СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Морфинг трехмерных объектов представляет собой технику анимации, при котором объект плавно переходит в другой так, что человеческий глаз не распознает момент, когда заканчивается первый и начинается второй. С помощью морфинга создают плавные анимационные эффекты в различных областях, включая кинопроизводство, рекламу, игры и дизайн.

Целью данной работы является разработка программы для морфинга объектов. Для достижения поставленной цели требуется решить следующие задачи:

1) проанализировать алгоритмы морфинга трехмерных моделей;
2) выбрать наиболее подходящий для достижения поставленной цели;
3) выбрать средства реализации программы;
4) разработать программу и реализовать выбранные алгоритмы;
5) реализовать графический интерфейс;
6) исследовать временные характеристики выбранного алгоритма морфинга объектов на основе созданной программы.

1 Аналитический раздел

В данном разделе представлен анализ существующих алгоритмов построения изображений и выбор подходящих алгоритмов для решения задачи.

1.1 Формализация объектов сцены

Сцена состоит из следующих объектов:

— объект (стартовый или получаемый) — трехмерная модель, представляющая собой каркасное тело;

— источник света — вектор направления света;

— камера — характеризуется своим положением и направлением просмотра.

1.2 Анализ способов описания трехмерных моделей

В компьютерной графике в основном используют три типа представления трехмерных объектов [1]: каркасная, поверхностная и твердотельная модели. Они предоставляют различные способы представления объектов и позволяют достичь правильного отображения их формы и размеров на сцене.

Каркасная модель

Каркасная модель это простейший вид моделей. В этой модели задается информация о вершинах и ребрах объектов. Этим видам модели присущ весьма существенный недостаток: не всегда модель правильно передает представление об объекте. К преимуществам же можно отнести сравнительно низкое число затрачиваемой памяти.

Поверхостная модель

Поверхностные модели несут информацию обо всех точках пространства, принадлежащих поверхности объекта, а внутренние точки в них не учитываются. К недостаткам данной модели относится отсутствие информации о том, с какой стороны поверхности находится материал.

Твердотельная модель

Твердотельная модель отличается от поверхностной наличием информации о том, где расположен материал. Проще всего это можно сделать путем указания направления внутренней нормали. [1]

Для решения поставленной задачи выбрана каркасная модель, так как для осуществления морфинга необходимо знать информацию только о вершинах и ребрах объекта. Хранить остальную информацию об объекте избыточно.

1.3 Анализ алгоритмов удаления невидимых поверхностей

Алгоритмы удаления невидимых линий и поверхностей определяют, какие линии, поверхности или объемы видимы или невидимы для наблюдателя, находящегося в определенной точке пространства.

Алгоритм Робертса

Алгоритм Робертса удаляет из каждого тела те ребра или грани, которые скрываются 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, скрываются этими телами. При этом вычислительная трудоемкость алгоритма Робертса растет теоретически, как квадрат числа объектов.

Алгоритм плавающего горизонта

Алгоритм плавающего горизонта чаше всего используется для удаления невидимых линий трехмерного представления функций, описывающих поверхность в виде $F(x, y, z) = 0$.

Идея метода заключается в сведении трехмерной задачи к двумерной путем пересечения исходной поверхности последовательностью параллельных секущих плоскостей, имеющих постоянные значения координаты z.

Алгоритм сначала упорядочивает плоскости $z = const$ по возрастанию расстояния до них от точки наблюдения. Затем для каждой плоскости, начиная с ближайшей к точке наблюдения, строится кривая, лежащая на ней, т.е. для каждого значения координаты в пространстве изображения определяется соответствующее значение y.

Если на текущей плоскости при некотором заданном значении x соответствующее значение y на кривой больше значения y для всех предыдущих кривых при этом значении x, то текущая кривая видима в этой точке; в противном случае она невидима. [2]

Алгоритм, использующий z-буфер

Алгоритм, использующий z-буфер, работает в пространстве изображения. Идея z-буфера является простым обобщением идеи о буфере кадра. Буфер кадра используется для запоминания атрибутов (интенсивности) каждого пикселя в пространстве изображения, z-буфер — это отдельный буфер глубины, используемый для запоминания координаты z или глубины каждого видимого пикселя в пространстве изображения. В процессе работы глубина или значение z каждого нового пикселя, который нужно занести в буфер кадра, сравнивается с глубиной того пикселя, который уже занесен в z-буфер. Если это сравнение показывает, что новый пиксель расположен впереди пикселя, находящегося в буфере кадра, то новый пиксель заносится в этот буфер и, кроме того, производится корректировка z-буфера новым значением z. Если же сравнение дает противоположный результат, то никаких действий не производится. По сути, алгоритм является поиском по x и y наибольшего значения функции $z(x, y)$.

Поскольку габариты пространства изображения фиксированы, оценка вычислительной трудоемкости алгоритма не более чем линейна. Так как элементы сцены или картинки можно заносить в буфер кадра или в z-буфер в произвольном порядке, их не нужно предварительно сортировать по приоритету глубины. Поэтому экономится вычислительное время, затрачиваемое на сортировку по глубине.

К недостаткам алгоритма относятся большой объем требуемой памяти и в высокой стоимости устранения лестничного эффекта, а также реализации эффектов прозрачности и просвечивания. Поскольку алгоритм заносит пиксели в буфер кадра в произвольном порядке, то нелегко получить информацию, необходимую для методов устранения лестничного эффекта, основывающихся на предварительной фильтрации. При реализации эффектов прозрачности и просвечивания пиксели могут заноситься в буфер кадра в некорректном порядке, что ведет к локальным ошибкам. [3]

Таблица 1.1 – Сравнение алгоритмов удаления невидимых поверхностей

Алгоритм	**Асимптотика**	**Типы объектов**
Алгоритм Робертса	$O(n^2)$, где n - число объектов	Выпуклые многогранники
Алгоритм плавающего горизонта	$O(x \cdot z)$, где $x = x_{max} - x_{min}$ $z = z_{max} - z_{min}$	Вида $F(x, y, z) = 0$
z-буфер	$O(w \cdot h)$, где w — ширина экрана h — высота экрана	Произвольные

Среди рассмотренных алгоритмов [2; 3], для поставленной задачи больше всего подходит алгоритм z-буфера, так как он может работать со сценами любой сложности и не требует больших вычислительных мощностей для сцен с множеством объектов, что необходимо при реализации морфинга.

1.4 Анализ алгоритмов закраски

Для придания реалистичности изображениям кроме удаления невидимых линий и поверхностей необходимо правильно отразить свет от присутствующих на сцене объектов. Существует несколько основных методов закрашивания изображаемых поверхностей. Среди них можно выделить [4]:

— метод однотонной закраски;

— метод, основанный на интерполяции значений интенсивности освещенности поверхности — метод Гуро;

— метод, основанный на интерполяции векторов нормалей к граням многогранника — метод Фонга.

