Содержание

Введение

В компьютерной графике большое внимание уделяется алгоритмам получения реалистических изображений. Они должны учитывать физические явления: преломление, отражение. При этом с повышением сложности алгоритмов растёт и их требовательность к системным ресурсам.

Актуальность работы обусловлена необходимостью создания реалистических изображений аппаратными методами, используя оптимальные для этой задачи алгоритмы.

Цель курсовой работы: разработать программу для построения реалистического изображения из перечня геометрических объектов: куб, конус, цилиндр и сфера.

Для достижение поставленное цели, требуется выполнить следующие задачи:

1. Провести анализ существующих алгоритмов и выбрать оптимальные пути для решения основной задачи.

2. Выбрать подходящий способ декомпозиции программы.

3. Выбрать подходящий язык программирования и среду разработки для выполнения работы.

4. Создать программный продукт для решения задачи, реализовать выбранные алгоритмы.

5. Реализовать понятный интерфейс для клиента.

6. Провести исследования на основе полученных результатов.

1 Аналитический раздел

В этом разделе будет приведено описание модели трёхмерного объекта в сцене, рассмотрены формализация объектов сцены и требования к работе программы. Будут рассмотрены алгоритмы построения трёхмерного изображения, методы закраски, модели освещения, а также способы текстуризации и закраски треугольников.

1.1 Описание модели трёхмерного объекта в сцене

Модель трёхмерного объекта в сцене в работе представлена в виде полигональной сетки.

Полигональная сетка – совокупность вершин, рёбер и граней, которые определяют форму многогранного объекта в трёхмерной компьютерной графике. Сетка может состоять из граней произвольной формы, но в курсовой работе грани имеют форму треугольников, так как любой полигон произвольной можно представить как треугольник.

Такой способ является универсальным, так как позволяет описывать трёхмерные объекты произвольной формы. Количество полигонов напрямую влияет на реалистичность визуализации модели, но при этом обработка большого количества граней приводит к увеличению требований к системным ресурсам.

Так как полигоны задают плоскости, то для достижения эффекта неровности можно изменять нормали к поверхностям. При создании модели также нужно указать текстурные координаты.

1.2 Формализация объектов синтезируемой сцены

Сцена состоит из следующих объектов:

1. Геометрический объект – представляется в виде полигональной сетки. В число рассматриваемых геометрических объектов входит куб, сфера, цилиндр и конус. Для описания геометрического объекта требуется указать координаты вершин, ребра и грани между ними, цвет объекта или текстура, которая его покрывает, а также свойства поверхности: коэффициент отражения, коэффициент прозрачности и коэффициент блеска.

2. Источник света – представляется в виде вектора направления света. Также у источника света есть следующие характеристики: место расположения, цвет и интенсивность излучения.

3. Камера - характеризуется своим пространственным положением и направлением просмотра.

Остальное пространство также представлено интенсивностью окружающего света и его цветом.

1.3 Требования к работе программы

Программа должна обеспечивать построение реалистического изображение, но также важно, чтобы пользователь имел возможность без задержек добавлять объекты в сцену и изменять их характеристики: пространственное расположение, свойства поверхности, спектральные характеристики.

Исходя из этого, программа должна работать в двух режимах. В первом режиме пользователь должен иметь возможность быстро проводить операции с объектами, следовательно, упор должен быть сделан на быстроту работы. Во втором режиме пользователь должен получить реалистическое изображение.

1.4 Анализ алгоритмов построения трёхмерного изображения

1.4.1 Алгоритм z-буфера

Алгоритм Z-буфера – один из простейших алгоритмов, который работает в пространстве изображения.

Буфер кадра (регенерации) используется для заполнения атрибутов (интенсивности) каждого пикселя в пространстве изображения. Для него требуется буфер регенерации, в котором запоминаются значения яркости, а также Z-буфер (буфер глубины), куда можно помещать информацию о координате z для каждого пикселя. Сначала в Z-буфер заносятся минимально возможные значения z, а буфер регенерации заполняется значениями пикселя, описывающими фон. Затем каждый многоугольник преобразуется в растровую форму и записывается в буфер регенерации, при этом не производится начального упорядочения.

В процессе работы глубина (значение координаты Z) каждого нового пикселя, который надо занести в буфер кадра, сравнивается с глубиной того пикселя, который уже занесён 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. Кроме того, производится корректировка Z-буфера: в него заносится глубина нового пикселя. Если же глубина (значение координаты Z) нового пикселя меньше глубины хранящегося в буфере, то никаких действий производить не надо.

На рисунке 1.1 представлена иллюстрация работы алгоритма Z-буфера

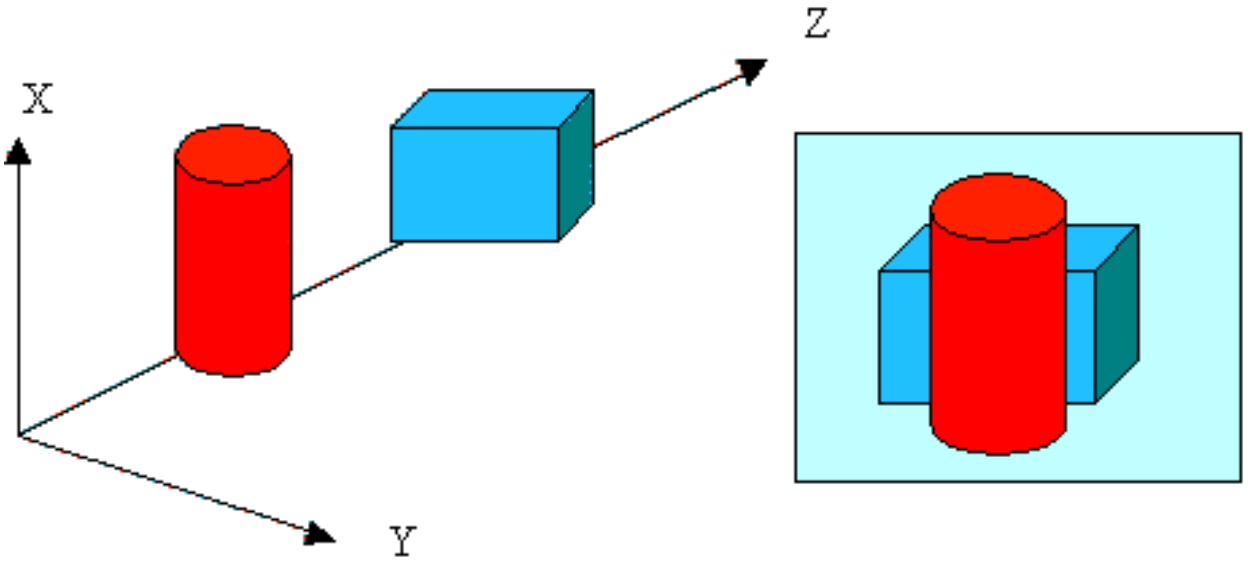

Рисунок. 1.1: Иллюстрация работы алгоритма Z-буфера

К преимуществам алгоритма можно отнести его простоту и отсутствие сортировок. Но у такого подхода есть и много недостатков: идёт большой перерасход памяти, так как хранятся два двухмерных массива. Эффекты прозрачности и просвечивания тяжело реализовать, также возникают проблемы с устранением лестничного эффекта [1].

1.4.2 Алгоритм обратной трассировки лучей

Алгоритм предлагает рассмотреть следующую ситуацию: через каждый пиксел изображения проходит луч, выпущенный из камеры, и программа должна определить точку пересечения этого луча со сценой. Первичный луч – луч, выпущенный из камеры.

На рисунке 1.2 представлена ситуация, когда первичный луч пересекает объект в точке H1:

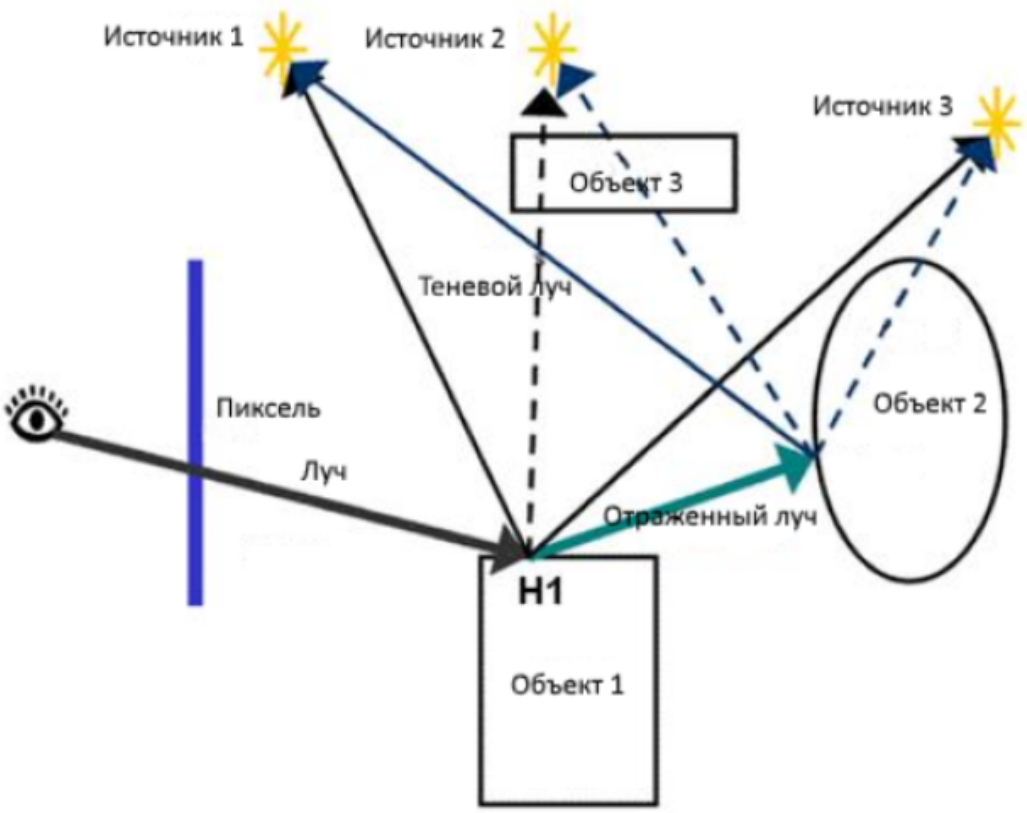

Рисунок. 1.2: Иллюстрация работы алгоритма обратной трассировки лучей

Для источника света определяется, видна ли для него эта точка. Чтобы это сделать, испускается теневой луч из точки сцены к источнику. Если луч пересёк какой-либо объект сцены, то значит, что точка находится в тени, и её не надо освещать. В обратном случае требуется рассчитать степень освещённости точки.

