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»

РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

Тема Редактор трехмерных моделей

Студент Солопов Ю. В.

Группа ИУ7-56Б

Научный руководитель Русакова З. Н.

Москва — 2022 г.

Оглавление

Введение

Компьютерная графика является одной из неотъемлемых частей современной жизни. Она используется повсеместно: в кинофильмах и мультфильмах, в компьютерных играх, а также для наглядного отображения данных. В связи с этим возрастает потребность в инструментах, позволяющих создавать трехмерные модели различной сложности. На данный момент существует множество подобных инструментов, предоставляющих обширный спектр различных функций, упрощающих процесс создания 3d модели.

Цель данного курсового проекта — разработка редактора трехмерных моделей, предоставляющего пользователю возможность производить базовые преобразования над полигональной моделью и ее составляющими частями (вершинами, ребрами, гранями).

Для достижения поставленной цели необходимо решить следующие задачи:

— изучить и провести анализ существующих алгоритмов компьютерной графики, выбрать наиболее подходящие из них для реализации редактора трехмерных моделей;

— спроектировать программное обеспечение, предоставляющее пользователю необходимые функции;

— реализовать спроектированное программное обеспечение;

— провести сравнительный анализ последовательной и параллельной реализаций алгоритма Z-буфера.

1 Аналитический раздел

В данном разделе будут формализованы объекты сцены, выбран способ заданий моделей, алгоритм удаления невидимых линий и поверхностей и алгоритм закраски.

1.1 Формализация объектов синтезируемой сцены

Сцена состоит из следующих объектов:

— **Камера** – специальный объект сцены, позволяющий просматривать ее сцены. Камера определяется двумя точками пространства: местоположением самой камеры, и точкой, к которой направлен объектив камеры;

— **Источник света** – объект сцены, который представляет из себя материальную точку, излучающую свет во всех направлениях. Источник света отображается как несколько концентрических окружностей;

— **Модель** – трехмерный объект, расположенный на сцене.

1.2 Анализ типов трехмерных моделей

Тип трехмерной модели определяет, как именно объект будет отображаться на сцене.

Модели могут задаваться в следующих формах:

— **Каркасная модель.** Это простейшая форма задания модели. Каркасная модель хранит информацию о вершинах и ребрах объекта. Основной недостаток такого отображения объектов заключается в том, что модель не всегда однозначно передает представление о форме объекта, в следствии чего ее можно неправильно интерпретировать.

— **Поверхностная модель.** Этот тип модели хранит информацию о каждой поверхности объекта. Поверхность может описываться аналитически, либо задаваться другим способом. Одним из недостатков поверхностной модели является отсутствие информации о том, с какой стороны поверхности находится материал.

— **Твердотельная модель.** Данный тип задания модели отличается от поверхностного тем, что в твердотельных моделях к информации о поверхностях добавляется информация о том, с какой стороны расположен материал. Это можно сделать, например, путем указания направления внутренней нормали.

Вывод

Использовать каркасный тип модели в моей задачи нецелесообразно, потому что такой тип в некоторых случаях может запутать пользователя, не давая полного представления об объекте.

Использовать твердотельную модель просто нет необходимости, т.к. пользователю моей программы не важно, с какой стороны от модели находится материал.

Поэтому я принял решение использовать полигональный тип модели, как самый подходящий для моей задачи. Тем не менее, моя программа предоставляет возможность отображения модели и в каркасной форме.

1.3 Анализ способов задания поверхностной модели

Поверхностную модель можно задать несколькими способами:

— **Аналитический способ.** Этот способ задания модели характеризуется описанием модели объекта, которое доступно в неявной форме. То есть для получения визуальных характеристик необходимо дополнительно вычислять некоторую функцию, которая зависит от параметра.

— **Полигональная сетка.** Данный способ позволяет задать модель совокупностью вершин, ребер и граней, определяющих ее форму в трехмерном пространстве.

Рассмотрим существующие способы хранения информации о полигональной сетке.

— **Вершинное представление.** Модель хранит множество вершин. Вершины указывают на другие вершины, с которыми они соединены.

— **Список граней.** Модель хранит множество граней и множество вершин. Каждая вершина при этом хранит информацию о соседних вершинах и о гранях, ее окружающих.

— **Крылатое представление.** Модель хранит информацию о вершинах, ребрах и гранях. При этом каждая вершина и грань хранят окружающие их ребра, а каждое ребро состоит из двух вершин (конечные точки), двух граней (по каждую сторону), и четырех ребер («крылья» ребра).

— **Таблица углов.** Модель задается таблицей, хранящей вершины. Обход заданной таблицы неявно задает полигоны.

Вывод

Использовать для моей задачи аналитический способ невозможно, так как задача предполагает возможность изменение модели посредством трансформации ее составляющих элементов.

Выбор конкретного способа хранения сетки напрямую зависит от поставленной задачи. Моя программа должна предоставлять возможность быстрого выбора и трансформации любых вершин, ребер и граней. Соответственно, решающим фактором является скорость выполнения этих операций. Поэтому полигональная сетка должна явно хранить информацию о каждой этой составляющей объекта.

Все вышеописанные способы, помимо крылатого представления, не хранят в явном виде информацию о ребрах, хотя в моей программе эта

информация необходима. Крылатое представление же избыточно для моей задачи.

Поэтому я разработал собственный способ хранения информации о сетке. В качестве основы был взят список граней, к которому я добавил информацию о ребрах, которые определяются двумя конечными точками. Иными словами, модель должна состоять из следующих элементов:

1. **Вершина.** Задается начальными координатами в пространстве и матрицей трансформации.
2. **Ребро.** Задается двумя его конечными вершинами.
3. **Грань.** Задается набором принадлежащих ей вершин и ребер.

Сама модель определяется набором принадлежащих ей граней, ребер и вершин.

1.4 Анализ способа выбора элементов модели на дисплее

Специфика моей задачи предполагает, что программа должна предоставлять пользователю возможность выбора необходимых ему вершин, ребер и граней при помощи клика мышью по дисплею вблизи них. Более того, пользователь должен иметь возможность выбрать только те элементы фигуры, которые являются видимыми.

Для решения этой проблемы было принято решение помимо буфера кадра хранить еще и буфер граней, каждая ячейка которого будет указывать на грань, которая отображается в соответствующей ячейке буфера кадров. Буфер граней позволит за константное время определить, какую именно грань выбрал пользователь и найти ближайшие к месту клика мыши вершину или ребро.

1.5 Анализ алгоритмов удаления невидимых линий и поверхностей

При выборе алгоритма удаления невидимых линий необходимо в первую очередь учесть особенности поставленной задачи. Моя задача подразумевает, что алгоритм удаления невидимых линий будет быстродействующим, поскольку пользователь должен видеть плавную анимацию при трансформации моделей или камеры. Также немаловажное значение имеет сложность реализации определенного алгоритма. Кроме того, выбранный алгоритм должен позволять быстро и эффективно заполнять буфер граней, описанный в предыдущем пункте.

При этом не имеет значения, в каком пространстве работает алгоритм, так как для моей задачи скорость важнее точности.

1.5.1 Алгоритм Робертса

Алгоритм Робертса [1] — алгоритм удаления невидимых граней и поверхностей, работающий в объектном пространстве, решая задачу только с выпуклыми телами.

Алгоритм выполняется в 3 этапа.

1. **Этап подготовки исходных данных.** На данном этапе должна быть задана информация о телах. Для каждого тела сцены должна быть сформирована матрица тела V. Размерность матрицы – $4 \cdot n$, где n – количество граней тела. Каждый столбец матрицы представляет собой четыре коэффициента уравнения плоскости $ax+by+cz+d = 0$, проходящей через очередную грань.

   Матрица тела должна быть сформирована корректно, то есть любая точка, расположенная внутри тела, должна располагаться по положительную сторону от каждой грани тела. В случае, если для очередной грани условие не выполняется, соответствующий столбец матрицы надо умножить на -1. Для проведения проверки следует взять точку, расположенную внутри тела.

2. **Этап удаления ребер, экранируемых самим телом.** На данном этапе рассматривается вектор взгляда E = {0, 0,-1, 0}. Для определения невидимых граней достаточно умножить вектор E на матрицу тела V. Отрицательные компоненты полученного вектора будут соответствовать невидимым граням.

