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 _____ **П. Г. Пересторонин**
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ **А. А. Оленев**
(Подпись, дата) (И.О.Фамилия)

2020 г.

Оглавление

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 средах, 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 они менее трудоёмкие в сравнении с физическими методами, а также более точны в следствие того, что модель точно задается и описывается функцией без аппроксимации и интерполяции[1]. Чаще всего 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, данные методы зачастую 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 динамеческой системы с точки зрения физики.

Физические методы решения задачи визуализации мышц нередко используются совместно с геометрическими, однако основная идея 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, представленная на конференции SIGGRAPH 2019 [14]. 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. 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[15].

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 1.1.2.

Ещё одним популярным продуктом в сфере визуализации мышц является X-Muscle System[16] - расширение для программы Blender[17], которое позволяет создавать мышцы в интерактивном режиме. В данном случае мышца состоит из так называемых *канонических костей*, которые имеют 3 параметра: позиция, направление и ориентация. На рисунках 1.3 и 1.4 показаны примеры мышц, на основе модели X-Muscle System.

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

16 ms / 62 fps

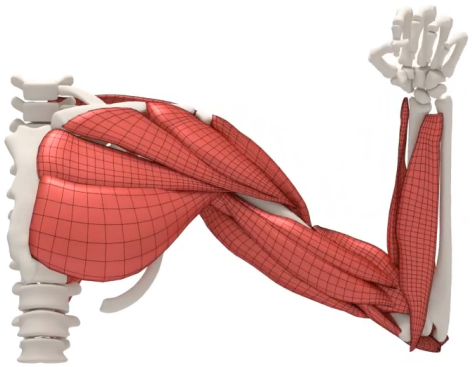

Рис. 1.2: Мышцы на основе VIPER, вид с сеткой.

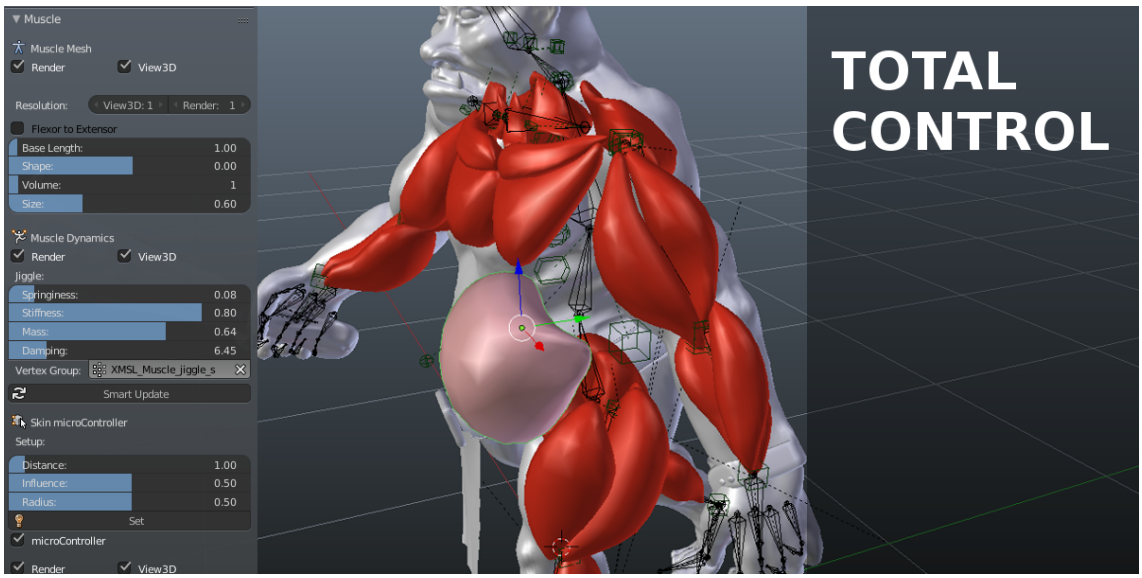

Рис. 1.3: Мышцы X-Muscle System. Проектирование.

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

1.3 Модели мышцы и каркаса

1.3.1 Модель мышцы

Бицепс является веретенообразной мышцой, что позволяет в полной мере использовать геометрические методы. К тому же программное обеспечение, получаемое в результате данной работы предусматривает статическую систему, что позволяет использовать все преимущства геометрических ме-

тодов и не иметь проблем их с недостатками.

