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 КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Морфинг трехмерных моделей»

Студент ИУ7-54Б (Группа) _____ (Подпись, дата) Писаренко Д. П. (И. О. Фамилия)

Руководитель курсовой работы _____ (Подпись, дата) Кузнецова О. В. (И. О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Морфинг трехмерных моделей представляет собой технику анимации в компьютерной графике, направленную на плавное переходное изменение одной трехмерной модели в другую. Этот процесс создает эффект плавного метаморфоза между двумя или более формами, что может быть использовано для создания динамичных и визуально привлекательных анимаций.

Морфинг может применяться в различных областях, таких как компьютерная анимация, визуализация данных, медицинская графика и игровая индустрия. Эта техника позволяет создавать плавные и естественные анимации, предоставляя художникам и разработчикам инструмент для трансформации объектов внутри виртуальных сцен.

Целью данной работы является разработка программного обеспечения для морфинга трехмерных моделей. Для достижения поставленной цели требуется решить следующие задачи.

1) Формализовать представление объектов и описать их.
2) Проанализировать алгоритмы морфинга трехмерных моделей и выбрать наилучшие для достижения цели.
3) Выбрать средства реализации алгоритмов.
4) Реализовать выбранные алгоритмы.
5) Реализовать графический интерфейс.
6) Исследовать временные характеристики выбранных алгоритмов на основе созданного программного обеспечения.

1 Аналитический раздел

В данном разделе производится формализация объектов сцены, анализ алгоритмов их визуализации, и выбираются наиболее подходящие для решения поставленных задач.

1.1 Формализация объектов сцены

На визуализируемой сцене могут находиться следующие объекты:

1) **Точечный источник света**

   Данный источник света испускает его равномерно во всех направлениях из фиксированной точки в трехмерном пространстве. Источник характеризуется:

   — интенсивностью;

   — позицией [1].

2) **Рассеянный источник света**

   Привносит часть освещения в каждую точку сцены, независимо от ее расположения. Источник характеризуется:

   — интенсивностью [1].

3) **Камера**

   В данном случае камера будет описана:

   — матрицей проекции. Она определяет границы видимости в трехмерном пространстве;

   — позицией.

4) **Трехмерные объекты**

   Трехмерные объекты будут описываться либо с помощью каркасной модели (во время морфинга), либо с помощью поверхностной модели (в статичном состоянии). Для этого задается:

   — набор вершин;

   — набор граней.

1.2 Способ задания моделей

Для универсальности нашей программы выберем один из самых популярных форматов - .obj файлы. Файлы формата .obj (Wavefront OBJ) представляют собой текстовый формат, используемый для хранения трехмерных графических моделей. Они могут содержать следующую информацию:

— **вершины** - координаты точек в трехмерном пространстве, которые определяют форму объекта;

— **нормали** - направления поверхности в каждой вершине, важные для расчета освещения;

— **текстурные координаты** - координаты, используемые для нанесения текстуры на поверхность модели;

— **грани** - определяют полигоны объекта, указывая индексы вершин, текстурных координат и нормалей, составляющих грани.

Поскольку мы будем использовать каркасные и поверхностные модели, из каждого файла нам нужно будет вычленять лишь вершины и грани. Таким образом, для определенного объекта будет составлен список его вершин, а также список граней, представляющих собой набор вершин.

1.3 Анализ алгоритмов морфинга

Морфинг — технология компьютерной графики, обеспечивающая плавный переход формы одного объекта в форму другого [2].

В последнее время наблюдается большой интерес к технологиям морфинга для создания плавных переходов между изображениями. Эти методы сочетают в себе 2D-интерполяцию формы и цвета для создания впечатляющих эффектов перехода. Отчасти привлекательность морфинга заключается в том, что создаваемые изображения могут выглядеть поразительно реалистичными и визуально убедительными. Несмотря на то, что они вычисляются с помощью преобразований 2D-изображений, эффективные морфинги могут предполагать естественную трансформацию объектов в 3D-мире.

1.3.1 Линейный морфинг

Линейный морфинг (англ. Linear Morphing) является самым простым и распространенным подходом. Он основан на линейной интерполяции между начальными и конечными позициями каждой точки или вершины модели. Путем изменения координат каждой точки по ходу времени, модель постепенно переходит из одной формы в другую [3].

Сам морфинг состоит из нескольких этапов:

1) Загрузка начальной и конечной модели морфинга, допускается несколько промежуточных состояний для плавности анимации.

2) Переход между позициями вершин.

3) Обновление текстурных координат.

4) Расчет промежуточных геометрических состояний.

Переход между позициями вершин — для каждой вершины модели вычисляются ее промежуточные позиции между начальной и конечной точками. Обычно это достигается путем линейной интерполяции координат вершин по временному параметру, который определяет текущий прогресс между началом и концом морфинга.

Обновление текстурных координат — аналогично с расчетом промежуточных позиций вершин необходимо с помощью интерполяции расчитать текстурные координаты.

Расчет промежуточных геометрических состояний — на основе расчитанных ранее координат возможно получить промежуточные геометрические состояния модели [3].

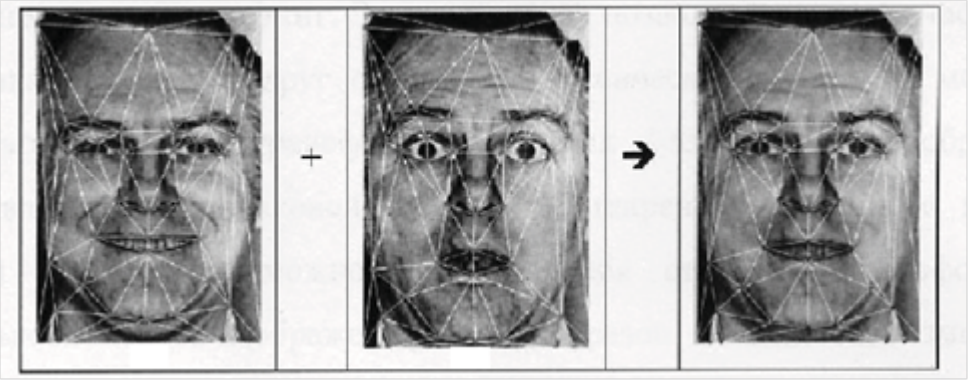

Рисунок 1.1 – Пример линейного морфинга человеческого лица

Плюсы:

1) Вычислительные затраты меньше, чем у других алгоритмов [4].

Минусы:

1) Ограниченные возможности для создания сложных форм и детализации.
2) Ограниченный контроль над динамическими атрибутами, такими как цвета или нормали.
3) Не всегда способен обеспечить реалистичный морфинг для моделей с большим количеством вершин или сложной геометрией [4].

1.3.2 Весовая деформация

Идея данного метода аналогична методу, описанному в 1.3, однако в данном случае каждая из вершин имеет собственный вес. Изначально каждая вершина присваивается какой-либо части тела объекта (кости). Каждая вершина модели привязывается к костям с помощью весов. Вес определяет, насколько каждая кость влияет на вершину. Чаще всего веса представлены числами от 0 до 1, где 0 означает полное отсутствие влияния, а 1 — полное влияние соответствующей кости. Весовые значения для каждой вершины нормализуются, так чтобы сумма весов для каждой вершины была равна 1. Это позволяет равномерно распределить влияние костей и избежать нежелательных артефактов. В процессе анимации, для каждого кадра, путем комбинирования позиций и ориентаций костей с их соответствующими весами, определяются новые позиции вершин. Это достигается путем линейной интерполяции или других методов комбинирования весовых значений и позиций костей [5].

Рисунок 1.2 – Иллюстрация веса вершины в зависимости от ее положения

На картинке 1.2 с помощью интенсивности красного цвета показывается значение веса вершины в зависимости от ее положения.

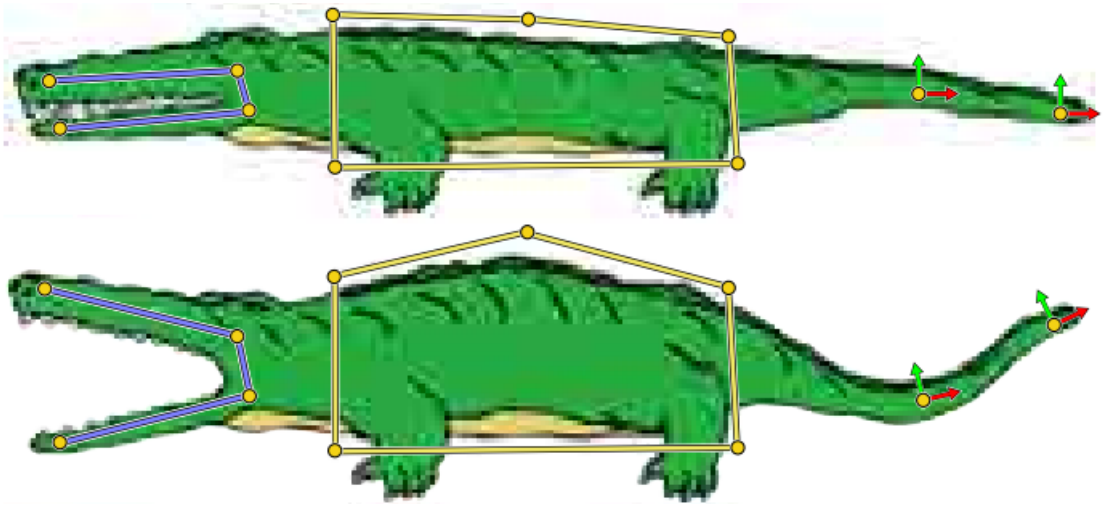

Рисунок 1.3 – Пример выбора вершин на скелете

Плюсы:

1) Гибкость в управлении деформацией и анимацией модели.
2) Возможность создания более сложных форм и эффектов деформации.
3) Мощный инструмент для контроля за определенными частями модели в процессе морфинга [4].

Минусы:

1) Дополнительная сложность в настройке и установке весовых значений для каждой вершины.
2) Проблемы с поддержкой ненатуральных или сложных деформаций.
3) Возможность создания артефактов, таких как рывки или изломы [4].

1.3.3 Морфинг на основе ключевых точек

При использовании данного метода морфинга выделяются несколько промежуточных этапов (точек) перехода одного объекта в другой. После чего, для каждого промежуточного этапа каждой вершине из предыдущего этапа ставится в соответствие вершина из последующего. Далее, для перехода из одного этапа в другой используется интерполяция.

