СОДЕРЖАНИЕ

ВВЕДЕНИЕ

TODO: inro.

1 Аналитическая часть

1.1 Явление морфинга

Слово «морфинг» происходит от слова «метаморфоза», которое, согласно Согласно Оксфордскому словарю [**?**], имеет следующее значение: "Процесс, в ходе которого кто-то/что-то полностью превращается во что-то другое".

Таким образом, в случае трехмерных объектов термин «морфинг» можно интерпретировать как построение последовательности кадров, соответствующей постепенному переходу между двумя различными объектами, так называемыми исходными (начальными) и целевыми (конечными) моделями. На рисунке 1.1 представлен пример морфинга трехмерных объектов.

Цель морфинга заключается в вычислении преобразования, обеспечивающего визуально приятный переход между исходной и целевой формами [1]. TODO: возможно здесь добавить про монотонность, сохранение точек интереса и тд (в зависимости от того, как будет работать feature preservation)

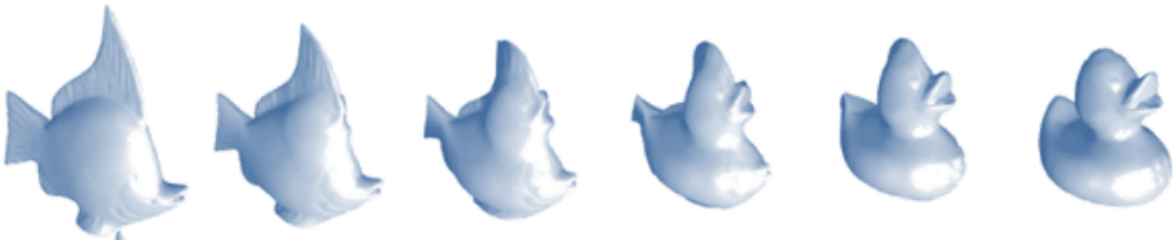

Рисунок 1.1 — Пример последовательности кадров морфинга

1.2 Формализация объектов сцен

На сцене выбора начальных и конечных объектов могут присутствовать 2 фрукта: начальный и конечный. Фрукт (объект) задатся моделью описания объектов, данными для этой модели, оптическими характеристиками поверхности.

На сцене просмотра морфинга находится результат морфинга (задается исходным и целевым объектами, стадией морфинга).

На обеих сценах могут присутствовать следующие типы объектов:

— источник света (задается положением в пространстве, цветом и интенсивностью света);

— наблюдатель (задается положением в пространстве, точкой в пространстве, на которую направлен взгляд, направлением верха обзора TODO);

1.3 Выбор модели описания объекта

Наиболее распространенные модели описания трехмерных объектов в компьютерной графике: *аналитичкская*, *полигональная модель*, *воксельная модель* [2].

1.3.1 Аналитическая модель

Аналитическая модель представляет собой описание поверхности математическими формулами [2]. Обычно поверхность задается уравнением вида $z = f(x, y)$ или $F(x, y, z) = 0$.

Отличительные черты [2]:

— легкая процедура рассчета координат точек, нормалей;

— небольшой объем информации для описания форм;

— сложные формулы, которые медленно вычисляются компьютером;

— задание объекта набором поверхностей, если невозможно описать его аналитически;

— отсутствие погрешности при задании сферического объекта;

1.3.2 Полигональная модель

В полигональной модели информация об объекте сосотоит из следущих компонентов [2]:

— вершина — точка ((x, y, z) в декартовой системе координат);

— отрезок прямой — задается двумя вершинами;

— полилиня — задается несколькими отрезками прямой;

— полигон — описывает плоскую грань объемного объекта в виде замкнутой линии;

Несколько граней (полигонов) составляют объемный объект в виде полигональной поверхности, также назваемой «полигональной сеткой».

1.3.3 Воксельная модель

TODO

1.3.4 Сравнение моделей описания объектов

TODO: а надо ли мне вообще выбирать, если у меня в ТЗ написанно НИЗКОПОЛИГОНАЛЬНЫЕ ОБЪЕКТЫ???? TODO: можно сравнить по критериям: сложность вычисления нормали, изображение любых объектов, сохранение качества при увеличении

Среди всех моделей наиболее подходящей является полигональная модель, т. к. с помощью нее можно описать объекты любой сложности, и сохраняет качестве при увеличении, что необходимо в задаче интерактивной визуализации морфинга, поэтому использованная будет именно она.

1.4 Основные этапы морфинга

1.4.1 Установление соответствия между объектами

3D-сетки часто различаются по топологии (число вершин/граней и связность), поэтому прямое сопоставление поверхностей затруднено. Соответствие получают косвенно через *параметризацию* — биективное отображение поверхности сетки в общую параметрическую область D, обычно единичный диск для незамкнутых сеток (*плоская парамеризация*) или единичная сфера для замкнутых (*сферическая параметризация*) [1].

Сферическая параметризация примяется к замкнутым поверхностям нулевого рода [1], или не имеющим отверстий, коими являются фрукты в рамках данной работы.

1.4.2 Параметризация

Выделяют 3 основных подхода к параметризации сеток: *плоская*, *сферическая*, *разбиение на участи с плоской параметризацией* [1, 3].

Плоская праметризация применима для незамкнутых сеток [1,3], поэтому она непригодна для морфинга замкнутных объектов, таких, как фрукты.

Если исходный и целевой объекты имеют существенно различающиеся формы или представляют собой поверхности разного рода, т. е. имеют разное число отверстий, применяют предварительное разбиение на плоские участки [1]. Пользователь вручную разбивает исходную и целевую сетки на участки и задаёт их соответствие; затем для каждого участка выполняется плоская

параметризация. Данный метод требует значительного участия пользователя, от которого зависит качество результата, поэтому в настоящей работе он не рассматривается.

Для сферической параметризации произвольных поверхностей нулевого рода, применяется метод релаксации, основаный на итеративном уточнении положений вершин [3]. Процесс релаксации сетки представлен на рисунке 1.2.

В качестве начального состояния строят грубую проекцию сетки на единичную сферу: выбирают любую внутреннюю точку модели в качестве центра сферы и проецируют все вершины на её поверхность (нормализуют радиус-векторы). Полученная начальная конфигурация обычно содержит значительные искажения и грани с неправильной ориентацией. Процесс релаксации продолжают до тех пор, пока все грани не приобретут корректную ориентацию — внешняя сторона каждой грани должна быть обращённой наружу сферы [3].

На каждом раунде релаксации вершины сдвигаются к центру масс своих соседей:

$$v_i^{k+1} = \frac{\sum_{j \in N(i)} v_j^k}{\| \sum_{j \in N(i)} v_j^k \|}, \tag{1.1}$$

где

— v_i^{k+1} — положение i-ой вершины после k-го раунда релаксации;

— v_i^k — положение i-ой вершины на момент k-го раунда релаксации;

— $N(i)$ — множество вершин, смежных с i-ой.

Для предотвращения коллапса всех вершин в одну точку после каждой итерации выполняется ре-центрирование всей сетки относительно начала координат [3]:

$$v_i' = v_i - \frac{\sum_{j=0}^{n} v_j}{n}, \tag{1.2}$$

где

— v_i' — новое положение i-ой вершины;

— n — количество вершин.

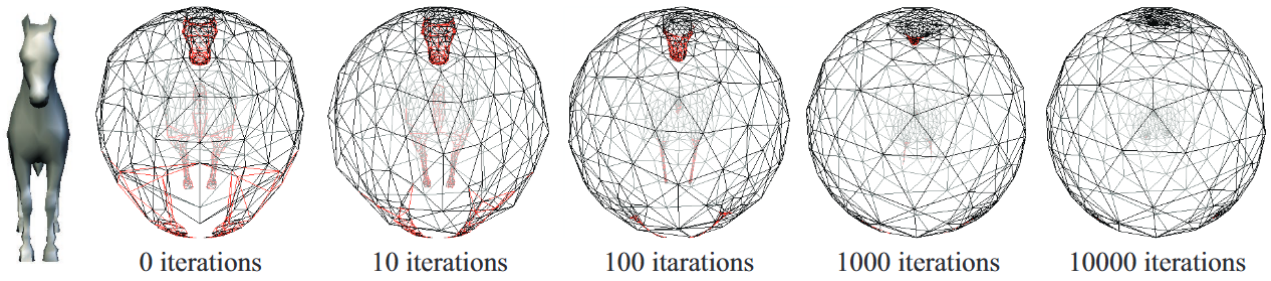

Рисунок 1.2 — Процесс релаксации сетки, красными отмечены грани с непрвильной ориентацией

1.4.3 Создание общего представления

После того как установлено соответствие, необходимо создать единую структуру, которая будет использоваться для всех промежуточных форм. Обычно для этого сроят *суперсетку* — сетку, содержащую вершины исходной и целевой сетки, а также их точки пересечения [1, 3] (пример на рисунке 1.3).

На параметрической сфере сетки накладываются друг на друга; в местах пересечения ребер создаются новые вершины, что формирует объединенную сетку, которая может принимать форму, как исходной так и целевой модели; поскольку полученная сетка обычно не является треугольной, выполняют её треангуляцию. Для этого используется треангуляция Делоне, поскольку она минимизирует количество узких треугольников, что улучшает качество сетки [1, 3].

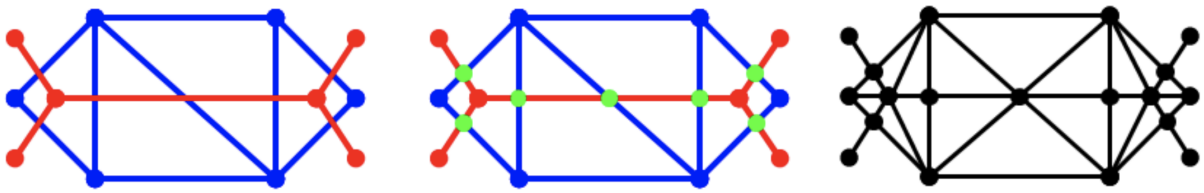

Рисунок 1.3 — Создание суперсетки

1.4.4 Интерполяция геометрии

Заключительный этап — вычисление траекторий движения вершин общего представления из их начальных положений до конечных. Для каждой вершины суперсетки необходимо определить ее координаты на исходной и целевой моделях. Процесс генерации промежуточных кадров сводится к интерполяции этих координат.