Ввиду сложности прямого описания поверхности мышцы некоторым уравнением, а также сложности конфигурирования мышцы через параметризованные функции, модель мышцы в данной работе будет представляться с помощью группы усечённых конусов, полученной путем вращения группы отрезков, находящихся между узлами модели. Очевидно, что 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 функции любого вида с произвольной 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.

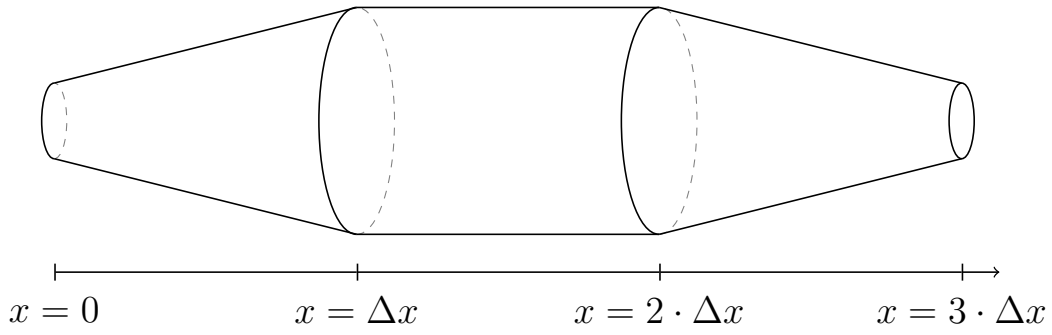

Рис. 1.5: Модель на 4 узлах.

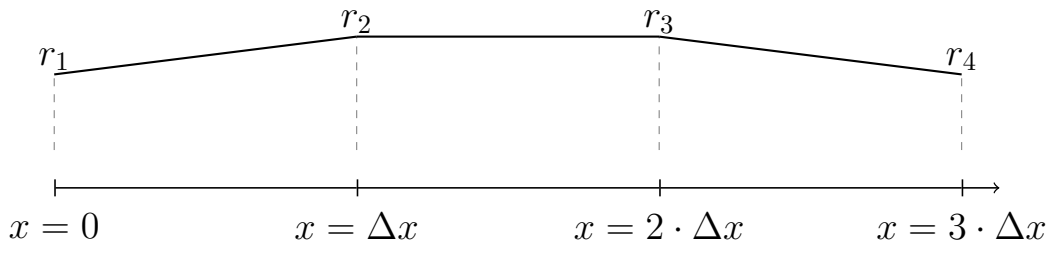

Рис. 1.6: Фигура, путём вращения которой получаются усечённые конусы, составляющие мышцу.

Для упрощения работы с моделью расстояние между узлами было выбрано постоянным. Данный факт не вносит никаких ограничений в представление модели, так как, описывая некоторый узел пропорционально со-

седям этого узла, получается представление, в котором визуально расстояние между этими двумя соседними узлами будет удвоено. Сделав обратное действие можно вводить новые узлы, уменьшая при этом расстояние между всеми узлами и увеличивая тем самым точность.

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

Однако для исчерпывающего описания свойств мышцы этого мало. Такой набор не хранит информации о поведении узлов при деформациях, поэтому также для каждого узла требуется добавить коэффициент приращения, который будет означать относительный прирост данного узла.

- 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 в данном случае проще всего задавать в виде прямой, заданной уравнением $y = ax + b$, которая ограничена по x: $x \in [0; \Delta x]$, где Δx - расстояние между узлами. Рассматривать для i-ого и $(i+1)$-ого узлов отрезок $[0; \Delta x]$ гораздо проще, чем отрезок $[(i-1) \cdot \Delta x; i \cdot \Delta x]$, в связи с чем во всех последующих вычислениях подразумевается, что пара соседних узлов с индексами i и $(i+1)$ сдвигается на $(i-1) \cdot \Delta x$ влево (можно заметить, что во всех последующих расчетах нас будет интересовать исключительно величина Δx, поэтому все последующие расчёты справделивы и для исходного положения составных частей модели). Площадь усечённого конуса в

таком случае можно рассчитывать по формуле (1.1):

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

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

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

Из чего следует (1.3):

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

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):

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

$$b = y_i \tag{1.5}$$

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):