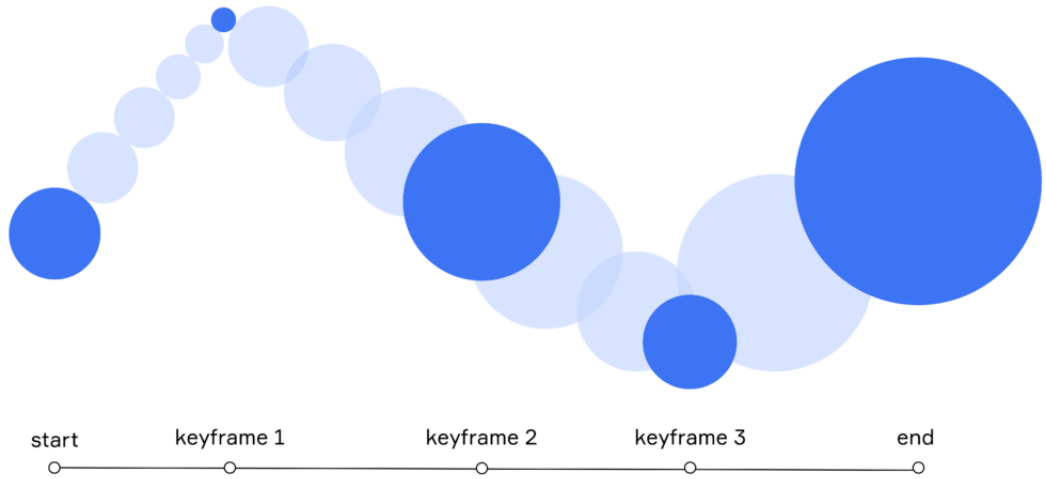

Рисунок 1.4 – Пример выделения ключевых точек

Плюсы:

1) Хорошая контролируемость и предсказуемость в анимации и морфинге моделей.

2) Возможность создания точных и выразительных анимаций с помощью набора ключевых точек.

3) Гибкость в управлении настройками и характеристиками каждой ключевой точки [4].

Минусы:

1) Необходимость тщательного планирования и размещения ключевых точек для достижения предполагаемого эффекта.

2) Ограничение в контроле над промежуточными состояниями между ключевыми точками.

3) Увеличение объема данных при использовании большого числа ключевых точек [4].

Вывод:

Проанализировав алгоритмы морфинга можно прийти к выводу, что наилучшим методом для решения поставленной задачи является линейный морфинг (англ. Linear morphing), так как он является наиболее быстродействующим из представленных алгоритмов, что играет значимую роль при большом количестве примитивов.

1.4 Анализ алгоритмов закраски

Для сглаживания и добавления реализма изображениям можно использовать алгоритмы закраски. В данном случае будут рассмотрены 3 основных алгоритма закраски:

1) Простая закраска.

2) Закраска методом Гуро.

3) Закраска методом Фонга [6].

1.4.1 Простая закраска

В случае использования простой закраски считается, что и источник света и наблюдатель находятся в бесконечности, так что диффузная составляющая одинакова (она зависит от угла падения), вектор наблюдения также будет для всех точек одинаковым, то есть зеркальная составляющая также не будет изменяться. Таким образом данный алгоритм является быстрым, однако в случае, если закрашиваемая грань является результатом аппроксимации тела, будет заметен резкий переход между интенсивностями [7].

Плюсы:

1) Простая реализация.

2) Время реализации меньше, чем у других алгоритмов [7].

Минусы:

1) Очень низкая реалистичность [7].

1.4.2 Закраска методом Гуро

В случае использования метода Гуро можно получить сглаженное изображение, для этого сначала определяется интенсивность вершин многоугольника, а затем с помощью билинейной интерполяции вычисляется интенсивность каждого пикселя на сканирующей строке.

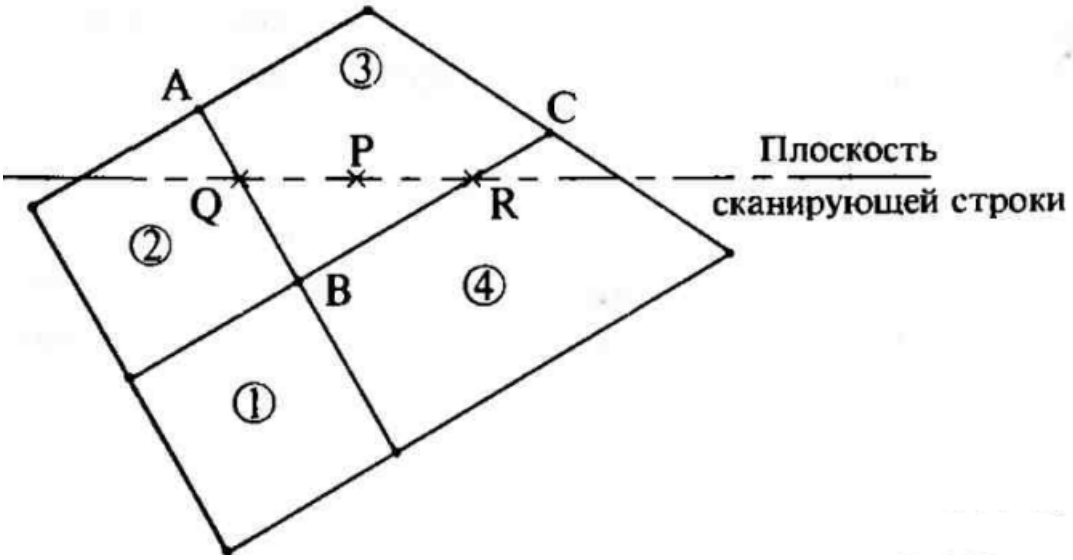

Рисунок 1.5 – Пример полигональной поверхности

Например, рассмотрим участок полигональной поверхности на рисунке 1.5. Значение интенсивности в точке P определяется линейной интерполяции значений интенсивностей в точках Q и R. Для получения интенсивности в точке Q можно провести линейную интерполяцию интенсивности в вершинах A и B по формуле

$$I_Q = uI_A + (1-u) * I_B \quad 0 \leq u \leq 1, t = \frac{AQ}{AB}. \tag{1.1}$$

Таким же образом рассчитывается интенсивность в точке B, после чего интерполяция используется еще раз, для поиска значения интенсивности в точке P [7].

$$I_P = tI_Q + (1-t) * I_R \quad 0 \leq t \leq 1, t = \frac{QP}{QR}. \tag{1.2}$$

Плюсы:

1) Получение сглаженного изображения.

2) Трудозатраты меньше, чем у любого другого алгоритма, кроме алгоритма простой закраски [7].

Минусы:

1) Появление полос Маха - оптической иллюзии. Она преувеличивает контраст между краями слегка отличающихся оттенков, в местах, где они соприкасаются друг с другом.

2) Не учитывает кривизны поверхности (случай, если нормали поверхностей одинаково ориентированы) [7].

1.4.3 Закраска методом Фонга

Закраска Фонга требует больших вычислительных затрат, однако она решает множество проблем метода Гуро. В данном методе вместо интерполяции интенсивностей света, производится интерполяция вектора нормали. Таким образом учитывается кривизна поверхности. В случае рисунка 1.5, нормали в соответствующих точках рассчитывалась бы формулами 1.3 [7].

$$\begin{aligned} n_Q &= un_A + (1-u)n_B \quad 0 \le u \le 1 \\ n_R &= wn_B + (1-w)n_C \quad 0 \le w \le 1 \\ n_P &= tn_Q + (1-t)n_R \quad 0 \le t \le 1. \end{aligned} \tag{1.3}$$

где

$$u = \frac{AQ}{AB}$$

$$w = \frac{BR}{BC} \tag{1.4}$$

$$t = \frac{QP}{QR}$$

Плюсы:

1) Получение сглаженного изображения.

2) Учет кривизны поверхностей.

3) Улучшенная реалистичность зеркальных бликов (по сравнению с методом Гуро) [7].

Минусы:

1) Трудозатратность алгоритма больше, чем у других алгоритмов [7].

Вывод:

На рисунке 1.6 представлены различные методы закраски. Заметим, что наиболее подходящей закраской для поставленной задачи является закраска Гуро, так как мы будем использовать модели с большим количеством примитивов и нам будет важна производительность. При этом вычислительных способностей достаточно, чтобы не использовать простую закраску.

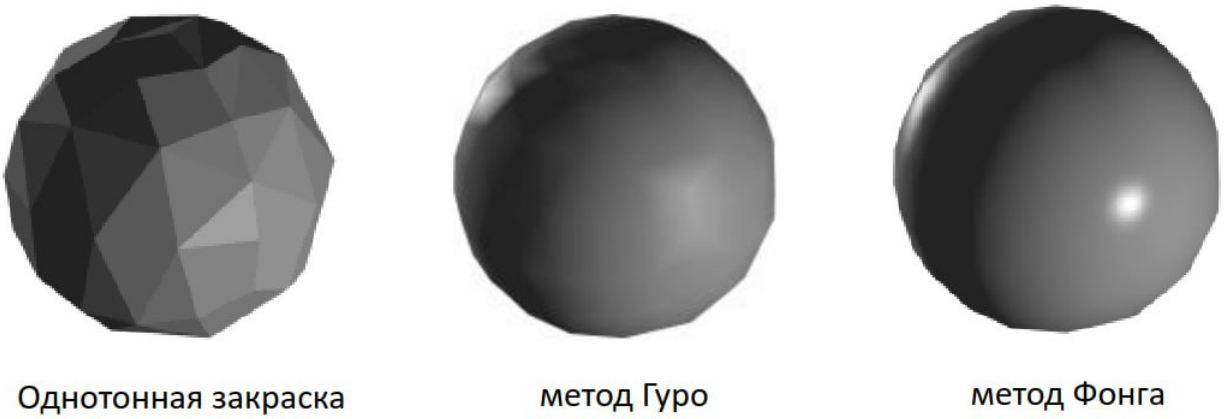

Рисунок 1.6 – Сравнение методов закраски

1.5 Анализ алгоритмов удаления невидимых линий и поверхностей

В рамках данной работы для визуализации процесса морфинга необходимо отсекать невидимые поверхности, для решения поставленной существует множество алгоритмов.