Однотонная закраска

При однотонной закраски для изображаемого объекта вычисляется один уровень интенсивности освещения, который и используется для закраски всего объекта. Следует заметить, что, если каждая плоская грань имеет один постоянный цвет, то различные цвета соседних граней очень заметны, и поверхность выглядит как многогранник. Этот эффект можно сгладить, если увеличить число рассматриваемых граней. Так как зрение человека

имеет способность подчеркивать перепады яркости на границах смежных граней (эффект Маха), для обеспечения иллюзии гладкой поверхности изображения нужно намного увеличить количество таких граней, а это приведет к снижению эффективности всей вычислительной процедуры.

Закраска по методу Гуро

Метод Гуро основывается на идее закрашивания каждой грани не одним цветом, а плавно изменяющимися оттенками, вычисляемыми путем интерполяции цветов примыкающих граней. Закрашивание граней по методу Гуро осуществляется в четыре этапа:

1) вычисляются нормали к поверхности;

2) определяются нормали в вершинах многогранника путем усреднения нормали по всем полигональным граням, которым принадлежит вершина;

3) используя нормали в вершинах и применяя произвольный метод закраски, вычисляют значения интенсивности освещения в вершинах;

4) каждый многоугольник закрашивают путем линейной интерполяции значений интенсивности в вершинах, сначала вдоль каждого ребра, а затем и между ребрами вдоль каждой сканирующей строки.

Закраска по методу Фонга

В методе закраски, разработанном Фонгом, используется интерполяция вектора нормали к поверхности вдоль видимого интервала на сканирующей строке внутри многоугольника, а не интерполяция интенсивности. Интерполяция выполняется между начальной и конечной нормалями, которые сами тоже являются результатами интерполяции вдоль ребер многоугольника между нормалями в вершинах. Нормали в вершинах, в свою очередь, вычисляются так же, как в методе закраски, построенном на основе интерполяции интенсивности. [4]

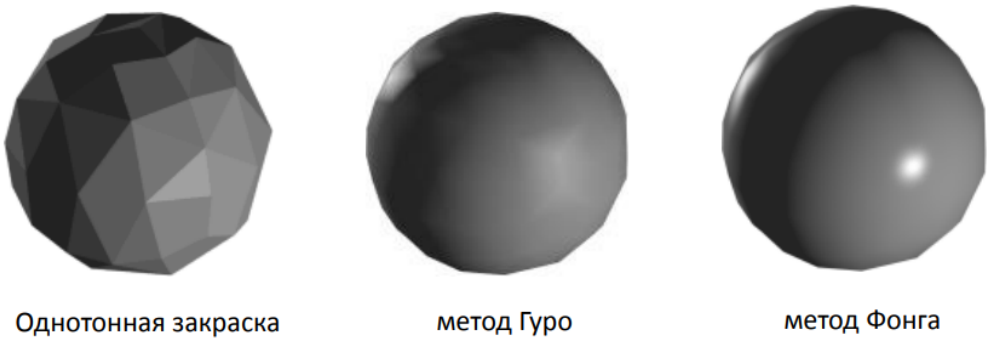

Рисунок 1.1 – Визуальное сравнение различных алгоритмов закраски

Наиболее подходящей закраской для поставленной задачи является закраска Гуро, так как программа подразумевает обработку моделей с большим количеством примитивов.

1.5 Анализ алгоритмов освещения

К основным моделям освещения относятся модели Ламберта и Фонга [5], построенные на основе трехкомпонентного освещения.

В общем виде модель освещения Фонга состоит из суммы фоновой, диффузной и зеркальной составляющей и имеет следующий вид:

$$I = I_a + I_d + I_s = m_a \cdot L_a + m_d \cdot k_d \cdot L_d + m_s \cdot k_s \cdot L_s \tag{1.1}$$

В общем виде модель освещения Ламберта состоит из суммы фоновой и диффузной компонент:

$$I = I_a + I_d = m_a \cdot L_a + m_d \cdot k_d \cdot L_d \tag{1.2}$$

Модель Ламберта является одной из самых простых моделей освещения. Данная модель очень часто используется как часть других моделей, поскольку практически в любой другой модели освещения можно выделить диффузную составляющую. Более-менее равномерная часть освещения (без присутствия какого-либо всплеска), как правило, будет представляться моделью Ламберта с определенными характеристиками. Данная модель может быть очень удобна для анализа свойств других моделей (за счет того, что ее легко выделить из любой модели и анализировать оставшиеся составляющие).

В частности, модель Ламберта является существенной частью модели Фонга, которая представляет собой комбинацию диффузной составляющей (модели Ламберта) и зеркальной составляющей. [5]

Наилучшим решением для поставленной задачи — модель освещения Ламберта.

1.6 Анализ алгоритмов морфинга

Среди алгоритмов морфинга можно выделить три алгоритма: линейный, слиянием и д-морфинг.

Линейный морфинг

Линейный морфинг основан на линейной интерполяции между начальными и конечными позициями каждой точки или вершины модели. Алгоритм состоит из следующих этапов:

1) выравнивание вершин. Чтобы линейный морфинг работал корректно, у исходной и конечной моделей должны быть одинаковое количество вершин и морфологическая структура. Обычно это требует предварительного выравнивания вершин;

2) интерполяция позиций вершин. Процесс начинается с вычисления позиций каждой вершины на промежуточных этапах между исходной и конечной моделями. Используется линейная интерполяция, чтобы постепенно изменять координаты вершин по оси x, y и z:

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

где P_t — позиция вершины на промежуточном этапе, P_0 — позиция вершины в исходной модели, P_1 — позиция вершины в конечной модели, а t — параметры интерполяции, принимающие значения от 0 до 1;

3) текстурирование и цвета. Наряду с изменением позиций вершин может также потребоваться интерполяция текстурных координат и цветов вершин, чтобы добиться качественного визуального эффекта;

4) анимация и кадры. Для создания анимации между двух моделей процесс интерполяции выполняется для каждого промежуточного кадра. Значение t увеличивается от 0 до 1 по мере прогресса анимации, что создает плавный переход.

Данный алгоритм плохо справляется с задачей плавного перехода, если число вершин между исходной и итоговой моделями сильно разнится. [6]

Морфинг слиянием

Алгоритм морфинга слиянием предполагает «надувание» исходного и итогового объектов до сферы. Оболочка каждого объекта проецируется на единичную сферу. Это отображение используется для идентификации соответствий между точками на двух исходных объектах путем связывания пар точек, которые отображаются в одно и то же место на сфере. Из этого этапа следует требование к объектам – Затем две топологии объединяются путем обрезки спроецированных граней одной модели спроецированными гранями другой. Объединенная топология затем отображается на поверхность обеих исходных моделей. Это создает две новые модели, которые имеют ту же форму, что и исходные две модели, но которые разделяют общую топологию. Это позволяет легко вычислить преобразование между двумя формами путем интерполяции координат каждой пары соответствующих вершин. [7]

Д-морфинг

Д-морфинг разделяется на два этапа: построение описывающих фигур и генерация морфинга.