Затем алгоритм рассматривает отражающие свойства объекта: если они есть, то из точки H1 выпускается отражённый луч, и процедура повторяется рекурсивно. Тоже самое происходит при рассмотрении свойств преломления. Этот алгоритм подходит для поставленной задачи, так как полученное изображение соответствует высокому уровню реалистичности. Также данный метод позволяет учесть все физические явления: отражение, преломление, воссоздание теней. [1]

У алгоритма существует недостаток: трассировка лучей каждый раз начинает заново процесс вычисления цвета пиксела, рассматривая каждый луч по отдельности. [2] При этом временные затраты у алгоритма суще-

ственно большие, чем у алгоритма Z-буфера.

1.4.3 Вывод

У программы два режима работы: в первом пользователь добавляет и редактирует объекты, изменяет спектральные характеристики, а во втором уже воссоздаётся реалистическое изображение. В первом режиме важна скорость, поэтому из всех алгоритмов был выбрал Z-буфер. Для второго же важна реалистичность, учёт физических и оптических эффектов, и для решения этой задачи был выбран алгоритм обратной трассировки лучей.

1.5 Анализ алгоритмов закраски

1.5.1 Метод закраски Гуро

Метод Гуро устранить дискретность изменения интенсивности и создать иллюзию гладкой криволинейной поверхности. Он основан на интерполяции интенсивности [3]. Сам алгоритм можно разбить на четыре основных этапа:

1. Вычисление нормалей ко всем полигонам.

2. Определение нормали в вершинах путём усреднения нормалей по всем полигональным граням, которым принадлежит рассматриваемая вершинам. На рисунке 1.3 представлен пример вычисления нормали.

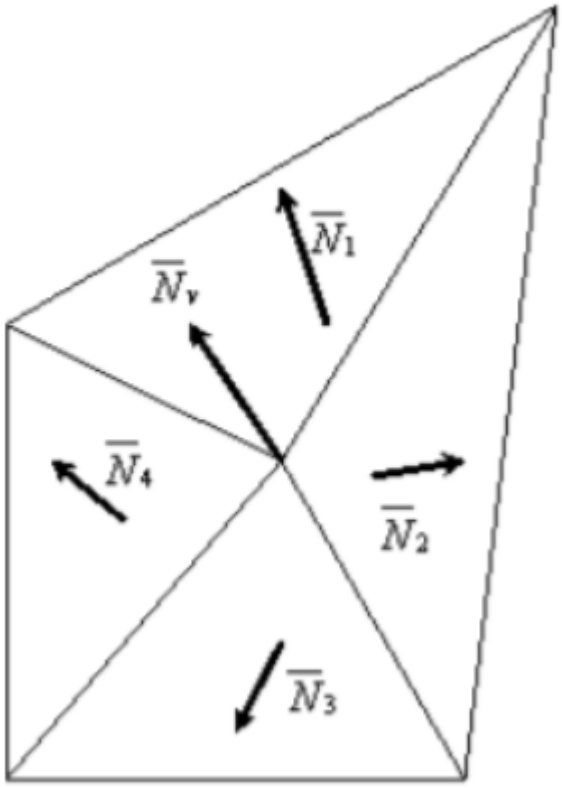

Рисунок. 1.3: Определение всех нормалей к вершине грани

3. Вычисление значения интенсивности в вершинах многоугольника на основе выбранной модели освещения.

4. Закраска каждого многоугольника путём линейной интерполяции в вершинах сначала вдоль рёбер, а затем и между ними.

Метод Гуро применим для небольших граней, расположенных на значительном расстоянии от источника света. Если же размер грани большой, то расстояние от источника света до центра будет меньше, чем до вершин, и центр грани должен выглядеть ярче, чем рёбра. Но из-за линейного закона, используемого в интерполяции, метод не позволяет это сделать, поэтому появляются участки с неестественной освещённостью [1].

1.5.2 Вывод

Вместе с алгоритмом z-буфера будет использоваться метод Гуро, так как он быстрый, и получается приемлемое качество изображения, в том числе можно заметить сглаживание тел. При создании реалистичного изображения будет использоваться алгоритм обратной трассировки лучей, который не требует какого-то ещё дополнительного алгоритма закраски.

1.6 Анализ алгоритмов моделирования освещения

Реалистичность изображения во многом зависит от правильного выбора алгоритма освещения. Все модели освещённости можно разделить на две группы: глобальные и локальные. Локальные модели учитывают только первичный источник света, а глобальные также рассматривают физические явления: отражение света от поверхностей, преломление света.

1.6.1 Модель Ламберта

В этой модели воспроизводится идеальное диффузное освещение. [3] Свет при попадании на поверхность равномерно рассеивается во все стороны. При расчёте учитывается только ориентация нормали поверхности (нормаль $\overline{N}$) и направление на источник света (вектор $\overline{L}$). Пусть I_d – рассеянная составляющая освещённости в точке, K_d – свойство материала воспринимать рассеянное освещение, I_0 – интенсивность рассеянного освещения.

Тогда интенсивность можно рассчитать по формуле:

$$I_d = K_d(\overline{L}, \overline{N})I_0 \tag{1.1}$$

Модель Ламберта является одной из самых простых моделей освещения. Она часто используется в комбинации с другими моделями, так как

практически в любой можно выделить диффузную составляющую. Равномерная часть освещения чаще всего представляется именно моделью Ламберта [4].

1.6.2 Модель Фонга

Модель Фонга основывается на предположении, что освещённость каждой точки можно разбить на три компоненты [4]:

- Фоновое освещение (ambient) – оно присутствует на любом участке сцены, не зависит от источников света, потому считается константой;
- Рассеянный свет (diffuse) – рассчитывается также, как в модели Ламберта;
- Бликовая составляющая (specular) – зависит от того, насколько близко находятся вектор отражённого луча и вектор до наблюдателя.

Свойства источника определяют мощность излучения для каждой из компонент, а свойства материала – способность объекта воспринимать свет.

Пусть $\overline{N}$ – вектор нормали к поверхности в точке, $\overline{L}$ – падающий луч, $\overline{R}$ – отражённый луч, $\overline{V}$ – вектор, направленный к наблюдателю, k_a – коэффициент фонового освещения, k_d – коэффициент диффузного освещения, k_s – коэффициент зеркального освещения, p – степень, аппроксимирующая пространственное распределение зеркально отражённого света. Тогда интенсивность света подсчитывается формулой 1.2:

$$I_a = K_a * I_a + K_d(\overline{N}, \overline{L}) + K_s(\overline{R}, \overline{V})^p \tag{1.2}$$

На рисунке 1.4 приведён пример работы модели Фонга:

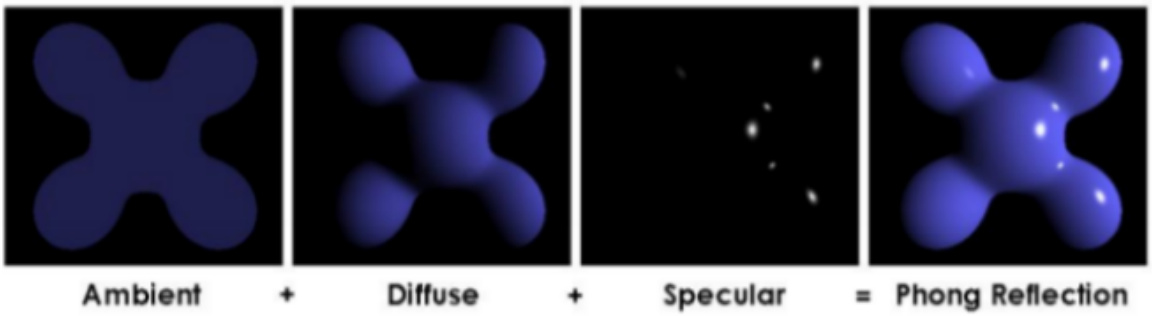

Рисунок. 1.4: Иллюстрация работы модели Фонга

1.6.3 Модель Уиттеда

Эта модель освещения позволяет рассчитать интенсивность отражённого к наблюдателю луча в каждом пикселе изображения. Используемый в проекте вариант учитывает также свет от других объектов сцены или пропущенный сквозь них. [4] Помимо направления взгляда, отражённого луча, источника света, также рассматривается и составляющая преломления. [2]

Пусть K_a – коэффициент рассеянного отражения,K_d – коэффициент диффузного отражения, K_s – коэффициент зеркальности, K_r – коэффициент отражения, K_t – коэффициент преломления, I_a – интенсивность фонового освещения, I_d – интенсивность для диффузного рассеивания, I_s – интенсивность для зеркальности, I_r - интенсивность излучения, приходящего по отраженному лучу, I_t интенсивность излучения, приходящего по преломленному лучу, - цвет поверхности.

Тогда интенсивность по модели Уиттеда можно рассчитать по формуле 1.3:

$$I = K_a I_a C + K_d I_d C + K_s I_s + K_r I_r + K_t I_t \tag{1.3}$$

1.6.4 Вывод

Для первого режима работы лучше подходит модель Ламберта, так как она быстрее и эффективно сочетается с Z-буфером. Для создания реалистического изображения выбрана модель Уиттеда, потому что здесь в первую очередь важно качество полученного изображения, и этот способ показыва-

ет высокую эффективность в комбинации с обратной трассировкой лучей.

1.7 Перспективно-корректное текстурирование объектов трёхмерной сцены

Пусть u, v – координаты текстуры, которые требуется найти для решения задачи наложения текстур на объект трёхмерной сцены. Метод перспективно-корректного текстурирования основан на приближении u и v кусочно-линейными функциями. При отрисовке каждая сканирующая строка разбивается на части, в начале и конце каждого куска вычисляются точные значения u и v, а внутри каждой части применяется линейная интерполяция. [1]

Пусть s_x и s_y – координаты, принадлежащие проекции текстурируемого треугольника. Тогда значения $\frac{1}{z}$, $\frac{u}{z}$ и $\frac{v}{z}$ линейно зависят от s_x и s_y. Следовательно, для каждой вершины достаточно подсчитать значения $\frac{1}{z}$, $\frac{u}{z}$ и $\frac{v}{z}$ и линейно их интерполировать.