3. **Этап удаления невидимых ребер, экранируемых другими телами сцены.** На данном этапе для определения невидимых точек ребра требуется построить луч, соединяющий точку наблюдения с точкой на ребре. Точка будет невидимой, если луч на своем пути встречает в качестве преграды рассматриваемое тело. Если тело является преградой, то луч должен пройти через тело. Если луч проходит через тело, то он находится по положительную сторону от каждой грани тела.

Основным преимуществом данного алгоритма является точность вычислений. Она достигается за счет работы в объектном пространстве, в отличии от большинства других алгоритмов.

Серьезным недостатком является вычислительная трудоемкость алгоритма. В теории она растет как квадрат количества объектов. Поэтому при большом количестве объектов на сцене, этот алгоритм будет работать достаточно медленно. Но для решения этой проблемы можно использовать разные оптимизации для повышения эффективности, например, сортировку по оси z. Тем не менее, некоторые из оптимизаций крайне сложны, что затрудняет реализацию этого алгоритма.

Еще один весомый недостаток этого алгоритма - все тела сцены должны быть выпуклыми. Данная проблема также приводит к усложнению алгоритма, так как потребуется прибегнуть к проверке объектов на выпуклость и их разбиению на выпуклые многоугольники, что сильно замедлит алгоритм при большом количестве тел.

Вывод

Алгоритм Робертса не подходит для решения поставленной задачи по следующим причинам:

— от программы не требуется той точности визуализации объектов, которую предоставляет этот алгоритм;

— на сцене может находиться множество объектов (зачастую - невыпуклых), что сильно замедлит скорость работы алгоритма. Таким образом, алгоритм не удовлетворяет требованиям к скорости выполнения алгоритма;

— реализация модификаций, позволяющих приблизить рост сложности алгоритма к линейной, очень трудозатратна;

— при использовании этого алгоритма возникают трудности с заполнением «буфера граней».

1.5.2 Алгоритм Варнока

Алгоритм Варнока [1] — алгоритм удаления невидимых граней и поверхностей, работающий в пространстве изображения.

Основной идеей данного алгоритма является рекурсивное разбиении области экрана на более мелкие подобласти (рисунок 1.1). Для каждой подобласти определяются связанные с ней многоугольники и те из них, видимость которых тривиальна, изображаются на экране. В случае невозможности однозначно определить видимость части многоугольника разбиение области повторяется, и для каждой из вновь полученных подобластей рекурсивно применяется процедура определения видимости. В результате работы алгоритма получаются области, содержащие не более одного многоугольника, либо разбиение продолжается до тех пор, пока размер области не станет равен одному пикселю. В этом случае для полученного пикселя вычисляется значение глубины каждого многоугольника (координата Z), и визуализируется тот из них, у которого значение этой координаты больше.

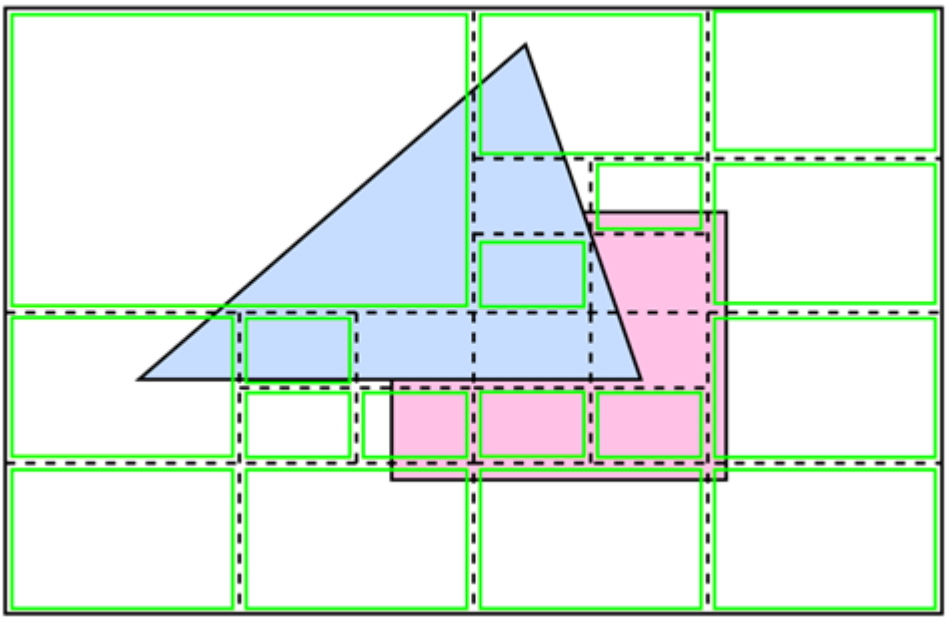

Рисунок 1.1 – Разбиении области экрана алгоритмом Варнока

Достоинством данного алгоритма является простота реализации и высокая эффективность в случае, если размеры перекрываемых областей невелики.

Его главным недостатком является использование рекурсивных вызовов, что значительно снижает скорость выполнения в случае больших размеров перекрываемых областей.

Вывод

Алгоритм Варнока не подходит для решения моей задачи по следующим причинам:

— алгоритм зависит от количества перекрывающихся областей на сцене, поэтому при его использовании отрисовка сцены может работать неоднородно;

— при использовании этого алгоритма возникают трудности с заполнением «буфера граней».

1.5.3 Алгоритм, использующий Z-буфер

Алгоритм, использующий Z-буфер [1] — алгоритм удаления невидимых граней и поверхностей, работающий в пространстве изображения.

Идея алгоритма, использующего Z-буфер (далее – алгоритм Z-буфера) заключается в использовании буфера, хранящего глубину каждого пикселя.

В ходе работы алгоритма значение глубины каждого нового пикселя, заносимого в буфер кадра, сравнивается с глубиной того пикселя, который уже занесен в Z-буфер. Если это сравнение показывает, что новый пиксель расположен ближе к наблюдателю, чем пиксель, уже находящийся в буфере кадра, то новый пиксель заносится в буфер кадра и производится корректировка Z-буфера: в него заносится глубина нового пикселя. Если же значение глубины нового пикселя меньше, чем хранящееся в буфере, то осуществляется переход к следующей точке.

Достоинством данного алгоритма является его простота, которая не мешает решению задачи удаления поверхностей и визуализации их пересечения. Также в этом алгоритме отсутствует необходимость предварительной сортировки объектов по глубине, то есть они могут обрабатываться в произвольном порядке. Более того, время работы алгоритм линейно зависит от количества граней на сцене, что делает его одним из самых быстродействующих. Помимо прочего, важным преимуществом этого алгоритма является возможность применения параллельных вычислений.

Недостаткам данного алгоритма является необходимость выделения памяти под два буфера, каждый из которых имеет размер равный количеству пикселей на экране, но для современных компьютеров этот недостаток является незначительным.

Вывод

Данный алгоритм наилучшим образом подходит для решения поставленной задачи, так как:

— из-за отсутствия необходимости предварительной сортировки, алго-

ритм способен работать быстрее других даже с множеством объектов на сцене;

— при использовании этого алгоритма можно реализовать заполнение «буфера граней» вместе с заполнением двух других буферов;

— при реализации этого алгоритма возможно использование многопоточности.

1.5.4 Алгоритм обратной трассировки лучей

Алгоритм обратной трассировки лучей [1] — алгоритм удаления невидимых граней и поверхностей, работающий в пространстве изображения.

Идея данного алгоритма заключается в том, что для каждого пикселя картинной плоскости определяется ближайшая к нему грань. Для этого через пиксель выпускается луч, находятся все пересечения луча с гранями и среди пересечений выбирается ближайшее.

К достоинствам данного алгоритма можно отнести возможность получения изображения гладких объектов без аппроксимации их примитивами (например, треугольниками). Благодаря отслеживанию пути, пройденного лучом, появляется возможность реализовать глобальную модель освещения, учитывающую отражения и преломления света. Качество полученного изображения получается очень реалистичным, этот метод отлично подходит для создания фотореалистичных сцен. Также важным преимуществом этого алгоритма является возможность применения параллельных вычислений, т.к. расчет отдельной точки выполняется независимо от других точек.

Главным недостатком алгоритма трассировки является необходимость создавать большое число лучей, проходящих через сцену, которые могут раздваиваться на отраженный и преломленный лучи, для которых все вычисления повторяются. Это приводит к существенному снижению скорости работы программы.

