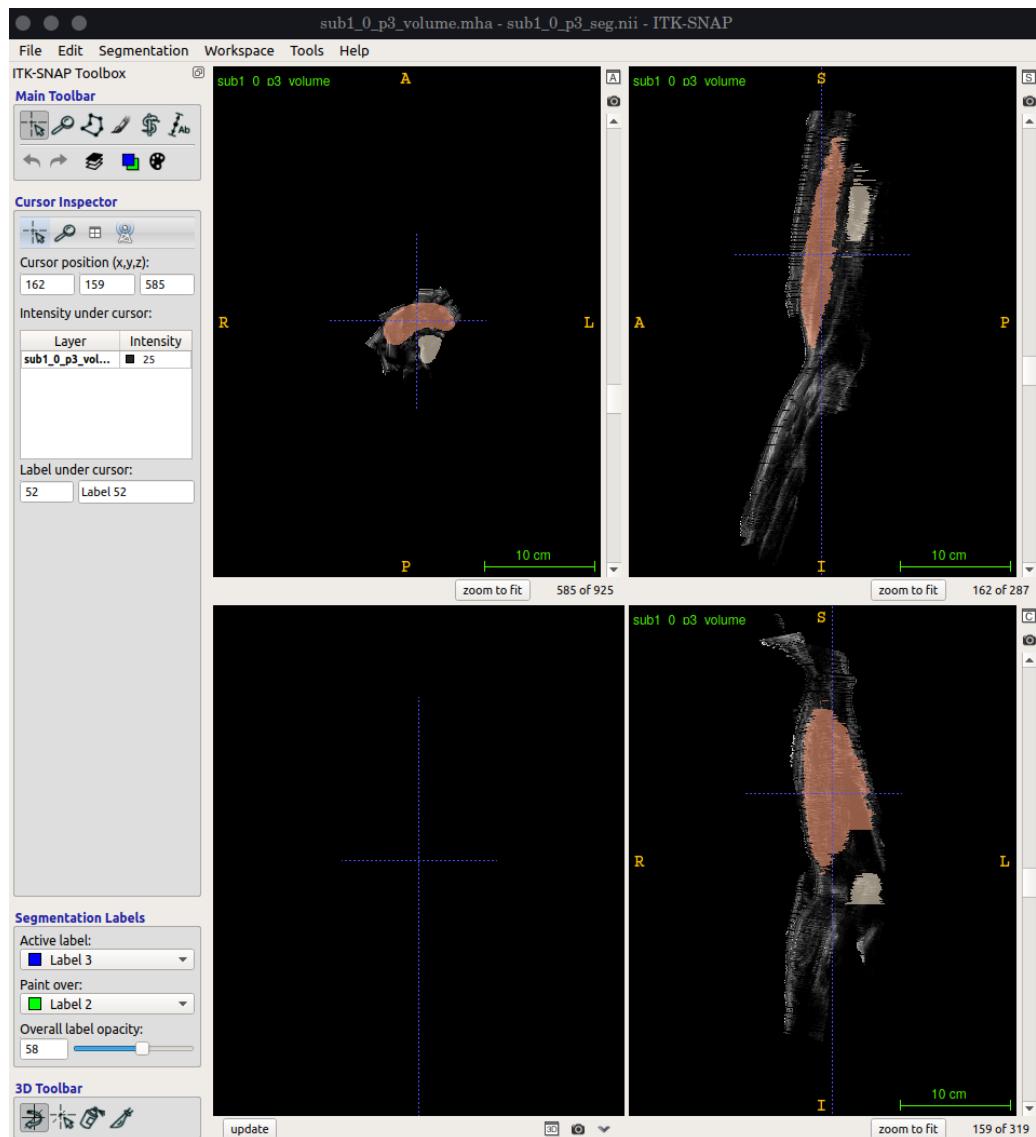


Рис. 4.5: Сегментированные данные.