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

После выноса за знак суммы Δx получается итоговая формула вычис-

ления объема (1.7):

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

Которую можно представить как функцию от расстояния между узлами Δx и массива радиусов узлов (1.8):

$$V(\Delta x, R) = \Delta x \cdot F(R), \tag{1.8}$$

где $F(R)$ - функция (1.9):

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

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

$$L = (n - 1) \cdot \Delta x \tag{1.10}$$

Откуда можно вывести зависимость расстояния между узлами от длины (1.11):

$$\Delta x = \frac{L}{n - 1} \tag{1.11}$$

При деформации модели (сокращении или растяжении) объем должен оставаться постоянным. Если посмотреть на выражение (1.8) становится очевидным, что для равенства $V = V'$, где V - объем до деформации, а V' - после, должно выполнятся равенство (1.12):

$$\Delta x \cdot F(R) = \Delta x' \cdot F(R') \tag{1.12}$$

где R' - массив радиусов узлов после деформации, $\Delta x'$, учитывая (1.11),

находится из (1.13):

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

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

Отсюда получается, что $F(R')$ можно найти, как (1.14):

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

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

$$r'_i = r_i + m_i \cdot \delta y \tag{1.15}$$

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

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

$$F(R') = \sum_{i=1}^{n-1}(\frac{((r_{i+1} + m_{i+1} \cdot \delta y) - (r_i + m_i \cdot \delta y))^2}{3} + \\ +2(r_i + m_i \cdot \delta y) \cdot (r_{i+1} + m_{i+1} \cdot \delta y - r_i + mi \cdot \delta y) + (r_i + m_i \cdot \delta y)^2) \tag{1.16}$$

После приведения подобных слагаемых относительно δy получается (1.17):

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

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

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

$$B = \sum_{i=1}^{n-1} (m_{i+1} r_i + m_i r_{i+1}) \tag{1.19}$$

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

Квадратное уравнение (1.17) будет иметь решения (1.21):

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

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

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

Таким образом при деформации модели на величину δx новые значения радиусов узлов будут иметь 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 2 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 есть параметры, которые от нее не зависят - точки крепления. Для представления каркаса будет достаточно следующих параметров:

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

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

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

Откуда можно выразить угол α как (1.24):

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

1.4 Анализ алгоритмов удаления невидимых линий и поверхностей

При выборе алгоритма удаления невидимых линий и поверхностей нужно учесть 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.

Идея: для определения цвета пиксела 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:

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

Минусы:

- трудоёмкие вычисления;
- высокая сложность реализации версии, учитывающей все физические явления.

Вывод: так как в поставленной задаче в первую очередь стоит требование быстроты работы алгоритма, а также ввиду того, что модели, использующиеся в программе, не являются прозрачными и имеют преимущественно диффузное отражение, данный алгоритм не подходит в рамках данной работы.

1.4.2 Алгоритм, использующий Z-буфер

Алгоритм работает в простанстве изображения.

Идея: имеется 2 буфера - буфер кадра, который используется для запоминания цвета каждого пиксела изображения, а также z-буфер - отдельный буфер глубины, используемый для запоминания координаты z (глубины) каждого видимого пиксела изображения. В процессе работы глубина или значение z каждого нового пиксела, который нужно занести в буфер кадра, сравнивается с глубиной того пиксела, который уже занесен в z-буфер. Если это сравнение показывает, что новый пиксел расположен выше пиксела, находящегося в буфере кадра ($z > 0$), то новый пиксел заносится в цвет рассматриваемого пиксела 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 поверхностей любой сложности;
- отсутствие требования сортировки объектов по глубине;
- высокая скорость работы.

Минусы:

- требование большого объема памяти (в рамках современных вычислительных систем несущественно);
- сложность работы с прозрачными и просвечивающими объектами.

Вывод: ввиду того, что алгоритм удовлетворяет главному требованию программы, а его минусы не входят в число требований к программе, данный алгоритм может быть выбран в качестве алгоритма удаления невидимых линий и поверхностей в данной работе.

1.4.3 Алгоритм Робертса

Алгоритм работает в объектном пространстве.

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:

- простые, но при этом точные математические методы;
- реализации алгоритма, использующие предварительную приоритетную сортировку вдоль оси z и простые габаритные или минимаксные тесты, демонстрируют почти линейную зависимость от числа объектов.