Вывод

Данный алгоритм не отвечает главному требованию – скорости работы. От реализуемого продукта не требуется высокой реалистичности синтезируемого изображения и возможности работы с поверхностями, заданными в математической форме. Также в моей программе не предполагается присутствие эффектов отражения и преломления света. Таким образом, при заметном замедлении работы программы, качество изображения заметно не улучшится. Указанные факты говорят о том, что использовать алгоритм обратной трассировки лучей для моей задачи нецелесообразно.

1.5.5 Выбор оптимального алгоритма

С учетом изложенных выше заключений, в качестве алгоритма удаления невидимых линий и поверхностей был выбран алгоритм, использующий Z-буфер.

1.6 Анализ методов закраски граней

Существуют три основных алгоритма, позволяющих закрасить полигональную модель: простая закраска, закраска по Гуро и закраска по Фонгу.

1.6.1 Простая закраска

Суть алгоритма простой закраски [1] заключается в том, что для каждой грани объекта находится вектор нормали, и с его помощью в соответствии с выбранной моделью освещения вычисляется значение интенсивности, с которой закрашивается вся грань.

Данный метод закраски обладает большим быстродействием, так как его сложность линейно зависит от количества граней на сцене, однако все

пиксели грани имеют одинаковую интенсивность, и сцена выглядит нереалистично. Тем не менее, этот метод крайне прост в реализации и совершенно не требователен к ресурсам.

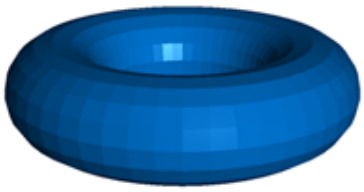

Рисунок 1.2 – Пример простой закраски

1.6.2 Закраска по Гуро

Основная идея алгоритма закраски по Гуро [1] заключается в билинейной интерполяции интенсивностей, за счет которой устраняется дискретность изменения интенсивности и создается иллюзия гладкой криволинейной поверхности. Иначе говоря, разные точки грани закрашиваются с разными значениями интенсивности.

С помощью этого метода получаются достаточно реалистичные изображения, однако все объекты кажутся матовыми.

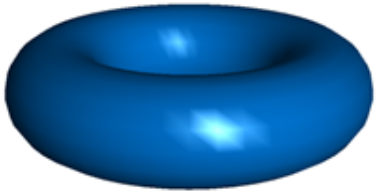

Рисунок 1.3 – Пример закраски по Гуро

1.6.3 Закраска по Фонгу

Закраска по Фонгу [1] по своей идее похожа на закраску по Гуро, но ее отличие состоит в том, что в методе Гуро по всем точкам полигона интерполируются значения интенсивностей, а в методе Фонга – вектора

нормалей, и с их помощью для каждой точки находится значение интенсивности.

Эта закраска требует больших вычислительных затрат, чем предыдущие, однако она позволяет достигнуть лучшей локальной аппроксимации кривизны поверхности и, следовательно, с ее помощью получается более реалистичное изображение.

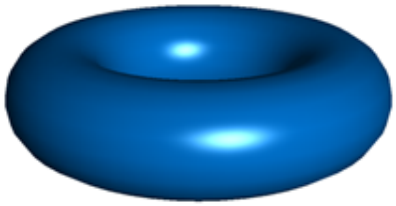

Рисунок 1.4 – Пример закраски по Фонгу

1.6.4 Выбор оптимального алгоритма закраски

Моя задача предполагает, что пользователь должен четко видеть ребра и вершины редактируемой модели. А закраски по Фонгу и Гуро будут сглаживать ребра, тем самым мешая пользователю сосредоточиться на моделировании. К тому же, они будут замедлять время отрисовки сцены.

Таким образом, для моей задачи оптимальным выбором будет простой алгоритм закраски.

2 Конструкторский раздел

В рамках данного раздела приведены основные требования к программному обеспечению (далее – ПО), описаны используемые структуры данных и алгоритмы.

2.1 Требования к программному обеспечению

Разрабатываемое ПО должно предоставлять следующие возможности:

— визуальное отображение сцены с объектами;

— добавление моделей из числа примитивов и их удаление;

— загрузка сцены из файла;

— выбор конкретной модели или её составляющей части (вершины, ребра или грани);

— преобразование конкретной модели или её составляющей части.

Более того, модели на сцене должны состоять только из треугольных граней.

2.2 Основные структуры данных

Для формализации общего алгоритма синтеза изображения необходимо определить использующиеся в программе структуры данных. Ниже приведены основные структуры данных и их составляющие.

1. **Сцена.** Состоит из

    — списка моделей,

    — списка источников света,

— камеры.

2. **Модель.** Состоит из

   — названия,

   — списка вершин,

   — списка ребер,

   — списка граней,

   — вершины, определяющей центр модели.

3. **Источник света.** Состоит из

   — названия,

   — вершины, определяющей местоположение источника,

   — интенсивности источника.

4. **Камера.** Состоит из

   — вершины, определяющей местоположение источника,

   — вершины, к которой направлен объектив камеры.

5. **Вершина.** Состоит из

   — начальных ее координат,

   — матрицы трансформации.

6. **Ребро.** Состоит из

   — начальной вершины ребра,

   — конечной вершины ребра.

7. **Грань.** Состоит из

   — списка принадлежащих ей вершин,

   — списка принадлежащих ей ребер.

2.3 Алгоритм визуального отображения сцены

Схема общего алгоритма визуального отображения сцены изображена на рисунке 2.1.

Схема алгоритма обработки грани в несколько потоков представлена на рисунке 2.2, а схема обработки обрамляющего прямоугольника грани – на рисунке 2.3.