4.2.3 Замеры реального бицепса

На рисунках 4.6 – 4.9 приведены замеры для 9 узлов мышцы в 4 положениях: под углом 0° , 30° , 60° и 90° . Данные замеров сведены в таблицу 4.1.

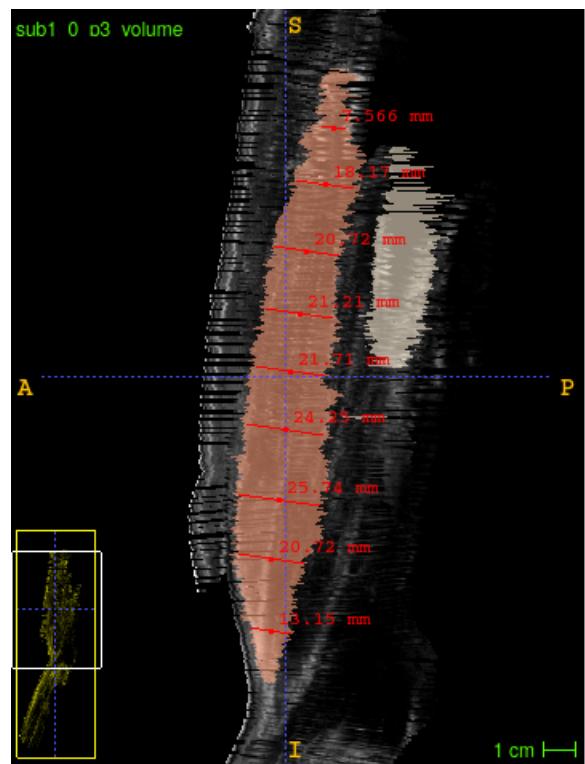


Рис. 4.6: Замеры для мышцы в положении 0° .

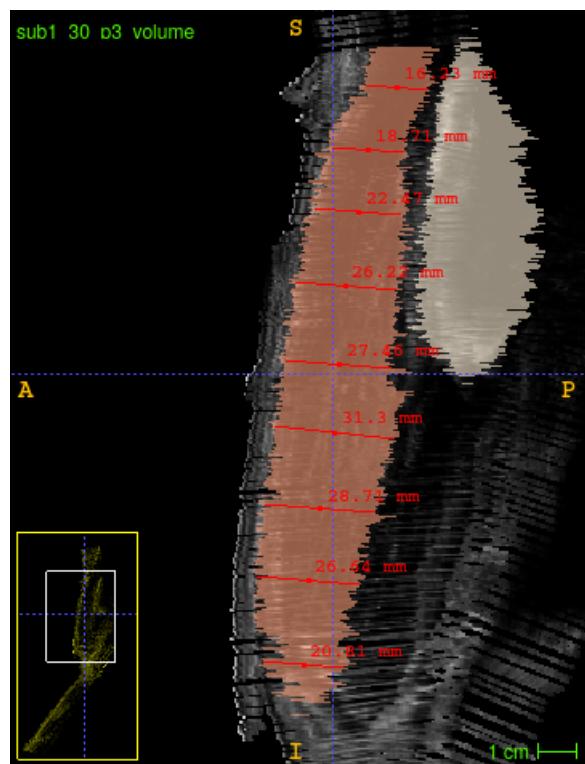


Рис. 4.7: Замеры для мышцы в положении 30° .