Точные значения u и v подсчитываются по формуле 1.4ы:

$$u = \frac{(u/z)}{1/z}, v = \frac{(u/z)}{1/z} \tag{1.4}$$

1.8 Алгоритм заполнения треугольника с использованием барицентрических координат

Барицентрические координаты – это координаты, в которых точка треугольника описывается как линейная комбинация вершин. [1]

В работе используется нормализованный вариант: суммарный вес трёх вершин равен единице:

$$\begin{aligned} p &= b_o v_0 + b_1 v_1 + b_2 v_2 \\ b_0 &+ b_1 + b_2 = 1 \end{aligned} \tag{1.5}$$

Такой выбор обусловлен тем, что барицентрические координаты легко вычислить, так как они равны отношению площадей треугольников, которые образует точка внутри треугольника и вершина, к общей площади треугольника.

Третья координата вычисляется через свойство нормировки, то есть фактически имеется только две степени свободы.

Барицентрические координаты позволяют интерполировать значение любого атрибута в произвольной точке треугольника: значение атрибута в заданной точке треугольника равно линейной комбинации барицентрических координат и значений атрибута в соответствующих вершинах:

$$T = T_0 b_0 + T_1 b_1 + T_2 b_2 \tag{1.6}$$

Алгоритм закраски с использованием барицентрических координат состоит из двух этапов:

1. Поиск прямоугольника, минимального по площади, чтобы он содержал в себе рассматриваемый треугольник.

2. Сканирование прямоугольника слева направо и вычисление барицентрических координат для каждого выбранного пикселя. Если значение координат находится в промежутке от 0 до 1, и сумма по всем координатам не превышает 1, то пиксель закрашивается.

1.9 Вывод

Было приведено описание модели трёхмерного объекта в сцене, рассмотрены формализация объектов сцены и требования к работе программы. Рассмотрены алгоритмы построения трёхмерного изображения, методы закраски, модели освещения, а также способы тексуризации и закраски треугольников.

2 Конструкторский раздел

В этом разделе будет рассмотрена аппроксимация трёхмерных объектов, описаны трёхмерные преобразования, приведены схемы разрабатываемых алгоритмов, описаны входные данные.

2.1 Аппроксимация трёхмерных объектов

Так как пользователь может в любой момент изменить параметры объекта, то нужно предусмотреть, чтобы аппроксимация происходила автоматически. Для цилиндра и конуса нужно выполнить следующие действия:

1. Выбрать требуемое количество разбиений N для тела вращения.

2. Подсчитать из этого угол поворота радиуса вектора для расчёта количества требуемых треугольников.

3. Вычислить координаты следующей вершины на окружности по формуле 2.1:

$$\begin{aligned} X &= X_0 * Rcos(\alpha * i) \\ Y &= Y_0 \\ Z &= Z_0 * Rsin(\alpha * i) \end{aligned} \tag{2.1}$$

4. Итоговая аппроксимация зависит от конкретного тела вращения. Для конуса: полученные точки соединяются с вершиной. Для цилиндра: из полученных точек получается прямоугольник.

Алгоритм аппроксимации сферы сложнее. В программе будет генерироваться икосфера – это многогранная сфера, состоящая из треугольников [5]. Её преимущества в том, что она изотропна – то есть свойства тела по всем направлениям меняться не будут. Кроме того, распределение треугольников будет равномерней, чем в других вариантах разбиения, что не будет приводить к странностям у полюсов сферы. Поэтому алгоритм будет такой:

1. На основе информации о сфере построить правильный икосаэдр, который состоит из 20 граней и 30 рёбер.

2. Итеративно каждый треугольник икосаэдра делить на четыре одинаковых треугольника. Координаты средней точки треугольника корректируются так, чтобы она лежала на сфере. Процесс повторять до тех пор, пока не будет достигнута требуемая величина аппроксимации.

На рисунке 2.1 представлен процесс итеративного приближения икосаэдра к сфере:

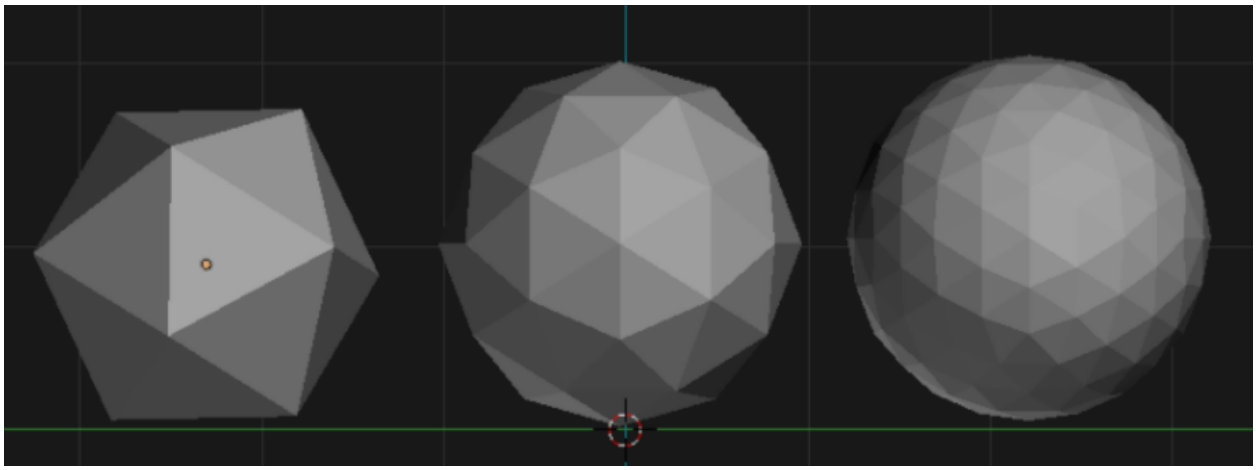

Рисунок. 2.1: Процесс итеративного приближения икосаэдра к сфере

Для того, чтобы икосаэдр был сгенерирован правильно, нужно, чтобы средняя точка треугольника вычислялась с коррекцией. Пусть координаты средней точки треугольника – (X_0, Y_0, Z_0), тогда коррекция будет выполняться по формуле 2.2:

$$\begin{aligned} L &= \sqrt{X_0^2 + Y_0^2 + Z_0^2} \\ X_1 &= \frac{X_0}{L} \\ Y_1 &= \frac{Y_0}{L} \\ Z_1 &= \frac{Z_0}{L} \end{aligned} \tag{2.2}$$

2.2 Описание трёхмерных преобразований

2.2.1 Способ хранения декартовых координат

Для хранения координат точек будет использоваться вектор-столбец, состоящий из четырёх координат: x, y, z, w, причём w по умолчанию равна 1. Это сделано для того, чтобы было удобно умножать вектор на матрицы трансформации, которые обладают размерностью 4x4.

2.2.2 Преобразование трёхмерных координат в двухмерное пространство экрана

Экран обладает только двумя координатами, поэтому нужно выбрать способ, каким образом трёхмерные объекты переносить на двухмерное пространство экрана. Каждый пиксель обладает определённым цветом, и за счёт этого нужно решить проблему передачи объёмности и реалистичности изображения. Алгоритм приведения координат к нужному виду следующий:

1. Перевести объект из собственного пространства в мировое.
2. Перевести объект из мирового пространства в пространство камеры.
3. Найти все проекции точек из пространства камеры в видимые точки, где координаты точек x, y , z находятся в диапазоне $[-w, w]$, а w находится в диапазоне [0, 1].
4. Масштабировать все точки, полученные в п.3, на картинку необходимого разрешения.

Чтобы выполнить все преобразования, нужно использовать матрицы преобразований. Сначала вычисляются все необходимые матрицы, затем они перемножаются в нужном порядке. Исходные координаты умножаются на получившийся результат, в результате чего координаты приводятся к нужной системе.

2.2.3 Преобразования трёхмерной сцены в пространство камеры

Чтобы привести трёхмерную сцену к пространству камеру, нужно умножить каждую вершину всех полигональных моделей на матрицу камеры. Сама же камера задаётся аргументами: положение камеры в мировом пространстве, вектор наблюдения взгляда, направление верха камеры. Пусть: α – координаты точки в пространстве, на которую смотрит камера, β – вектор, который указывает, куда смотрит верх камеры, ψ– ортогональный вектор к векторам направления взгляда и вектору направления.

Тогда матрица будет выглядеть так:

$$A = \begin{pmatrix} \alpha_x & \beta_x & \psi_x & 0 \\ \alpha_y & \beta_y & \psi_y & 0 \\ \alpha_z & \beta_z & \psi_z & 0 \\ -(P * \alpha) & -(P * \beta) & -(P * \psi) & 1 \end{pmatrix}$$

2.2.4 Матрица перспективной проекции

После перехода в пространство камеры нужно умножить каждую вершину всех полигональных моделей на матрицу проекции. Эта матрица преображает заданный диапазон усечённой пирамиды в пространство отсечение, и изменяет w-компоненту так, что чем дальше от наблюдателя находится вершина, тем больше возрастает w. После преобразования координат в пространство отсечения, координаты x и y попадают в промежуток $[-w, w]$, а вершина z – [-0, w]. Всё, что находится вне диапазона, отсекается.

Пусть AR – отношение ширины изображения к его высоте, α – угол обзора камеры, Z_n – координата z ближайшей к камере плоскости отсечения пирамиды видимости, Z_f – координата z дальней от камеры плоскости отсечения пирамиды видимости. Тогда матрица перспективной проекции

принимает вид:

$$A = \begin{pmatrix} \frac{cot(\frac{\alpha}{2})}{AR} & 0 & 0 & 0 \\ 0 & cot(\frac{\alpha}{2}) & 0 & 0 \\ 0 & 0 & \frac{Z_f \times Z_n}{Z_f - Z_n} & 1 \\ 0) & 0 & \frac{Z_f}{Z_f - Z_n} & 0 \end{pmatrix}$$

Следующий этап – спроецировать все координаты на одну плоскость, разделив всё на координату z. После умножения вектора координат на матрицу перспективной проекции, реальная координата z заносится в w-компоненту, так что вместо деления на z делят на w.

2.2.5 Преобразования трёхмерной сцены в пространство области изображения

Чтобы преобразовать спроецированные координаты в координаты области изображения, нужно умножить вектор координат на матрицу. Пусть W – ширина изображения, H – высота изображения, hW – половина ширины изображения, hH – половина высоты изображения, тогда матрица преобразования сцены в пространство изображения принимает вид:

$$A = \begin{pmatrix} hW & 0 & 0 & hW \\ 0 & hH & 0 & hH \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.3 Алгоритм Z-буфера

Алгоритм Z-буфера используется в первом режиме работы программы, в котором пользователь производит работу с объектами. Совместно с алгоритмом Z-буфера применяется для закраски метод Гуро, а в качестве модели освещения используется модель Ламберта.