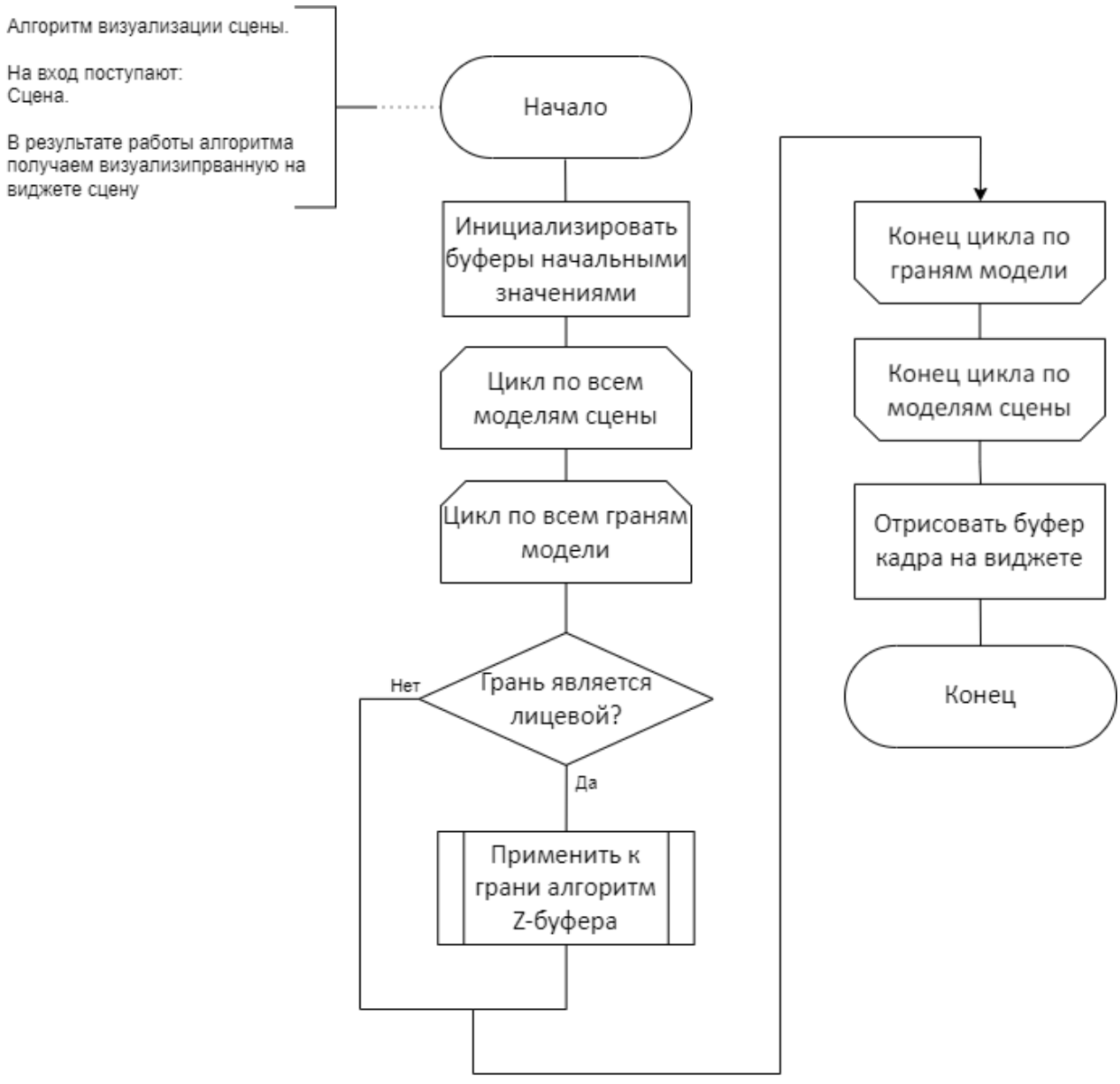

Рисунок 2.1 – Алгоритм визуализации сцены

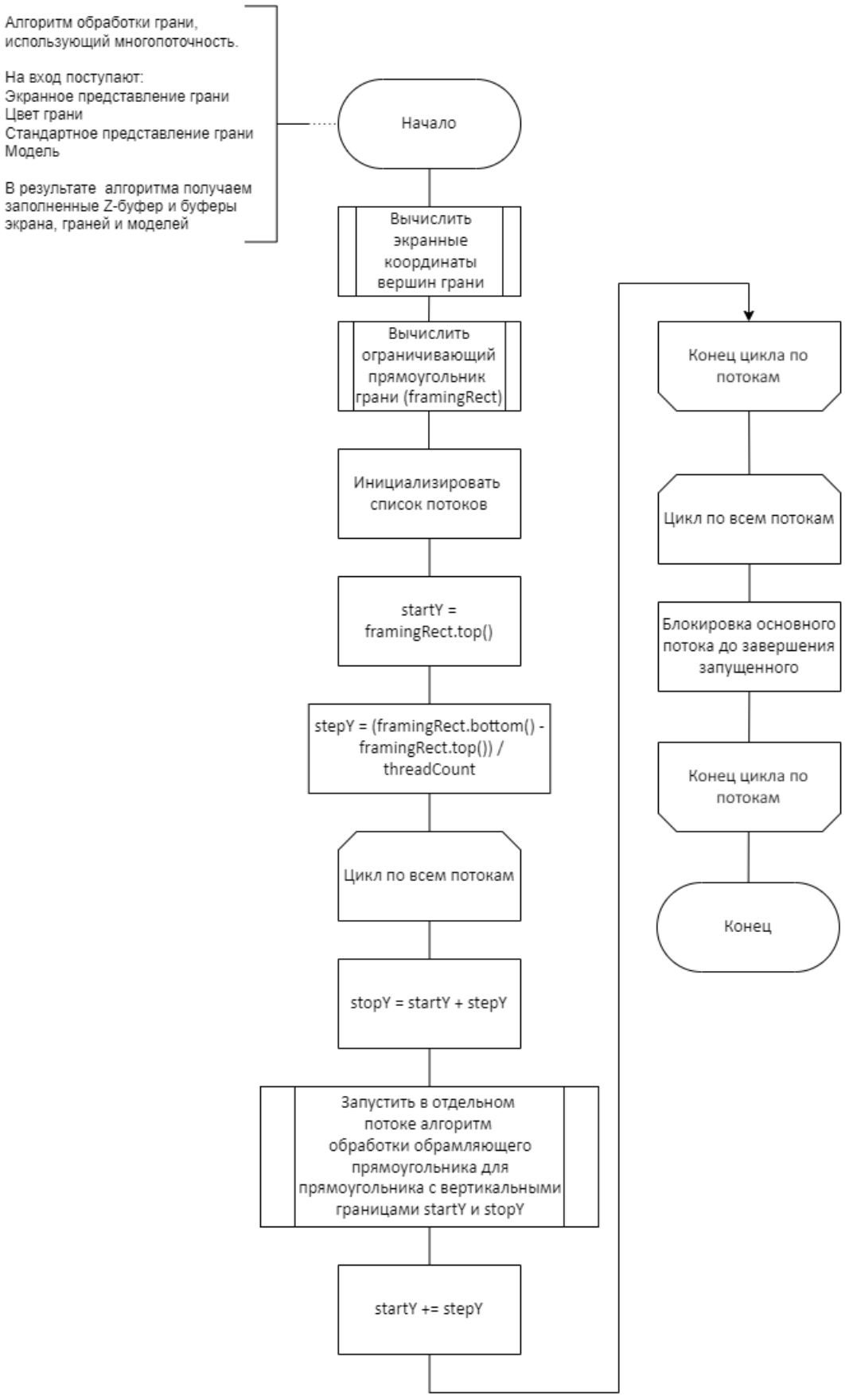

Рисунок 2.2 – Алгоритм обработки грани с использованием многопоточности

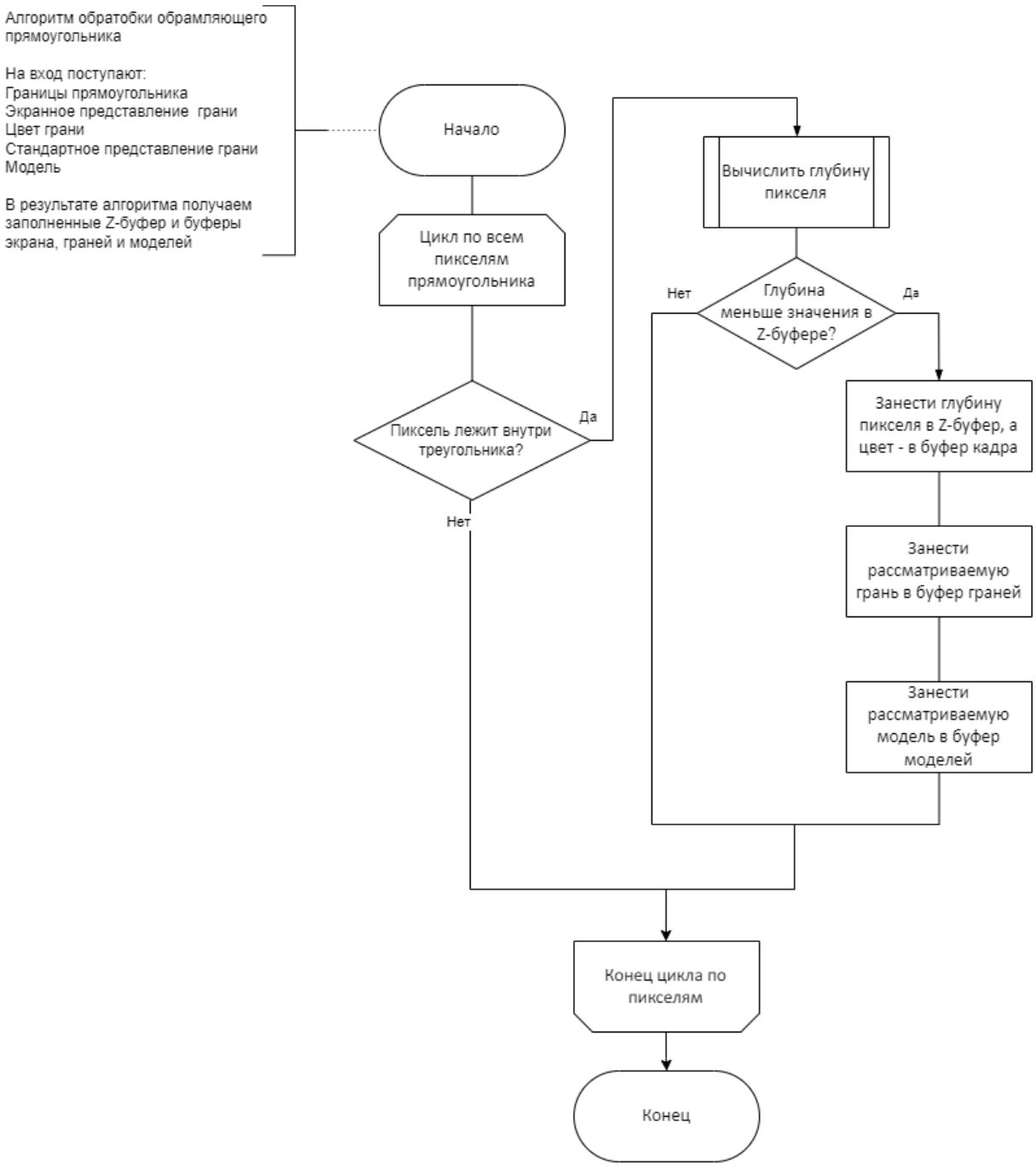

Рисунок 2.3 – Алгоритм обработки обрамляющего прямоугольника

2.4 Поиск экранных координат вершины

В рамках алгоритма обработки грани необходимо вычислять экранные координаты вершин грани.