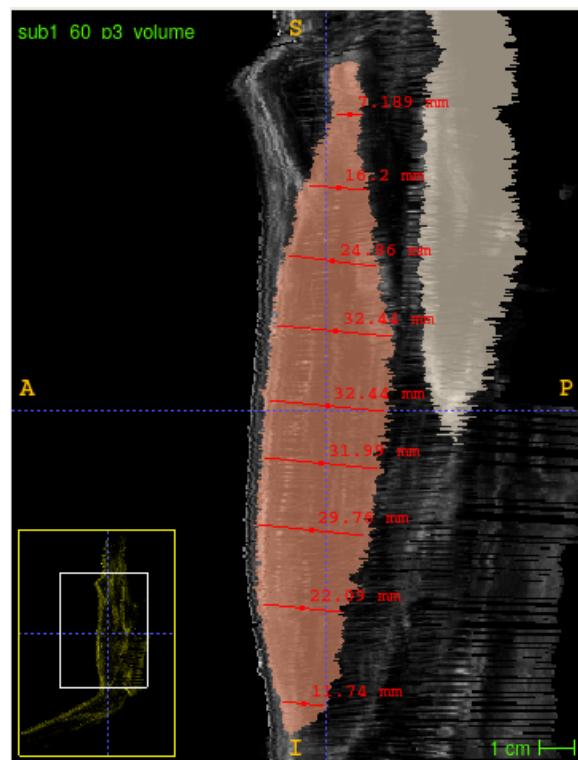


Рис. 4.8: Замеры для мышцы в положении 60°.

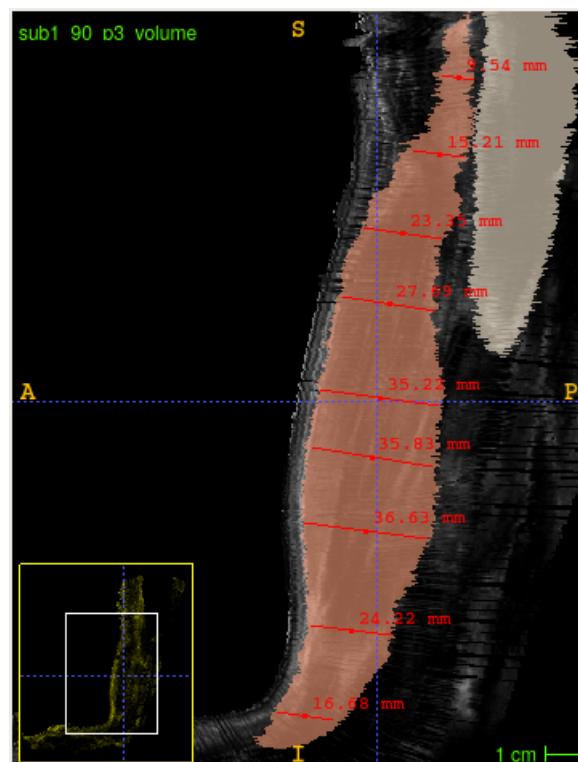


Рис. 4.9: Замеры для мышцы в положении 90°.

Номер узла	0°	30°	60°	90°
1	7.6	16.2	7.8	9.5
2	18.2	18.7	16.2	15.2
3	20.7	22.5	24.9	23.4
4	21.2	26.2	32.4	27.7
5	21.7	27.5	32.4	35.2
6	24.3	31.3	32.0	35.8
7	25.7	28.7	29.8	36.6
8	20.7	26.6	22.0	24.2
9	13.2	20.8	12.7	16.7

Таблица 4.1: Радиусы узлов при различном угле наклона в локте

4.2.4 Замеры разработанной модели

Для измерения геометрических характеристик модели необходимо сперва ее сконфигурировать. Конфигурация будет выполняться исходя из данных реальной мышцы: начальные значения узлов модели пропорциональны начальным значениям узлов реальной мышцы, коэффициенты роста пропорциональны приросту реальной мышцы при переходе из начального в конечное состояние.

Таким образом конфигурация будет иметь вид, описанный в листинге 4.2.

Листинг 4.2: Конфигурация модели, параметры которой пропорциональны реальной мышце.

```

1 ---
2 muscle_config:
3   radiiuses: [5.0, 7.6, 18.2, 20.7, 21.2, 21.7, 24.3, 25.7, 20.7, 13.2, 5.0]
4   grow_mults: [0.0, 1.9, -3.0, 2.7, 5.5, 13.5, 11.5, 10.9, 3.5, 3.5, 0.0]
5   len: 250.0
6
7 carcass_config:
8   data: [[30.0, 220.0], [30.0, 120.0]]
9   thickness: 5.0

```

Вид модели представлен на рисунках 4.10 – 4.11. Результаты представлены в таблице 4.2.