Положение вершины на каждой из моделей определяется с помощью барицентрических координат в параметрическом пространстве [1, 3]. Алгоритм сводится к следующим шагам:

Процесс вычисления состоит из следующих шагов:

1) Поиск содержащего треугольника: Для каждой определяется треугольник исходной (или целевой) сетки, в который она попадает в общем параметрическом пространстве (на сфере).
2) Вычисление барицентрических координат данной вершины относительно вершин найденного треугольника.
3) Барицентрические координаты применяются к вершинам соответствующего треугольника в мировых координатах.

После получения начальных и конечных положений $P0$ и $P1$ для каждой вершины траектории генерируют посредством линейной интерполяции [1, 3]. Положение вершины в момент времени $t \in [0, 1]$ вычисляется по формуле (**??**).

$$P(t) = (1 - t) \cdot P_0 + t \cdot P_1 \tag{1.3}$$

TODO: добавить про интерполяцию материала и нормалей

1.5 Анализ алгоритмов удаления невидимых линий и поверхностей

Алгоритмы удаления невидимых линий и поверхностей служат для удаления ребер, поверхностей или объемов, которые видимы или невидимы для наблюдателя, находящегося в заданной точке пространства [4].

Рассмотрим следующие алгоритмы: *алгоритм Робертса*, *алгоритм, использоующий z-буфер*, *алгоритм трассировки лучей*.

1.5.1 Алгоритм Робертса

Данный алгоритм применим только к выпуклым телам. Если обрабатываемое тало невыпуклое — его необходимо предварительно разбить на выпуклые [4].

Алгоритм состоит из следующих этапов [4]:

1) Удаление граней, экранируемых самим телом.

2) Удаление граней, экранируемых другими телами.

3) Удаление линий пересечения тел, экранируемых самими телами.

Ассимптотическая оценка трудоемкости: $O(N^2)$, где N — количество граней. TODO: проверить

1.5.2 Алгоритм, использующий *z*-буфер

Идея z-буфера является обобщением идеи буфера кадра. Буфер кадра используется для запоминания аттрибутов (интенсивности) каждого пикселя в пространстве изображения. z-буфер — это отдельны буфер глубины, используемый для запоминания координаты z каждого видимого пикселя в пространстве изображения [4].

Этапы работы алгоритма [4]:

1) Заполнить буфер кадра фоновым значением.

2) Заполнить z-буфер минимальным значением глубины.

3) Выполнить переход в пространство изображения.

4) Для каждого пикселя (x, y), пренадлежащего телу вычислить его глубину $z(x, y)$.

5) Если глубина $z(x, y) >$ *z-буфер*(x, y), то записать атрибут текущего тела в *буфер-кадра*(x, y), записать глубину $z(x, y)$ в *z-буфер*(x, y).

Асимптотическая оценка трудоекости: $O(N)$, где N — количество граней. TODO: проверить

1.5.3 Алгоритм художника

Идея алгоритма состоит в том, чтобы подобно художнику отрисовывать объекты по мере их приближения к наблюдателю.

Основные этапы алгорима [4]:

1) Отсортировать грани по минимальному или максимальному значению глубины.

2) Отрисовать грани в отсортированном порядке.

Простая сортировка не всегда дает корректный список приоритетов, тогда приходится использовать дополнительные методы разрешения конфликтов [4].

Алгоитм не справляется со случаями циклического перекрытия и пересечения многоугольников.

Ассиптотическая оценка трудоемкости: $O(N)$, где N — количество граней, однако стоит дополнительно учитывать трудоемкость предварительной сортировки.

Алгоритм трассировки лучей

Идея метода заключается в том, что наблюдатель видит любой объект посредством испускаемого неким источником света, который падает на этот объект и некоторомы путем доходит до наблюдателя. Однако обычно используют алгоритм обратной трассировки лучей, в котором лучи пускаются от наблюдателя [4].

Полагая, что объекты сцены уже находятся в простратсве изображения, выполняются следующие этапы для каждого пикселя [4]:

1) Составить уравнение отслеживаемого луча.
2) Найти пересечение луча со всеми гранями.
3) Среди всех пересечений найти ближайшее к наблюдатею и использовать аттрибуты объекта, которому соответвствует пересечение для определения цвета пикселя. Если пересечений нет — закрасить пиксель цветом фона.

Ассимптотическая оценка трудоемкости: $O(WHN)$, где W — ширина экрана в пикселях, H — высота экрана в пикселях, N — количество граней.

1.5.4 Сравнение алгоритмов

В таблице 1.1 представленные результаты сравнения алгоритмов и использованы следующией обозначения:

— Р — алгоритм Робертса;
— ZБ — алгоритм, использующий z-буфер;
— Х — алгоритм Художника;
— ТЛ — алгоритм трассировки лучей.

Таблица 1.1 — Сравнение алгоритмов удаления невидимых линий и поверхностей

	Р	**ZБ**	**Х**	**ТЛ**
Совместимость с любыми телами	-	+	+	+
Возможность использования без сортировки	+	+	-	+
Возможность учета прозрачности	-	+	+	+
Асиптотическая оценка трудоемкости	$O(N^2)$	$O(N)$	$O(N)$	$O(WHN)$

Среди всех расмотренных алгоритмов, алгоритм, использующий z-буфер подходит больше остальных, т. к. он имеет лучшую асимптотическую оценку трудоемкости и применим к любым телам, что необходимо при решении задачи визуализации морфинга.

1.6 Модель освещения

В компьютерной графике наиболее распространенными являются две модели освещения: *локальная* и *глобальная* [4].

Локальная модель учитывает только свет, падающий от источника (источников), и ориентацию поверхности [4].

Глобальная модель освещения учитывает также свет, отраженный от других объектов сцены или пропущенный через них [4].

Поскольку на сцене будет находиться только один объект, то будет использована локальная модель освещения.

Интенсивность I в точке P в локальной модели вычисляется по формуле (**??**) [4].

$$I = k_a I_a + \frac{I_l}{d + K}[k_d(\hat{\mathbf{n}} \cdot \hat{\mathbf{L}}) + k_s(\hat{\mathbf{R}} \cdot \hat{\mathbf{S}})^{\alpha}], \tag{1.4}$$

где

k_a — коэффициент TODO;

I_a — интенсивность фонового освещения;

I_l — интенсивность источника света;

d — расстояние от источника света до точки P;

K — добавка уменьшения интенсивности света с расстоянием (выбирается из эстетических предпочтений);
k_d — коэффициент диффузного отражения поверхности;
$\mathbf{n}$ — вектор нормали к поверхности в точке P;
$\mathbf{L}$ — вектор, обратный вектору падения луча;
k_s — коэффициент зеркального отражения поверхности;
$\mathbf{R}$ — вектор, отраженного луча;
$\mathbf{S}$ — вектор, направленный на наблюдателя из точки P;
α — степень, аппроксимирующая пространственное распределение зеркально отраженного света.

1.7 Анализ алгоритмов закраски

Основными алгоритмами закраски в комьютерной графике являются: *однотонная закраска*, *закраска методом Гуро*, *закраска методом Фонга* [4].

1.7.1 Однотонная закраска

При однотонной закраске для каждой грани (многоугольника) полигональной поверхности вычисляется один уровень интенсивности, с которым закрашивается вся грань. В результате такой закраски изображенный состоит из отдельных иногоугольников и объект выглядит, как многогранник [4].

1.7.2 Закраска методом Гуро

Этот метод предназначен для создания иллюзии гладкой криволинейной поверхности, описанной в виде многогранников или полигональной сетки с плоскиими гранями [2, 4, 5]

Метод Гуро основан на интерполяции интенсивности каждого пикселя при закраске. Закрашивание граней по методу гуро осуществляется в четыре этапа [2]:

1) Определение нормали к каждой грани.
2) Определение нормалей в вершинах путем усреднения нормалей прилежащих граней.
3) На основе нормалей в вершинах вычисляются значения интенсивностей в вершинах согласно выбранной модели освещения.

4) Закрашиваются полигоны граней цветом, соответствующим интерполяции значений интенсивности в вершинах.

1.7.3 Закраска методом Фонга

Аналогичен методу Гуро, но при использовании метода Фонга для определения цвета в каждой точке интерполируются не интенсивности отраженного света, а векторы нормалей. При этом значение интенсивность вычисляется в каждом внутреннем пикселе грани [2, 4].

1.7.4 Сравнение алгоритмов закраски

Визуальное сравнение алгоитмов представленно на рисунке 1.4.

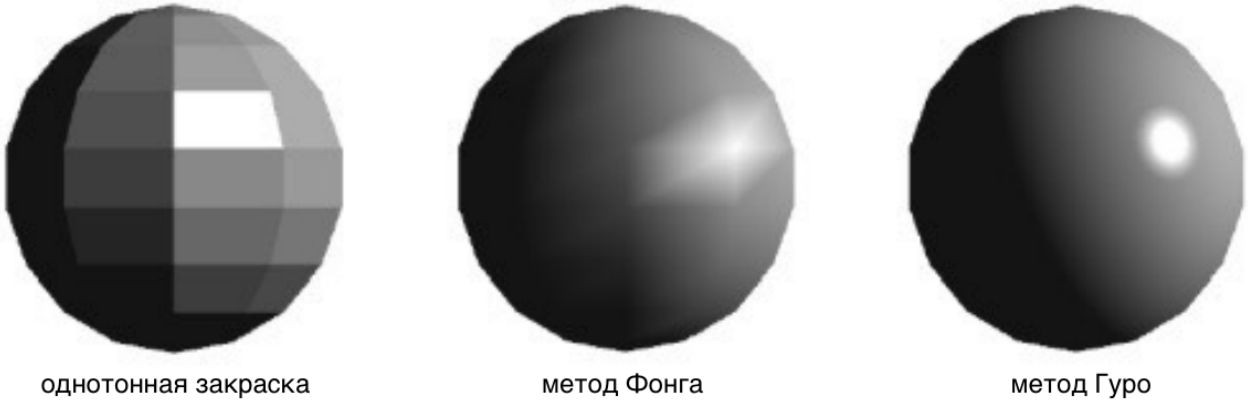

Рисунок 1.4 — Визуальное сравнение методов закрски

Поскольку объекты сцены (фрукты) представлены низкополигональными объектами, предпочтительной является однотонная закраски, обеспечивающая дискретность освещения между смежными гранями. В результате мофринга часто возникает множество полигонов, составляющих одну грань, из-за чего при однотонной закраске возникают световые артефакты, как на рисунке 1.5, а.