Поиск экранных координат вершины предполагает последовательное применение к начальным ее координатам следующих преобразований:

1. аффинных преобразований;
2. преобразований камеры;
3. перспективных преобразований, если включена соответствующая опция.

2.4.1 Аффинные преобразования

Первым этапом получения экранных координат вершины является применение к ней аффинных преобразований. Это преобразование осуществляется путем умножения матрицы трансформации вершины на матрицу соответствующего аффинного преобразования.

В данном проекте над вершиной можно произвести следующие аффинные преобразования:

Перенос.

Перенос в трехмерном пространстве задается значениями переноса вдоль осей OX, OY, OZ - dx, dy, dz соответственно. Матрица переноса имеет вид:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{pmatrix} \tag{2.1}$$

Поворот.

Поворот описывается углом θ и осью вращения. Матрицы поворота имеют вид:

— Вокруг оси OX:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\ \theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.2}$$

— Вокруг оси OY:

$$\begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.3}$$

— Вокруг оси OZ:

$$\begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.4}$$

Масштабирование.

Масштабирование в трехмерном пространстве задается значениями масштабирования вдоль осей OX, OY, OZ - kx, ky, kz соответственно. Матрица масштабирования имеет вид:

$$\begin{pmatrix} kx & 0 & 0 & 0 \\ 0 & ky & 0 & 0 \\ 0 & 0 & kz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.5}$$

При этом важно заметить, что перед применением преобразований поворота и масштабирования необходимо предварительно осуществить пе-

ренос вершины так, чтобы центр соответствующего преобразования совпал с центром координат.

2.4.2 Приведение к пространству камеры

Приведение к пространству камеры производится путем умножения матрицы трансформации вершины на матрицу преобразования камеры. При этом стоит учитывать, что над камерой возможно осуществления лишь 2 вида преобразования: перенос и поворот, которые производятся над вершиной, задающей местоположение камеры.

2.4.3 Проецирование вершины

После перехода в пространство камеры необходимо спроецировать вершину на картинную плоскость. В данном курсовом проекте используется перспективная и ортогональная проекции.

Для получения ортогональной проекции вершины не требуется каких-либо дополнительных преобразований.

В моем проекте используется одноточечная перспективная проекция [2]. Для получения перспективной проекции вершины необходимо умножить полученную на предыдущих шагах матрицу ее трансформации на матрицу одноточечной перспективной проекции:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{pmatrix}, \text{где } r = \frac{1}{F}, F\text{– фокус} \tag{2.6}$$

2.4.4 Определение нелицевых граней

С помощью отбрасывания нелицевых граней моделей при построении изображения можно существенно сократить временные затраты, так как грани, невидимые по отношению к камере, визуализироваться не будут.

Пусть $\bar{N}$ — вектор внешней нормали к грани модели, $\bar{V}$ – вектор от камеры до любой точки грани.

Для определения видимости грани используется формула:

$$(\bar{N}, \bar{V}) = \begin{cases} \geq 0, & \text{если грань невидима} \\ < 0, & \text{если грань видима} \end{cases} \tag{2.7}$$

2.5 Вычисление глубины пикселя

В рамках алгоритма обработки обрамляющего прямоугольника требуется определить, лежит ли пиксель внутри треугольника, задающего грань. И, если лежит, вычислить его глубину.

Для решения этих задач для каждого пикселя ограничивающего прямоугольника находятся его барицентрические координаты относительно вершин треугольной грани.

Пусть A, B, C – вершины полигона, P – пиксель внутри ограничивающего прямоугольника.

Площадь треугольника можно найти по следующей формуле:

$$square = (A_y - C_y) \cdot (B_x - C_x) + (B_y - C_y) \cdot (C_x - A_x) \tag{2.8}$$

Тогда барицентрические координаты пикселя равны:

$$\alpha = \frac{(P_y - C_y) \cdot (B_x - C_x) + (B_y - C_y) \cdot (C_x - P_x)}{square} \tag{2.9}$$

$$\beta = \frac{(P_y - A_y) \cdot (C_x - A_x) + (C_y - A_y) \cdot (A_x - P_x)}{square} \tag{2.10}$$

$$\gamma = 1 - \ \alpha - \ \beta \tag{2.11}$$

В случае, если хоть одна из барицентрических координат отрицательна, то пиксель лежит вне полигона. Если пиксель лежит внутри треугольника, то найти значение его глубины можно по следующей формуле:

$$z = \frac{1}{\frac{\alpha}{A_z} + \frac{\beta}{B_z} + \frac{\gamma}{C_z}} \tag{2.12}$$

2.6 Простая модель освещения

В простом методе освещения цвет пикселя рассчитывается по закону Ламберта. При этом положение наблюдателя не имеет значения, т.к. диффузно отраженный свет рассеивается равномерно по всем направлениям (рисунок 2.4).

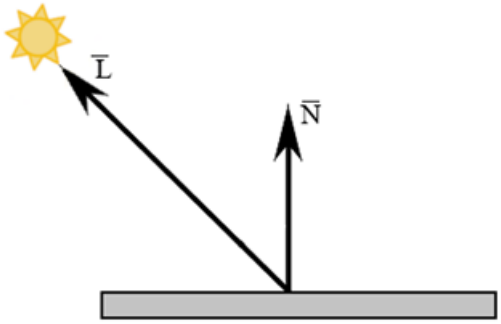

Рисунок 2.4 – Простая модель освещения

Пусть I – результирующая интенсивность света в точке, $\bar{L}$ – вектор от точки до источника, $\bar{N}$ – вектор нормали, I_0 – интенсивность источника, k_d – коэффициент диффузного освещения.

Формула расчёта интенсивности имеет следующий вид:

$$I = I_0 \cdot k_d \cdot \left(\bar{L}, \bar{N}\right) \tag{2.13}$$

2.7 Алгоритм выбора составляющей части модели

Общая схема алгоритма выбора составляющей части модели изображена на рисунке 2.5.