На первом этапе строятся два новых объекта: исходная фигура, измененная с помощью паралельных переносов плоскостей, полностью включающая в себя итоговую, и наоборот. Чтобы получить морфинг, мы просто перемещаем все плоскости одновременно из их начальных положений в их конечные положения. Форма, образованная их пересечением, будет кадром морфинга. [8]

Для поставленной задачи лучше всего подходит морфинг слиянием. Такой выбор обусловлен тем, что линейный морфинг хуже работает с объектами, имеющими разное количество вершин, а д-морфинг позволяет обрабатывать только выпуклые объекты.

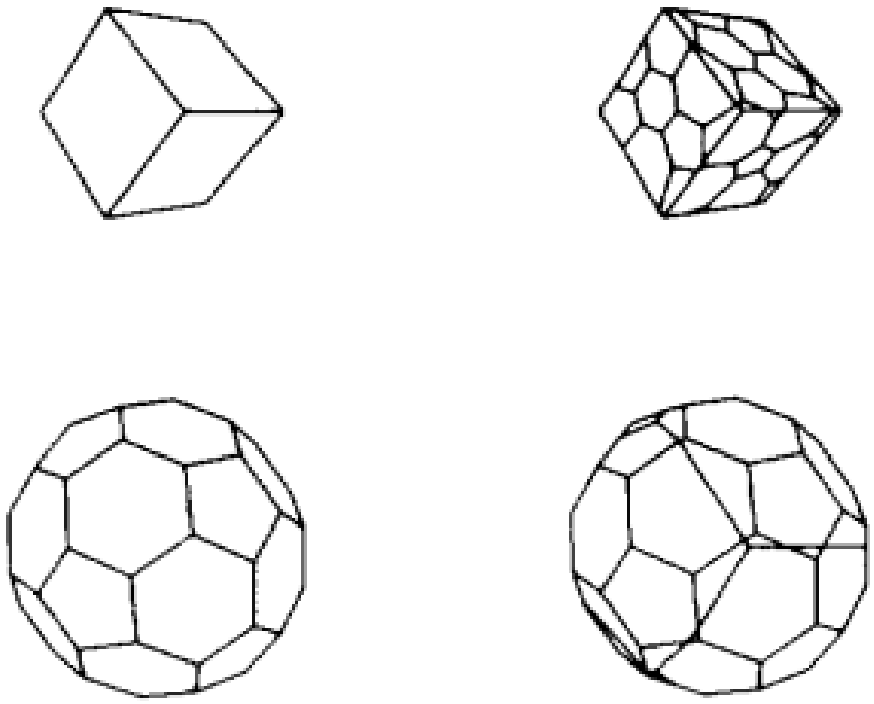

Рисунок 1.2 – Пара исходных моделей и та же пара с объединенными топологиями

Таблица 1.2 – Сравнение алгоритмов морфинга

Алгоритм	**Разность количества вершин**	**Типы многогранников**
Линейный морфинг	Чем больше, тем хуже качество	Произвольные
Морфинг слиянием	Не важна	Практически произвольные
Д-морфинг	Не важна	Выпуклые

Вывод

Были выбраны следующие алгоритмы:

— алгоритм, использующий z-буфер для удаления невидимых поверхностей;

— закраска по методу Гуро и модель Ламберта в качестве алгоритма закраски и модели освещения;

— морфинг слиянием.

2 Конструкторский раздел

2.1 Требования к программе

Программа должна предоставлять пользователю следующую функциональность:

— загрузка объектов в программу в формате .obj;

— выбор цвета загружаемых объектов;

— отображение загруженных объектов;

— возможность просмотра объектов, путем масштабирования, перемещения и поворота;

— перемещение источника света;

— морфинг.

Программа должна корректно реагировать на любые действия пользователя.

2.2 Разработка алгоритмов

2.2.1 Z-буфер

На рисунке 2.1 приведена схема алгоритма, использующего z-буфер.

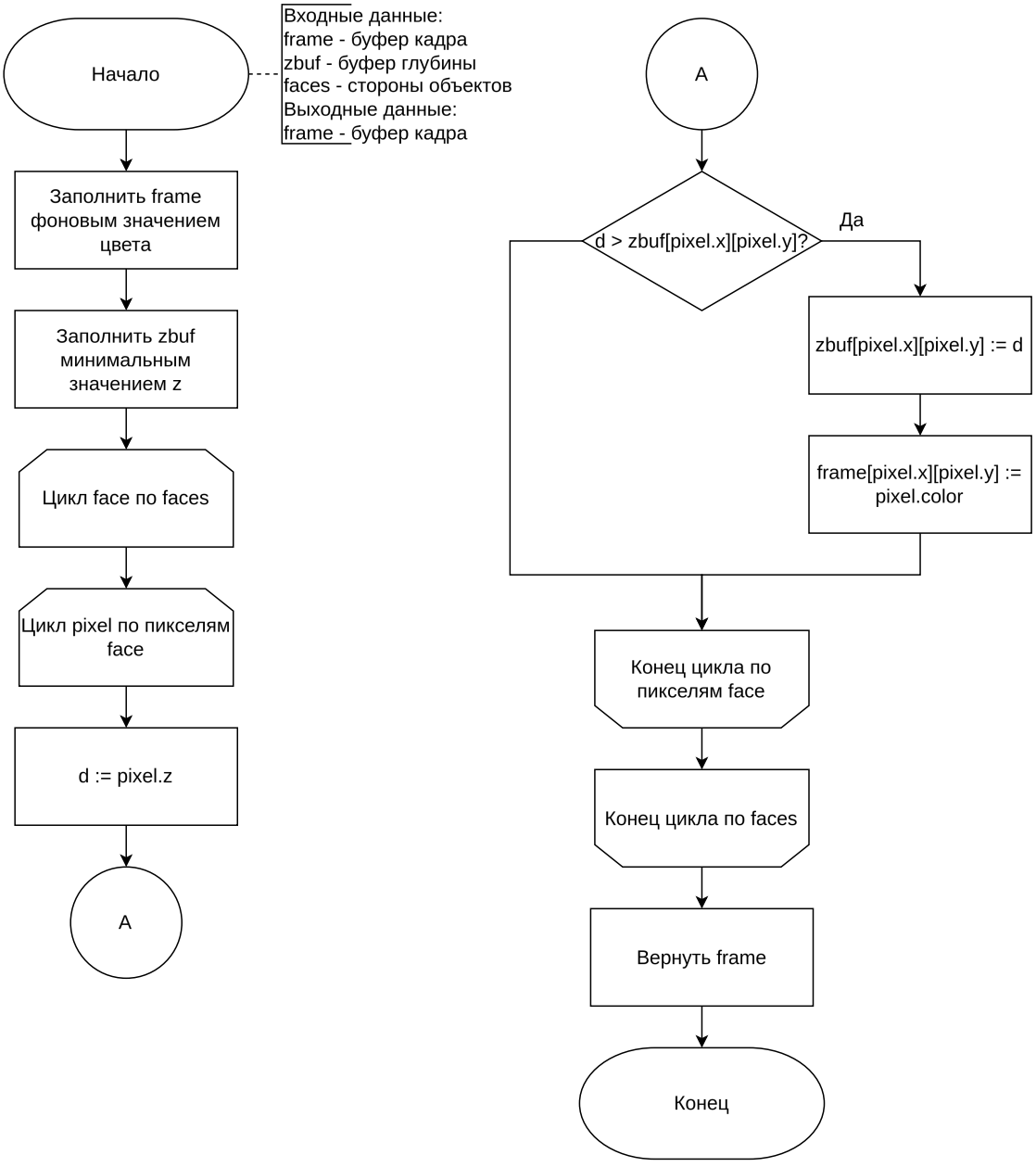

Рисунок 2.1 – Схема алгоритма, использующего z-буфер

2.2.2 Закраска по методу Гуро

На рисунке 2.2 приведена схема алгоритма закраски по методу Гуро.