Для устранения этой проблемы будет использована модифицированная версия закраски Гуро: значение интенсивности вершин в пределах одной грани вычисляется с помощью нормали к этой грани, а не нормалей в вершинах. Это обеспечивает равномерное распределение интенсивности по грани и предотвращает сглаживание рёбер, что продемонстрированно на рисунке 1.5, б.

а) б)

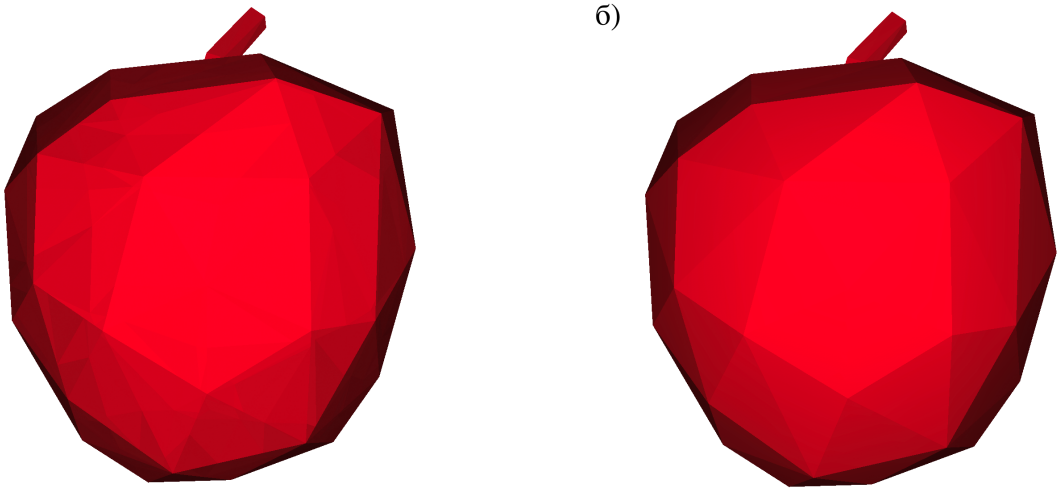

Рисунок 1.5 — Результат морфинга с использованием различных алгоритмов закраски: а) однотонная закраска; б) закраска модифицированным методом Гуро

Вывод

В аналитической части были рассмотрены основные этапы морфинга, проанализированны модели представления объектов и алгоритмы решения основных задач компьютерной графики. По результатам проведенного анализа были выбраны: полигональное представление модели, алгоритм, использующий z-буфер для удаления невидимых линий и поверхностей, простая модель освещения, модифиицированная закраска Гуро.

2 Конструкторская часть

2.1 Требования к программному обеспечению

Программа должна предоставлять пользователю графический интерфейс, позволяющий:

— загрузить исходный и целевой объект из *.obj* файла;

— изменить цвет и оптические свойства поверхности загруженных объектов;

— просматривать каждый объект и результат морфинга посредством поворота и масштабирования;

— изменять масштабирование и поворот исходного и целевого объектов морфинга;

— выбирать стадию морфинга объектов.

Программа должна корректно реагировать на любые действия пользователя.

2.2 Используемы типы и структуры данных

1) Сцена состоит из:
   — объекта;
   — камеры;
   — источника света.
2) Объект состоит из:
   — массива точек;
   — массива треугольников (индексы вершин);
   — массива внешних нормалей к треугольникам;
   — материала;
   — матрицы преобразований.
3) Результат морфинга, состоит из:
   — массива точек;
   — массива функций, интерполирующих вершины в момент t;
   — массива треугольников (индексы вершин);
   — массива внешних нормалей к треугольникам;
   — массива функций, интерполирующих внешние нормали в момент

t;

— материала;

— функции интерполирующей материал в момент t;

— матрицы преобразований.

4) камера состоит из:

— положения;

— точки, на которую направлен взгляд;

— угол обзора в радианах;

— отношение сторон;

— уаление ближней плоскости пирамиды видимости;

— удаление дальней плоскости пирамды видимости.

5) источник света состоит из:

— положения;

— интесивности;

— цвета; TODO: возможно убрать

6) материал состоит из:

— цвета в формате *RGB*;

— коэффициента диффузного отражения;

— коэффициента зеркального отражения;

— степени, аппроксимирующей пространственное распределение зеркально отраженного света.

Для построения суперсетки используется структура данных DCEL (doubly connected edge list) [3]. Визуальное представление структуры данных приведено на рисунке **??**.

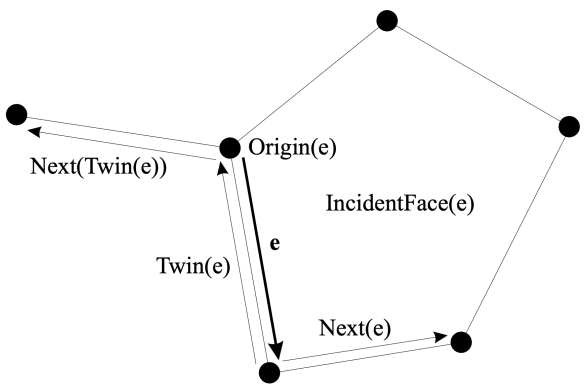

Рисунок 2.1 — DCEL (doubly connected edge list) [3]

DCEL состоит из:

— массив вершин;

— массив полуребер;

— массив граней;

Полуребро состоит из:

— индекса вершины-начала;

— индекса полуребра-двойника;

— индекса смежной грани;

— индекса следующего полуребра;

Грань состоит из индекса полуребра, связанного с ней.

2.3 Математические основы алгоритмов

2.3.1 Барицентрические координаты

Барицентрические координаты α, β и γ определяют положение точки P относительно вершин треугольника A, B и C [5].

Точка P находится внутри или на границах треугольника ABC, если она может быть представлена в виде аффинной комбинации вершин [5]:

$$P = \alpha A + \beta B + \gamma C, \text{ где } \alpha + \beta + \gamma = 1, \text{ и } \alpha, \beta, \gamma \geq 0 \text{ [5]} \tag{2.1}$$

Если хотя бы одна из координат α, β, γ отрицательна, то точка P лежит вне треугольника [5].

Барицентрическая координата α точки P пропорциональна площади треугольника PBC (см. рисунок **??**) (треугольника, противолежащего вершине A) [5].

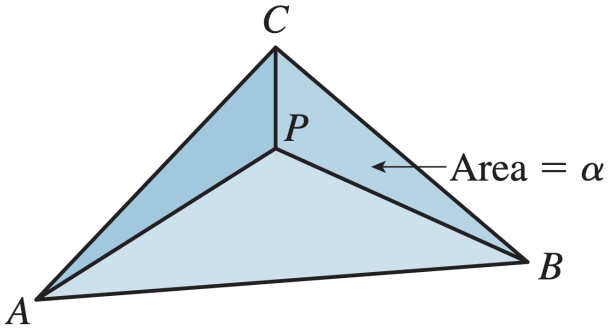

Рисунок 2.2 — Точка P разделяет треугольник ABC на три меньших треугольника, площади которых соотносятся как α, β и γ; Барицентрические координаты точки P равны (α, β, γ) [5]

Координата α точки P определяется как отношение площади треугольника PBC к площади всего треугольника ABC [5]:

$$\alpha = \frac{S_{\triangle PBC}}{S_{\triangle ABC}} \tag{2.2}$$

Аналогичные соотношения используются для β (пропорционально $S_{\triangle PAC}$) и γ (пропорционально $S_{\triangle PAB}$).

На плоскости барицентрические координаты точки P в треугольнике ABC могут быть вычислены при помощи косого произведения:

$$\begin{cases} \alpha = \frac{|BC \times BP|}{|AB \times AC|} = \frac{(C-B)_x \cdot (P-B)_y - (C-B)_y \cdot (P-B)_x}{(B-A)_x \cdot (C-A)_y - (B-A)_y \cdot (C-A)_x} \\ \beta = \frac{|CA \times CP|}{|AB \times AC|} = \frac{(A-C)_x \cdot (P-C)_y - (A-C)_y \cdot (P-C)_x}{(B-A)_x \cdot (C-A)_y - (B-A)_y \cdot (C-A)_x} \\ \gamma = 1 - \alpha - \beta \end{cases} \tag{2.3}$$

2.3.2 Поиск пересечения дуг на единичной сфере

Для нахождения пересечения дуг необходимо:

1) найти прямую по которой пересекаются плоскости, содержащие дуги;
2) найти точки пересечение прямой с единичной сферой;
3) проверить принадлежность точек обеим дугам.

Пусть дуга A ограничена точками A_0, A_1, а дуга B — B_0, B_1.