На рисунке 2.2 изображена схема алгоритма Z-буфера с применением метода Гуро для закраски:

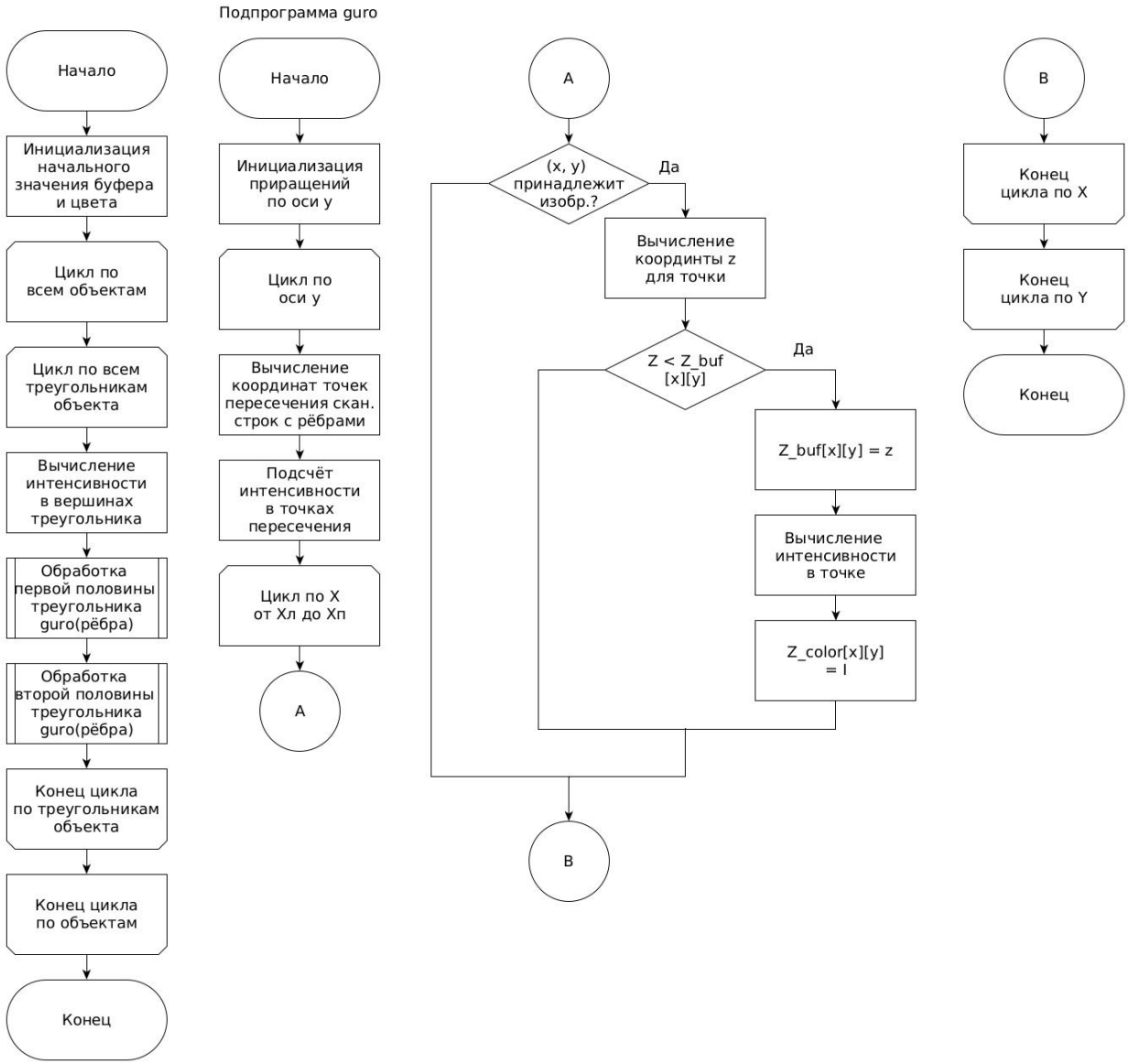

Рисунок. 2.2: Схема реализации алгоритма Z-буфера совместно с методом Гуро

2.4 Алгоритм обратной трассировки лучей

Алгоритм обратной трассировки лучей используется во втором режиме работы программы для построения реалистического изображения. Совместно с ним используется модель освещения Уиттеда.

На рисунке 2.3 приведена схема алгоритма обратной трассировки лучей.

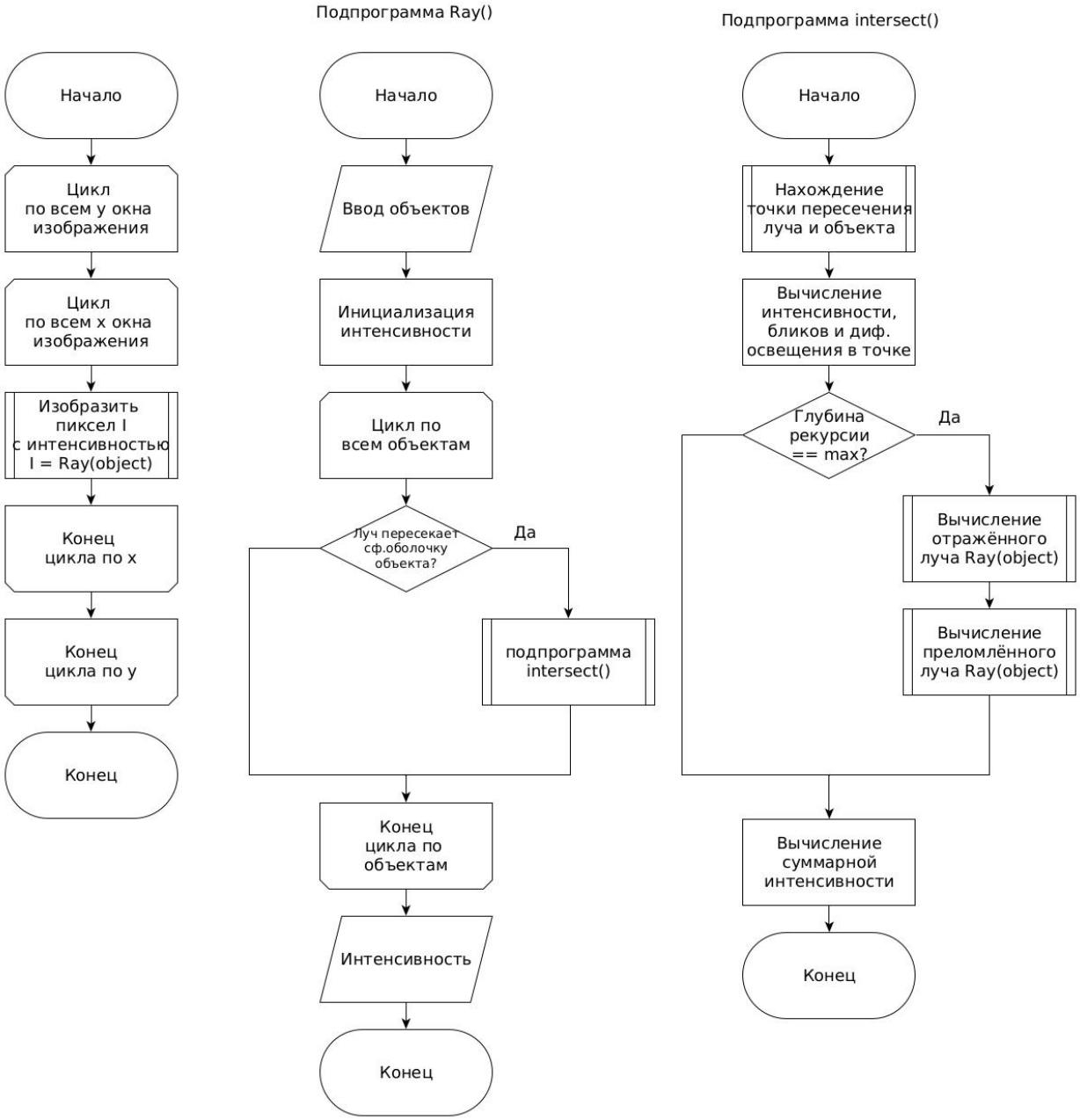

Рисунок. 2.3: Схема реализации алгоритма обратной трассировки лучей

Для алгоритма обратной трассировки лучей нужно уметь находить отражённый и преломлённый лучи, при этом учитывая модель освещения Уиттеда. Отражённый луч можно найти, зная направление падающего луча и нормаль к поверхности.

Пусть $\overline{L}$ – направление луча, а $\overline{n}$ – нормаль к поверхности. Луч можно разбить на две части: , $\overline{L_p}$ которая перпендикулярна нормали, и $\overline{L_n}$ – параллельна нормали.

Представленная ситуация изображена на рисунке 2.4:

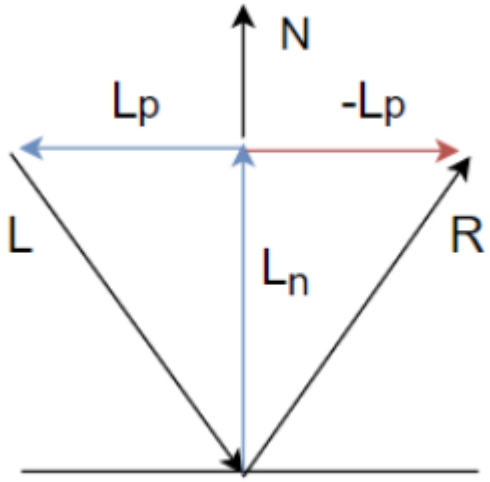

Рисунок. 2.4: Рассматриваемые векторы для расчёта отражённого луча.

Учитывая свойства скалярного произведения $\overline{L_n} = \overline{n} * (\overline{n}, \overline{L})$ и $\overline{L_p} = \overline{L} - \overline{n} * (\overline{n}, \overline{L})$ Так как отражённый луч выражается через разность этих векторов, то отражённый луч выражается по формуле 2.3:

$$R = 2 * \overline{n} * (\overline{n}, \overline{L}) - \overline{L} \tag{2.3}$$

По закону преломления падающий, преломлённый луч и нормаль к поверхности лежат в одной плоскости. Пусть μ_i – показатели преломления сред, а η_i – углы падения и отражения света соответственно. Применяя закон Снеллиуса, параметры преломлённого луча можно вычислить по формуле 2.4:

$$\begin{aligned} R &= \frac{\mu_1}{\mu_2}\overline{L} + (\frac{\mu_1}{\mu_2}cos(\eta_1) - cos(\eta_2))\overline{n} \\ cos(\eta_2) &= \sqrt{1 - (\frac{\mu_1}{\mu_2})^2 * (1 - cos(\eta_1))^2} \end{aligned} \tag{2.4}$$

2.5 Алгоритм пересечения луча с параллелепипедом

При работе алгоритма обратной трассировки лучей крайне неэффективно при каждой трассировке луча искать пересечения со всеми полигонами каждого объекта в сцене, поэтому имеет смысл заключить объект в параллелепипед, который бы полностью его включал.