1.5.1 Алгоритм Робертса

Преимуществом этого алгоритма является его эффективное использование мощных и точных математических методов. На первом этапе происходит удаление ребер или граней, которые находятся за самим объектом, а затем рассматриваются грани, закрытые другими объектами на сцене. Если объекты пересекаются друг с другом, удаляются невидимые линии пересечения. Однако недостатком алгоритма является то, что вычислительная сложность растет квадратично с увеличением числа объектов, что может привести к замедлению работы алгоритма в случае большого числа объектов на сцене. Приведенный недостаток является критически важным для поставленно задачи, однако возможна модификация с использованием сортировки по оси Z, что улучшит производительность алгоритма [7].

1.5.2 Алгоритм Z-буфера

Данный алгоритм позволяет не только удалять невидимые линии и поверхности, но и визуализировать их пересечение. При работе данного алгоритма происходит поиск наиближайшего объекта к каждому пикселю экрана. Сцена может быть любой сложности, а так как размер изображения ограничен размером экрана, сложность алгоритма зависит линейно от количества поверхностей.

У этого алгоритма есть и существенные недостатки. Он требует больший объем памяти для своей реализации, так как необходимо хранить буфер кадра и Z-буфер. Кроме того, его сложно использовать для реализации эффектов, таких как прозрачность и освещение, ввиду отсутствия наблюдения за лучами света [7].

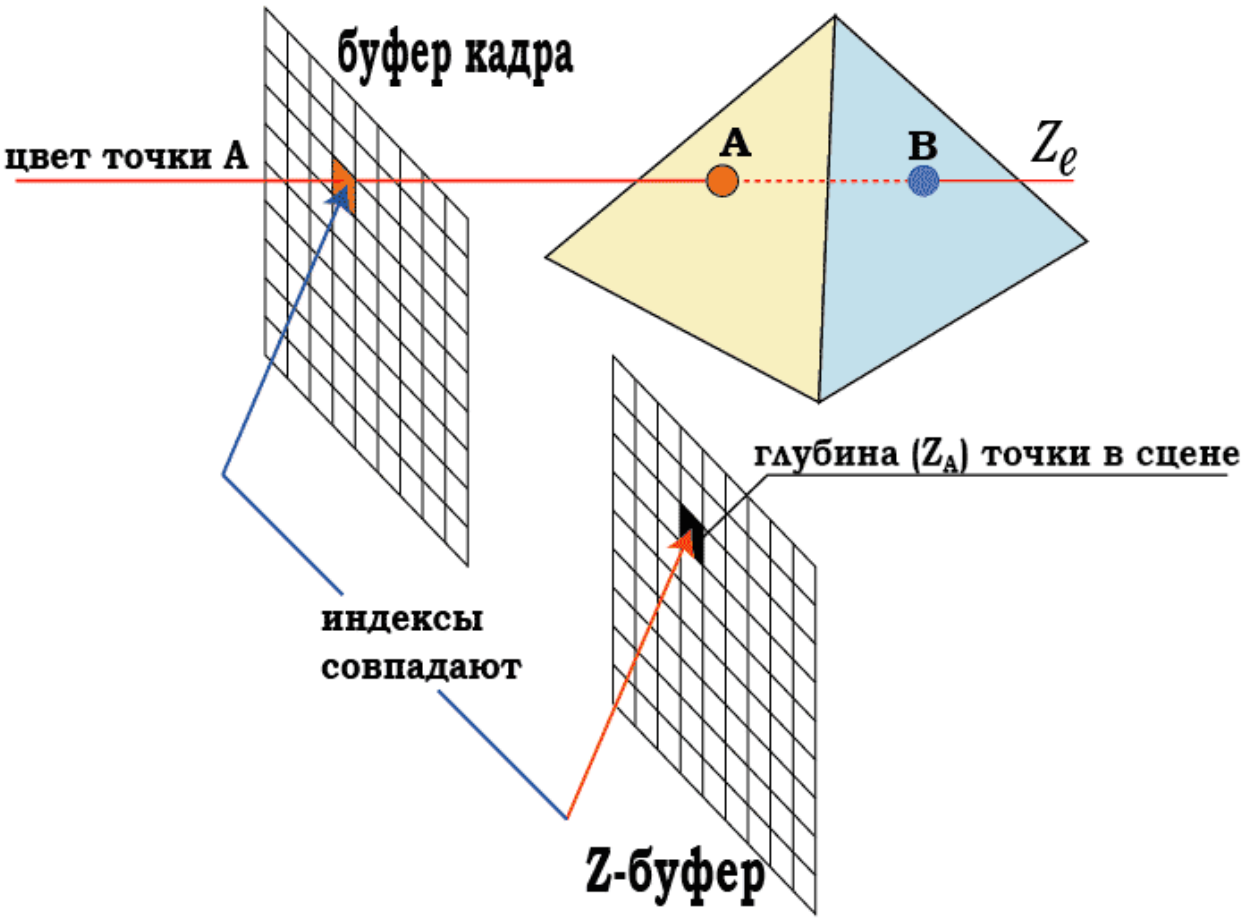

Рисунок 1.7 – Пример работы алгоритма, использующего Z-буфер

1.5.3 Обратная трассировка лучей

В данном алгоритме предполагается, что точка зрения находится в бесконечности на положительной полуоси Z, что делает все световые лучи параллельными этой оси. Алгоритм отслеживает траекторию каждого луча и определяет, с какими объектами сцены, если они существуют, он пересекается. В случае если пересечение существует, то из данного луча образуются преломленный и отраженный.

Благодаря этому алгоритму можно достичь эффектов, таких как отражение и преломление, что делает изображение более реалистичным. Каждый пиксель изображения вычисляется независимо от других, что обеспечивает высокую параллельность расчетов.

Однако алгоритм обратной трассировки лучей имеет и недостатки. Данный алгоритм очень ресурсоемок, так как каждый луч порождает несколько отраженных и преломленных лучей, которые также необходимо хранить и обрабатывать. Также для поиска пересечения одного луча необходимо про-

анализировать все имеющиеся на сцене примитивы, что значительно влияет на вычислительную сложность алгоритма [7].

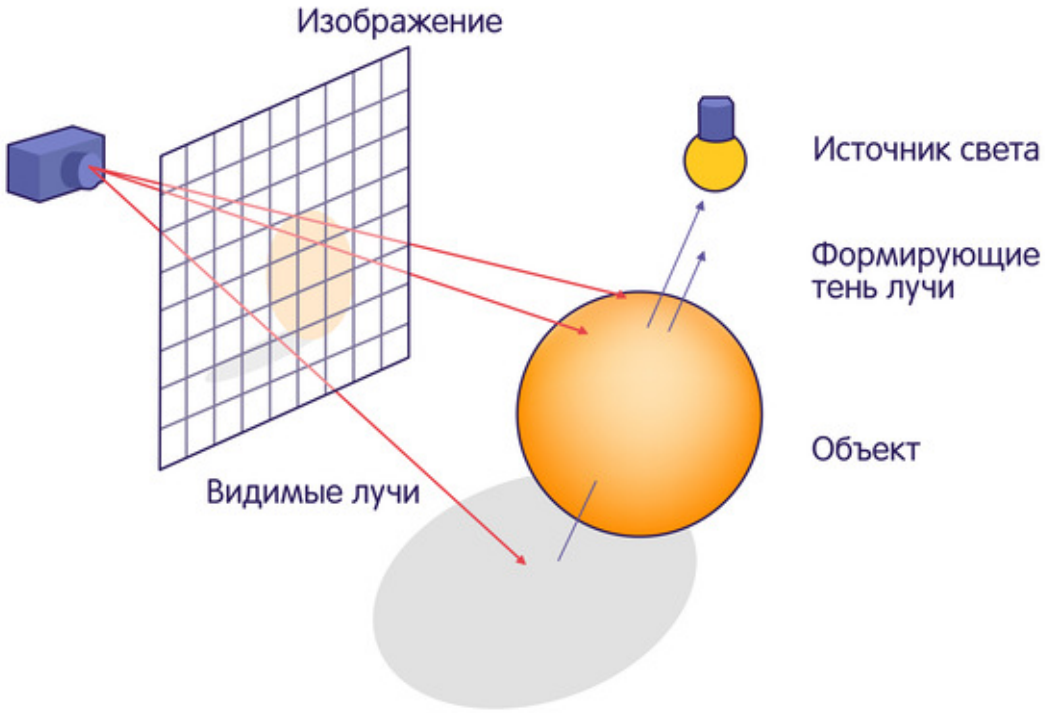

Рисунок 1.8 – Пример работы алгоритма обратной трассировки лучей

Вывод

Проанализировав алгоритмы удаления невидимых линий и поверхностей, заметим, что 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.

Выводы из аналитического раздела

В данном разделе были проанализированы алгоритмы морфинга, удаления невидимых линий, а также методы закраски. Таким образом были выбраны следующие алгоритмы.

1) **Алгоритм морфинга** - линейный морфинг.

2) **Метод закраски** - закраска Гуро.

3) **Алгоритм удаления невидимых линий** - алгоритм Z-буфера.

Входными данными для полученной системы будут являться:

— вершины и ребра модели;

— локальные координаты модели;

— угол поворота модели по каждой оси;

— текущий масштаб модели;

— положение камеры;

— положение источника света.

2 Конструкторский раздел

В данном разделе будут более подробно рассмотрены выбранные в предыдущем разделе методы и алгоритмы, а также предоставлены требования к программному обеспечению.

2.1 Требования к программному обеспечению

Программа должна:

— загружать модели из .obj файлов;

— визуализировать трехмерные каркасные модели;

— визуализировать морфинг двух трехмерных моделей;

— преобразовывать модель: поворот, масштабирование и сдвиг;

— изменять положение источника света;

— изменять положение камеры;

— изменять тип вершин, ребер (в режиме морфинга);

— изменять размер вершин и ребер (в режиме морфинга).

2.2 Считывание моделей

Для задания моделей был выбран один из наиболее популярных и универсальных методов – .obj файлы. Рассмотрим этапы считывания данных.

1) **Открытие файла .obj**

   Программа открывает файл, предоставляя путь к нему в качестве входного параметра.

2) **Чтение вершин**

   Из файла считываются строки, содержащие координаты вершин. Каждая строка представляет собой набор координат в формате 'v x y z', где x, y и z - координаты вершины.

3) **Чтение граней**

Происходит чтение строк, представляющих грани. Формат строки - 'f v1 v2...vn', где v1, v2...vn - индексы вершин, образующих грань.