Направляющий вектор прямой можно вычислить как векторное произведение нормалей к плоскостям, содержащим дуги:

$$\mathbf{d} = \mathbf{n_A} \times \mathbf{n_B} = (\mathbf{A_0} \times \mathbf{A_1}) \times (\mathbf{B_0} \times \mathbf{B_1}) \tag{2.4}$$

Точки пересечения с единичной сферой равны:

$$P_{1,2} = \pm\frac{\mathbf{d}}{\parallel \mathbf{d} \parallel} \tag{2.5}$$

Т. к. длина радиус-векторов всех точек на единичной сфере равна 1, скалярное произведение равно косинусу угла между векторами. В таком случае точка $P_{1,2}$ принадлежит обеим дугам A и B, а следовательно является их пересечением если:

$$\begin{cases} A_0 \cdot A_1 \leq A_0 \cdot P_{1,2} \\ A_0 \cdot A_1 \leq A_1 \cdot P_{1,2} \\ B_0 \cdot B_1 \leq B_0 \cdot P_{1,2} \\ B_0 \cdot B_1 \leq B_1 \cdot P_{1,2} \end{cases} \tag{2.6}$$

2.3.3 Поиск точки внутри объекта

Основные шаги:

1) Испустить луч из центра масс O любой грани в направлении внутренней нормали.
2) Найти все точки пересечения с другими гранями.
3) Определить ближайшую к началу луча точку пересечения I.
4) Вернуть точку по середине отрезка OI.

Пересечение луча с полигоном

В параметрическом представлении луча задается уравнением (**??**).

$$P(t) = P_0 + \mathbf{d} \cdot t, \tag{2.7}$$

где P_0 — начальная точка луча; $\mathbf{d}$ — направляющий вектор луча.

Произвольная точка M лежит на плоскости A, если для нее выполняется равенство (**??**).

$$(M - M_A) \cdot \mathbf{n_A} = 0, \tag{2.8}$$

где M — произвольная точка

M_A — известная точка на плоскости A

$\mathbf{n_A}$ — вектор нормали к A.

Подставив (**??**) в (**??**), получим уравнение (**??**) для нахождения параметра t точки пересечения луча с плоскостью полигона.

$$t = \frac{(M_A - P_0) \cdot \mathbf{n_A}}{\mathbf{d} \cdot \mathbf{n_A}} \tag{2.9}$$

Если знаменатель $\mathbf{d} \cdot \mathbf{n_A}$ равен нулю, то луч параллелен плоскости полигона. Если $t < 0$, то пересечение находится позади начала луча, и такая точка отбрасывается, так как поиск ведется внутри объекта.

Найдя точку пересечения луча с плоскостью, необходимо определить, лежит ли она внутри полигона. Для этого можно использовать барицентрические координаты.

2.3.4 Матрицы преобразований

Матрицы преобразований в мировое пространство

Для перевода объектов из их локального пространства в мировое пространство используются матрицы масштабирования и поворота [2, 4, 5]. Матрицы поворота вокруг осей X, Y и Z представлены в формулах (**??**), (**??**) и (**??**) соответственно.

$$R_X(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

$$R_Y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.11}$$

$$R_Z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.12}$$

Матрица масштабирования представлена в формуле (**??**).

$$S = \begin{bmatrix} S_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.13}$$

2.4 Разработка алгоритмов

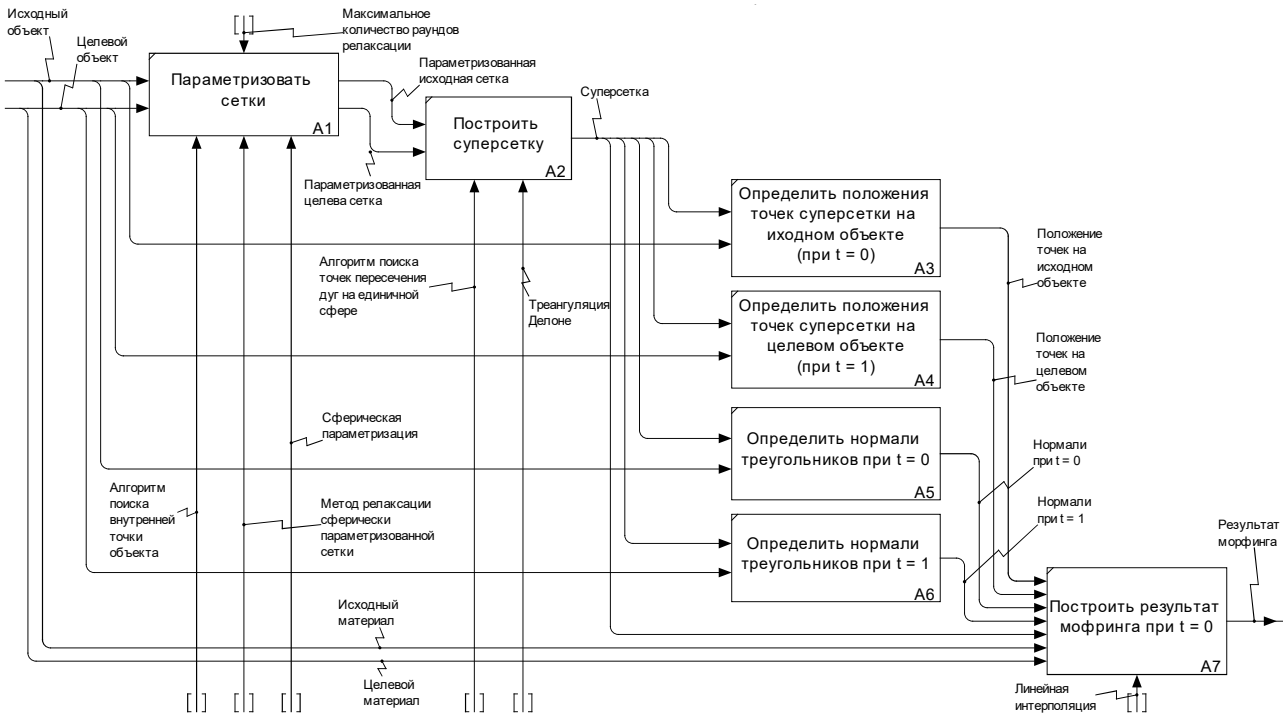

Рисунок 2.3 — Функциональная модель построения морфинга

Схемы основных алгоритмов, используемых в программном обеспечены представлены на рисунках **??**–**??**.

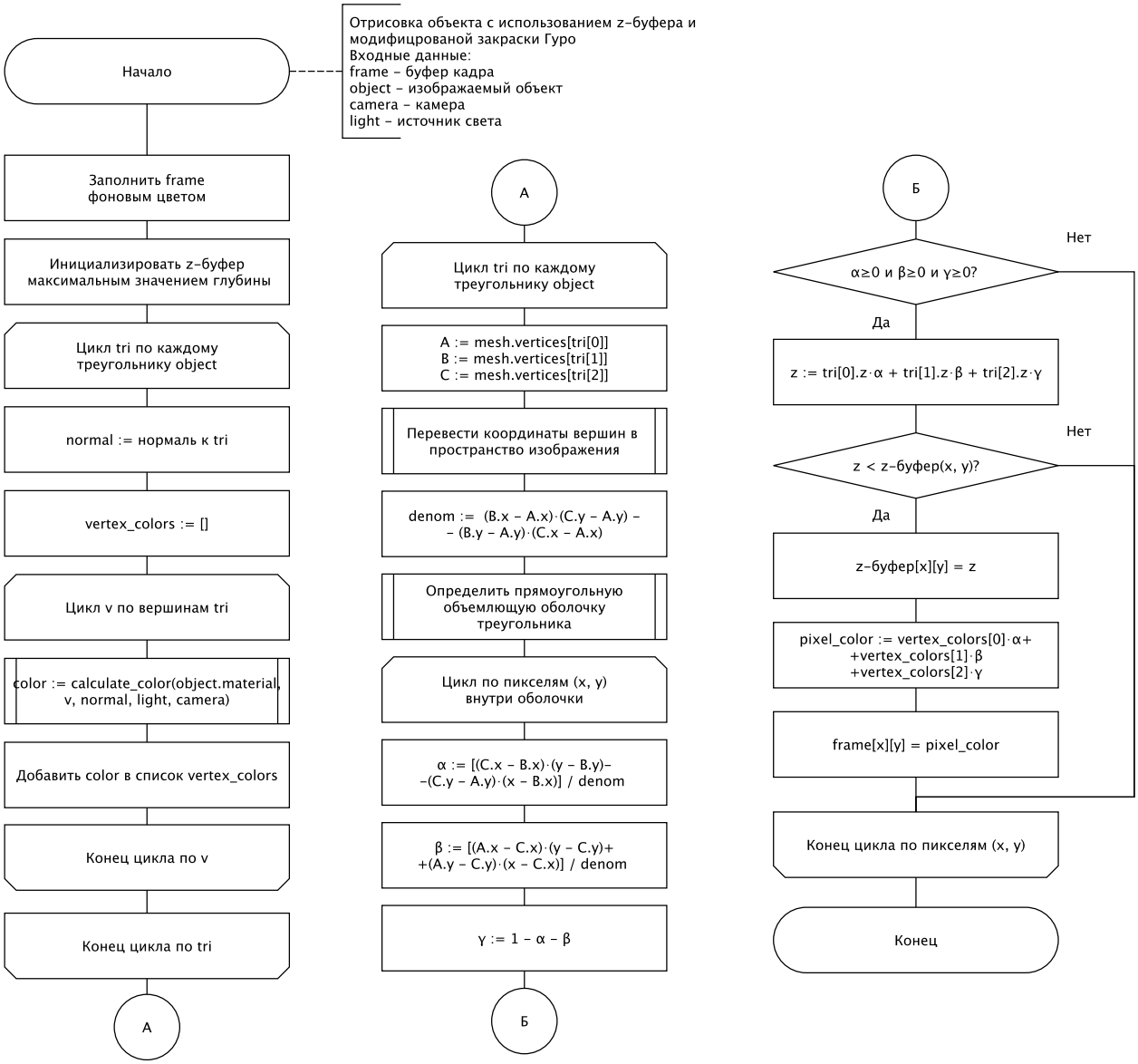

Рисунок 2.4 — Схема алгоритма отрисовки объекта с использованием *z*-буфера и модифицированной закраски Гуро