Данный параллелепипед задаётся координатами двух вершин: с минимальными и максимальными значениями координат x, y, z. Таким образом это позволяет задать шесть плоскостей, ограничивающих параллелепипед, и при этом все они будут параллельны координатным плоскостям.

Рассмотрим пару плоскостей, параллельных плоскости yz: X = x1 и X = x2. Пусть $\overline{D}$ – вектор направления луча. Если координата x вектора D равна 0, то заданный луч параллелен этим плоскостям, и, если $x_0 < x_1$ или $x_0 > x_1$, то он не пересекает рассматриваемый прямоугольный параллелепипед. Если же Dx не равно 0, то вычисляются отношения 2.5:

$$\begin{aligned} t_{1x} &= \frac{x_1 - x_0}{D_x} \\ t_{2x} &= \frac{x_2 - x_0}{D_x} \end{aligned} \tag{2.5}$$

Можно считать, что найденные величины связаны неравенством $t_{1x} < t_{2x}$. Пусть $t_n = t_{1x}$, $t_f = t_{2x}$. Считая, что D_y не равно нулю, и рассматривая вторую пару плоскостей, несущих грани заданного параллелепипеда, $Y = y_1$, $Y = y_2$, вычисляются величины:

$$\begin{aligned} t_{1y} &= \frac{y_1 - y_0}{D_y} \\ t_{2y} &= \frac{y_2 - y_0}{D_y} \end{aligned} \tag{2.6}$$

Если $t_{1y} > t_n$, то тогда $t_n = t_{1y}$. Если $t_{2y} < t_f$, то тогда $t_f = t_{2y}$. При $t_n > t_f$ или при $t_f < 0$ заданный луч проходит мимо прямоугольного параллелепипеда.

Считая, что D_z не равно нулю, и рассматривая вторую пару плоскостей, несущих грани заданного параллелепипеда, $Z = z_1$, $Z = z_2$, вычисляются

величины:

$$t_{1z} = \frac{z_1 - z_0}{D_z}$$
$$t_{2z} = \frac{z_2 - z_0}{D_z} \tag{2.7}$$

и повторяются сравнения, приведённые для формулы 2.6.

Если в итоге всех проведённых операций получается, что $0 < t_n < t_f$ или $0 < t_f$, то заданный луч пересечёт исходный параллелепипед со сторонами, параллельными координатным осям.

Следует отметить, что при пересечении лучом параллелепипеда извне знаки t_n и t_f должны быть равны, в противном случае можно сделать вывод, что луч пересекает параллелепипед изнутри.

2.6 Описание входных данных

В разработанной программе входные данные подаются в виде файла с расширением .obj. В этом формате помимо координат вершин можно передавать информацию о текстурах и нормалях.

Формат файла:

1. Список вершин с координатами (x, y, z):

   v 0.56 0.19 -1.32

2. Текстурные координаты (u, v):

   vt 0.430 0.2

3. Координаты нормалей (x, y, z):

   vn 0.5 -1 -0.2

4. Определение поверхности задаётся в формате i1/i2/i3, где i1 – индекс координаты вершины, i2 – индекс координаты текстуры, i3 – индекс координаты нормали:

   f 1/3/5 2/4/6 3/9/7

2.7 Вывод

В этом разделе была рассмотрена аппроксимация трёхмерных объектов, описаны трёхмерные преобразования, приведены схемы разрабатываемых алгоритмов, описаны входные данные.

3 Технологический раздел

В этом разделе будет приведено описание структура программы, выбраны средства реализации ПО, приведены листинги кода, и продемонстрирован интерфейс программы.

3.1 Средства реализации программного обеспечения

В качестве языка программирования для решения поставленных задач был выбран язык программирования C++, поскольку:

- имеется опыт разработки на этом языке;
- C++ обладает достаточной производительностью для быстрого исполнения трассировки лучей;

В качестве IDE была выбрана среда разработки QT Creator, так как:

- имеется опыт разработки с взаимодействием с данной IDE;
- есть возможность создать графический интерфейс;

3.2 Описание структуры программы

Структура программы основана на парадигмах ООП. В программе реализованы следующие классы:

- class Manager – хранит сцену, описывает методы взаимодействия со сценой и объектами на ней;
- class Model – описывает представление трёхмерного объекта в программе и методы работы с ним;
- class Light – описывает источники света и методы взаимодействия с ним;

- class Camera – описывает камеру и методы взаимодействия с ней;
- class objLoader – описывает работу с файлами расширения .obj;
- class Face – описывает полигоны для представления трёхмерного объекта;
- class Vertex – описывает вершину объекта;
- class Vec3, Vec4 – реализация векторов размерности 3 и 4.
- class Mat – описывает матрицы и методы взаимодействия с ними.
- class BoundingBox – описывает ограничивающий параллелепипед и методы работы с ним;
- class RayThread – описывает работу отдельного потока при трассировки лучей;
- class PixelShader – содержит функции для вычисления атрибутов объекта в конкретном пикселе
- class VertexShader – содержит функции для преобразования атрибутов модели при переходе к мировому пространству из объектного;
- class TextureShader – содержит функции для интерполяции значения текстурных координат в конкретном пикселе;
- class GeometryShader – содержит функции для преобразования атрибутов модели при переходе из мирового пространства в пространство нормализированных координат;

На рисунке 3.1 представлена диаграмма классов в виде UML-диаграммы:

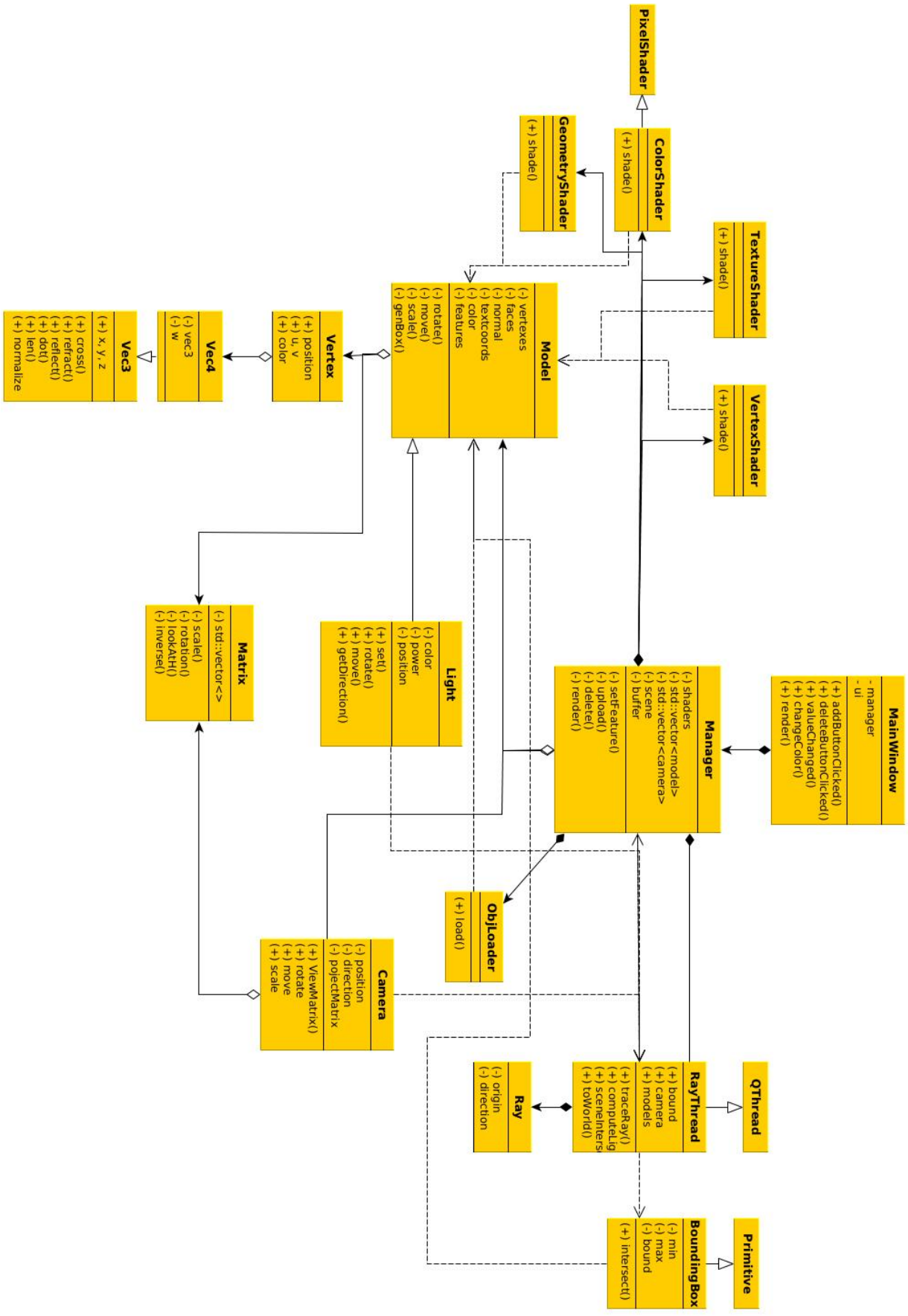

Рисунок. 3.1: Диаграмма классов

3.3 Листинг кода

На листинге 3.1 представлен код отрисовки модели в первом режиме работы программы:

Листинг 3.1: Код отрисовки модели в первом режиме работы программы

```
void SceneManager::rasterize(Model& model)
{
    auto camera = camers[curr_camera];
    auto projectMatrix = camera.projectionMatrix;
    auto viewMatrix = camera.viewMatrix();

    auto rotation_matrix = model.rotation_matrix;
    auto objToWorld = model.objToWorld();

    for (auto& face: model.faces)
    {

        auto a = vertex_shader->shade(face.a, rotation_matrix, objToWorld,
            camera);
        auto b = vertex_shader->shade(face.b, rotation_matrix, objToWorld,
            camera);
        auto c = vertex_shader->shade(face.c, rotation_matrix, objToWorld,
            camera);

        if (backfaceCulling(a, b, c))
            continue;

        a = geom_shader->shade(a, projectMatrix, viewMatrix);
        b = geom_shader->shade(b, projectMatrix, viewMatrix);
        c = geom_shader->shade(c, projectMatrix, viewMatrix);


        rasterBarTriangle(a, b, c);
    }
}
```