Минусы:

- вычислительная трудоёмкость алгоритма теоретически растет, как квадрат числа объектов;
- большое количество этапов обработки и предварительных вычислений (из чего следует большое количество кода и высокая асимптотическая константа);
- отсутствие возможности работы с прозрачными и просвечивающими объектами.

Вывод: данный алгоритм сложен в реализации в виду большого количества требуемых оптимизаций, а так же будет работать медленнее с аппроксимированными фигурами вращения, нежели со стандартными многогранниками, что приведет к медленной работе и не будет удовлетворять главному требованию программы.

Вывод

В таблице 1.1 представлено сравнение алгоритмов 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 объектами сложной формы (СФ).

Алгоритм	С	ММ	МЭ	СФ
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 рас-

сматриваемого полигона и источника света находится уровень освещенности по закону Ламберта (1.25):

$$I_\alpha = I_0 \cdot \cos(\alpha) \tag{1.25}$$

где I_α - уровень освёщенности в рассматриваемой точке, I_0 - максимальный уровень освёщенности, 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. 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:

- быстрая работа;
- хорошо подходит для многогранников, обладающих преимущественно диффузным отражением.

Минусы:

- плохо подходит для фигур вращения: видны ребра.

Вывод: учитывая, что в рамках данной работы все модели - фигуры вращения, данных алгоритм не является подходящим.

1.5.2 Закраска по Гуро

Идея: билинейная интерполяция в каждой точке интенсивности освещения в вершинах.

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 проходится 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):

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

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):

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

Плюсы:

- хорошо подходит для фигур вращения, аппроксимированных полигонами, с диффузным отражением.

Минусы:

- при закраске многогранников ребра могут стать незаметными.

Вывод: с учетом наличия в данной работе исключительно фигур вращения данных алгоритм подходит гораздо больше предыдущего.

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. 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:

- хорошо отображает блики, вследствие чего подходит для закраски фигур с зеркальным отражением.

Минусы:

- самый трудоёмкий алгоритм из рассмотренных.

Вывод

В таблице 1.2 представлено сравнение алгоритмов 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

Вывод

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[18]. Данный выбор обусловлен тем, что данный язык предоставляет весь требуемый функционал для решения поставленной задачи, а также обладает связанным с ним пакетным менеджером Cargo[19], который содержит инструменты для тестирования разрабатываемого ПО[20].

Для создания пользовательского интерфейса программного обеспечения была использована библиотека gtk-rs[21]. 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[19], поставляемого вместе с компилятором языка при стандартном способе установке, описанном на официальном сайте языка[18].

В процессе разработки был использован инструмент RLS[22] (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[23].

В качестве среды разработки был выбран текстовый редактор VIM[24], поддерживающий возможность установки плагинов[25], в том числе для работы с RLS[22].