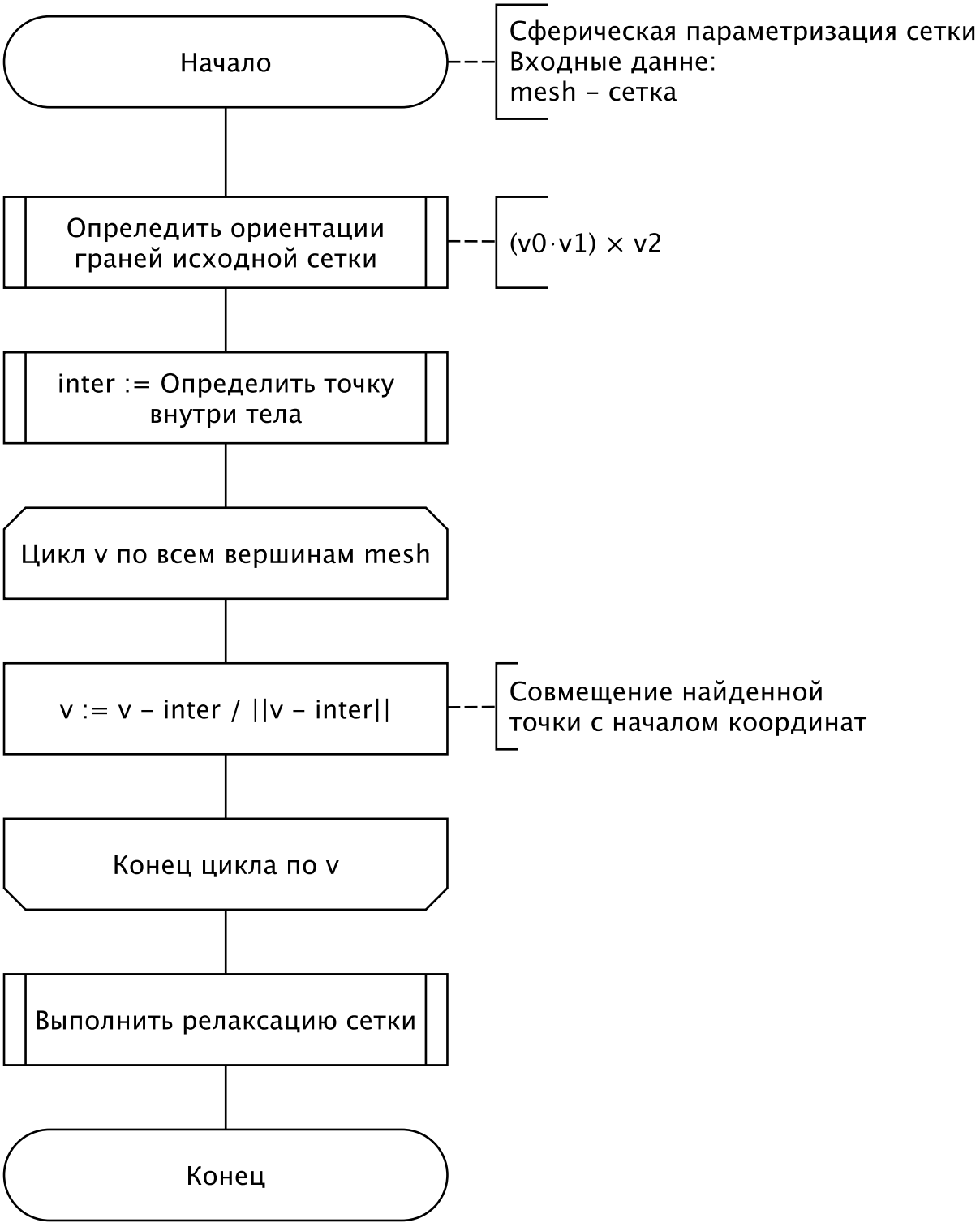

Рисунок 2.5 — Схема алгоритма сферической параметризации сетки

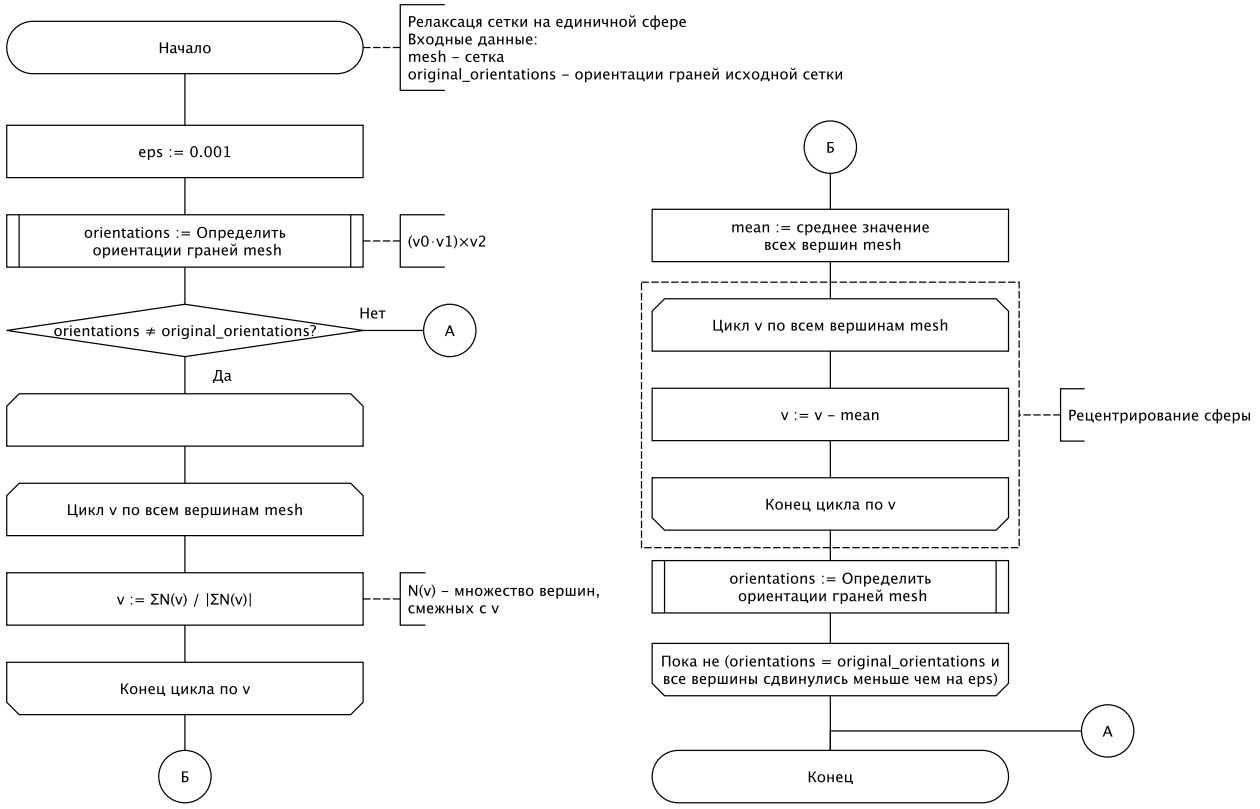

Рисунок 2.6 — Схема алгоритма релаксации сетки

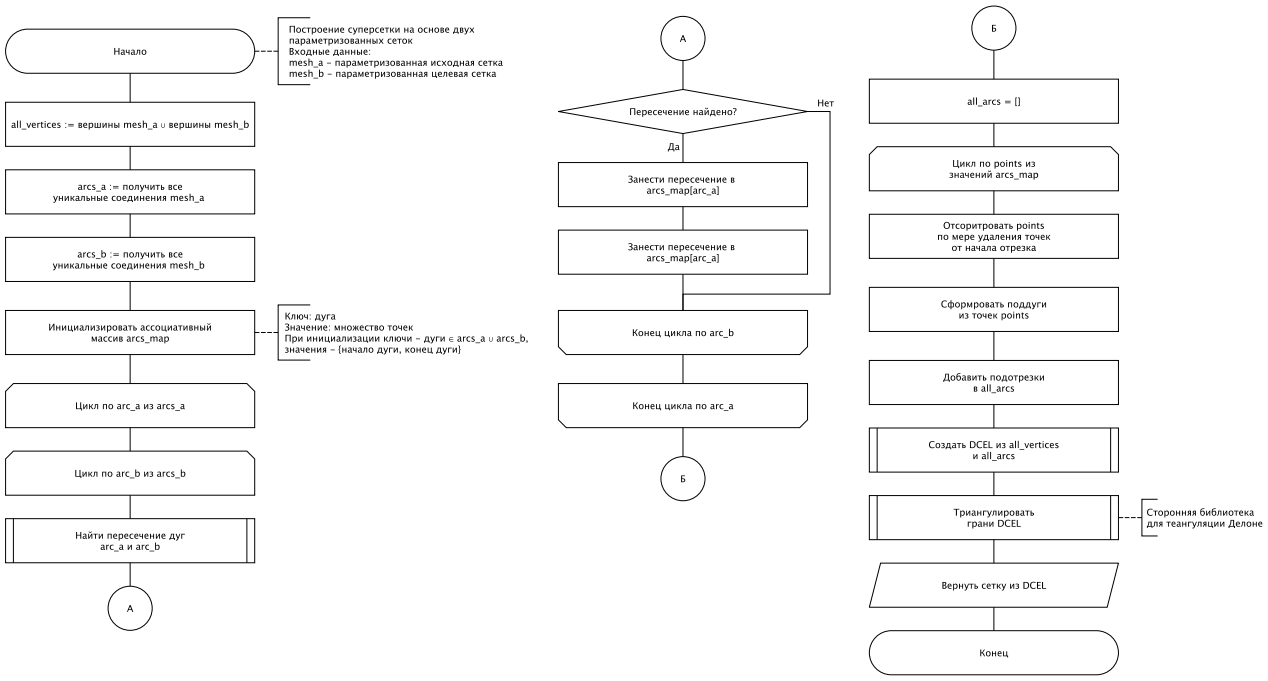

Рисунок 2.7 — Схема алгоритма построения суперсетки на единичной сфере

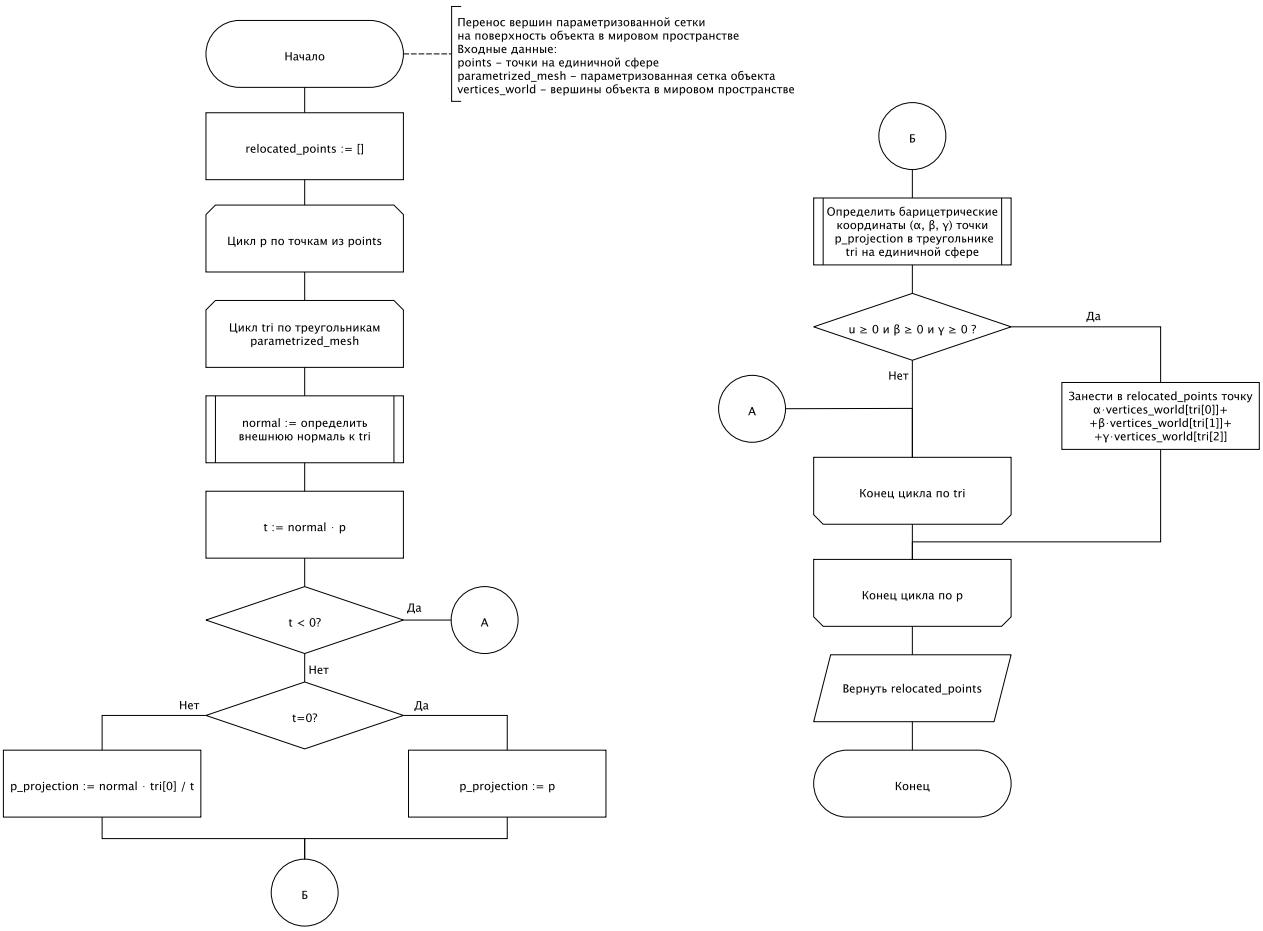

Рисунок 2.8 — Схема алгоритма переноса вершин суперсетки с единичной сферы в мировое пространство

Вывод

3 Технологическая часть

3.1 Средства реализации

В качестве языка программирования для реализации алгоритмов выбран язык *Rust* [6], так как он удовлетворяет всем требования.

Для реализации графического интерфейса использована библиотека *egui* [7].

В качестве среды разработки была выбрана IDE *RustRover* [8].

3.2 Формат входных и выходных данных

3.3 Реализация алгоритмов

В данном разделе приведены листинги кода, реализующего основные алгоритмы.

3.3.1 Реализация алгортмов растеризации TODO: название

.

На листингах 3.1, 3.2 приведена реализация алгоритма отрисовки объекта с использованием z-буфера и модифицированной закраски Гуро.

Листинг 3.1 — Реализация алгоритма отрисовки объекта с использованием z-буфера и модифицированной закраски Гуро (часть 1)

```
    fn draw_object(
        &mut self,
        image: &mut RgbImage,
        model: &dyn Model3D,
        camera: &Camera,
        light_source: &LightSource,
    ) {
        let (width, height) = image.dimensions();
        let mvp_matrix = camera.camera_matrix * model.
           model_matrix();
        let viewport_matrix = Self::calculate_viewport_matrix(
           width, height);
        let mvpv_matrix = viewport_matrix * mvp_matrix;

        let screen_vertices: Vec<Point3<f64>> = Self::
           transform_vertices_to_screen(
```

```
            model.vertices(),
            &mvpv_matrix,
        );

        for (i, tri) in model.triangles().iter().enumerate() {
            let tri_colors = [tri.0, tri.1, tri.2].map(|v_idx| {
                calculate_color(
                    &model.material(),
                    &model.normals()[i].xyz(),
                    &model.vertices_world()[v_idx],
                    &light_source,
                    &camera.pos,
                )
            });

            self.draw_triangle(
                image,
                &[
                    screen_vertices[tri.0],
                    screen_vertices[tri.1],
                    screen_vertices[tri.2],
                ],
                &tri_colors,
            );
        }
    }
}
```

Листинг 3.2 — Реализация алгоритма отрисовки объекта с использованием z-буфера и модифицированной закраски Гуро (часть 2)

```
    fn draw_triangle(
        &mut self,