4) **Обработка и хранение данных**

Прочитанные вершины и грани сохраняются в структуры данных, предназначенные для хранения геометрических данных модели. Вершины сохраняются в массиве, а грани - в структуре, содержащей информацию о связи вершин и порядке их соединения.

2.3 Общий алгоритм построения изображения

Рассмотрим алгоритм построения кадра для каркасной модели, он представлен на рисунке 2.1. Для визуализации объектов используется алгоритм Z-буфера представлен на рисунке 2.2. Он был рассмотрен для полигонов поверхностной модели.

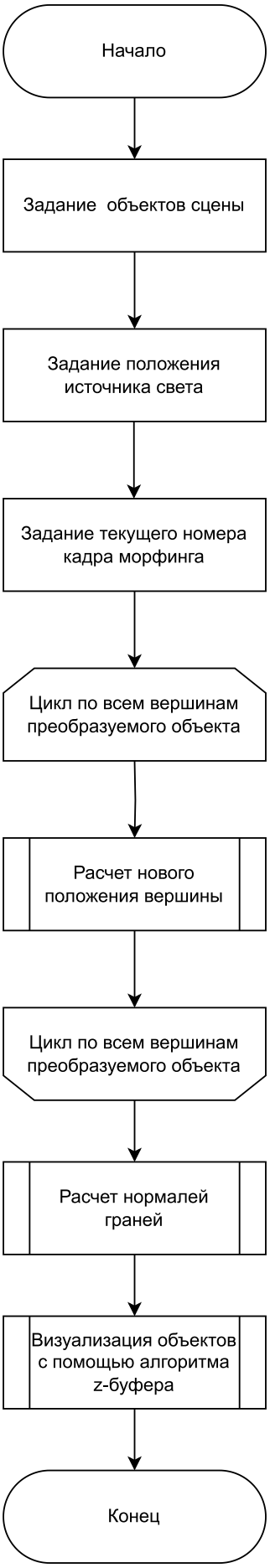

Рисунок 2.1 – Общий алгоритм построения кадра

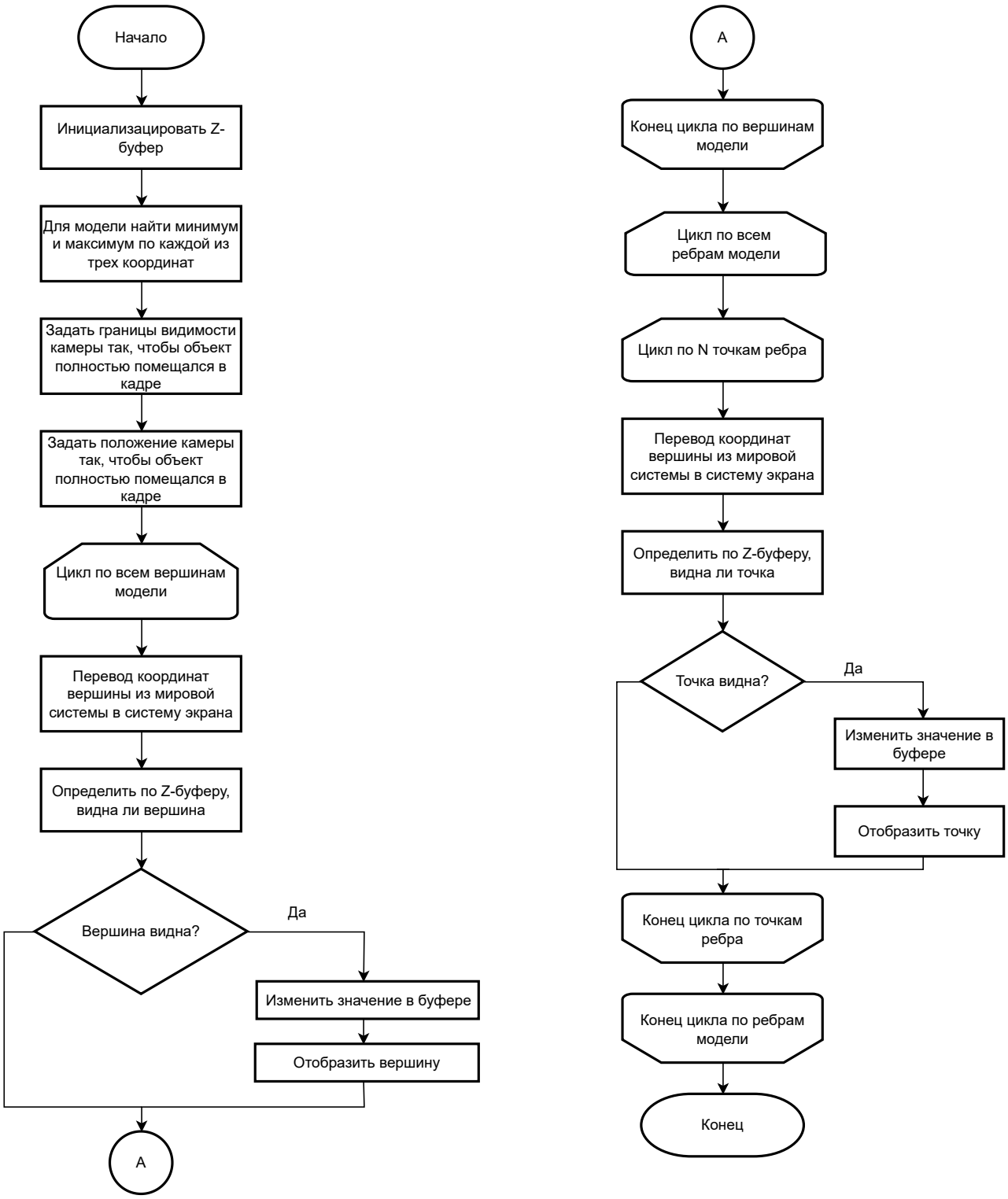

Рисунок 2.2 – Алгоритм Z-буфера

Математические соображения об алгоритме Гуро приведены в разделе 1.4. Данный алгоритм позволяет получить довольно гладкое изображение при малых вычислительных затратах, что важно при необходимости постоянно пересчета вершин при морфинге. Необходимо отметить, что интенсивность в вершинах полигона будет определяться как скалярное произведение вектора светового луча (вектора направленного из вершины к источнику света) и нормали в вершине.

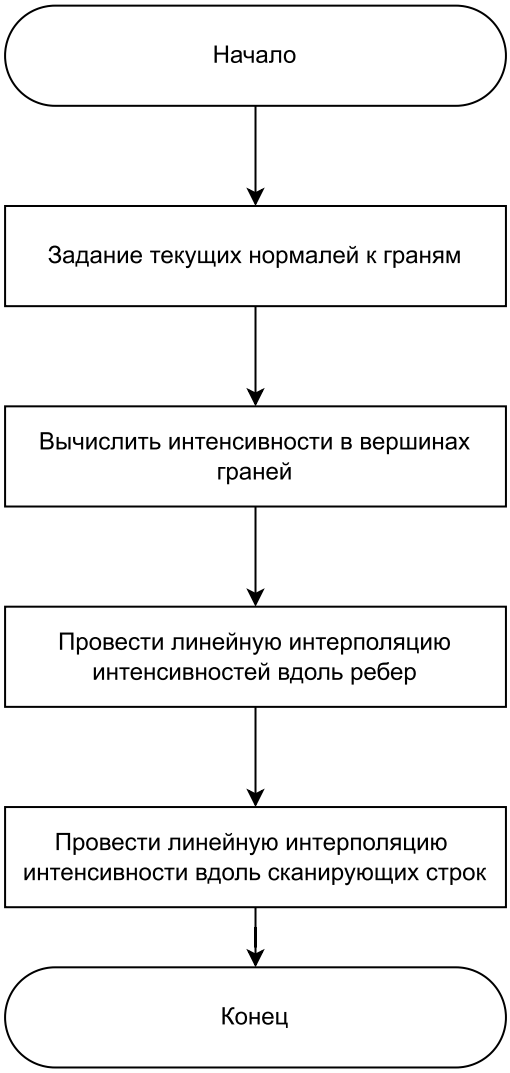

Рисунок 2.3 – Алгоритм закраски по Гуро

2.4 Алгоритм морфинга

Рассмотрим алгоритм морфинга двух объектов. Он представлен на рисунке 2.4.