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.

```
use super::prelude::*;
use std::vec::Vec;

pub struct Muscle {
    radiuses: Vec<f64>,
    grow_mults: Vec<f64>,
    dx: f64,
    min_dx: f64,
    max_dx: f64,
}

impl Muscle {
    pub fn new(radiuses: Vec<f64>, grow_mults: Vec<f64>, len: f64) -> Self {
        let dx = len / (radiuses.len() - 1) as f64;
        Self {
            radiuses,
            grow_mults,
            dx,
            min_dx: dx * constants::MIN_PART,
            max_dx: dx * constants::MAX_PART,
        }
    }

    fn get_angle(&self, i: usize) -> f64 {
        let last = self.radiuses.len() - 1;
        match i {
            0 => {
                f64::atan((self.radiuses[1] - self.radiuses[0]) / self.dx)
                    + std::f64::consts::PI / 2f64
            }
            val if val == last => {
                f64::atan((self.radiuses[last] - self.radiuses[last - 1]) / self.dx)
                    + std::f64::consts::PI / 2f64
            }
            val => {
                let a1 = (self.radiuses[val] - self.radiuses[val - 1]) / self.dx;
                let a2 = (self.radiuses[val + 1] - self.radiuses[val]) / self.dx;
                (f64::atan(a1) + f64::atan(a2)) / 2f64 + std::f64::consts::PI / 2f64
```

```
            }
        }
    }

    fn normal_ep(&self, i: usize) -> Point3d {
        let angle = self.get_angle(i);
        Point3d::new(
            self.dx * i as f64 + f64::cos(angle),
            self.radiuses[i] + f64::sin(angle),
            0_f64,
        )
    }

    fn find_intersections(&self, mut i1: usize, mut i2: usize) -> (Point3d, Point3d) {
        if i1 > i2 {
            std::mem::swap(&mut i1, &mut i2);
        }
        (
            Point3d::new(self.dx * i1 as f64, self.radiuses[i1], 0_f64),
            Point3d::new(self.dx * i2 as f64, self.radiuses[i2], 0_f64),
        )
    }

    fn fill_pn_connectors(
        &self,
        points: &mut Vec<Vec<Point3d>>,
        normal2points: &mut Vec<Vec<Point3d>>,
    ) {
        for i in 0..(self.radiuses.len() - 1) {
            let (p1, p2) = self.find_intersections(i, i + 1);

            let (mut new_points, mut new_norm2points) = rotate_intersections(
                &[p1, p2],
                &[self.normal_ep(i), self.normal_ep(i + 1)],
                constants::MUSCLE_STEP,
            );
            cycle_extend(&mut new_points, 2);
            cycle_extend(&mut new_norm2points, 2);

            points.push(new_points);
            normal2points.push(new_norm2points);
        }
    }

    fn fill_pn_spheres(
        &self,
        points: &mut Vec<Vec<Point3d>>,
        normal2points: &mut Vec<Vec<Point3d>>,
```

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

```
            res += mults[1] * rads[0] + mults[0] * rads[1];
        }

        res
    }

    fn find_c(&self, g: f64) -> f64 {
        let mut res = 0_f64;

        for rads in self.radiuses.windows(2) {
            res += 1_f64 / 3_f64 * f64::powi(rads[1] - rads[0], 2) + rads[0] * rads[1];
        }

        res - g
    }

    fn update_radiuses(&mut self, dy: f64) {
        for (rad, mult) in self.radiuses.iter_mut().zip(self.grow_mults.iter()) {
            *rad += mult * dy;
        }
    }

    pub fn deform(&mut self, diff: f64) {
        let new_dx = self.dx + diff / (self.radiuses.len() - 1) as f64;
        if new_dx < self.min_dx || new_dx > self.max_dx {
            return;
        }

        let g2 = self.find_volume() / new_dx;

        let a = self.find_a();
        let b = self.find_b();
        let c = self.find_c(g2);

        let dy = solve_quad_eq(a, b, c);
        if let Some(dy) = dy.1 {
            self.update_radiuses(dy);
            self.dx = new_dx;
        }
    }
}
```