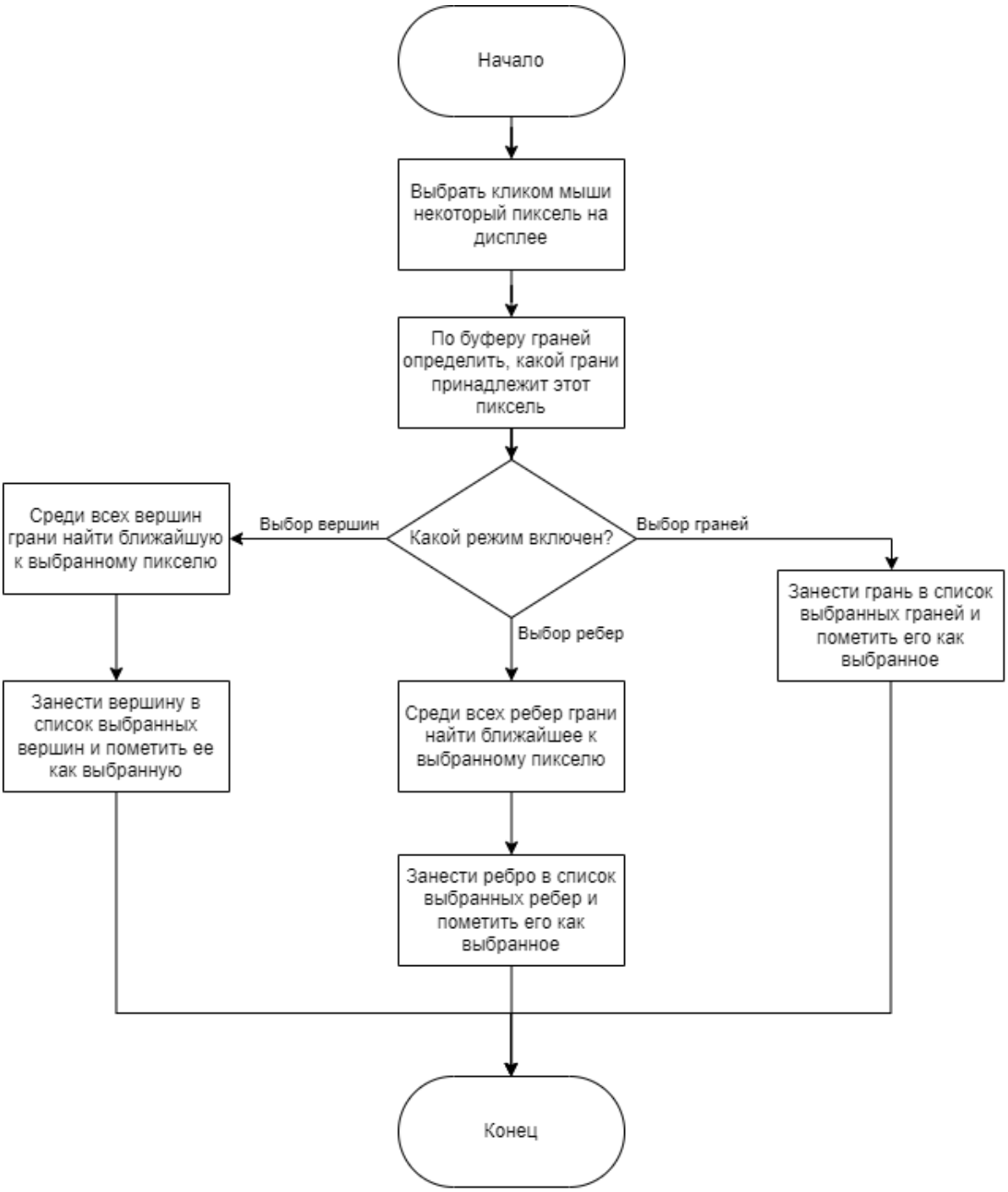

Рисунок 2.5 – Алгоритм выбора составляющей части модели

3 Технологический раздел

В данном разделе выбраны средства реализации, приведена структура программы и описание классов, продемонстрирован интерфейс программы.

3.1 Выбор средств реализации

В качестве языка программирования был выбран C++ [3], так как:

— Я ознакомился с этим языком в рамках курса по ООП.

— Данный язык программирования поддерживает объектно-ориентированную модель разработки, что позволяет четко структурировать программу.

— Язык C++ поддерживает многопоточность и предоставляет встроенные средства для осуществления замеров времени работы.

В качестве графического фреймворка был выбран Qt [4], так как этот фреймворк поддерживает Qt Designer, приложение для разработки UI и предоставляет множество средств для отображения визуальной сцены на виджете в приложении.

В качестве среды разработки была выбрана «Microsoft Visual Studio» [5], так как:

— Для студентов это ПО распространяется бесплатно.

— Эта среда разработки поддерживает интеграцию с Qt.

— Она имеет встроенную технологию автодополнения IntelliSense[6], позволяющую сильно облегчить процесс разработки.

— Имеет встроенный отладчик.

3.2 Структура программы

Схема UML классов моей программы изображена на рисунке 3.1. Приведенная схема отображает отношения между кассами и их интерфейс.

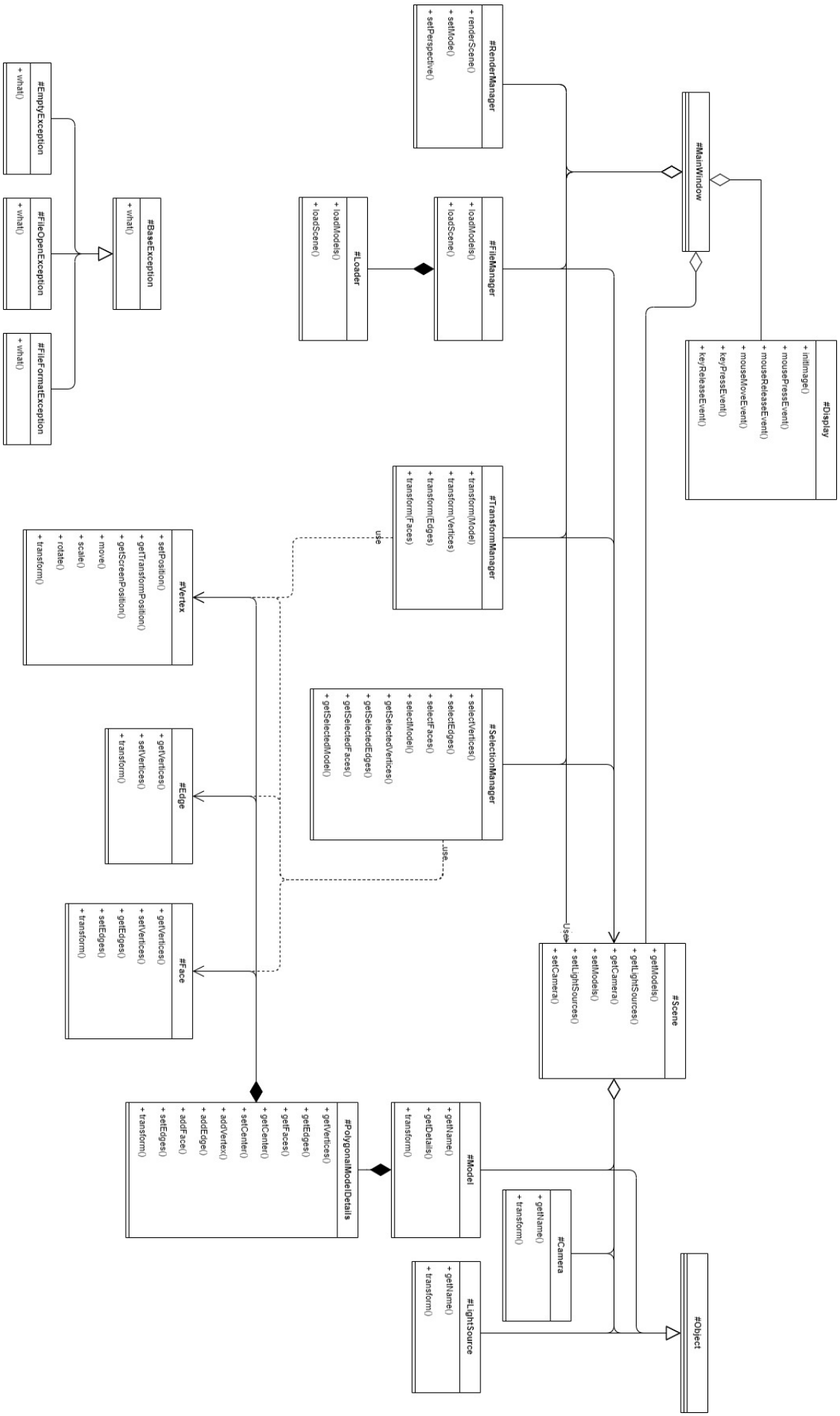

Рисунок 3.1 – UML-схема классов

3.3 Описание классов и модулей программы

Интерфейс:

— MainWindow (MainWindow.h, MainWindow.cpp) – класс, описывающий интерфейс главного окна приложения. Наследуется от QMainWindow;

— Display (Display.h, Display.cpp) – класс, описывающий виджет для отображения буфера кадра. Наследуется от QWidget;

— MainWindow.ui – форма пользовательского интерфейса главного окна приложения.

Сцена:

Scene (Scene.h, Scene.cpp) – класс, описывающий сцену. Этот класс является контейнером для хранения объектов сцены: источников света, моделей и камеры.

Объекты сцены:

— Object (Object.h) – абстрактный класс объекта;

— LightSource (LightSource.h, LightSource.cpp) – класс, описывающий точечный источник освещения;

— Camera (Camera.h, Camera.cpp) – класс, описывающий камеру;

— Model (Model.h, Model.cpp) – класс, описывающий трехмерную модель.

Составляющие части модели:

— Vertex (Vertex.h, Vertex.cpp) – класс, описывающий вершину модели;

— Edge (Edge.h, Edge.cpp) – класс, описывающий ребро модели;

— Face (Face.h, Face.cpp) – класс, описывающий грань модели;

— PolygonalModelDetails (PolygonalModelDetails.h, PolygonalModelDetails.cpp) – класс, описывающий реализацию полигональной модели.