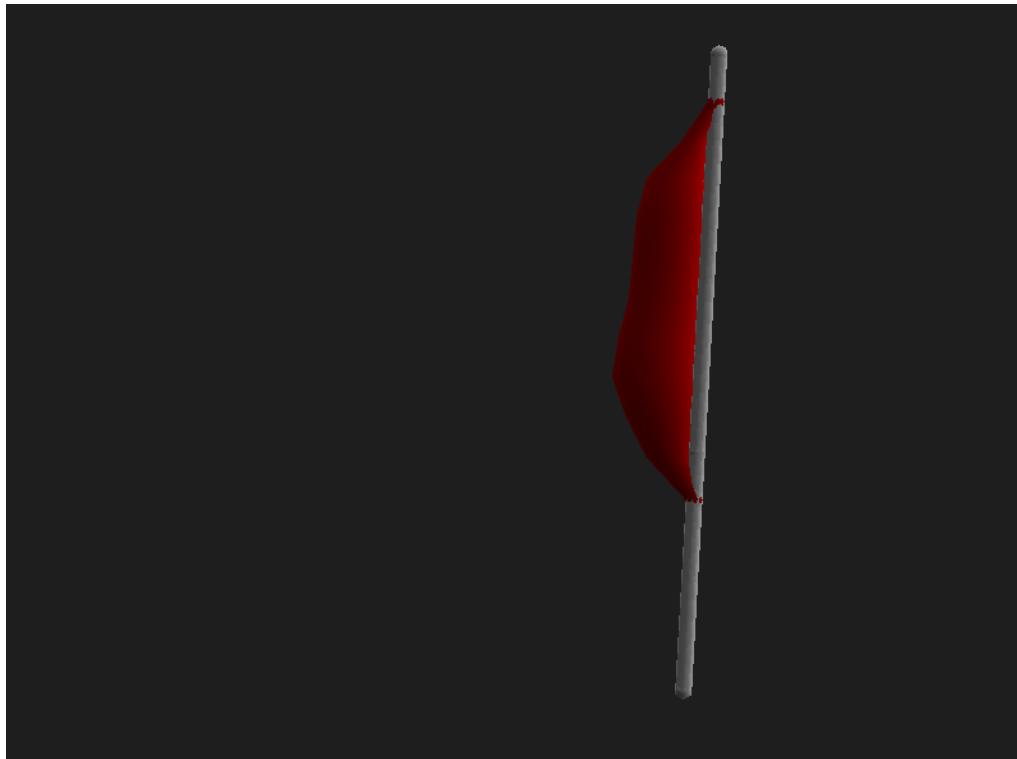


Рис. 4.10: Модель в положении 0° .