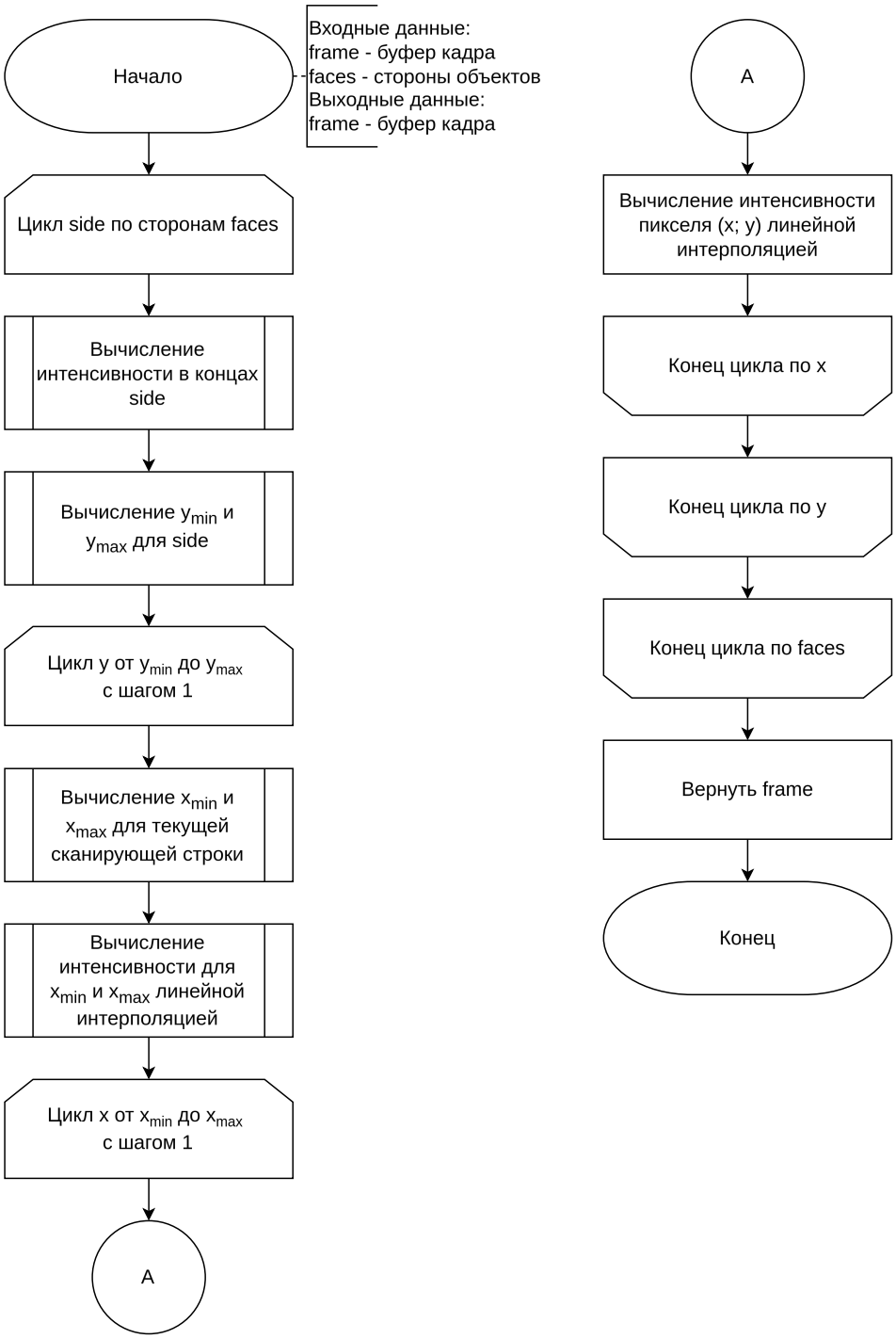

Рисунок 2.2 – схема алгоритма закраски по методу Гуро

2.2.3 Морфинг слиянием

На рисунках 2.3, 2.4 и 2.4 приведена схема алгоритма морфинга слиянием.