На листинге 3.2 представлен код закраски треугольника в первом режиме работы программы:

Листинг 3.2: Код закраски треугольника в первом режиме работы программы

```
void SceneManager::rasterBarTriangle(Vertex p1_, Vertex p2_, Vertex p3_)
{
    if (!clip(p1_) && !clip(p2_) && !clip(p3_))
```

```
    {
        return;
    }

    denormolize(width, height, p1_);
    denormolize(width, height, p2_);
    denormolize(width, height, p3_);

    auto p1 = p1_.pos;
    auto p2 = p2_.pos;
    auto p3 = p3_.pos;

    float sx = std::floor(Min(Min(p1.x, p2.x), p3.x));
    float ex = std::ceil(Max(Max(p1.x, p2.x), p3.x));

    float sy = std::floor(Min(Min(p1.y, p2.y), p3.y));
    float ey = std::ceil(Max(Max(p1.y, p2.y), p3.y));

    for (int y = static_cast<int>(sy); y < static_cast<int>(ey); y++)
    {
        for (int x = static_cast<int>(sx); x < static_cast<int>(ex); x++)
        {
            Vec3f bary = toBarycentric(p1, p2, p3, Vec3f(static_cast<float
               >(x), static_cast<float>(y)));
            if ( (bary.x > 0.0f || fabs(bary.x) < eps) && (bary.x < 1.0f
               || fabs(bary.x - 1.0f) < eps) &&
                 (bary.y > 0.0f || fabs(bary.y) < eps) && (bary.y < 1.0f
                    || fabs(bary.y - 1.0f) < eps) &&
                 (bary.z > 0.0f || fabs(bary.z) < eps) && (bary.z < 1.0f
                    || fabs(bary.z - 1.0f) < eps))
            {
                auto interpolated = baryCentricInterpolation(p1, p2, p3,
                   bary);
                interpolated.x = x;
                interpolated.y = y;
                if (testAndSet(interpolated))
                {
                    // Рисуем
                    auto pixel_color = pixel_shader->shade(p1_, p2_, p3_,
                       bary) * 255.f;
                    img.setPixelColor(x, y, qRgb(pixel_color.x,
                       pixel_color.y, pixel_color.z));
                }
            }
        }
    }

}
```

На листинге 3.3 представлен код алгоритма трассировки одного луча:

Листинг 3.3: Код алгоритма трассировки

```
Vec3f RayThread::cast_ray(const Ray &ray, int depth)
{

    InterSectionData data;
    if (depth > 4 || !sceneIntersect(ray, data))
        return Vec3f{0.f, 0, 0};

    float di = 1 - data.model.specular;

    float distance = 0.f;

    float occlusion = 1e-4f;

    Vec3f ambient, diffuse = {0.f, 0.f, 0.f}, spec = {0.f, 0.f, 0.f},
       lightDir = {0.f, 0.f, 0.f},
            reflect_color = {0.f, 0.f, 0.f}, refract_color = {0.f, 0.f, 0.
               f};

    if (fabs(data.model.refractive) > 1e-5)
    {
        Vec3f refract_dir = refract(ray.direction, data.normal, power_ref,
            data.model.refractive).normalize();
        Vec3f refract_orig = Vec3f::dot(refract_dir, data.normal) < 0 ?
           data.point - data.normal * 1e-3f : data.point + data.normal * 1
           e3f;
        refract_color = cast_ray(Ray(refract_orig, refract_dir), depth +
           1);
    }

    if (fabs(data.model.reflective) > 1e-5)
    {
        Vec3f reflect_dir = reflect(ray.direction, data.normal).normalize
           ();
        Vec3f reflect_orig = Vec3f::dot(reflect_dir, data.normal) < 0 ?
           data.point - data.normal * 1e-3f : data.point + data.normal * 1
           e-3f;
        reflect_color = cast_ray(Ray(reflect_orig, reflect_dir), depth +
           1);
    }

    for (auto &model: models)
    {
        if (model->isObject()) continue;
        Light* light = dynamic_cast<Light*>(model);
        if (light->t == Light::light_type::ambient)
```

```
            ambient = light->color_intensity;
        else
        {
            if (light->t == Light::light_type::point)
            {
                lightDir = (light->position - data.point);
                distance = lightDir.len();
                lightDir = lightDir.normalize();
            } else{
                lightDir = light->getDirection();
                distance = std::numeric_limits<float>::infinity();
            }

            auto tDot = Vec3f::dot(lightDir, data.normal);

            Vec3f shadow_orig = tDot < 0 ? data.point - data.normal*
               occlusion : data.point + data.normal*occlusion; // checking
                if the point lies in the shadow of the lights[i]
            InterSectionData tmpData;
            if (sceneIntersect(Ray(shadow_orig, lightDir), tmpData))
                if ((tmpData.point - shadow_orig).len() < distance)
                    continue;

            diffuse += (light->color_intensity * std::max(0.f, Vec3f::dot(
               data.normal, lightDir)) * di);
            if (fabs(data.model.specular) < 1e-5) continue;
            auto r = reflect(lightDir, data.normal);
            auto r_dot = Vec3f::dot(r, ray.direction);
            auto power = powf(std::max(0.f, r_dot), data.model.n);
            spec += light->color_intensity * power * data.model.specular;
        }

    }

    return data.color.hadamard(ambient +
                               diffuse +
                               spec +
                               reflect_color * data.model.reflective +
                               refract_color * data.model.refractive).
                                  saturate();
}
```

На листинге 3.4 представлен код функции поиска пересечения луча с ограничивающим параллелепипедом:

Листинг 3.4: Код отрисовки модели в первом режиме работы программы

```
bool BoundingBox::intersect(const Ray &r) const
{
```

```
    float txmin, txmax, tymin, tymax, tzmin, tzmax;

    txmin = (bounds[r.sign[0]].x - r.origin.x) * r.invdirection.x;
    txmax = (bounds[1-r.sign[0]].x - r.origin.x) * r.invdirection.x;

    tymin = (bounds[r.sign[1]].y - r.origin.y) * r.invdirection.y;
    tymax = (bounds[1-r.sign[1]].y - r.origin.y) * r.invdirection.y;

    tzmin = (bounds[r.sign[2]].z - r.origin.z) * r.invdirection.z;
    tzmax = (bounds[1-r.sign[2]].z - r.origin.z) * r.invdirection.z;

    if ((txmin > tymax) || (tymin > txmax))
        return false;
    if (tymin > txmin)
        txmin = tymin;
    if (tymax < txmax)
        txmax = tymax;

    if ((txmin > tzmax) || (tzmin > txmax))
        return false;
    if (tzmin > txmin)
        txmin = tzmin;
    if (tzmax < txmax)
        txmax = tzmax;

    return (SIGN(txmin) == SIGN(txmax));
}
```

3.4 Описание интерфейса

На рисунке 3.2 представлен стартовый экран программы. Он предоставляет пользователю возможность добавить объект, изменить параметры окружающего света и добавления точечного источника.

Рисунок. 3.2: Стартовый экран программы

Для добавления объекта модели её нужно выбрать в выпадающем меню в левом верхнем углу, а затем нажать на кнопку добавления. Модель появится на графике, а также в списке в верхнем правом текстовом поле.

На рисунке 3.3 представлен результат добавления на сцену цилиндра, а также красным подчёркнуты места интерфейса, которые были задействованы:

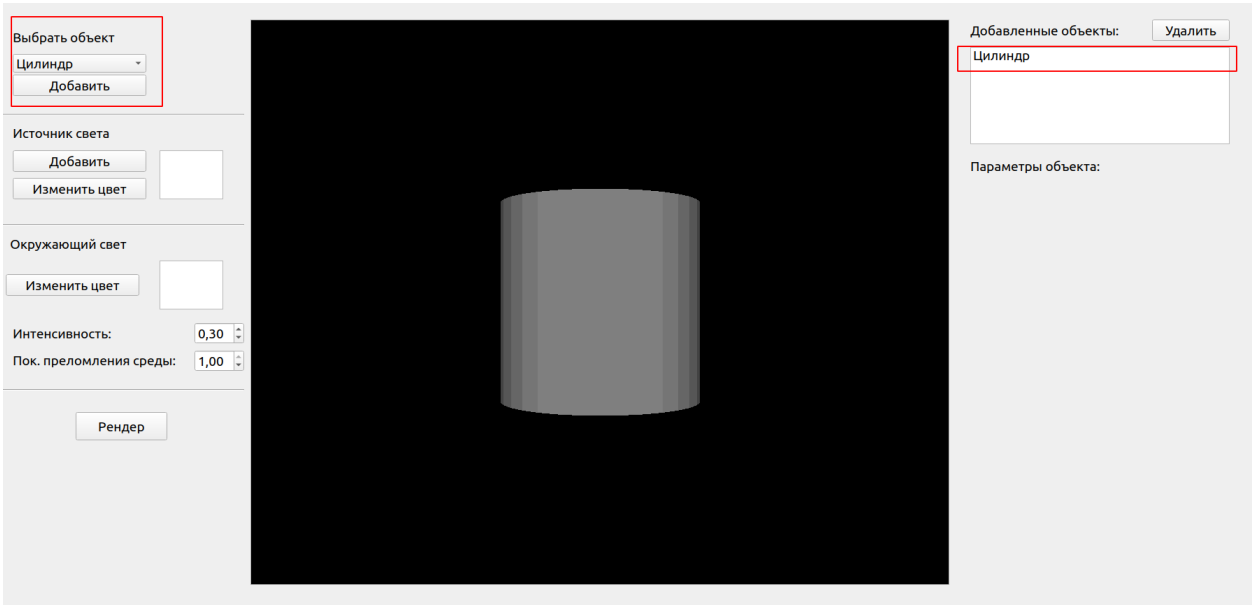

Рисунок. 3.3: Экран после добавления на сцену цилиндра

При нажатии на название объекта в верхнем правом текстовом поле

на экране появляется интерфейс, позволяющий изменить параметры модели: пространственное положение, размеры, цвет поверхности или текстуру, физические показатели.