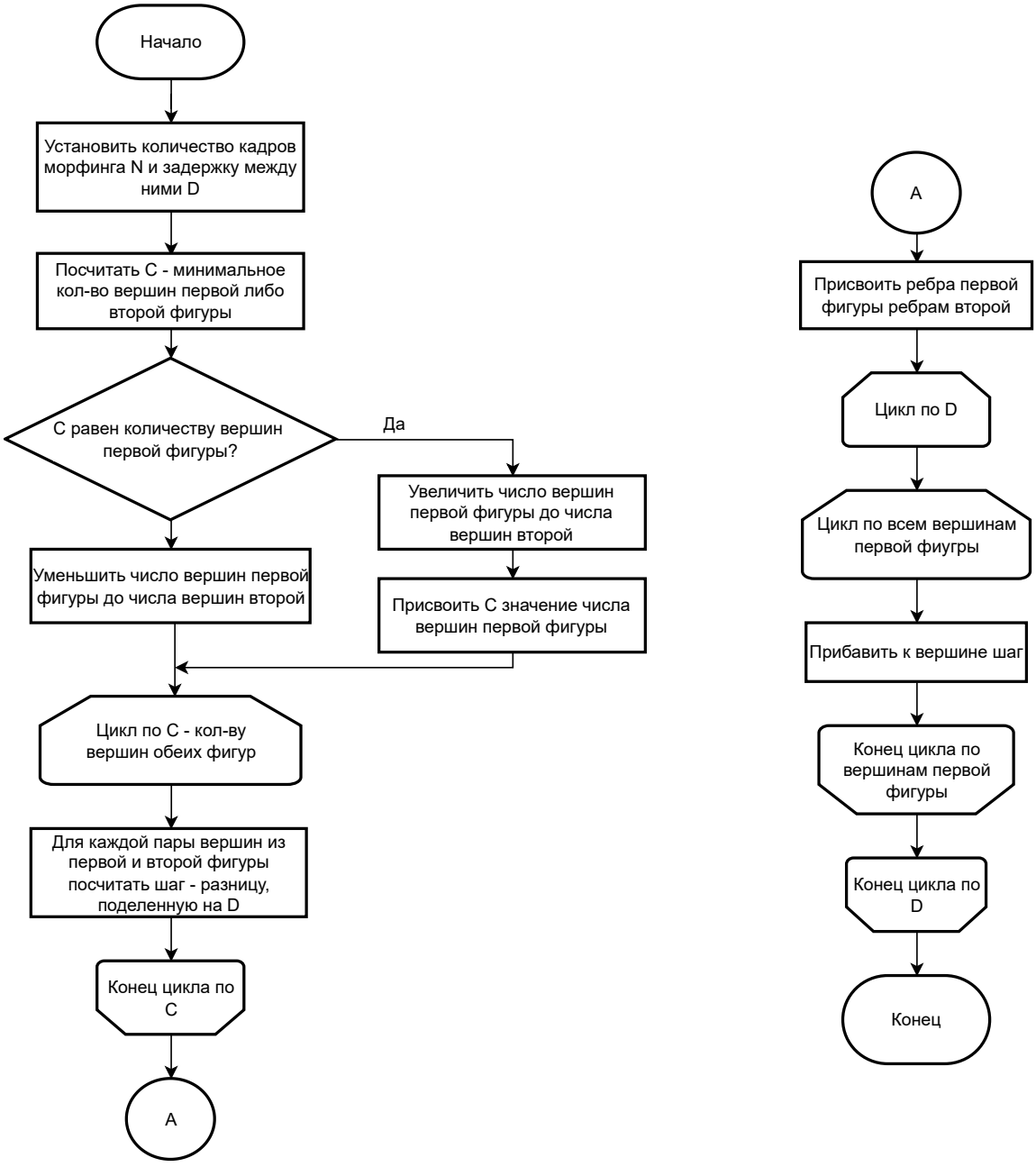

Рисунок 2.4 – Алгоритм морфинга двух объектов

Выводы из конструкторского раздела

В данном разделе была разобрана реализация выбранного алгоритма построения изображения, а также рассмотрены алгоритмы Z-буфера, закраски по Гуро и морфинга двух фигур.

3 Технологический раздел

В данном разделе производится выбор средств реализации программного обеспечения, описываются типы и структуры данных, листинги кода, а так же будет продемонстрирован интерфейс программы.

3.1 Выбор языка программирования и среды разработки

Для реализации программного обеспечения был выбран язык C++ по нескольким причинам:

— поддерживает объектно-ориентированную парадигму программирования;

— обладает достаточно высокой производительностью;

— обладает широким набором функций из стандартной библиотеки, в том числе функция замера процессорного времени, которая будет использована в дальнейшем исследовании.

В качестве среды разработки был выбран Qt Creator. Он обладает всем необходимым функционалом для написания и отладки программ. Данная среда поставляется с фреймворком Qt, который будет использоваться для создания графического интерфейса программы.

3.2 Описание структуры программы

В данной работе необходимо создать следующие структуры данных:

— структура, характеризующая перемещение модели;

— структура, характеризующая вращение модели;

— структура, характеризующая масштабирование модели;

— класс, описывающий точку;

— класс, описывающий точки;

— структура, описывающая ребро;

— класс, описывающий ребра;

— структура, описывающая грань;

— класс, описывающий грани;

— класс, описывающий фигуру;

— класс, описывающий отрисовываемое полотно;

— класс, необходимый для взаимодействия пользователя с программой.

3.3 Структура программы

На рисунке 3.1 представлена структура программы в виде UML-диаграммы.

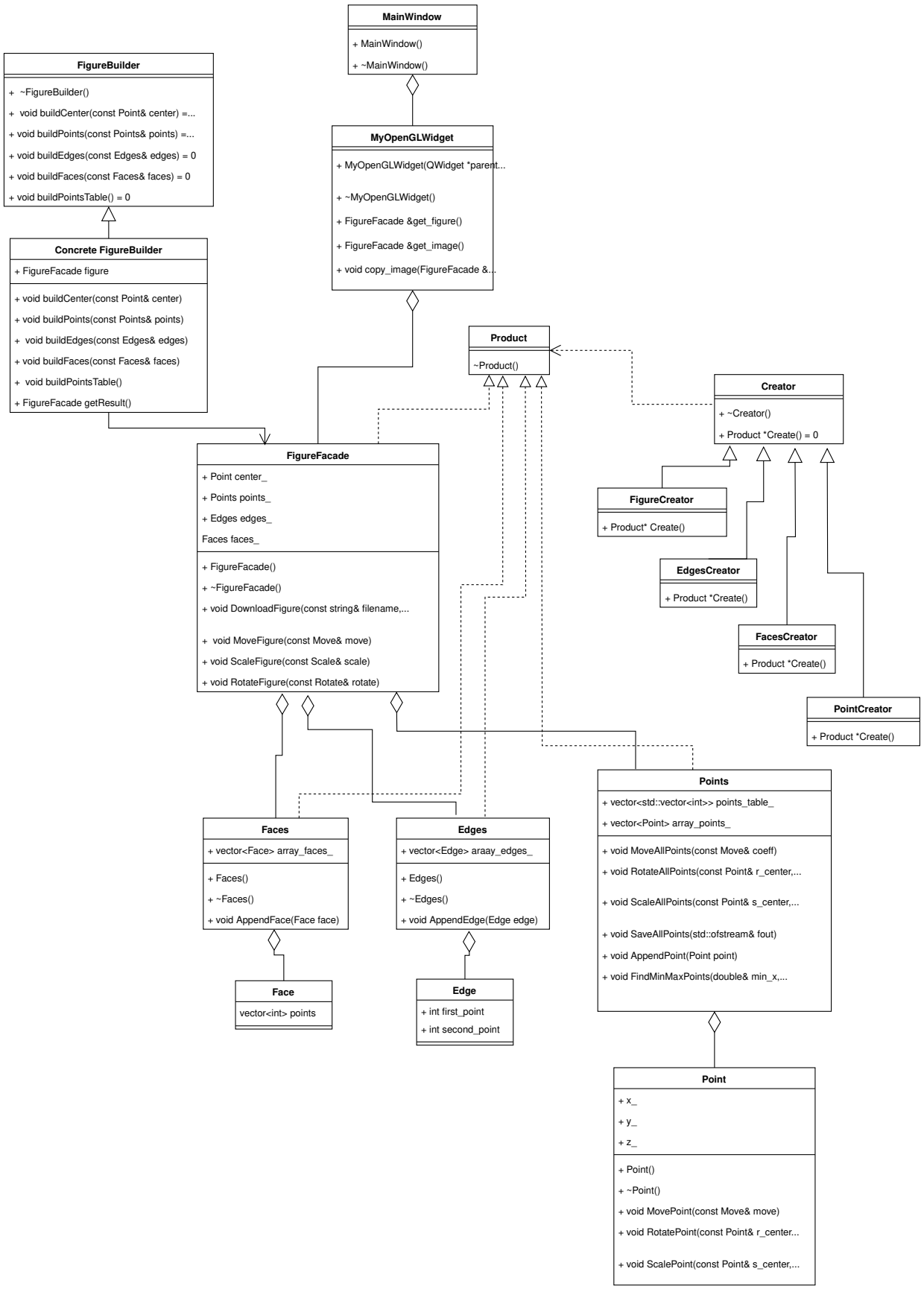

Рисунок 3.1 – Структура программы

3.4 Реализация алгоритмов

Реализация алгоритма создания фигуры и считывания ее вершин и ребер из .obj файла предоставлена на листингах 3.1 – 3.6. Необходимо отметить, что очередное ребро не просто заносится в список. Сначала проверяется его наличие (см. листинги 3.3 – 3.4). Так же для построения фигуры используется паттерн строитель – так код становится логически проще. За выделение памяти отвечает фабричный метод.

Листинг 3.1 – Создание и загрузка фигуры

```
1  void FigureFacade::DownloadFigure(const std::string& filename,
       ConcreteFigureBuilder& builder)
2  {
3      std::ifstream fin(filename);
4      if (fin.is_open())
5      {
6          ReadFigure(fin, builder);
7          fin.close();
8      }
9  }
10
11 void FigureFacade::ReadFigure(std::ifstream& fin, FigureBuilder&
       builder)
12 {
13     std::string line;
14
15     EdgesCreator edgesCreator;
16     Edges* edges = (Edges*)edgesCreator.Create();
17
18     PointsCreator pointsCretor;
19     Points* points = (Points*)pointsCretor.Create();
20
21     FacesCreator facesCreator;
22     Faces* faces = (Faces*)facesCreator.Create();
23
24     while (std::getline(fin, line))
25     {
26         if (line.substr(0, 2) == "v ")
27         {
28             Point vertex;
29             ReadVertex(vertex, line);
```

```
30              points->AppendPoint(vertex);
31          }
32          else if (line.substr(0, 2) == "f ")
33          {
34              Face face;
35              ReadFace(face, line);
36              faces->AppendFace(face);
37          }
38      }
39      builder.buildFaces(*faces);
40      builder.buildPoints(*points);
41      builder.buildPointsTable();
42
43      for (size_t i = 0; i < faces->array_faces_.size(); ++i)
44          FaceToEdges(faces->array_faces_[i], *edges);
45      builder.buildEdges(*edges);
46  }
```

Листинг 3.2 – Чтение вершин из файла

```
1  void FigureFacade::ReadVertex(Point& point, const std::string&
       line)
2  {
3      std::istringstream iss(line);
4      std::string v;
5      iss >> v >> point.x_ >> point.y_ >> point.z_;
6  }
```

Листинг 3.3 – Чтение граней из файла

```
1  void FigureFacade::ReadFace(Face& face, const std::string& line)
2  {
3      std::istringstream iss(line);
4      std::string token;
5      int num_tokens = 0;
6
7      while (iss >> token)
8      {
9          if (num_tokens > 0)
10         {
11             int point_index;
12             std::istringstream(token) >> point_index;
13             face.points.push_back(point_index);
14         }
```

```
15          num_tokens++;
16      }
17  }
```

Листинг 3.4 – Получение ребер из граней

```
1  void FigureFacade::FaceToEdges(const Face& face, Edges& edges)
2  {
3      for (size_t i = 0; i < face.points.size(); ++i)
4      {
5          Edge edge;
6          edge.first_point = face.points[i];
7          edge.second_point = face.points[(i + 1) %
               face.points.size()];
8          if (!ExistEdge(edge))
9              edges.AppendEdge(edge);
10     }
11 }
```