Листинг 3.2: Реализация алгоритмов компьютерной графики.

```
use super::prelude::*;

static mut Z_BUFFER: [[f64; constants::WIDTH]; constants::HEIGHT] =
    [[f64::MIN; constants::WIDTH]; constants::HEIGHT];
static mut COLOR_BUFFER: [[u32; constants::WIDTH]; constants::HEIGHT] =
```

```
    [[constants::DEFAULT_COLOR; constants::WIDTH]; constants::HEIGHT];

// INPUT: points with normals, transformation matrix, light_source, color of input figure.
// RESULT: flushes all visible parts of transformed figure in internal COLOR_BUFFER.
pub unsafe fn transform_and_add(
    (points_groups, normals_groups): &(Vec<Vec<Point3d>>, Vec<Vec<Point3d>>),
    matrix: &Matrix4,
    light_source: Point3d,
    color: u32,
) {
    // for every triangulated group of input figure:
    for (points, normals) in points_groups.iter().zip(normals_groups.iter()) {
        let (p1, p2) = (
            transform_and_normalize(points[0], normals[0], matrix),
            transform_and_normalize(points[1], normals[1], matrix),
        );
        let mut current_window = [p1, p2, (Point3d::default(), Vec3d::default())];

        // for every triangle (3 points + 3 normal points):
        for (change_index, (&new_point, &new_normal)) in (2..)
            .map(|elem| elem % 3)
            .zip(points.iter().skip(2).zip(normals.iter().skip(2)))
        {
            // transform new point
            current_window[change_index] = transform_and_normalize(new_point, new_normal, matrix);
            // check if any part of triangle visible and triangle isn't rotated to background
            if check_pos_all(current_window.iter().map(|elem| elem.0))
                && check_normals_all(current_window.iter().map(|elem| elem.1))
            {
                // divide triangle on points array and normal points array
                let points = [
                    current_window[0].0,
                    current_window[1].0,
                    current_window[2].0,
                ];
                let normals = [
                    current_window[0].1,
                    current_window[1].1,
                    current_window[2].1,
                ];
                // add transformed triangle polygon to buffer
                add_polygon(points, normals, &light_source, color);
            }
        }
    }
```

```rust
}

unsafe fn add_polygon(
    points: [Point3d; 3],
    mut normals: [Vec3d; 3],
    light_source: &Point3d,
    color: u32,
) {
    // cast Y coordinate to integer (coordinates of the screen are integers)
    let mut int_points = [
        IntYPoint3d::from(points[0]),
        IntYPoint3d::from(points[1]),
        IntYPoint3d::from(points[2]),
    ];
    // sort points by Y coordinate
    sort_by_y(&mut int_points, &mut normals);
    // find brightnesses for all vertexes for furhter processing by Gouraud algorithm
    let brightnesses = find_brightnesses(points, normals, light_source);
    // divide triangle on 2 pairs of sections, which make up 2 triangles with
    // parallel to X axis edge
    let sections = divide_on_sections(int_points, brightnesses);
    process_sections(sections, color);
}

// application of Gouraud and Z-buffer algorithms for 2 processed triangles
unsafe fn process_sections(mut sections: [Section; 4], color: u32) {
    for pair in sections.chunks_mut(2) {
        if pair[0].x_start > pair[1].x_start {
            continue;
        }

        if pair[0].y_start < 0 {
            let diff = (-pair[0].y_start) as f64;
            for sec in pair.iter_mut() {
                sec.x_start += diff * sec.x_step;
                sec.br_start += diff * sec.br_step;
                sec.z_start += diff * sec.z_step;
            }
            pair[0].y_start = 0;
        }

        for y in (pair[0].y_start..=pair[0].y_end)
            .filter(|&elem| elem < constants::HEIGHT as i16)
            .map(|y| y as usize)
        {
            let x_from = f64::round(pair[0].x_start) as usize;
            let x_to = f64::round(pair[1].x_start) as usize;
            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 }
```

Вывод

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

Вывод

Заключение

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. с. 22–30.

[3] J. E. Chadwick D. R. Haumann, Parent R. E. Layered construction for deformable animated characters // SIGGRAPH Computer Graphics. 1989. с. 243–252.

[4] Komatsu K. Human skin model capable of natural shape variation // The Visual Computer, vol. 3, no. 5. 1988. с. 265–271.

[5] Blinn J. F. A generalization of algebraic surface drawing // ACM Transactions on Graphics, vol. 1, no. 3. 1982. с. 235–256.

[6] Bloomenthal J., Shoemake K. Convolution surfaces // SIGGRAPH Computer Graphics. 1991. с. 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. с. 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. с. 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] Что такое ACM SIGGRAPH? Режим доступа: https://www.siggraph.org/about/what-is-acm-siggraph/russian/ (дата обращения: 07.06.2020).

[15] 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.

[16] X-Muscle System. Режим доступа: https://blendermarket.com/products/x-muscle-system (дата обращения: 14.07.2020).

[17] Blender. Режим доступа: https://www.blender.org/ (дата обращения: 14.07.2020).

[18] Rust Programming Language [Электронный ресурс]. Режим доступа: https://doc.rust-lang.org/std/index.html (дата обращения: 20.07.2020). 2017.

[19] The Cargo Book. Режим доступа: https://doc.rust-lang.org/cargo/ (дата обращения: 21.07.2020).

[20] 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).

[21] Gtk-rs. Режим доступа: https://gtk-rs.org/ (дата обращения: 21.07.2020).

[22] Rust Language Server (RLS). Режим доступа: https://github.com/rust-lang/rls (дата обращения: 21.07.2020).

[23] Rustbook. What is Ownership? Режим доступа: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (дата обращения: 20.07.2020).

[24] VIM the editor. Режим доступа: https://www.vim.org/ (дата обращения: 20.07.2020).

[25] VimAwesome. Режим доступа: https://vimawesome.com/ (дата обращения: 20.07.2020).