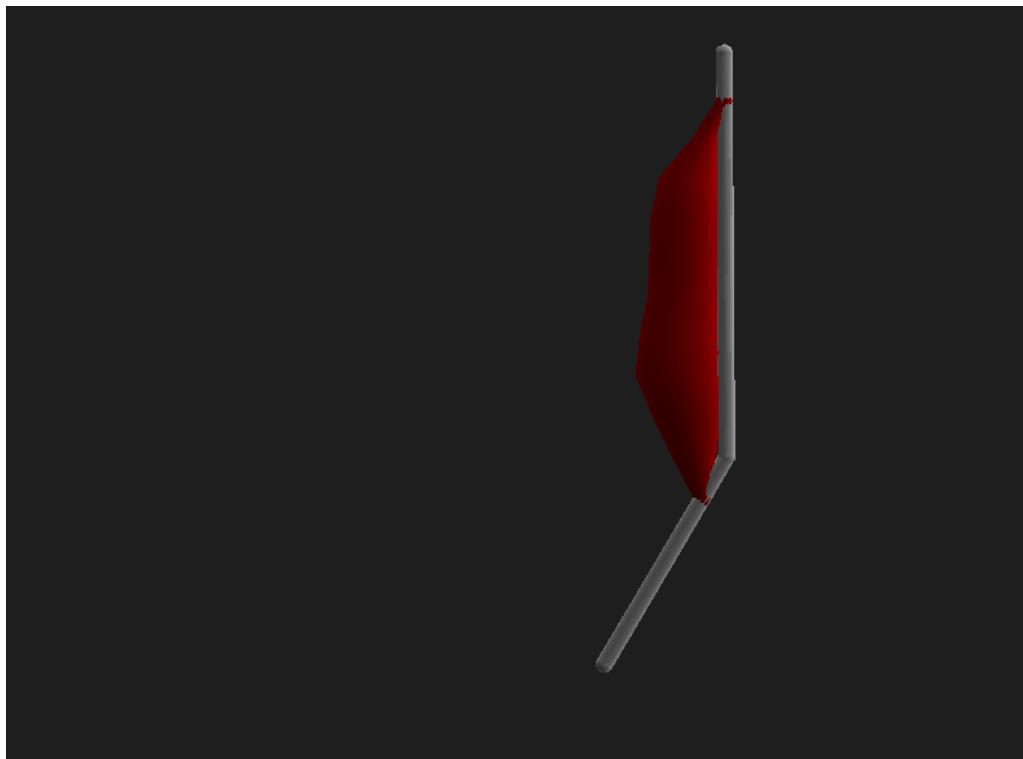


Рис. 4.11: Модель в положении 30° .

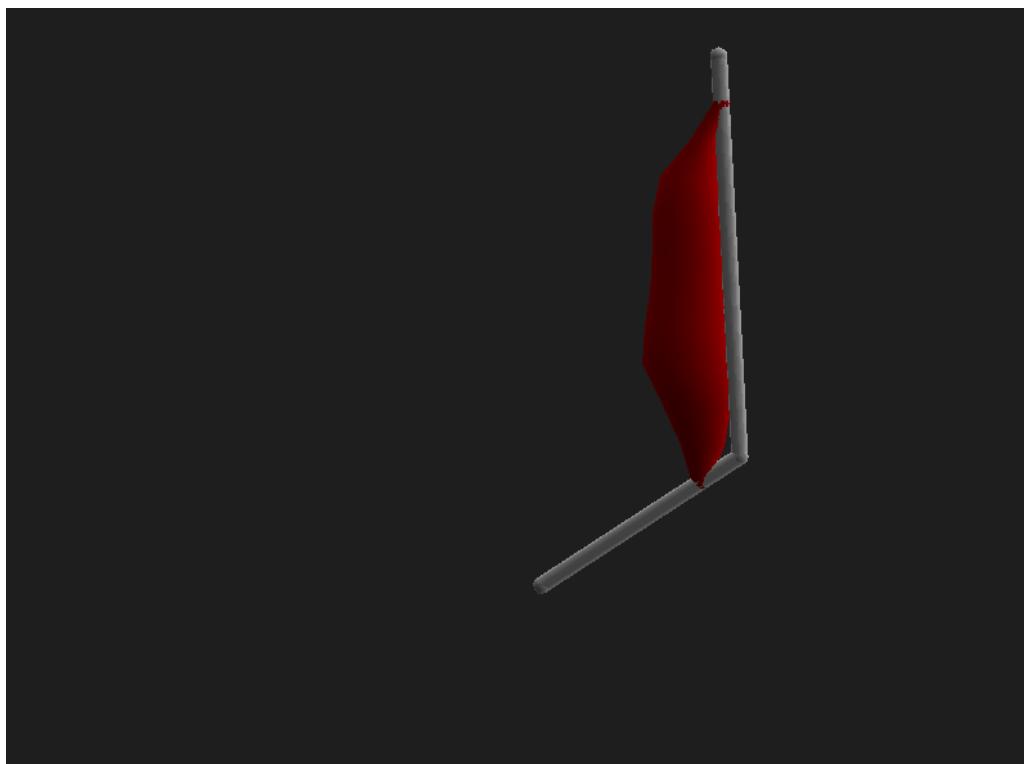


Рис. 4.12: Модель в положении 60° .