Листинг 3.5 – Заполнение таблицы смежности для точек

```
1  void buildPointsTable() override
2  {
3      for (size_t i = 0; i < figure.edges_.array_edges_.size();
           ++i)
4      {
5          if (static_cast<size_t>
6          (figure.edges_.array_edges_[i].first_point) <
7          figure.points_.points_table_.size() &&
8          static_cast<size_t>
9          (figure.edges_.array_edges_[i].second_point) <
10         figure.points_
11         .points_table_[figure.edges_.array_edges_[i].first_point]
12         .size())
13         {
14             figure.points_
15             .points_table_[figure.edges_.array_edges_[i]
16             .first_point]
17             [figure.edges_.array_edges_[i].second_point] = 1;
18             figure.points_
19             .points_table_[figure.edges_.array_edges_[i]
20             .first_point]
21             [figure.edges_.array_edges_[i].second_point] = 1;
22         }
```

```
23          else
24          {
25              return;
26          }
27      }
28  }
```

Листинг 3.6 – Проверка ребра на наличие в таблице смежности

```
1  bool FigureFacade::ExistEdge(const Edge& edge)
2  {
3      if (static_cast<size_t>(edge.first_point) <
          points_.points_table_.size() &&
4      static_cast<size_t>(edge.second_point) <
5      points_.points_table_[edge.first_point].size())
6      {
7          return points_.
8          points_table_[edge.first_point][edge.second_point] == 1;
9      } else
10     {
11         return false;
12     }
13 }
```

Реализация алгоритма отображения сцены предоставлена на листингах 3.7 – 3.11. Здесь алгоритм Z-буфера и настройка камеры и света реализованы средствами OpenGL. Так повышается производительность программы, ведь методы OpenGL используют аппаратное ускорение (многие методы выполняются на графическом процессоре) и параллелизм при обработке вершин. Настройка камеры происходит относительно размеров фигуры. Так она всегда находится в области видимости камеры.

Листинг 3.7 – Инициализация и перерисовка сцены

```
1  void MyOpenGLWidget::paintGL()
2  {
3      glClearColor(background_color.redF(),
          background_color.greenF(), background_color.blueF(),
          1.0f);
4      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
5
6      if (MORPH)
7          init_camera(image);
8      else
```

```
9           init_camera(figure);
10
11      if (OK)
12      {
13          if (MORPH)
14          {
15              draw_edges();
16              draw_vertices();
17          }
18          else
19          draw_faces();
20      }
21      else
22      {
23          figure_default();
24          image_default();
25          OK = true;
26      }
27  }
```

Листинг 3.8 – Настройка камеры и света

```
1   void MyOpenGLWidget::init_camera(FigureFacade &fig)
2   {
3       glMatrixMode(GL_PROJECTION);
4       glLoadIdentity();
5
6       double min_x = -1.0;
7       double max_x = 1.0;
8       double min_y = -1.0;
9       double max_y = 1.0;
10      double min_z = 1.0;
11      double max_z = 10.0;
12
13      if (!fig.points_.array_points_.empty())
14          fig.points_.FindMinMaxPoints(min_x, max_x, min_y, max_y,
                min_z, max_z);
15
16      double padding = 2.0;
17
18      double width = max_x - min_x + 2 * padding;
19      double height = max_y - min_y + 2 * padding;
20      double depth = max_z - min_z + 2 * padding;
```

```
21
22      double centerX = (max_x + min_x) / 2;
23      double centerY = (max_y + min_y) / 2;
24      double centerZ = (max_z + min_z) / 2;
25
26      double halfWidth = width / 2;
27      double halfHeight = height / 2;
28      double halfDepth = depth / 2;
29
30      double diagonal = sqrt(halfWidth * halfWidth + halfHeight *
            halfHeight +
31      halfDepth * halfDepth);
32
33      min_x = centerX - diagonal;
34      max_x = centerX + diagonal;
35      min_y = centerY - diagonal;
36      max_y = centerY + diagonal;
37      min_z = centerZ - diagonal;
38      max_z = centerZ + diagonal;
39
40      GLfloat lightPos[] = {static_cast<GLfloat>(max_x * 1.5) +
            lightD[0],
41      static_cast<GLfloat>(max_y * 1.5) + lightD[1],
            static_cast<GLfloat>(max_z + 2.0f) +
42      lightD[2], 1.0f};
43      GLfloat lightAmbient[] = {0.2f, 0.2f, 0.2f, 1.0f};
44      GLfloat lightDiffuse[] = {1.0f, 1.0f, 1.0f, 1.0f};
45
46      glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
47      glLightfv(GL_LIGHT0, GL_AMBIENT, lightAmbient);
48      glLightfv(GL_LIGHT0, GL_DIFFUSE, lightDiffuse);
49
50      glOrtho(min_x, max_x, min_y, max_y, 0,
            std::numeric_limits<float>::max());
51      glTranslatef(0.0f + cameraD[0], 0.0f + cameraD[1],
            -fabs(min_z) + cameraD[2]);
52      glMatrixMode(GL_MODELVIEW);
53      glLoadIdentity();
54  }
```

Листинг 3.9 – Отрисовка вершин

```
1  void MyOpenGLWidget::draw_vertices()
```

```
{
    if (vertex_type != NONE)
    {
        if (vertex_type == CIRCLE)
            glEnable(GL_POINT_SMOOTH);

        glColor3f(vertices_color.redF(), vertices_color.greenF(),
        vertices_color.blueF());
        glPointSize(vertices_width);
        glBegin(GL_POINTS);

        for (size_t i = 0; i <
           image.points_.array_points_.size(); i++)
        {
            glVertex3f(image.points_.array_points_[i].x_,
            image.points_.array_points_[i].y_,
            image.points_.array_points_[i].z_);
        }
        glEnd();

        if (vertex_type == CIRCLE) glDisable(GL_POINT_SMOOTH);
    }
}
```

Листинг 3.10 – Отрисовка ребер

```
void MyOpenGLWidget::draw_edges()
{
    if (edges_type == DOTTED)
    {
        glEnable(GL_LINE_STIPPLE);
        glLineStipple(1, 0x00FF);
    }

    glColor3f(edges_color.redF(), edges_color.greenF(),
       edges_color.blueF());
    glLineWidth(edges_width);
    glBegin(GL_LINES);

    for (size_t i = 0; i < image.edges_.array_edges_.size(); ++i)
    {
        int v1 = image.edges_.array_edges_[i].first_point - 1;
        int v2 = image.edges_.array_edges_[i].second_point - 1;
```

```
17
18          glVertex3f(image.points_.array_points_[v1].x_,
19          image.points_.array_points_[v1].y_,
20          image.points_.array_points_[v1].z_);
21
22          glVertex3f(image.points_.array_points_[v2].x_,
23          image.points_.array_points_[v2].y_,
24          image.points_.array_points_[v2].z_);
25      }
26      glEnd();
27
28      if (edges_type == DOTTED)
29          glDisable(GL_LINE_STIPPLE);
30  }
```

Листинг 3.11 – Отрисовка граней

```
1   void MyOpenGLWidget::draw_faces()
2   {
3       glColor3f(edges_color.redF(), edges_color.greenF(),
            edges_color.blueF());
4       glLineWidth(edges_width);
5
6       for (size_t i = 0; i < image.faces_.array_faces_.size(); ++i)
7       {
8           const auto& current_face =
                image.faces_.array_faces_[i].points;
9           size_t num_points = current_face.size();
10          glBegin(num_points == 3 ? GL_TRIANGLES : GL_POLYGON);
11
12          for (size_t j = 0; j < num_points; ++j)
13          {
14              int vertex_index = current_face[j] - 1;
15              glVertex3f(image.points_.
16              array_points_[vertex_index].x_,
17              image.points_.array_points_[vertex_index].y_,
18              image.points_.array_points_[vertex_index].z_);
19          }
20          glEnd();
21      }
22  }
```

Реализация морфинга двух фигур представлена на листинге 3.12.

Листинг 3.12 – Морфинг двух фигур

```
void MyOpenGLWidget::Morph(FigureFacade &figure, FigureFacade
   &image)
{
    if (image.points_.array_points_.empty())
    {
        copy_image(figure, image);
        return;
    }

    int transformTime = 100;
    int delayMilliseconds = 100;
    QElapsedTimer tm;
    tm.start();
    size_t count = std::min(figure.points_.array_points_.size(),
       image.points_.array_points_.size());

    if (count == image.points_.array_points_.size())
    {
        for (size_t i = count; i <
           figure.points_.array_points_.size(); ++i)
        {
            image.points_.array_points_.
            push_back(image.points_.array_points_[i - count]);
        }
        count = figure.points_.array_points_.size();
    }
    else
    {
        size_t imageSize = image.points_.array_points_.size();
        for (size_t i = count; i < imageSize; ++i)
            image.points_.array_points_.
            erase(image.points_.array_points_.begin() + i);
    }
    std::vector<Point> steps(count);

    for (size_t i = 0; i < count; ++i)
    {
        steps[i].x_ = (figure.points_.array_points_[i].x_ -
           image.points_.array_points_[i].x_) / transformTime;
```

```
36          steps[i].y_ = (figure.points_.array_points_[i].y_ -
               image.points_.array_points_[i].y_) / transformTime;
37          steps[i].z_ = (figure.points_.array_points_[i].z_ -
               image.points_.array_points_[i].z_) / transformTime;
38      }
39      image.edges_ = figure.edges_;
40      QTimer* timer = new QTimer(this);
41      QEventLoop eventLoop;
42      connect(timer, &QTimer::timeout, this, [this, figure,
           &image, steps, transformTime, timer, &eventLoop]() mutable
43      {
44          MorphNextStep(image, steps);
45          --transformTime;
46          if (transformTime < 0)
47          {
48              copy_image(figure, image);
49              timer->stop();
50              timer->deleteLater();
51              eventLoop.quit();
52          }
53      });
54
55      MORPH = true;
56      timer->start(delayMilliseconds);
57      eventLoop.exec();
58      QThread::msleep(1000);
59      MORPH = false;
60  }
61
62  void MyOpenGLWidget::MorphNextStep(FigureFacade &image,
       std::vector<Point> &steps)
63  {
64      for (size_t i = 0; i < steps.size(); ++i)
65      {
66          image.points_.array_points_[i].x_ += steps[i].x_;
67          image.points_.array_points_[i].y_ += steps[i].y_;
68          image.points_.array_points_[i].z_ += steps[i].z_;
69      }
70
71      update();
72  }
```

3.5 Интерфейс программы

Программа предоставляет следующий графический интерфейс (см. рис. 3.1). При первом запуске все виджеты имеют значения по умолчанию, при следующих запусках программы значения сохраняются. Пользователь видит сцену (справа), виджеты (слева) и меню (сверху). С помощью меню пользователь может выполнять следующие команлы.