Менеджеры:

— TransformManager (TransformManager.h, TransformManager.cpp) – класс, описывающий менеджера трансформации. Этот менеджер осуществляет за перенос, поворот и масштабирование моделей и их составляющих частей;

— FileManager (FileManager.h, FileManager.cpp) – класс, описывающий менеджера работы с файлами. Этот менеджер осуществляет загрузку сцены из файла и ее сохранение;

— RenderManager (RenderManager.h, RenderManager.cpp) – класс, описывающий менеджера отрисовки. Этот менеджер осуществляет отрисовку сцены в буфер кадра;

— SelectionManager (SelectionManager.h, SelectionManager.cpp) – класс, описывающий менеджера выбора. Этот менеджер отвечает за выбор моделей и их составляющих частей на дисплее.

Работа с файлами:

Loader (Loader.h, Loader.cpp) – класс, осуществляющий загрузку из файла и валидацию сцены.

3.4 Реализации алгоритмов

В листингах 3.1, 3.2 представлена реализация алгоритмов обработки пикселей обрамляющего прямоугольника и многопоточной обработки грани модели.

Листинг 3.1 – Реализация алгоритма обработки пикселей обрамляющего прямоугольника

```
1  void RenderManager::processFramingRectPixel(
2  unique_ptr<ThreadParams> params
3  ) {
4      int startY = params->startY;
5      int stopY = params->stopY;
6      int startX = params->startX;
7      int stopX = params->stopX;
8
9      const ScreenFace& face = params->face;
10     double square = params->square;
11     const QRgb& color = params->color;
12     const shared_ptr<Face>& basicFace = params->basicFace;
13     const shared_ptr<Model>& model = params->model;
14
15     for (int y = startY; y <= stopY; y++) {
16         for (int x = startX; x <= stopX; x++) {
17
18             auto barCoords =
                   calculateBarycentric(QPoint(x, y), face,
                   square);
19
20             if (barCoords.x() >= -EPS && barCoords.y() >=
                  -EPS && barCoords.z() >= -EPS) {
21                 double z =
                       baryCentricInterpolation(face[0],
                       face[1], face[2], barCoords);
22
23                 this->processPixel(x, y, z, color);
24                 this->updateFaceBuffer(Vector2d(x, y), z,
                       basicFace);
25                 this->updateModelBuffer(Vector2d(x, y),
                       z, model);
26             }
27         }
28     }
29 }
```

Листинг 3.2 – Реализация алгоритма обработки грани в несколько потоков

```
1         void RenderManager::processFace(
2         const ScreenFace& face, const QRect& framingRect,
3         const QRgb& color, const shared_ptr<Face>& basicFace,
4         const shared_ptr<Model>& model
5         ) {
6             double square = (face[0].y() - face[2].y()) *
                  (face[1].x() - face[2].x()) +
7             (face[1].y() - face[2].y()) * (face[2].x() -
                  face[0].x());
8
9             int threadCount = config.threadCount;
10            vector<thread> threads(threadCount);
11
12            double startY = framingRect.top();
13            double stepY = (framingRect.bottom() -
                  framingRect.top()) / double(threadCount);
14
15            for (auto& thread : threads) {
16                unique_ptr<ThreadParams> params =
                      unique_ptr<ThreadParams>(new ThreadParams{
17                        0, 0, framingRect.left(),
                              framingRect.right(), face, square, color,
                              basicFace, model
18                    });
19
20                params->startY = round(startY);
21                params->stopY = round(startY + stepY);
22                thread = std::thread(
23                &RenderManager::processFramingRectPixel,
24                this, move(params));
25
26                startY += stepY;
27            }
28
29            for (auto& thread : threads) {
30                thread.join();
31            }
```

3.5 Интерфейс программы

Главное окно программы (рисунок 3.2) имеет несколько виджетов на которых располагаются основные элементы интерфейса:

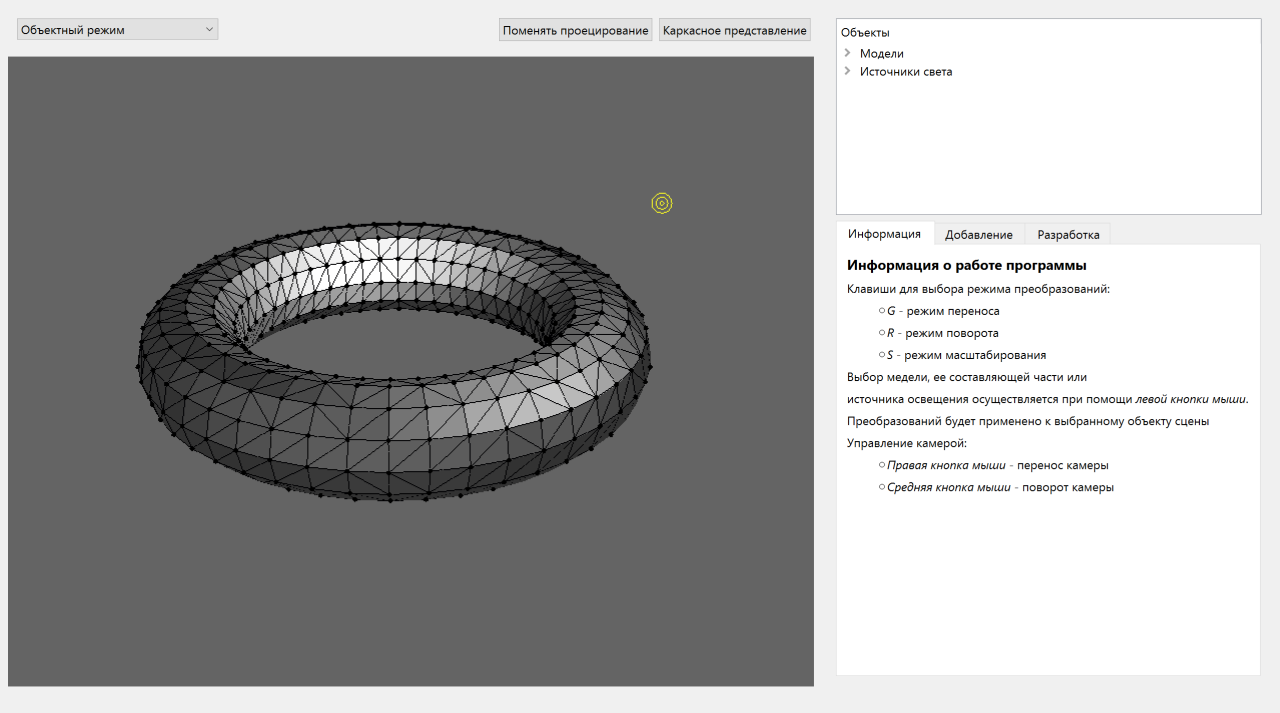

Рисунок 3.2 – Основное окно программы

— ПО центру экрана находится виджет, на котором визуализируется сцена;

— Справа сверху располагается виджет с наименованиями объектов на сцене;

— Справа снизу располагается страничный виджет с тремя вкладками:

1. Информация – вкладка с информацией об элементах управления;
2. Добавление (рисунок 3.3) – вкладка с кнопками для добавление примитивов на сцену;
3. Разработка – служебная вкладка с вспомогательными функциями для отладки программы.

— Сверху располагается «Комбобокс» для выбора режима редактирования и кнопки для изменения проекции и представления модели.

Интерфейс программы также содержит меню со следующими пунктами:

— Файл – позволяет загрузить сцену или модель;

— Справка – отображает справочную информацию о программе.

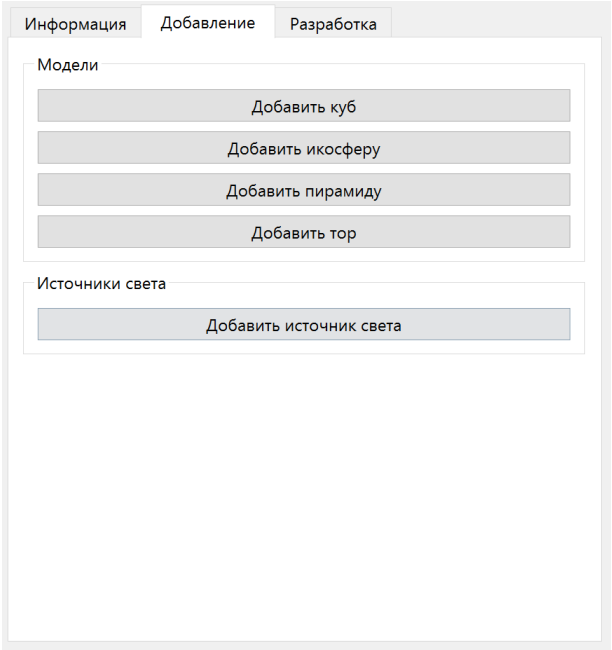

Рисунок 3.3 – Вкладка «Добавление»

4 Исследовательский раздел

В данном разделе будут приведён пример работы программы, а также проведён сравнительный анализ многопоточной и однопоточной реализаций.