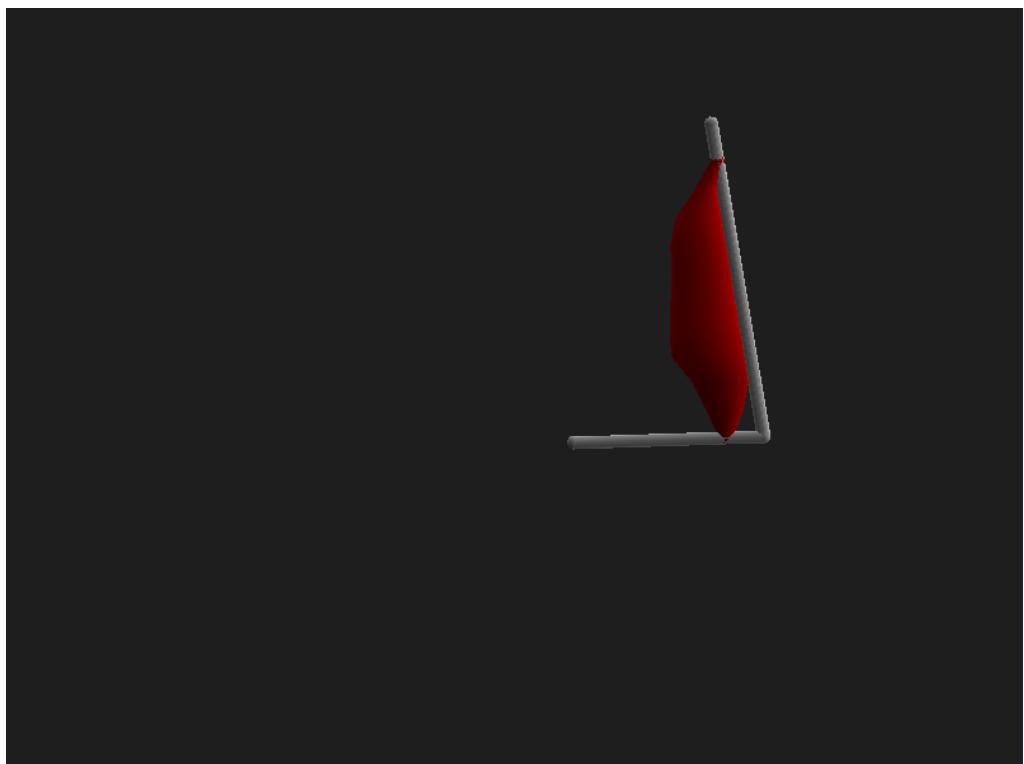


Рис. 4.13: Модель в положении 90° .

Номер узла	0°	30°	60°	90°
1	7.6	7.7	7.8	8.0
2	18.2	18.1	17.9	17.6
3	20.7	20.8	21.0	21.2
4	21.2	21.3	21.7	22.3
5	21.7	22.1	23.0	24.4
6	24.3	24.6	25.4	26.6
7	25.7	26.0	26.7	27.9
8	20.7	20.8	21.0	21.4
9	13.2	13.3	13.5	13.9

Таблица 4.2: Радиусы узлов модели при различном угле наклона в локте

4.2.5 Сравнение замеров

В таблице 4.3 приведены приросты для всех узлов для реального бицепса (РБ) и модели относительно при переходе из состояния 0° в состояние 90° .