1) Загружать новую модель. Если модель уже была на сцене, начнется процесс морфинга.

2) Менять отрисовку с помощью виджетов:

   — перемещать модель вдоль осей;

   — поворачивать модель вокруг осей;

   — масштабировать модель;

   — изменять размер ребер (видно при морфинге);

   — изменять размер вершин (видно при морфинге);

   — изменять тип вершин (видно при морфинге);

   — изменять тип ребер (видно при морфинге);

   — перемещать камеру;

   — перемещать источник света.

На рисунке 3.2 изображен интерфейс программы, процесс морфинга изображен на рисунке 3.3. Загруженная модель изображена на рисунке 3.4.

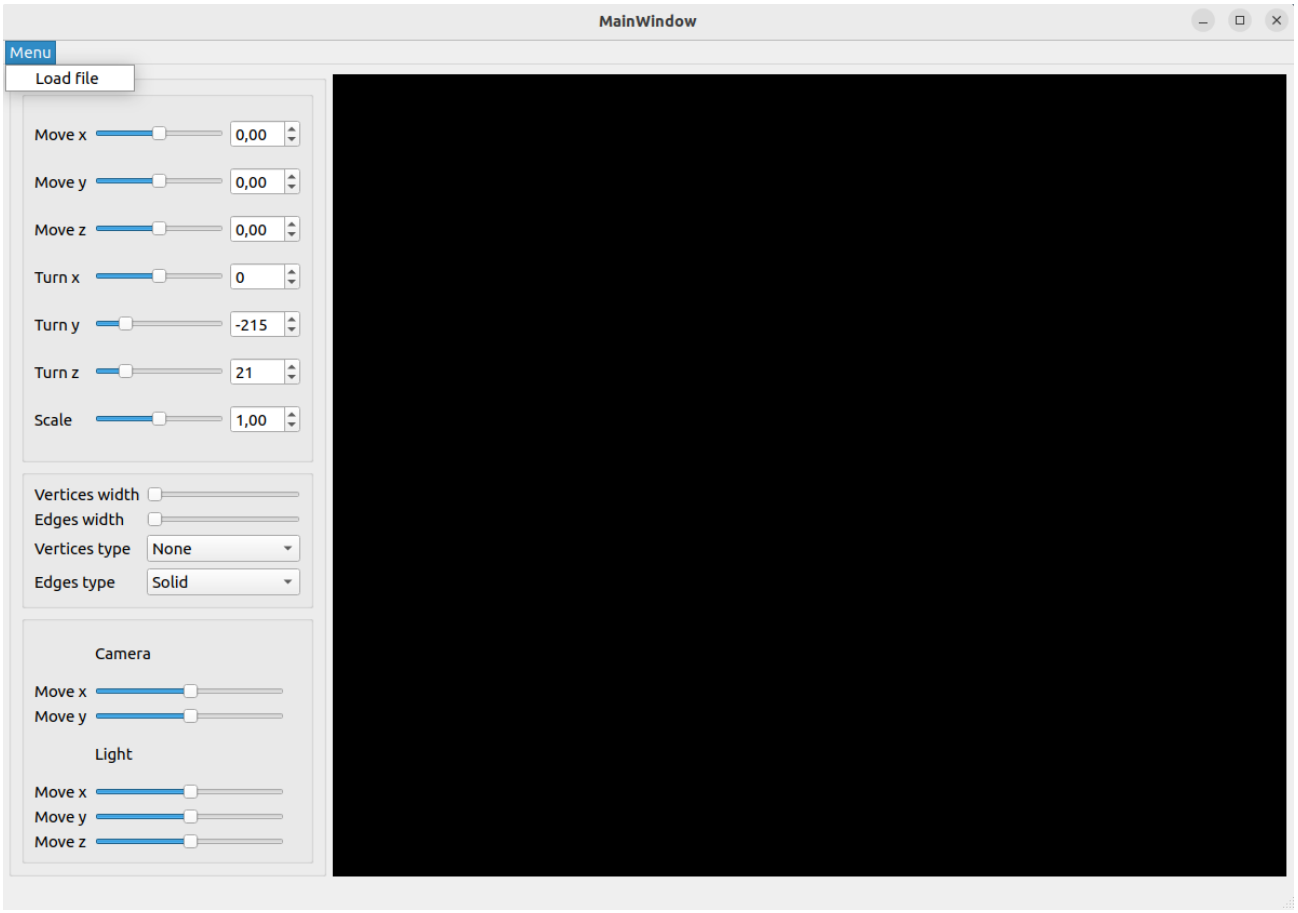

Рисунок 3.2 – Интерфейс программы

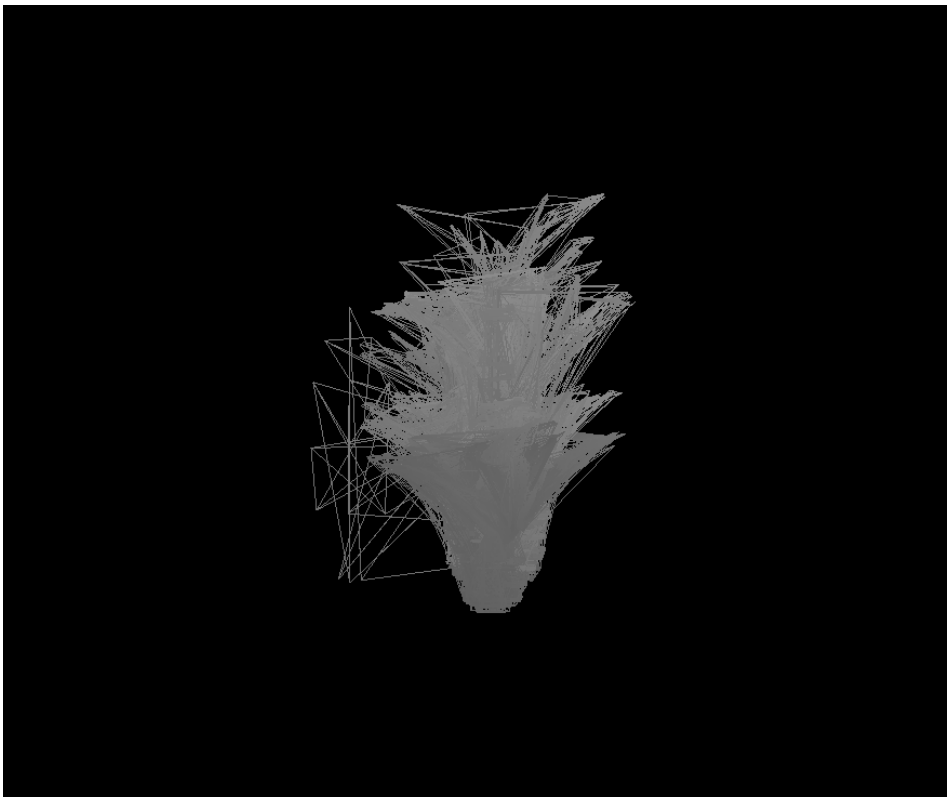

Рисунок 3.3 – Процесс морфинга

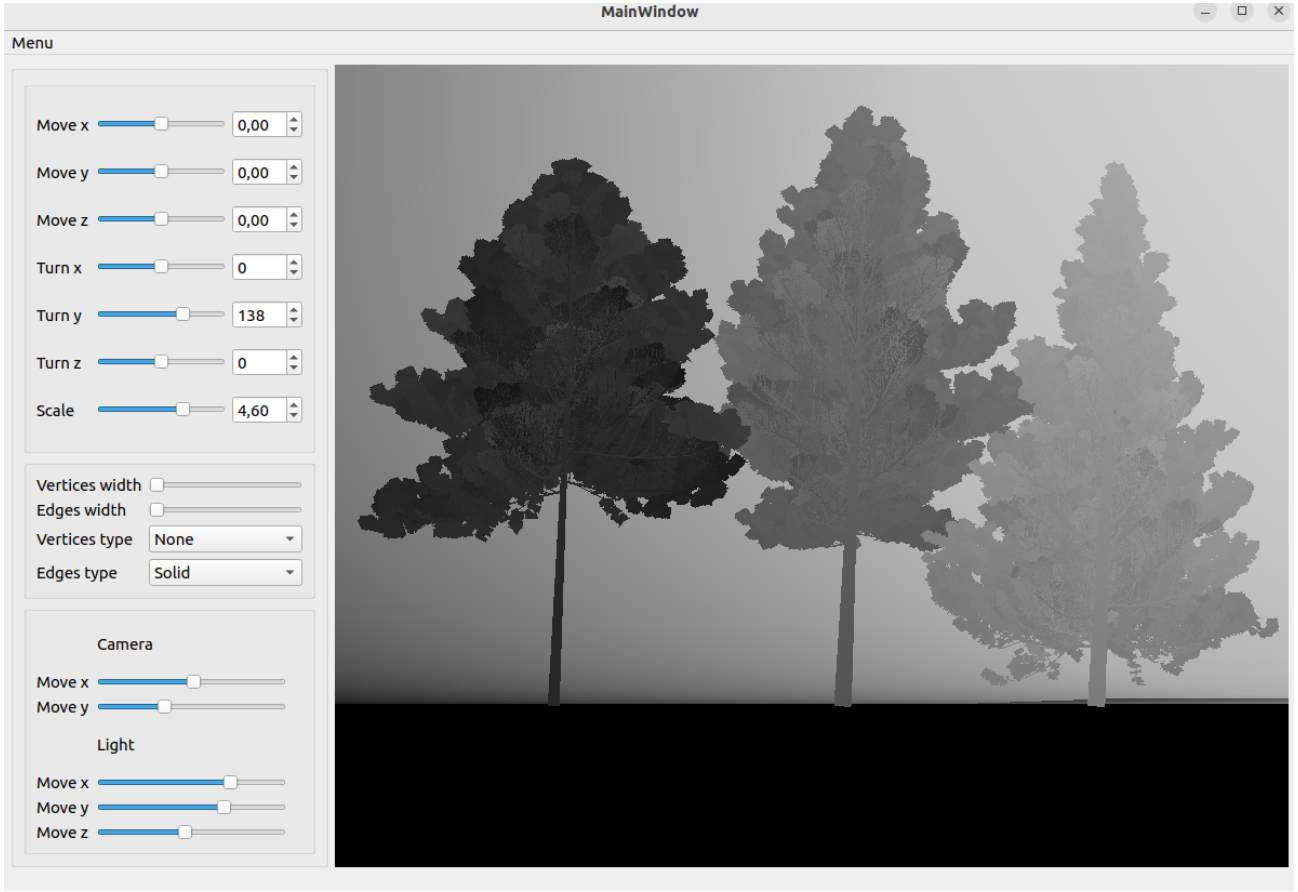

Рисунок 3.4 – Загруженная модель

Выводы из технологического раздела

В этом разделе были выбраны средства реализации программы и необходимые структуры данных, описана структура программы, а также предоставлена реализация алгоритмов и пользовательский интерфейс.