        image: &mut RgbImage,
        tri: &[Point3<f64>; 3],
        tri_colors: &[Rgb<u8>; 3],
    ) {
        let [p1, p2, p3] = *tri;

        // Находим ограничивающий прямоугольник, ограничивая разм
           ерами изображения.
```

```
let min_x = (p1.x.min(p2.x).min(p3.x).round() as u32).max
   (0);
let max_x = (p1.x.max(p2.x).max(p3.x).round() as u32).min
   (self.width - 1);
let min_y = (p1.y.min(p2.y).min(p3.y).round() as u32).max
   (0);
let max_y = (p1.y.max(p2.y).max(p3.y).round() as u32).min
   (self.height - 1);

// Предварительно вычисляем общие компоненты, чтобы избеж
   ать избыточных вычислений в цикле.
let denom = (p2.x - p1.x) * (p3.y - p1.y) - (p2.y - p1.y)
    * (p3.x - p1.x);

for y in min_y..=max_y {
    for x in min_x..=max_x {
        // Вычисляем барицентрические координаты.
        let u =
            ((p3.x - p2.x) * (y as f64 - p2.y) - (p3.y -
               p2.y) * (x as f64 - p2.x)) / denom;
        let v =
            ((p1.x - p3.x) * (y as f64 - p3.y) - (p1.y -
               p3.y) * (x as f64 - p3.x)) / denom;

        let bary = Point3::new(u, v, 1.0 - u - v);

        // Проверяем, находится ли пиксель внутри треугол
           ьника.
        if bary.x > -f64::EPSILON && bary.y > -f64::
           EPSILON && bary.z > -f64::EPSILON {
            let z = p1.z * bary.x + p2.z * bary.y + p3.z
               * bary.z;

            // Выполняем проверку по Z-буферу.
            if z < self.get_depth(x, y) {
                self.set_depth(x, y, z);

                // Интерполируем цвета корректно для кажд
                   ого канала.
                let r = (bary.x * tri_colors[0].0[0] as
```

```
                        f64
                            + bary.y * tri_colors[1].0[0] as f64
                            + bary.z * tri_colors[2].0[0] as f64)
                            .clamp(0.0, 255.0) as u8;
                        let g = (bary.x * tri_colors[0].0[1] as
                        f64
                            + bary.y * tri_colors[1].0[1] as f64
                            + bary.z * tri_colors[2].0[1] as f64)
                            .clamp(0.0, 255.0) as u8;
                        let b = (bary.x * tri_colors[0].0[2] as
                        f64
                            + bary.y * tri_colors[1].0[2] as f64
                            + bary.z * tri_colors[2].0[2] as f64)
                            .clamp(0.0, 255.0) as u8;

                        image.put_pixel(x, y, Rgb([r, g, b]));
                    }
                }
            }
        }
    }
```

3.3.2 Реализация алгоритмов морфинга

На листинге 3.3 представлена реализация алгоритма построения объекта морфинга из исходной и целевой сеток.