Номер узла	РБ	модель
1	1.9	0.4
2	-3.0	-0.6
3	2.7	0.5
4	6.5	1.1
5	13.5	2.7
6	11.5	2.3
7	10.9	2.2
8	3.5	0.7
9	3.5	0.7

Таблица 4.3: Радиусы узлов при различном угле наклона в локте

Из таблицы видно, что реализованная модель может сохранять задаваемые пропорции, но ввиду того, что модель сохраняет объем, все приращения получились в 5-6 раз меньше.

Из таблиц 4.1 и 4.2 можно заметить, что приращения модели при сокращении мышцы строго положительные, в то время как с реальным бицепсом

могут быть как положительные, так и отрицательные.

Вывод

В данном разделе были рассмотрены примеры работы программного обеспечения, а также были вычислены и сравнены радиусы узлов реального бицепса и сделанной модели. В результате сравнения были получены следующие результаты:

- Модель способна сохранять пропорции приращения, заданные реальной мышцой.
- Модель ведёт себя монотонно, то есть, например, при сокращении радиусы не перестают расти (или убывать, если коэффициент роста меньше 0), в то время как узел в реальном бицепсе при сокращении может как увеличиться, так и уменьшиться.
- Ввиду ограничения, связанного с постоянством объема, радиусы узлов модели получают прирост в 5-6 раз меньший, нежели узлы реальной мышцы.

Заключение

В ходе курсового проекта было разработано программное обеспечение, которое предоставляет возможности загрузки параметров геометрической модели бицепса на узлах из конфигурационного файла, изменения этих параметров в интерактивном режиме, управления состоянием модели (сокращение и растяжение), а также положением (вращение, перемещение и масштабирование). В процессе выполнения данной работы были выполнены следующие задачи:

- формально описана структура моделей мышцы и каркаса;
- рассчитаны формулы деформации геометрической модели с сохранением объема;
- выбраны алгоритмы трехмерной графики, визуализирующие модель;
- реализованы алгоритмы для визуализации описанных выше объектов.

В процессе исследовательской работы было выяснено, что полученная геометрическая модель имеет ограничения, не позволяющие полностью воспроизвести поведение реального бицепса, однако реализованная модель способна получить схожее поведение: сохранить пропорции с уменьшенной в 5-6 раз амплитудой роста/уменьшения (из-за ограничения, связанного с постоянством объема), а также с достаточной степенью визуализации

Литература

- [1] Modeling and Simulation of Skelatal Muscle for Computer Graphics: A Survey. Режим доступа: <http://www.cs.toronto.edu/pub/reports/na/MuscleSimulationSurvey.pdf> (дата обращения: 07.07.2020). 2011.
- [2] Wilhelms J. Animals with anatomy // IEEE Computer Graphics and Applications, vol. 17, no. 3. 1997. c. 22–30.
- [3] J. E. Chadwick D. R. Haumann, Parent R. E. Layered construction for deformable animated characters // SIGGRAPH Computer Graphics. 1989. c. 243–252.
- [4] Komatsu K. Human skin model capable of natural shape variation // The Visual Computer, vol. 3, no. 5. 1988. c. 265–271.
- [5] Blinn J. F. A generalization of algebraic surface drawing // ACM Transactions on Graphics, vol. 1, no. 3. 1982. c. 235–256.
- [6] Bloomenthal J., Shoemake K. Convolution surfaces // SIGGRAPH Computer Graphics. 1991. c. 251–256.
- [7] Nedel L. P., Thalmann D. Real time muscle deformations using massspring systems, // Proceedings of Computer Graphics International, IEEE. 1998.
- [8] Strang G., Fix G. An Analysis of the Finite Element Method. WellesleyCambridge, 2nd Edition. 2008.
- [9] A. Nealen M. Mueller, Carlson M. Physically based deformable models in computer graphics // Computer Graphics Forum, vol. 25, no. 4. 2006. c. 809–836.
- [10] LeVeque R. J. Finite Volume Methods for Hyperbolic Problems // Cambridge University Press. 2002.
- [11] J. Teran S. Blemker, Fedkiw R. Finite volume methods for the simulation of skeletal muscle // SCA '03: Proceedings of the 2003 ACM

SIGGRAPH/Eurographics Symposium on Computer Animation. 2003.
c. 68–74.