4.1 Технические характеристики

Замеры времени выполнялись на личном ноутбуке. Технические характеристики устройства, на котором выполнялось тестирование представлены далее:

— операционная система Windows 10 Домашняя;

— память 16 Гбайт;

— процессор 3.20 ГГц 4-ядерный процессор Intel Core i5 11-го поколения.

— процессор имеет 4 физических и 8 логических ядер.

Во время замеров ноутбук был включен в сеть электропитания.

Для корректности сравнения многопоточной и однопоточной реализаций была выключена оптимизация при компиляции.

4.2 Время выполнения алгоритмов

Для замера времени работы алгоритмов использовалась функция chrono::high_resolution_clock::now(...) из библиотеки chrono [7] на C++.

Результаты замеров времени работы многопоточной и однопоточной реализаций алгоритмов приведены в таблице 4.1. Строчка с количеством потоков 0 означает не распараллеленную реализацию алгоритма. На рисунке 4.1 приведена графическая интерпретация замеров времени. Замеры производились по 50 раз для каждого числа порожденных потоков на модели, которая представляет собой квадратную плоскость, состоящую из двух граней.

Таблица 4.1 – Результаты замеров времени

Количество порожденных потоков	Время, мкс
0	254569
1	293457
2	207791
4	128807
8	84151
16	66569
32	61264
64	65488
128	75354

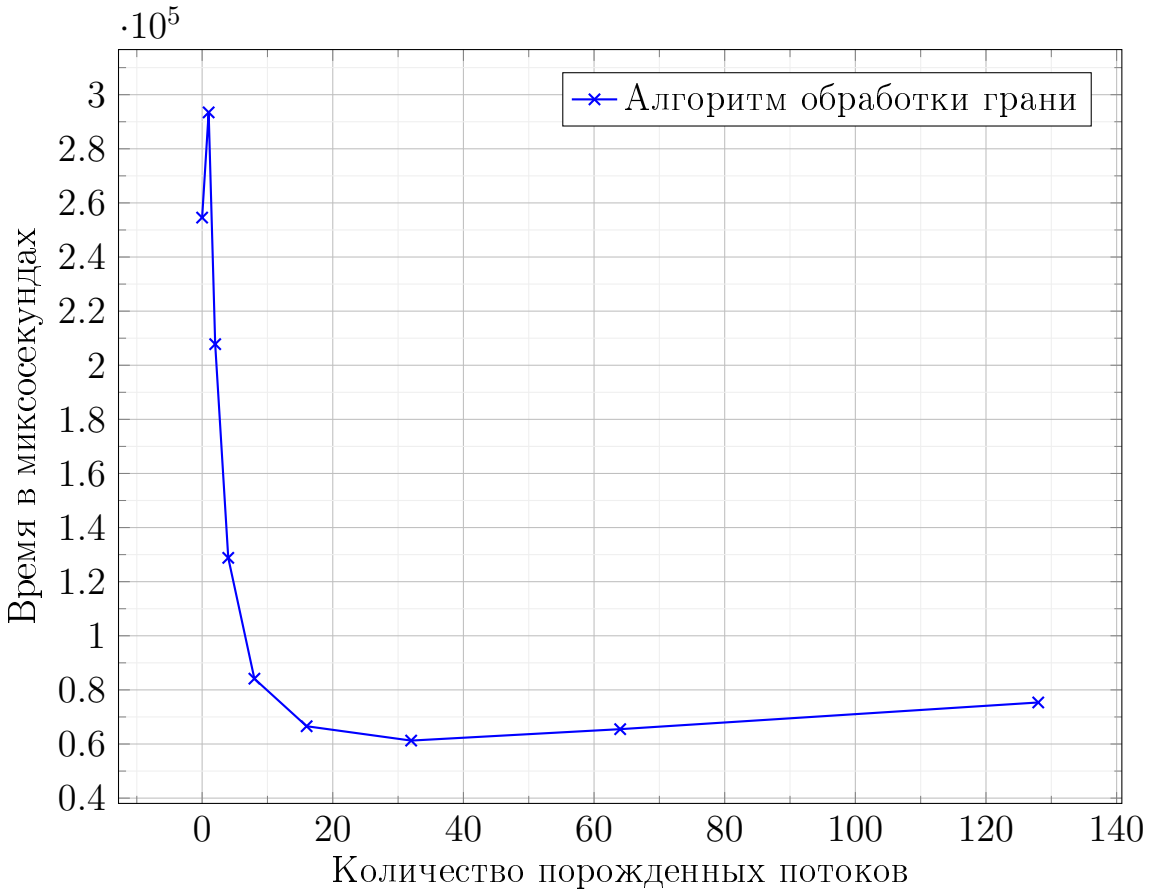

Рисунок 4.1 – Время работы реализации алгоритма обработки грани в зависимости от количества потоков

Из результатов замеров видно, что при числе порожденных потоков M = 1, многопоточный алгоритм работает в 1.15 раз дольше, чем алгоритм с отсутствующей параллельностью. Это происходит из-за затрат на диспетчеризацию этого потока.

Тем не менее, при M = 2 и более, многопоточный алгоритм начинает сильно выигрывать по времени у не распараллеленного алгоритма. Наибольшую эффективность по времени работы удается достичь при M = 32, тогда многопоточный алгоритм работает примерно в 4.1 раза быстрее, чем алгоритм без использования многопоточности. Это происходит из-за того, что при M > 32 затраты на диспетчеризацию M потоков превышают преимущество от использования многопоточности.

Также важно заметить, что замеры времени производились на программе, скомпилированной без оптимизаций по скорости работы. При включенной оптимизации результаты могут отличаться.

Вывод

Многопоточная реализация с использованием 32 порожденных потоков показала наилучший результат. Такая реализация оказалась эффективнее примерно в 4.1 раз чем реализация, не использующая многопоточность.

Рекомендуется использовать число порожденных потоков примерно в 4 раза большее, чем число логических ядер.

Заключение

Поставленная цель была достигнута — разработан редактор трехмерных моделей, предоставляющий пользователю возможность производить базовые преобразования над полигональной моделью и ее составляющими частями (вершинами, ребрами, гранями).

Для достижения поставленной цели были решены следующие задачи:

— изучены существующие алгоритмы компьютерной графики и проведен их анализ. Выбраны наиболее подходящие из них для реализации редактора трехмерных моделей;

— спроектировано программное обеспечение, предоставляющее пользователю необходимые функции;

— реализовано спроектированное программное обеспечение;

— проведен сравнительный анализ последовательной и параллельной реализаций алгоритма Z-буфера.

Разработанный программный продукт в режиме реального времени синтезирует трехмерное изображение при помощи алгоритмов компьютерной графики. Программа реализована таким образом, что пользователь может добавлять новые объекты на сцену, производить трансформацию над их составными частями, изменять положение источника света.

Из результатов проведенного сравнительного анализа следует, что наилучший результат показала многопоточная реализация с использованием 32 порожденных потоков. Таким образом, наиболее оптимальное число порожденных потоков примерно в 4 раза превышает число логических ядер.

Список использованных источников

[1] Ф. Роджерс Д. Алгоритмические основы машинной графики. – М.: Издательство «Мир», 1989.

[2] Ф. Роджерс Д. Математические основы машинной графики. – М.: Издательство «Мир», 2001.

[3] Программирование на C/C++ [Электронный ресурс]. Режим доступа: `http://www.c-cpp.ru/` (дата обращения: 11.10.2022).

[4] Qt framework [Электронный ресурс]. Режим доступа: `https://www.qt.io/product/framework` (дата обращения: 11.10.2022).

[5] Visual Studio – Разработка приложений на C и C++ [Электронный ресурс]. Режим доступа: `https://visualstudio.microsoft.com/ru/vs/features/cplusplus/` (дата обращения: 11.10.2022).

[6] IntelliSense [Электронный ресурс]. Режим доступа: `https://learn.microsoft.com/ru-ru/visualstudio/ide/visual-cpp-intellisense?view=vs-2022` (дата обращения: 11.10.2022).

[7] Date and time utilities [Электронный ресурс]. Режим доступа: `https://en.cppreference.com/w/cpp/chrono` (дата обращения: 11.10.2022).