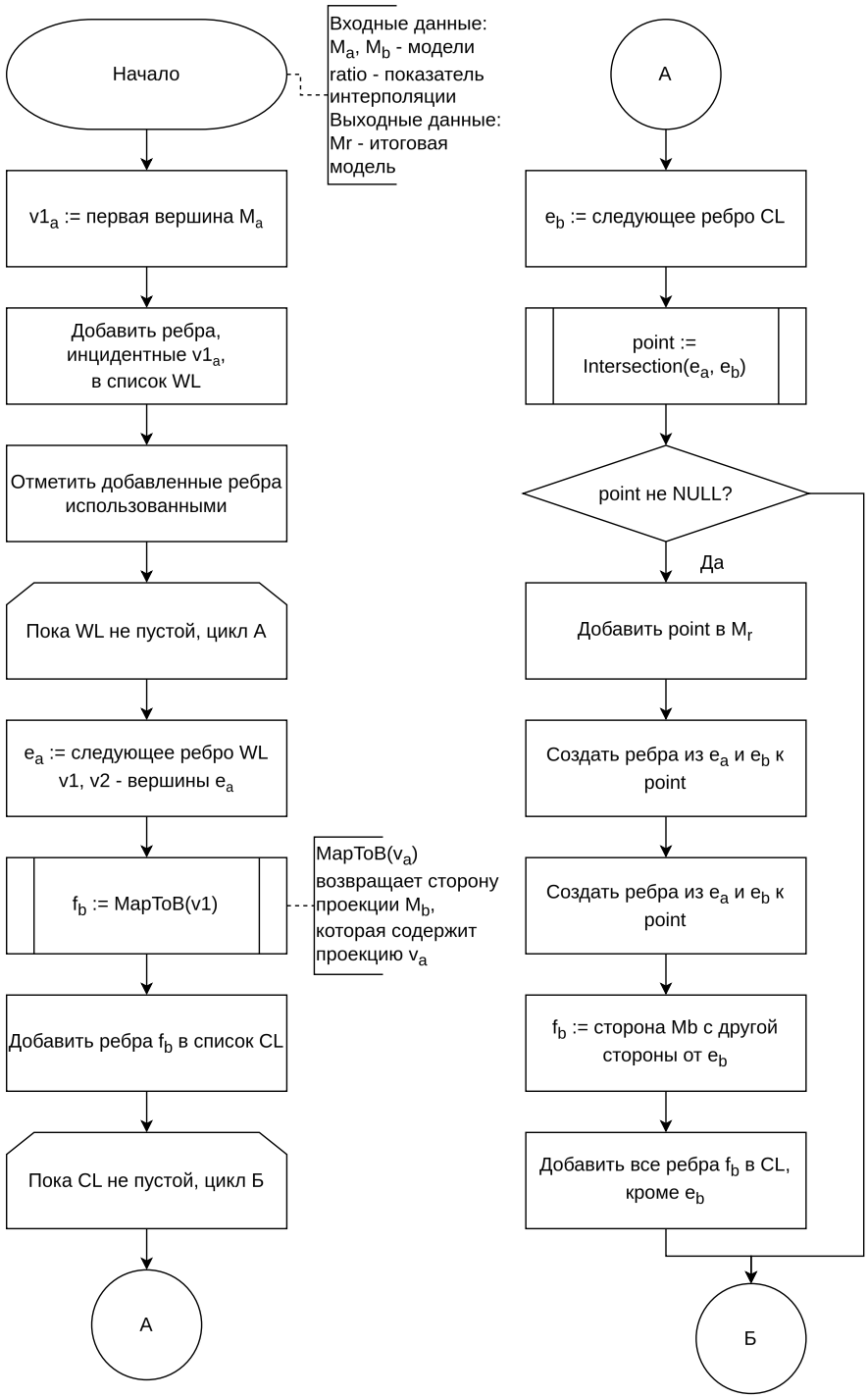

Рисунок 2.3 – схема алгоритма морфинга слиянием, часть 1

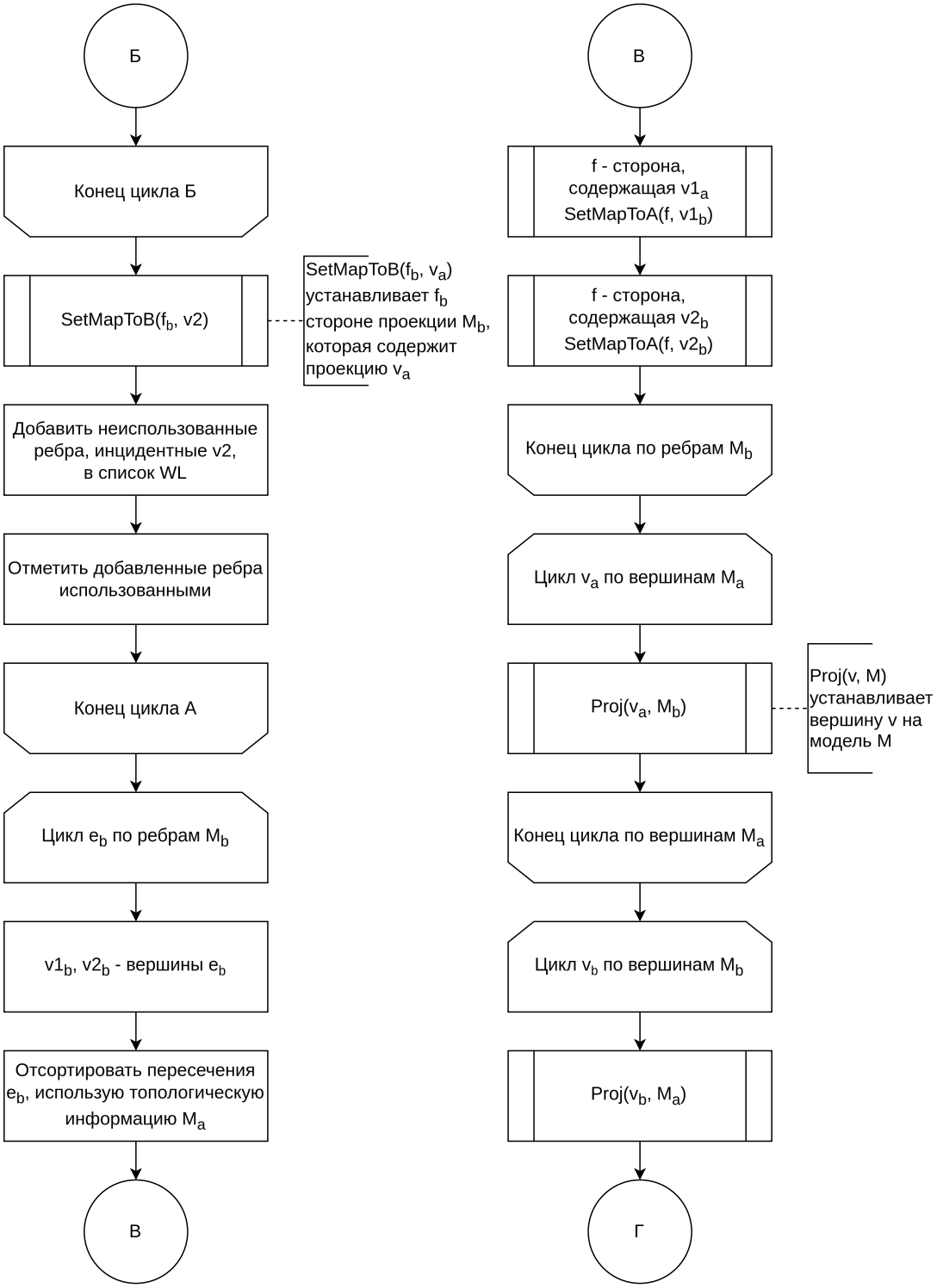

Рисунок 2.4 – схема алгоритма морфинга слиянием, часть 2

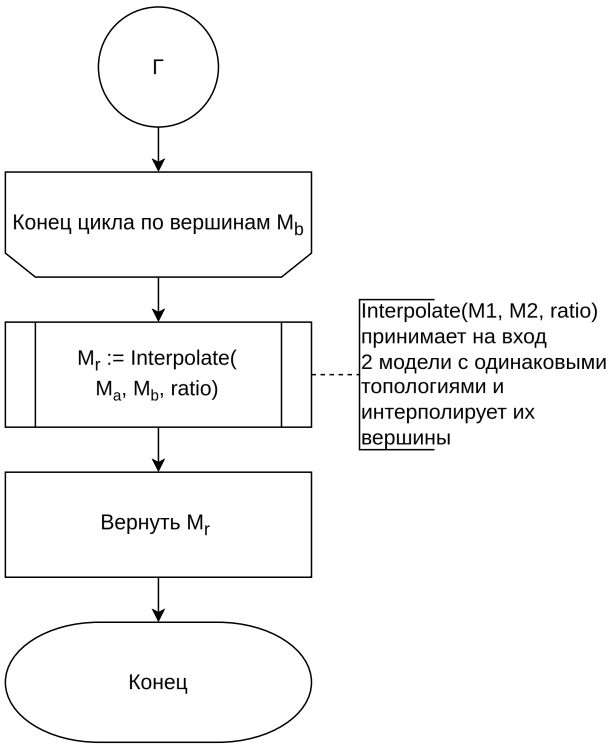

Рисунок 2.5 – схема алгоритма морфинга слиянием, часть 3

Вывод

Были описаны алгоритмы, использующего z-буфер, закраски по Гуро и морфинга слиянием. Было выдвинуты требования к программе.