- [12] VIPER: Volume Invariant Position-based Elastic Rods. Режим доступа: <https://media.contentapi.ea.com/content/dam/ea/seed/presentations/viper-volume-invariant-position-based-elastic-rods.pdf> (дата обращения: 04.07.2020).
- [13] Position-based elastic rods. Режим доступа: <https://dl.acm.org/doi/10.5555/2849517.2849522> (дата обращения: 03.07.2020).
- [14] K. Min S. Baek, Park C. Anatomically-based modeling and animation of human upper limbs // Proceedings of International Conference on Human Modeling and Animation. 2000.
- [15] X-Muscle System. Режим доступа: <https://blendermarket.com/products/x-muscle-system> (дата обращения: 14.07.2020).
- [16] Blender. Режим доступа: <https://www.blender.org/> (дата обращения: 14.07.2020).
- [17] Трассировка лучей в реальном времени. Режим доступа: <https://www.ixbt.com/3dv/directx-raytracing.html> (дата обращения: 13.08.2020).
- [18] Cost Analysis of a Ray Tracing Algorithm. Режим доступа: <https://www.graphics.cornell.edu/~bjw/mca.pdf> (дата обращения: 05.09.2020).
- [19] Алгоритм Z-буфера. Режим доступа: <http://compgraph.tpu.ru/zbuffer.htm> (дата обращения: 21.08.2020).
- [20] Алгоритм Робертса. Режим доступа: <http://compgraph.tpu.ru/roberts.htm> (дата обращения: 14.05.2020).
- [21] Д. Роджерс. Алгоритмические основы машинной графики. 1989.

- [22] Модели затенения. Плоская модель. Затенение по Гуро и Фонгу. Режим доступа: <https://compgraphics.info/3D/lighting/shading-model.php> (дата обращения: 18.06.2020).
- [23] Rust Programming Language [Электронный ресурс]. Режим доступа: <https://doc.rust-lang.org/std/index.html> (дата обращения: 20.07.2020). 2017.
- [24] The Cargo Book. Режим доступа: <https://doc.rust-lang.org/cargo/> (дата обращения: 21.07.2020).
- [25] Документация по ЯП Rust: бенчмарки [Электронный ресурс]. Режим доступа: <https://doc.rust-lang.org/1.7.0/book/benchmark-tests.html> (дата обращения: 21.09.2020).
- [26] Gtk-rs. Режим доступа: <https://gtk-rs.org/> (дата обращения: 21.07.2020).
- [27] Rust Language Server (RLS). Режим доступа: <https://github.com/rust-lang/rls> (дата обращения: 21.07.2020).
- [28] Rustbook. What is Ownership? Режим доступа: <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html> (дата обращения: 20.07.2020).
- [29] VIM the editor. Режим доступа: <https://www.vim.org/> (дата обращения: 20.07.2020).
- [30] VimAwesome. Режим доступа: <https://vimawesome.com/> (дата обращения: 20.07.2020).
- [31] OpenArm 2.0. Режим доступа: <https://berkeley.app.box.com/v/openarm-2p0-data/file/445757315449> (дата обращения: 11.12.2020).
- [32] OpenArm 1.0. Режим доступа: <https://berkeley.app.box.com/s/mj9yano1umbtbi1aj013b7y2pm864wnb/file/692415066210> (дата обращения: 11.12.2020).
- [33] ITK-SNAP. Режим доступа: <http://www.itksnap.org/pmwiki/pmwiki.php> (дата обращения: 11.12.2020).