Листинг 3.3 — Реализация алгоритма построения мофринга из исходной и целевой сеток

```
impl Morph {
    pub fn new(source_object: TriangleMesh, target_object:
       TriangleMesh) -> Result<Self, String> {
        // 1. Параметризация исходных сеток
        let mut parametrized_source_mesh = source_object.clone();
        parametrize_mesh(&mut parametrized_source_mesh);

        let mut parametrized_target_mesh = target_object.clone();
        parametrize_mesh(&mut parametrized_target_mesh);

        // 2. Построение суперсетки
```

```
        let (vertices, triangles) =
            create_supermesh(&parametrized_source_mesh, &
               parametrized_target_mesh)?;

        // 3. Находим положения точек на исходной и целевой сетка
           x
        let src_vertices = relocate_vertices_on_mesh(
            &vertices,
            &parametrized_source_mesh,
            source_object.vertices_world(),
        );
        let dst_vertices = relocate_vertices_on_mesh(
            &vertices,
            &parametrized_target_mesh,
            target_object.vertices_world(),
        );

        let src_normals = find_normals(
            &vertices,
            &triangles,
            &parametrized_source_mesh,
            source_object.normals(),
        );
        let dst_normals = find_normals(
            &vertices,
            &triangles,
            &parametrized_target_mesh,
            target_object.normals(),
        );

        // 4. Строим интерполяции
        let vertex_interpolations: Vec<VertexInterpolation> =
           src_vertices
            .into_iter()
            .zip(dst_vertices.into_iter())
            .map(|(src_v, dst_v)| -> VertexInterpolation {
                Box::new(move |t: f64| Point::from((1. - t) *
                   src_v.coords + t * dst_v.coords))
            })
            .collect();
```

```
let normals_interpolations: Vec<NormalInterpolation> =
   src_normals
    .into_iter()
    .zip(dst_normals.into_iter())
    .map(|(src_n, dst_n)| -> NormalInterpolation {
        Box::new(move |t: f64| lerp(src_n, dst_n, t))
    })
    .collect();

let src_material = source_object.material().clone();
let dst_material = target_object.material().clone();
let material_interpolation: MaterialInterpolation =
    Box::new(move |t: f64| Material::lerp(&src_material,
       &dst_material, t));

// 5. Строим интерполяции при t=0
// 5.1 Строим вершины
let vertices: Vec<Point> = vertex_interpolations.iter().
   map(|lerp| lerp(0.)).collect();
let vertices_world = vertices.clone();

// 5.2 Строим нормали
let normals: Vec<Vector4<f64>> =
    normals_interpolations.iter().map(|lerp| lerp(0.)).
       collect();
let normals_world = normals.clone();

// 5.3 Строим материал
let material = material_interpolation(0.);

Ok(Morph {
    vertices,
    vertices_world,
    triangles,
    normals,
    normals_world,
    material,
    vertex_interpolations,
    normals_interpolations,
```

```
            material_interpolation,
```

На листингах 3.4–3.8 представлены реализации алгоритмов основных этапов построения морфинга.

Листинг 3.4 — Реализация алгоритма сферической параметризации сетки

```
pub fn parametrize_mesh(mesh: &mut TriangleMesh) {
    let inner_point = find_inner_point(mesh).unwrap();
    for v in mesh.vertices_world_mut() {
        *v = Point3::from((v.coords - inner_point.coords).
           normalize());
    }

    let vertices_world = mesh.vertices_world();
    let original_orientations = izip!(mesh.triangles(), mesh.
       normals())
        .map(|(tri, normal)| {
            let origin = vertices_world[tri.0].coords - normal.
               xyz();
            let v0 = vertices_world[tri.0].coords - origin;
            let v1 = vertices_world[tri.1].coords - origin;
            let v2 = vertices_world[tri.2].coords - origin;
            v0.cross(&v1).dot(&v2).signum()
        })
        .collect();

    relax_mesh(mesh, &original_orientations);

    mesh.vertices = mesh.vertices_world().clone();
    mesh.model_matrix = Matrix4::identity();
}
```

Листинг 3.5 — Реализация алгоритма релаксации сетки на единичной сфере

```
fn relax_mesh(parametrized_mesh: &mut TriangleMesh,
   original_orientations: &Vec<f64>) {
    let epsilon_threshold = 1e-2;

    let neighbors = collect_neighbors(parametrized_mesh);

    // Релаксация сетки
    let mut orientations = get_orientations(
```

```
        parametrized_mesh.vertices_world(),
        parametrized_mesh.triangles(),
    );
    let mut orientations_established = original_orientations.iter
       ().eq(orientations.iter());
    let mut epsilon_reached = true;
    let mut round_no: usize = 0;

    while (!(orientations_established && epsilon_reached)) &&
       round_no < RELAXATION_ROUNDS_LIMIT {
        // 1. Сохраняем положение вершин перед релаксацией
        let prev_vertices = parametrized_mesh.vertices_world().
           clone();

        // 2. Выполняем один раунд релаксации
        {
            let vertices = parametrized_mesh.vertices_world_mut()
               ;

            for i in 0..vertices.len() {
                let new_pos = neighbors[i]
                    .iter()
                    .map(|neighbor_idx| prev_vertices[*
                       neighbor_idx].coords)
                    .sum::<Vector3<f64>>()
                    .normalize();
                vertices[i] = Vertex::from(new_pos);
            }

            // Достигнут эпсилон-порог (вершины почти не сдвинули
               сь)
            epsilon_reached = prev_vertices
                .iter()
                .zip(vertices.iter())
                .all(|(prev, curr)| (prev - curr).norm() <
                   epsilon_threshold);

            // Центрирование сферы для избежания коллапса вершин
            let mean: Vector3<f64> =
                vertices.iter().map(|v| v.coords).sum::<Vector3<
```

```
                f64>>() / vertices.len() as f64;
            vertices.iter_mut().for_each(|v| *v -= mean);
        }

        // Главное условие остановки: Ориентации граней совпадают
            с оригинальными (нет вывернутых граней)
        orientations = get_orientations(
            parametrized_mesh.vertices_world(),
            parametrized_mesh.triangles(),
        );
        orientations_established = original_orientations.iter().
           eq(orientations.iter());

        round_no += 1;
    }

    println!("{}", round_no);
}
```

Листинг 3.6 — Реализация алгоритма релаксации сетки на единичной сфере

```
fn relax_mesh(parametrized_mesh: &mut TriangleMesh,
   original_orientations: &Vec<f64>) {
    let epsilon_threshold = 1e-2;

    let neighbors = collect_neighbors(parametrized_mesh);

    // Релаксация сетки
    let mut orientations = get_orientations(
        parametrized_mesh.vertices_world(),
        parametrized_mesh.triangles(),
    );
    let mut orientations_established = original_orientations.iter
       ().eq(orientations.iter());
    let mut epsilon_reached = true;
    let mut round_no: usize = 0;

    while (!(orientations_established && epsilon_reached)) &&
       round_no < RELAXATION_ROUNDS_LIMIT {
        // 1. Сохраняем положение вершин перед релаксацией
        let prev_vertices = parametrized_mesh.vertices_world().
```

```
        clone();

    // 2. Выполняем один раунд релаксации
    {
        let vertices = parametrized_mesh.vertices_world_mut()
            ;

        for i in 0..vertices.len() {
            let new_pos = neighbors[i]
                .iter()
                .map(|neighbor_idx| prev_vertices[*
                    neighbor_idx].coords)
                .sum::<Vector3<f64>>()
                .normalize();
            vertices[i] = Vertex::from(new_pos);
        }

        // Достигнут эпсилон-порог (вершины почти не сдвинули
            сь)
        epsilon_reached = prev_vertices
            .iter()
            .zip(vertices.iter())
            .all(|(prev, curr)| (prev - curr).norm() <
                epsilon_threshold);

        // Центрирование сферы для избежания коллапса вершин
        let mean: Vector3<f64> =
            vertices.iter().map(|v| v.coords).sum::<Vector3<
                f64>>() / vertices.len() as f64;
        vertices.iter_mut().for_each(|v| *v -= mean);
    }

    // Главное условие остановки: Ориентации граней совпадают
        с оригинальными (нет вывернутых граней)
    orientations = get_orientations(
        parametrized_mesh.vertices_world(),
        parametrized_mesh.triangles(),
    );
    orientations_established = original_orientations.iter().
        eq(orientations.iter());
```

```
        round_no += 1;
    }

    println!("{}", round_no);
}
```

Листинг 3.7 — Реализация алгоритма построения суперсетки (часть 1)

```
        let mut has_very_long_edges = false;
        for i in 0..vertex_indices.len() {
            let v1_idx = vertex_indices[i];
            let v2_idx = vertex_indices[(i + 1) % vertex_indices.
               len()];
            let v1 = &dcel.vertices[v1_idx];
            let v2 = &dcel.vertices[v2_idx];

            // Вычисляем длину дуги
            let dot = v1.coords.dot(&v2.coords).clamp(-1.0, 1.0);
            let arc_length = dot.acos();

            // Если длина дуги > 90 градусов, это подозрительно д
               линное ребро
            if arc_length > std::f64::consts::FRAC_PI_2 {
```