3 Технологический раздел

3.1 Средства реализации

В качестве языка программирования был выбран *Rust*. Данный выбор был обусловлен следующими факторами:

— средствами языка можно реализовать все алгоритмы, выбранные в результате проектирования;

— *Rust* обладает высокой производительностью. [9]

3.2 Реализация алгоритмов

В листингах 3.1, 3.2 и 3.3 приведена реализация алгоритма z-буфера, совмещенного с закраской по Гуро.

Листинг 3.1 – Реализация алгоритма z-буфера, совмещенного с закраской по Гуро, часть 1

```
pub fn draw_object(&mut self, object: &Object, mut
   light_direction: Vertex) {
    light_direction.normalize();
    for face_ind in 0..object.nfaces() {
        let world_coords = object.face_coords(face_ind);
        let mut intensities = vec![];
        for normal in object.face_normals(face_ind) {
            intensities.push(light_direction * normal);
        }
        let mut screen_coords = vec![];
        for i in 0..3 {
            screen_coords.push((
                world_coords[i].world_to_screen(self.height,
                   self.width),
                intensities[i],
            ));
        }
        self.draw_triangle(screen_coords, object.color());
    }
}
```

Листинг 3.2 – Реализация алгоритма z-буфера, совмещенного с закраской по Гуро, часть 2

```
fn draw_triangle(&mut self, mut coords: Vec<(Vertex, f64)>,
   color: Color) {
    if coords[0].0.y == coords[1].0.y && coords[1].0.y ==
       coords[2].0.y {
        return;
    }
    coords.sort_by(|a, b|
       a.0.y.partial_cmp(&b.0.y).expect("draw_triangle:
       sorting"));
    let total_height = (coords[2].0.y - coords[0].0.y) as i32;
    for i in 0..total_height {
        let is_second_half =
            (i > (coords[1].0.y - coords[0].0.y) as i32) ||
               (coords[1].0.y == coords[0].0.y);
        let segment_height = if is_second_half {
            coords[2].0.y - coords[1].0.y
        } else {
            coords[1].0.y - coords[0].0.y
        };
        let alpha = i as f64 / total_height as f64;
        let mut a_side_intensity = coords[0].1 + (coords[2].1 -
           coords[0].1) * alpha;
        let beta = if is_second_half {
            (i as f64 - coords[1].0.y + coords[0].0.y) /
               segment_height
        } else {
            i as f64 / segment_height
        };
        let mut a = coords[0].0 + (coords[2].0 - coords[0].0) *
           alpha;
        let mut b = if is_second_half {
            coords[1].0 + (coords[2].0 - coords[1].0) * beta
        } else {
            coords[0].0 + (coords[1].0 - coords[0].0) * beta
        };
        let mut b_side_intensity = if is_second_half {
            coords[1].1 + (coords[2].1 - coords[1].1) * beta
        } else {
```

Листинг 3.3 – Реализация алгоритма z-буфера, совмещенного с закраской по Гуро, часть 3

```
            coords[0].1 + (coords[1].1 - coords[0].1) * beta
        };
        a.round();
        b.round();
        if a.x > b.x {
            swap(&mut a, &mut b);
            swap(&mut a_side_intensity, &mut b_side_intensity);
        }
        for j in a.x as i32..=b.x as i32 {
            let phi = if a.x == b.x {
                1.
            } else {
                (j as f64 - a.x) / (b.x - a.x)
            };
            let mut p = a + (b - a) * phi;
            p.round();
            let idx = (p.x as u32 + p.y as u32 * self.width) as
               usize;
            if idx < self.zbuffer.len() && self.zbuffer[idx] <
               p.z {
                self.zbuffer[idx] = p.z;
                let p_int = a_side_intensity + (b_side_intensity
                   - a_side_intensity) * phi;
                let mut cur_color = color.clone();
                cur_color.set_rgb(
                    (cur_color.r() as f64 * p_int) as u8,
                    (cur_color.g() as f64 * p_int) as u8,
                    (cur_color.b() as f64 * p_int) as u8,
                );
                self.set_pixel(p.x.round() as u32, p.y.round()
                   as u32, cur_color);
            }
        }
    }
}
```

В листингах 3.4, 3.5, 3.6, 3.7 и 3.8 приведена реализация алгоритма морфинга слиянием.

Листинг 3.4 – Реализация алгоритма морфинга слиянием, часть 1

```
pub fn new(src_proj: Projection, dst_proj: Projection) ->
   Result<Self, ()> {
    let n = src_proj.nvertexes();
    let m = dst_proj.nvertexes();
    let mut sphere_vertexes = Vec::with_capacity(n + m);
    let mut edges = EdgeSet::new();
    for i in 0..n {
        sphere_vertexes.push(SphereVertex {
            vertex: src_proj.sphere_vertex(i),
            index: i,
            origin_id: 1,
        });
    }
    for i in 0..m {
        sphere_vertexes.push(SphereVertex {
            vertex: dst_proj.sphere_vertex(i),
            index: i,
            origin_id: 2,
        });
    }
    for src_edge in src_proj.edges_iter() {
        edges.insert(src_edge);
    }
    for dst_edge in dst_proj.edges_iter() {
        let dst_edge = Edge::new(dst_edge.from + n, dst_edge.to
           + n);
        let v1 = sphere_vertexes[dst_edge.from].vertex;
        let v2 = sphere_vertexes[dst_edge.to].vertex;
        let dst_arc = Arc::new(v1, v2, dst_edge.from,
           dst_edge.to);
        let mut intersections = vec![(0., dst_edge.from), (1.,
           dst_edge.to)];
        let mut is_skip_add = false;
        for src_edge in edges.clone().iter() {
```

Листинг 3.5 – Реализация алгоритма морфинга слиянием, часть 2

```
            let u1 = sphere_vertexes[src_edge.from].vertex;
            let u2 = sphere_vertexes[src_edge.to].vertex;
            let src_arc = Arc::new(u1, u2, src_edge.from,
               src_edge.to);
            match Arc::intersect(&src_arc, &dst_arc) {
                ArcIntersectionResult::T1(index, k) =>
                   intersections.push((k, index)),
                ArcIntersectionResult::T2(index, ..) => {
                    edges.remove(src_edge);
                    edges.add(src_edge.from, index);
                    edges.add(src_edge.to, index);
                }
                ArcIntersectionResult::X(vertex, k) => {
                    let id = sphere_vertexes.len();
                    sphere_vertexes.push(SphereVertex {
                        vertex,
                        origin_id: 0,
                        index: 0,
                    });
                    edges.remove(src_edge);
                    edges.add(src_edge.from, id);
                    edges.add(src_edge.to, id);
                    intersections.push((k, id));
                }
                ArcIntersectionResult::I((id1, k1), (id2, k2))
                   => {
                    edges.remove(src_edge);
                    if k1 > 0. {
                        intersections.push((k1, id1));
                    } else if k1 < 0. {
                        edges.add(id1, intersections[0].1);
                    }
                    if k2 < 1. {
```