На рисунке 3.4 изображен результат изменения характеристик цилиндра, а также красным подчёркнуты места интерфейса, которые были задействованы:

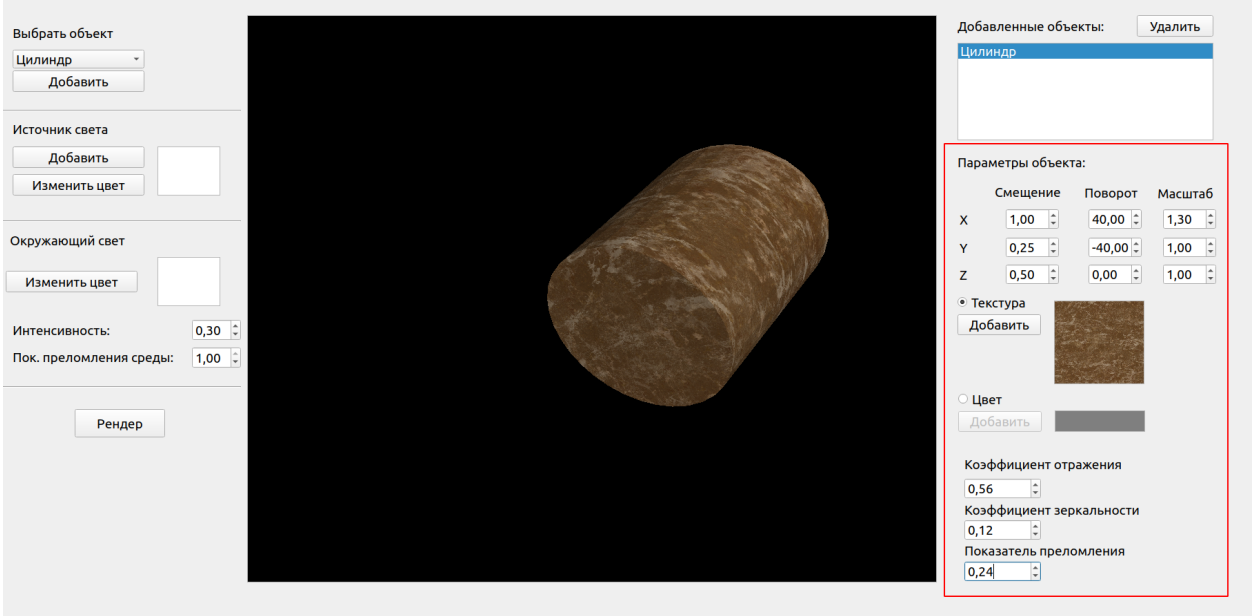

Рисунок. 3.4: Результат изменения характеристик цилиндра

Также слева в поле источника света пользователь может добавить источник света. После добавления он появится на графике в виде серой точки и в верхнем правом текстовом поле.

На рисунке 3.5 изображен результат добавления источника света на сцену:

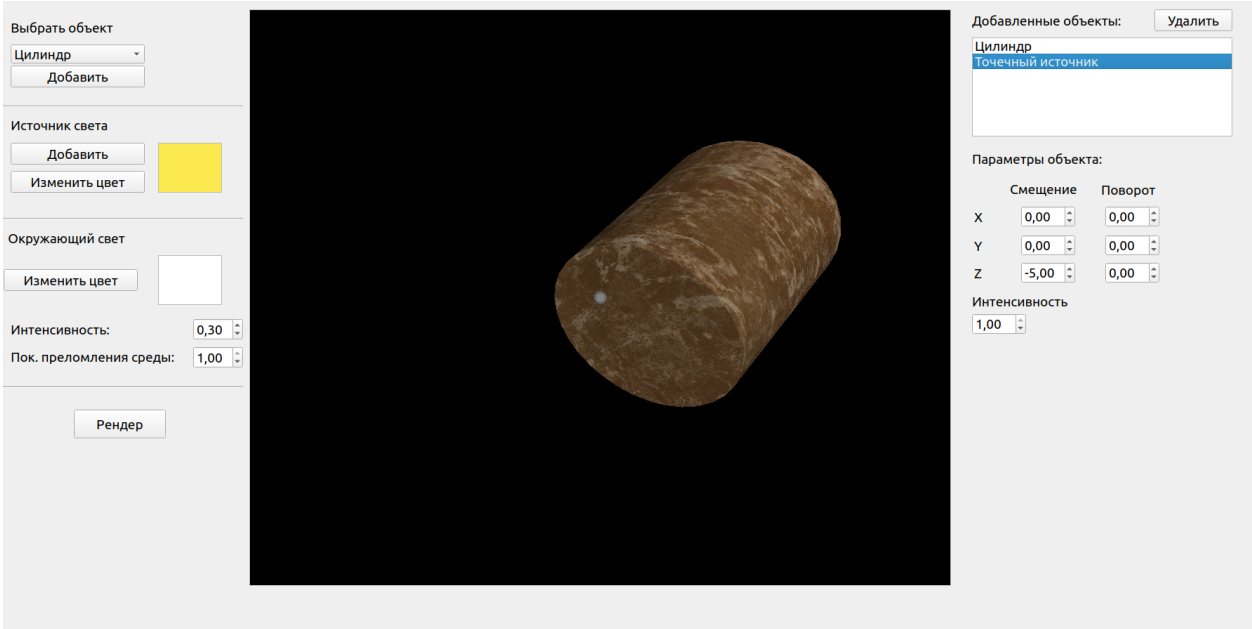

Рисунок. 3.5: Результат добавления источника света на сцену

В поле 'Окружающая среда' пользователь может изменить цвет фонового освещения, его интенсивность, а также изменить показатель преломления среды. На рисунке 3.6 изображены части интерфейса, которые для этого нужно задействовать.

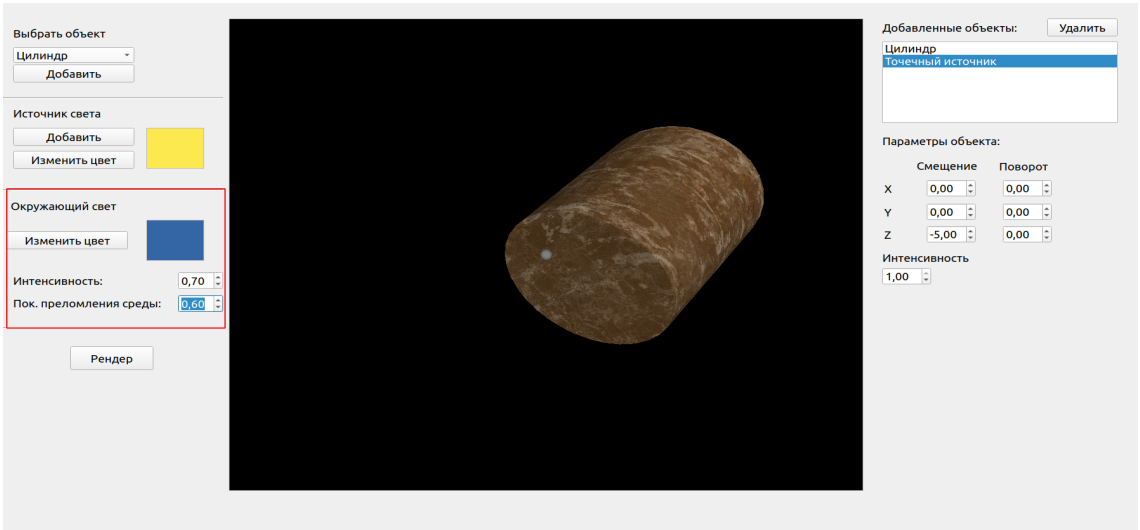

Рисунок. 3.6: Части интерфейса для изменения параметров окружающей среды

При нажатии на кнопку 'Рендер' программа запускает алгоритм обратной трассировки лучей. Все кнопки блокируются, пока алгоритм не завершит свою работу. По его окончанию на графике будет отображено реали-

стическое изображение. На рисунке 3.7 показано состояние программы в момент рендеринга.

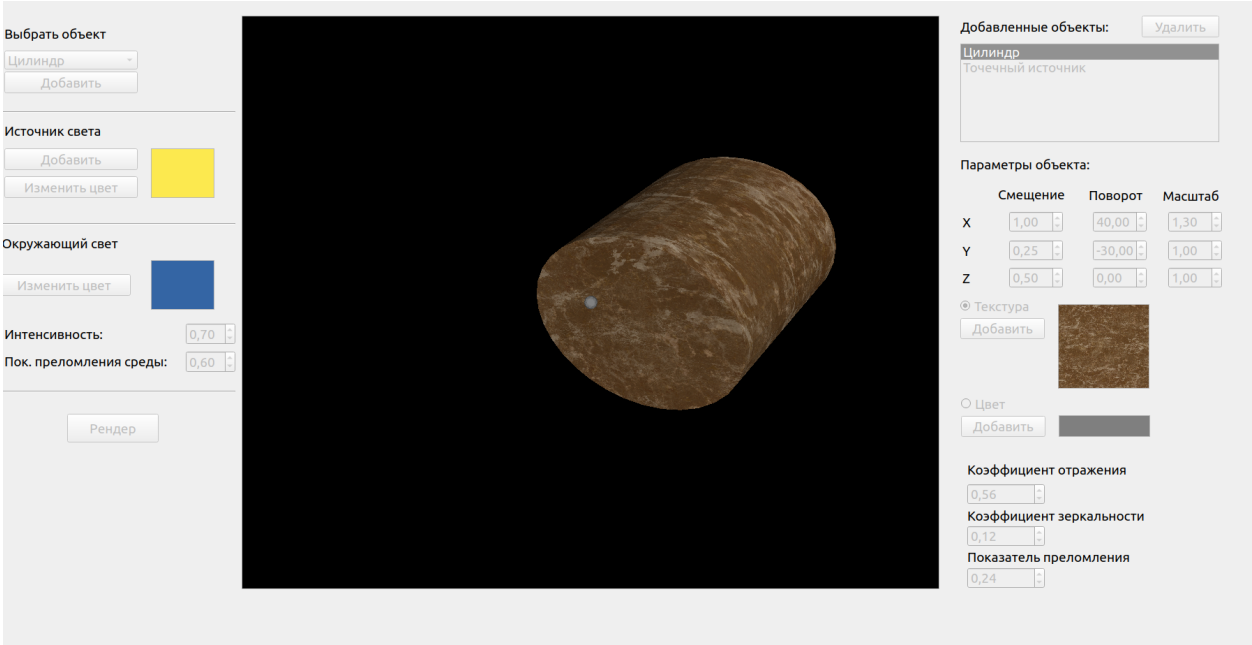

Рисунок. 3.7: Процесс рендеринга

На рисунке 3.8 показан конечный результат работы алгоритма обратной трассировки лучей.