Листинг 3.8 — Реализация алгоритма построения суперсетки (часть 2)

```
    let (mut all_vertices, mapping_a, mapping_b) =
       create_unified_vertex_map(mesh_a, mesh_b);

    println!(
        "{} {} {}",
        mesh_a.vertices.len(),
        mesh_b.vertices.len(),
        all_vertices.len()
    );

    // 2. Получаем сегменты из обеих сеток с правильными индексам
       и
    let segments_a: Vec<Segment> = get_mesh_segments(mesh_a)
        .into_iter()
        .map(|s| {
```

```
        let mut mapped_segment = [mapping_a[s[0]], mapping_a[
            s[1]]];
        mapped_segment.sort_unstable();
        mapped_segment
    })
    .collect();

let segments_b: Vec<Segment> = get_mesh_segments(mesh_b)
    .into_iter()
    .map(|s| {
        let mut mapped_segment = [mapping_b[s[0]], mapping_b[
            s[1]]];
        mapped_segment.sort_unstable();
        mapped_segment
    })
    .collect();

// 3. Ассоциативный массив для хранения всех вершин, которые
   лежат на каждом отрезке
let mut segment_map: HashMap<Segment, HashSet<usize>> =
   HashMap::new();

// Добавляем все отрезки в ассоциативный массив
for &s in &segments_a {
    segment_map.entry(s).or_insert_with(HashSet::new);
}
for &s in &segments_b {
    segment_map.entry(s).or_insert_with(HashSet::new);
}

// 4. Находим вершины, которые лежат на рэбрах другой сетки
// Проверяем вершины сетки A на рэбрах сетки B
find_vertices_on_edges(&mapping_a, &segments_b, &all_vertices
   , &mut segment_map);

// Проверяем вершины сетки B на рэбрах сетки A
find_vertices_on_edges(&mapping_b, &segments_a, &all_vertices
   , &mut segment_map);

// 5. Находим точки пересечения между дугами
```

```
eprintln!("\n=== ПОИСК ПЕРЕСЕЧЕНИЙ ДУГ ===");
let mut intersection_count = 0;

for &seg_a in &segments_a {
    for &seg_b in &segments_b {
        // Пропускаем, если сегменты имеют общие вершины
        if seg_a[0] == seg_b[0]
            || seg_a[0] == seg_b[1]
            || seg_a[1] == seg_b[0]
            || seg_a[1] == seg_b[1]
        {
            continue;
        }

        let arc_1 = [&all_vertices[seg_a[0]], &all_vertices[
           seg_a[1]]];
        let arc_2 = [&all_vertices[seg_b[0]], &all_vertices[
           seg_b[1]]];

        if let Some(intersection_point) = intersect_arcs(
           arc_1, arc_2) {
            intersection_count += 1;

            // Проверяем, является ли точка пересечения одной
                из существующих вершин
            let mut is_existing = false;
            for (idx, v) in all_vertices.iter().enumerate() {
                if (v.coords - intersection_point.coords).
                   norm() < VERTEX_MATCH_EPS {
                    is_existing = true;
                    eprintln!("   Пересечение #{}: сегменты
                       [{}-{}] и [{}-{}] -> существующая v{}
                       ({:.6}, {:.6}, {:.6})",
                        intersection_count, seg_a[0], seg_a
                           [1], seg_b[0], seg_b[1],
                        idx, intersection_point.x,
                           intersection_point.y,
                           intersection_point.z);
                    break;
                }
```

```
                }

                if !is_existing {
                    let new_idx = all_vertices.len();
                    eprintln!("   Пересечение #{}: сегменты
                       [{}-{}] и [{}-{}] -> НОВАЯ точка v{}
                       ({:.6}, {:.6}, {:.6})",
                        intersection_count, seg_a[0], seg_a[1],
                           seg_b[0], seg_b[1],
                        new_idx, intersection_point.x,
                           intersection_point.y,
                           intersection_point.z);
                }

                let inter_idx = find_or_add_vertex(&mut
                   all_vertices, intersection_point);
                segment_map.get_mut(&seg_a).unwrap().insert(
                   inter_idx);
                segment_map.get_mut(&seg_b).unwrap().insert(
                   inter_idx);
            }
        }
    }

    eprintln!("Всего найдено пересечений: {}", intersection_count
       );
    eprintln!("=== КОНЕЦ ПОИСКА ПЕРЕСЕЧЕНИЙ ===\n");

    // 6. Генерируем финальный список подотрезков
    let mut all_segments: HashSet<Segment> = HashSet::new();

    for ([start_idx, end_idx], points_idx_set) in segment_map.
       into_iter() {
        let mut points_indices: Vec<usize> = points_idx_set.
           into_iter().collect();

        // Сортируем точки вдоль дуги на основе их расстояния от
           начальной точки
        let start_coords = all_vertices[start_idx].coords;
```

```
        points_indices.sort_unstable_by(|&a_idx, &b_idx| {
            let a_coords = all_vertices[a_idx].coords;
            let b_coords = all_vertices[b_idx].coords;

            let dist_a = (start_coords - a_coords).norm_squared()
                ;
            let dist_b = (start_coords - b_coords).norm_squared()
                ;
```

3.4 Интерфейс программы

На рисунках **??**, **??** представлен пример работы программы.

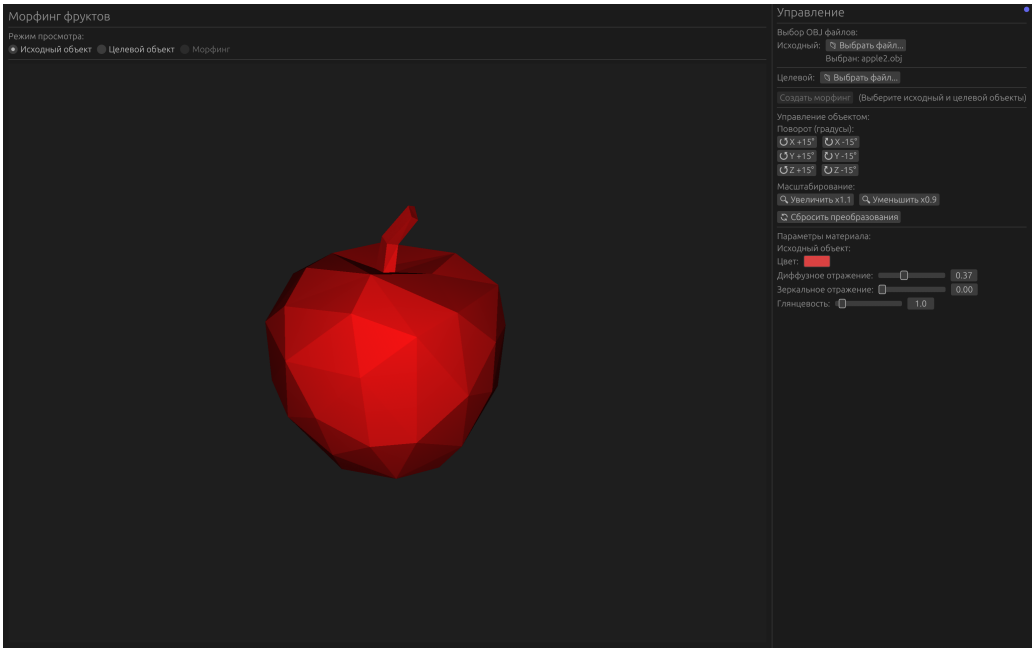

Рисунок 3.1 — Пример графического интерфейса программы — управление исходным объеком

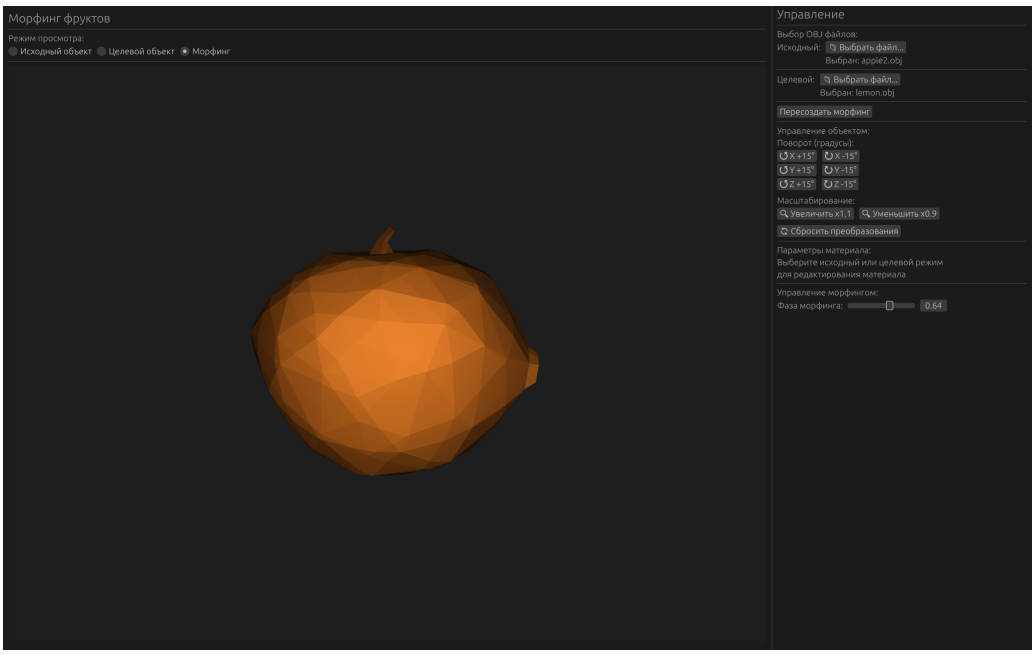

Рисунок 3.2 — Пример графического интерфейса программы — управления морфингом

Для загрузки объектов используются кнопки «Загрузить исходный объект» и «Загрузить целевой объект».

После того как объекты загружены, становятся доступна кнопка «Создать морфинг», по нажатию на которую выполняется построение объекта морфинга. После того, как объект морфинга построен, становится доступна вкладка «Морфинг».

На вкладках «Исходный объект» и «Целевой объект» выполняется просмотр и изменение параметры соответствующих объектов. Пример приведен на рисунке **??**.

На вкладке «Морфинг» можно управлять стадией морфинга, путем изменения параметра «Фаза морфинга» и просматривать результат.

Поворот и масштабирование объектов выполняется с помощью мыши: зажатие левой кнопки мыши и движение мыши выполняет поворот объекта, а прокрутка колесика мыши выполняет масштабирование объекта. Либо при помощи соответствующих кнопок на боковой панели.

Вывод

В этом разделе были описаны средства реализации, представлены реализации основных алгоритмов, продемонстрирован интерфейс программы.

4 Исследовательская часть

4.1 Вывод

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Mocanu Bogdan-Costel. 3D Mesh Morphing. Thèse de doctorat / Doctoral thesis: TELECOM SudParis; Universitatea Politehnica Bucuresti. France / Roumanie, 2013. Numéro national de thèse : 2013TELE0010; HAL Id: tel-00836048.

2. Порев В. Компьютерная графика. СПб.: БХВ-Петербург., 2002.

3. Alexa M. Mesh Morphing. 2001.

4. Роджерс Д. Алгоритмические основы машинной графики: Пер. с англ. — СПб: БХВ-Петербург, 1989. — С. 512.

5. Foley J. Computer Graphics: Principles and Practice. 3 edition. Addison Wesley, 2013.

6. The Rust Programming Language. URL: https://doc.rust-lang.org/book/ (дата обращения: 26.10.2025).

7. egui - Rust. URL: https://docs.rs/egui/latest/egui/ (дата обращения: 26.10.2025).

8. RustRover: IDE для Rust. URL: https://www.jetbrains.com/ru-ru/rust/ (дата обращения: 27.10.2025).