Листинг 3.6 – Реализация алгоритма морфинга слиянием, часть 3

```
                        intersections.push((k2, id2));
                    } else if k2 > 1. {
                        edges.add(id2, intersections[1].1);
                    }
                }
                ArcIntersectionResult::L(id1, id2) => {
                    if id2 == dst_edge.from {
                        intersections[0].1 = id1;
                    } else if id2 == dst_edge.to {
                        intersections[1].1 = id1;
                    }
                }
                ArcIntersectionResult::S => {
                    is_skip_add = true;
                    break;
                }
                ArcIntersectionResult::N => {}
            }
        }
        if is_skip_add {
            continue;
        }
        intersections.sort_by(|a, b|
           a.partial_cmp(b).unwrap_or(Ordering::Equal));
        for i in 0..intersections.len() - 1 {
            edges.add(intersections[i].1, intersections[i +
               1].1);
        }
    }
    let mut vertexes_pairs = Vec::new();
    let mut normals_pairs = Vec::new();
    let mut vp = (Vertex::default(), Vertex::default());
```

Листинг 3.7 – Реализация алгоритма морфинга слиянием, часть 4

```
    let mut np = (Vertex::default(), Vertex::default());
    for vertex in sphere_vertexes.iter() {
        match vertex.origin_id {
            1 => {
                let dst =
                   dst_proj.project_from_sphere(vertex.vertex)?;
                vp = (src_proj.vertex(vertex.index), dst.0);
                np = (src_proj.normal(vertex.index), dst.1);
            }
            2 => {
                let src =
                   src_proj.project_from_sphere(vertex.vertex)?;
                vp = (src.0, dst_proj.vertex(vertex.index));
                np = (src.1, dst_proj.normal(vertex.index));
            }
            _ => {
                let src =
                   src_proj.project_from_sphere(vertex.vertex)?;
                let dst =
                   dst_proj.project_from_sphere(vertex.vertex)?;
                vp = (src.0, dst.0);
                np = (src.1, dst.1);
            }
        };
        vp.0 -= *src_proj.center();
        vp.1 -= *dst_proj.center();
        vertexes_pairs.push(vp);
        normals_pairs.push(np);
    }
    let sphere_vertexes = sphere_vertexes.iter().map(|v|
       v.vertex).collect();
    let faces = Self::resolve_faces(&sphere_vertexes, &edges);
    let mut triangle_faces = Vec::new();
    let mut set =
       std::collections::BTreeSet::<Vec<usize>>::new();
```

Листинг 3.8 – Реализация алгоритма морфинга слиянием, часть 5

```
    for f in faces.into_iter() {
        if f.len() > 3 {
            for i in 1..f.len() - 1 {
                let mut tri = vec![f[0], f[i], f[i + 1]];
                tri.sort();
                if f[0] == f[i] || f[0] == f[i + 1] {
                    continue;
                }
                if set.insert(tri.clone()) {
                    Self::adjust_order(&mut tri,
                       &sphere_vertexes, Vertex::new(0.0, 0.0,
                       0.0));
                    triangle_faces.push(tri);
                }
            }
        } else {
            let mut tri = f;
            tri.sort();
            if set.insert(tri.clone()) {
                Self::adjust_order(&mut tri, &sphere_vertexes,
                   Vertex::new(0.0, 0.0, 0.0));
                triangle_faces.push(tri);
            }
        }
    }
    Ok(Self {
        vertexes_pairs,
        normals_pairs,
        faces: triangle_faces,
        color_pairs: (src_proj.color().clone(),
           dst_proj.color().clone()),
    })
}
```

3.3 Интерфейс программы

На рисунках 3.1 и 3.2 представлен интерфейс программы.

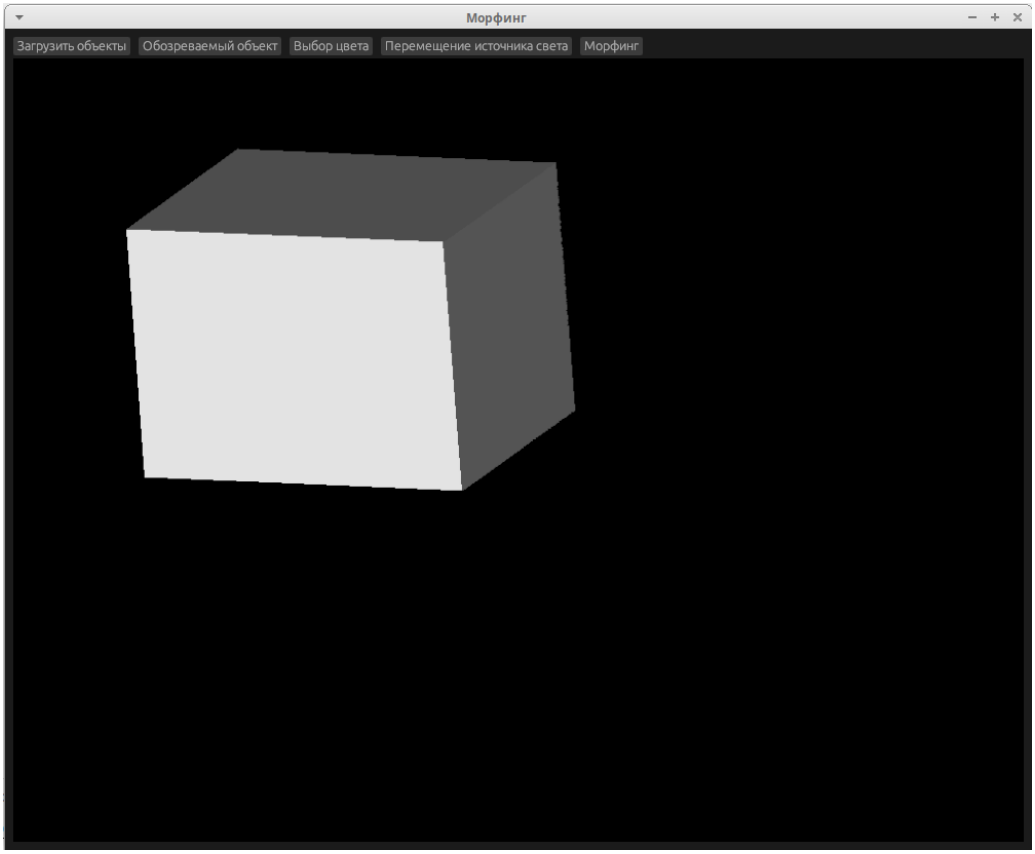

Рисунок 3.1 – Общий вид программы

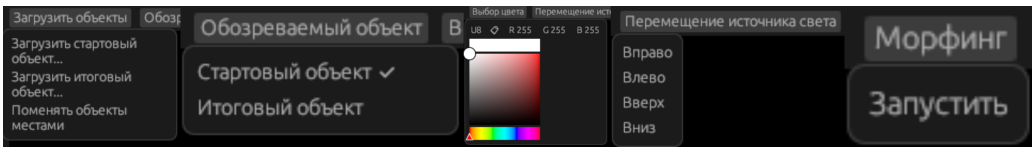

Рисунок 3.2 – Внутренние меню программы

Во вкладке «Загрузить объекты» пользователь загружает 2 объекта в формате *.obj*, цвета которых предварительно выбираются во вкладке «Выбор цвета», а также имеет возможность поменять местами стартовый и итоговый объекты.