4 Исследовательский раздел

В данном разделе приведены технические характеристики устройства, на котором проводилось измерение времени работы программного обеспечения, а также результаты замеров времени работы программы.

Целью исследования является провести анализ скорости работы алгоритма морфинга изображений с использованием алгоритма Z-буфера для визуализации.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

1) Процессор Intel(R) Core(TM) i7-11700K @ 3.60МГц.

2) Оперативная память - 32 GB.

Во время тестирования программа была запущена на виртуальной машине Ubuntu 22.04 из операционной системы Windows 10. Технические характеристики виртуальной машины:

1) Число ядер - 4.

2) Оперативная память - 20.9 GB.

4.2 Результаты исследования

Для исследования зависимости времени от размеров фигуры использовались фигуры с различным числом вершин и ребер. На графиках учитываются вершины и ребра результирующих фигур, так как до начала морфинга каждая фигура достраивается либо разрушается так, чтобы количество вершин и ребер стало таким же, как у фигуры, к которой она будет преобразована.

Для каждого случая было проведено по 10 тестов, выбрано среднее значение по времени. Результаты исследования представлены на рисунках 4.1 - 4.2 и в таблице 4.1.

Таблица 4.1 – Результаты исследования

Кол-во вершин	Кол-во ребер	Время (мс)
8	24	1
281	1 124	1
1 136	6 246	2
2 210	8 568	4
5 530	21 712	8
7 338	29 344	10
10 194	40 772	17
25 075	91 680	30
35 986	143 936	53
40 062	161 472	56

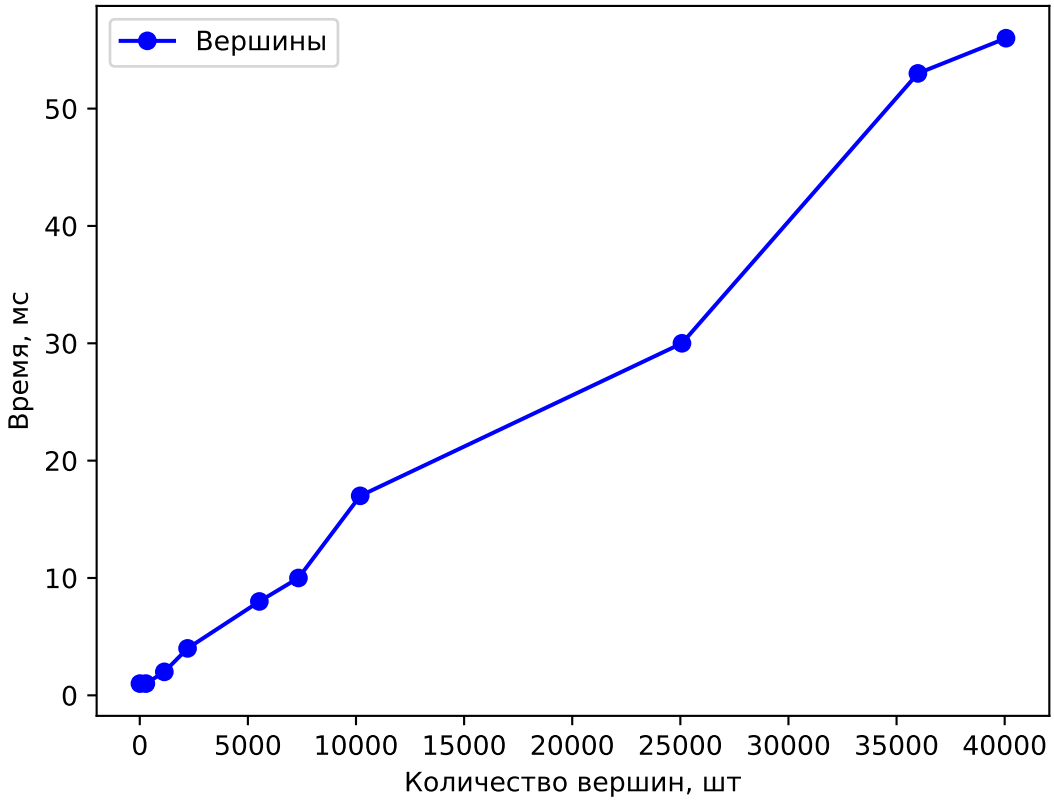

Рисунок 4.1 – Результаты исследования

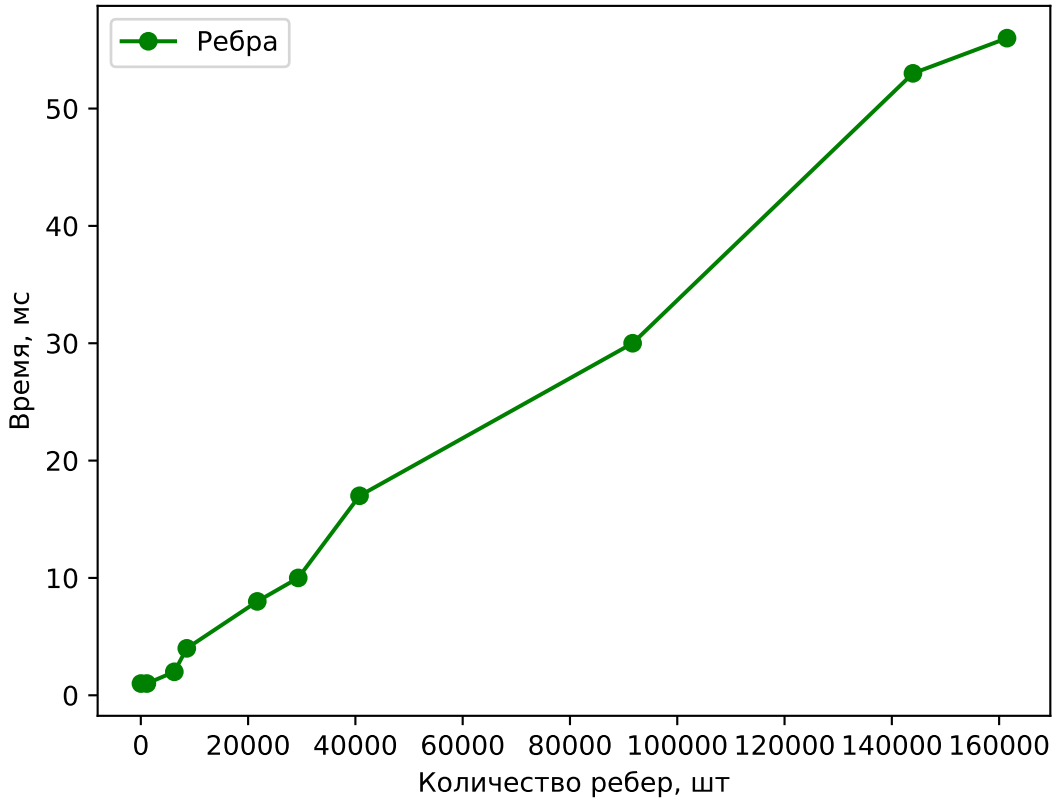

Рисунок 4.2 – Результаты исследования

Из проведенного исследования можно сделать вывод, что время визуализации сцены во время морфинга линейно зависит от количества вершин и ребер результирующей фигуры.

Выводы из исследовательского раздела

В данном разделе приведены результаты работы программного обеспечения и проведено исследование, показывающее зависимость времени визуализации сцены во время морфинга от количества вершин и ребер результирующей фигуры.

Результаты исследования совпали с ожидаемыми, так как в ходе исследования было установлено, что время работы увеличивается с увеличением количества вершин и ребер у объекта.

ЗАКЛЮЧЕНИЕ

Была достигнута цель данной работы - разработано программное обеспечение для морфинга трехмерных моделей.

Были выполнены следующие задачи.

1) Формализованы представления объектов.
2) Проанализированы алгоритмы морфинга трехмерных моделей.
3) Выбраны средства реализации алгоритмов.
4) Реализованы выбранные алгоритмы.
5) Реализован графический интерфейс.
6) Исследованы временные характеристики выбранных алгоритмов на основе созданного программного обеспечения.

Этот проект подчеркивает важность правильного выбора алгоритмов и методов в компьютерной графике. Исследования в области морфинга и визуализации трехмерных объектов оставляют простор для будущих улучшений и оптимизаций. Полученные знания и опыт станут основой для разработки более реалистичных и впечатляющих трехмерных сцен в будущем.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Гамбетта Г.* Компьютерная графика. Рейтрейсинг и растеризация //. — Питер, 2022.
2. Алгоритм Морфинга для создания сферических панорам [Электронный ресурс]. — Режим доступа: `https://clck.ru/35ghkJ` (дата обращения: 20.07.2023).
3. DMorph [Электронный ресурс]. — Режим доступа: `https://www.graphicon.ru/oldgr/courses/cg/assigns/2004/hw4/g5078.pdf` (дата обращения 25.07.23).
4. A Comparison of Mesh Morphing Methods [Электронный ресурс]. — Режим доступа: `https://clck.ru/35on9b` (дата обращения 25.07.23).
5. Weight morphing [Электронный ресурс]. — Режим доступа: `https://igl.ethz.ch/projects/bbw/bounded-biharmonic-weights-siggraph-2011-compressed-jacobson-et-al.pdf` (дата обращения 01.08.23).
6. Методы закраски [Электронный ресурс]. — Режим доступа: `https://clck.ru/35qmXd` (дата обращения 24.09.23).
7. *Роджерс Д.* Алгоритмические основы машинной графики //. — Москва: Мир, 1989.