Во вкладке «Обозреваемый объект» выбирается объект для его просмотра и преобразования: перемещения, поворота и масштабирования.

Вывод

Программа была реализована.

4 Исследовательский раздел

4.1 Технические характеристики

Технические характеристики используемого оборудования:

— операционная система: Ubuntu 22.04.4 LTS;

— оперативная память: 16 ГБ;

— процессор: Intel(R) Core(TM) i5-10300H.

4.2 Временные замеры

Для проведения временных замеров использовалась библиотека *Criterion*. Количество замеров для каждого набора входных данных — 100.

Так как время выполнения алгоритма зависит от количества ребер и пересечений их проекций на сфере, замеры проводились для двух случаев:

1) стартовый и конечный объекты одинаковые;

2) стартовый и конечный объекты разные.

Таблица 4.1 – Результаты временных замеров алгоритма морфинга слиянием

Количество ребер	**Время, с**	
	Одинаковые объекты	**Разные объекты**
408	0.001	0.005
1338	0.008	0.236
2268	0.017	0.489
3198	0.024	0.707
4128	0.036	0.868
5368	0.206	1.342
6608	0.341	1.670
7848	0.557	2.298

На рисунке 4.1 представлены результаты временных замеров алгоритма морфинга слиянием.

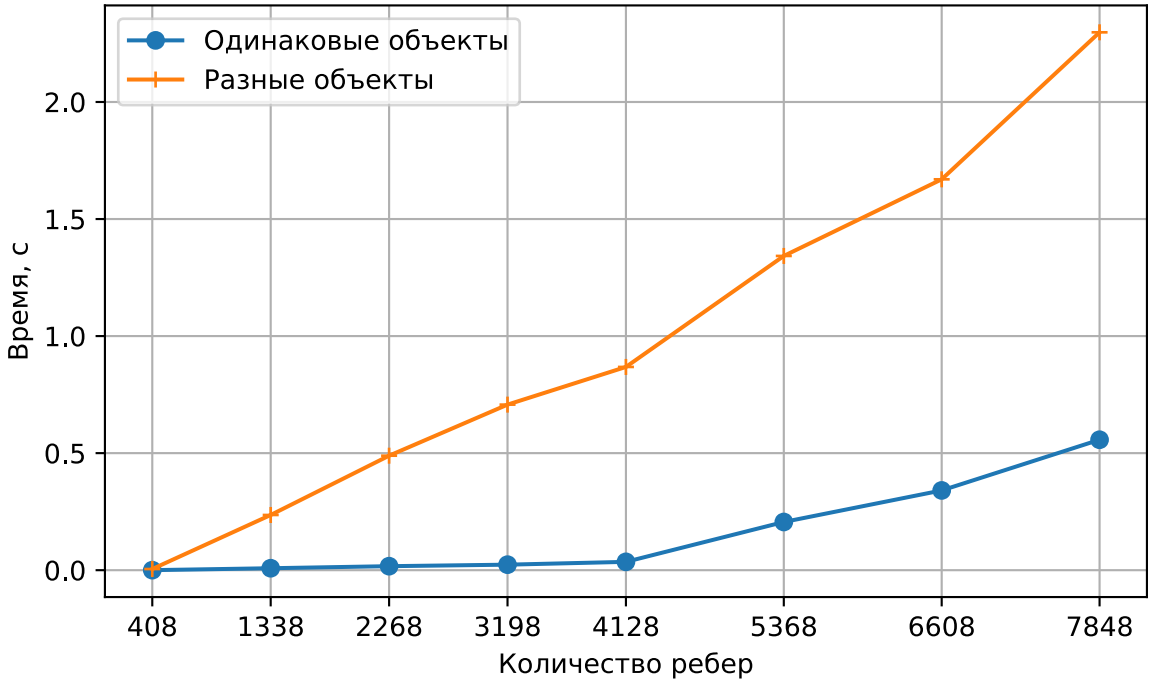

Рисунок 4.1 – Результаты временных замеров алгоритма морфинга слиянием

С увеличением количества ребер и количества пересечений их проекций, время выполнения растет. На 7848 ребрах морфинг с нулем пересечений ребер (с одинаковыми объектами) выполнялся в среднем за 0.557 с, а с некоторым ненулевым числом пересечений за 2.298 с, то есть примерно в 4.13 раз медленнее.

На 1338 ребрах морфинг с разными объектами выполнялся 0.236 с, а на 7848 — 2.298 с, то есть время выполнения растет быстрее, чем количество вершин.

Вывод

В данном разделе были проведены исследования по замеру времени морфинга объектов в зависимости от наличия пересечений ребер и количества их.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были решены следующие задачи:

1) проанализированы алгоритмы морфинга трехмерных моделей;
2) выбраны наиболее подходящий для достижения поставленной цели;
3) выбраны средства реализации программы;
4) разработана программа и реализованы выбранные алгоритмы;
5) реализован графический интерфейс;
6) исследованы временные характеристики алгоритма морфинга слиянием на основе созданной программы.

Цель данной лабораторной работы, а именно разработка программы для морфинга объектов была достигнута. По результатам анализа временных характеристик получили, что на скорость работы морфинга слиянием влияют количество ребер и пересечений их проекций на сфере. Так, при сумме ребер 7848 двух объектов морфинга, время нахождения их слияния составило 2.298 секунд.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Косников Ю. Н.* Поверхостные модели в системах трехмерной компьютерной графики. — Пензенский государственный университет, 2007.
2. Алгоритмы удаления невидимых линий и поверхностей [Электронный ресурс]. — URL: `https://astro.tsu.ru/KGaG/text/5_6.html` (дата обращения: 10.9.2024).
3. Алгоритм, использующий z-буфер [Электронный ресурс]. — URL: `https://compgraph.tpu.ru/zbuffer.htm` (дата обращения: 30.9.2024).
4. *Роджерс Д.* Алгоритмические основы машинной графики. — Мир, 1989.
5. *Задорожный А. Г.* Модели освещения и алгоритмы затенения в компьютерной графике. — Новосибирский государственный технический университет, 2020.
6. *Дивеев Д. А.*, *Хозе Е. Г.* Современные технологии трансформации изображений в изучении восприятия человека по выражению его лица. — 2009.
7. *Кент Д.*, *Карлсон У.*, *Парент Р.* Shape Transformation for Polyhedral Objects. — 1992.
8. *Гласснер А.* DMorph. — 2003.
9. *Блэнди Д.*, *Орендорф Д.* Программирование на языке Rust. — ДМК Пресс, 2018.

ПРИЛОЖЕНИЕ А