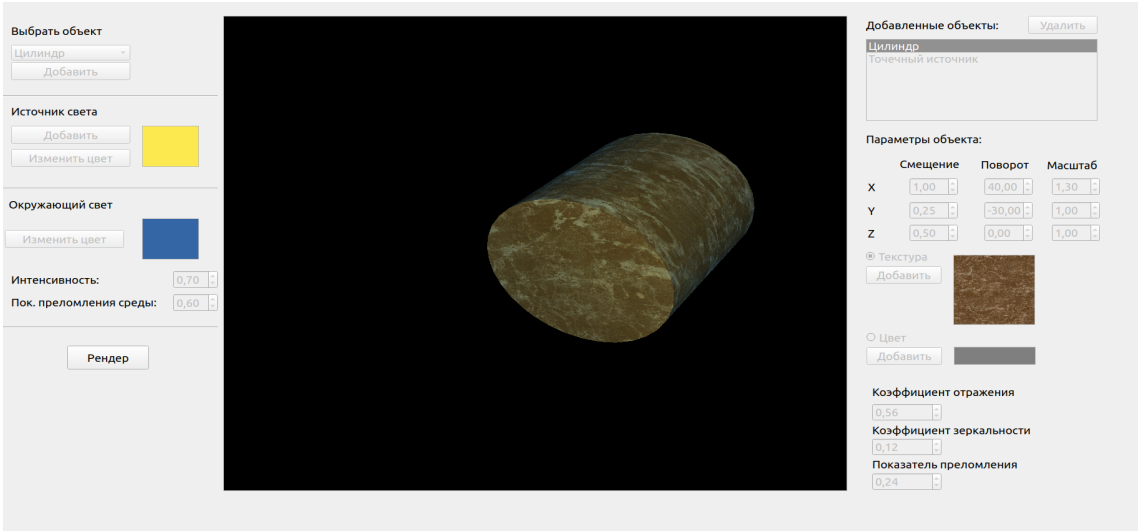

Рисунок. 3.8: Результат работы программы

Для того, чтобы вернуться к редактированию изображению, пользователь может нажать на кнопку ’Рендер’. Управление камерой осуществляется при помощи клавиатуры кнопками W, E, S, D, X, Home, End, PageUp, PageDown.

3.5 Вывод

Было приведено описание структура программы, выбраны средства реализации ПО, приведены листинги кода, и продемонстрирован интерфейс программы.

4 Исследовательская часть

В этом разделе будут проведены исследования полученных результатов.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование, следующие:

- операционная система: Ubuntu 20.04.1 LTS; [6]
- память: 8 GB;
- процессор: Intel Core i5-1135G7 @ 2.40GHz [7].
- количество ядер процессора: 8

Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.2 Метод тестирования

Для проведения замеров времени в экспериментах будет использована формула 4.1:

$$t = \frac{T_n}{N} \tag{4.1}$$

где t – время выполнения реализации алгоритма, N - количество экспериментов, T_n - время выполнения замеров.

Неоднократное измерение времени необходимо для построения более гладкого графика и получения усреднённого значения. Для замеры общего времени используется библиотека chrono.

4.3 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы программы:

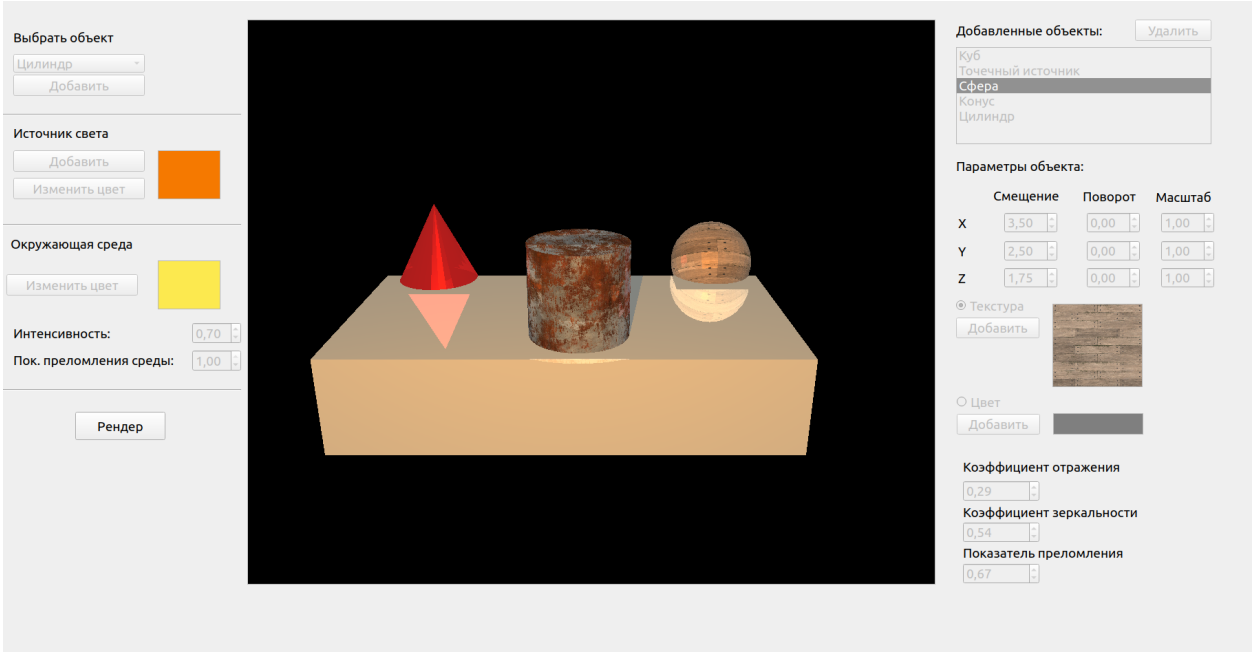

Рисунок. 4.1: Демонстрация работы программы

4.4 Зависимость времени работы алгоритма обратной трассировки лучей от количества объектов в сцене

Время работы алгоритма обратной трассировки лучей имеет зависимость от количества объектов в сцене. Для измерения времени работы алгоритма на сцены добавлялось N сфер, для которых и подсчитывалось время работы. Также на сцене находится один точечный источник.

Тестирование производилось на следующих параметрах:

1. Количество потоков – 8.
2. Глубина рекурсии – 2.
3. Интенсивность окружающего света – 0.5.
4. Коэффициент преломления внешней среды – 1.
5. Коэффициент отражения объекта – 0.2.
6. Коэффициент преломления объекта – 0.5.

Тестирование также проводилось для двух случаев:

1. Сфера закрашивается цветом.

2. Сфера имеет текстуру.

В таблице 4.1 приведены результаты тестирования работы алгоритмов в случае закраски цветом:

Таблица 4.1: Результаты тестов

Количество объектов	Время (в мс)
1	12160274
2	25475533
3	42793308
4	62433652

В таблице 4.2 приведены результаты тестирования работы алгоритмов в случае текстуризации:

Таблица 4.2: Результаты тестов

Количество объектов	Время (в мс)
1	13439190
2	22993890
3	49343762
4	89433652

На рисунке 4.2 представлены графики зависимости времени работы алгоритма от количества объектов в сцене

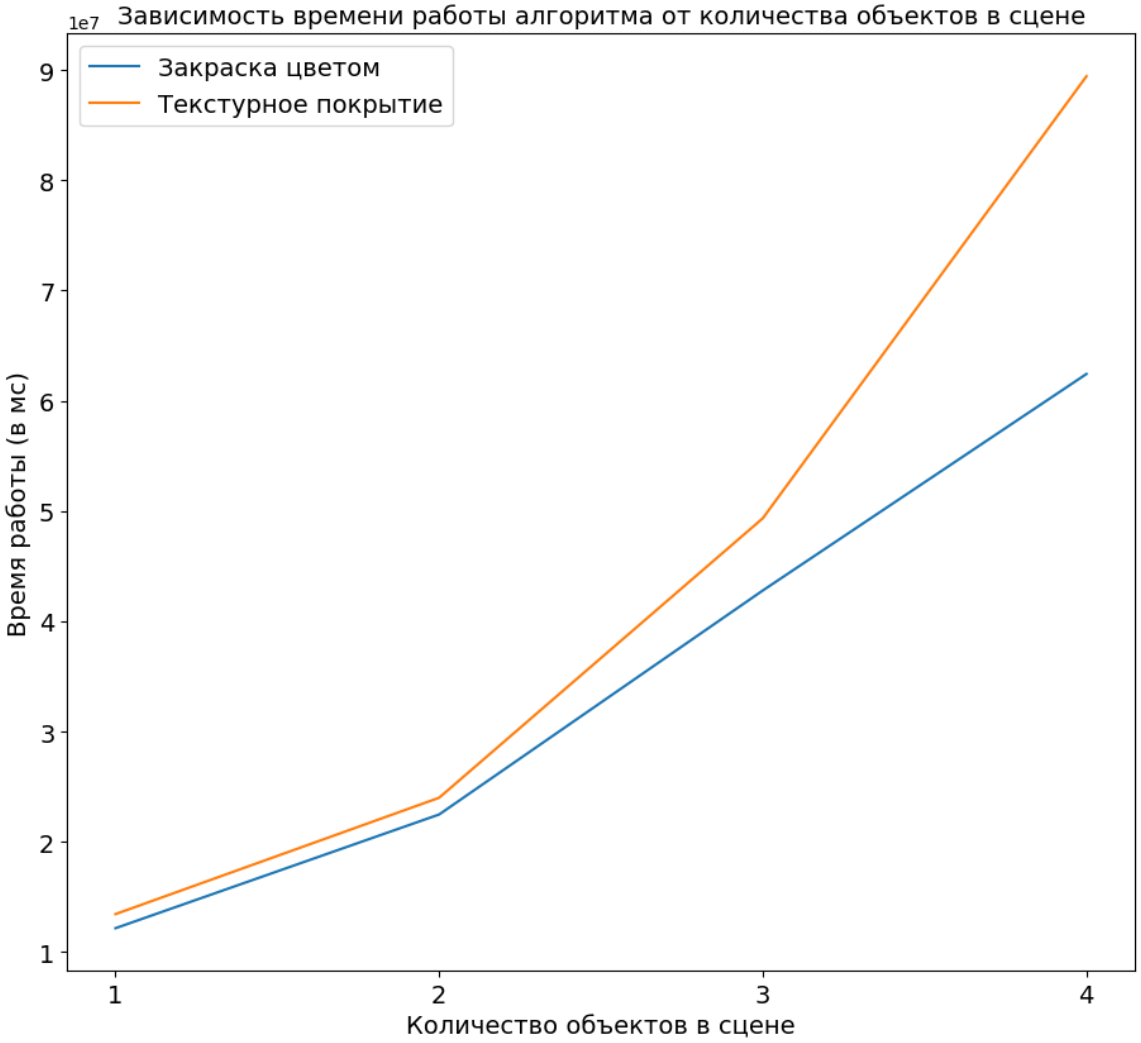

Рисунок. 4.2: График зависимости времени работы алгоритма от количества объектов в сцене

4.5 Зависимость времени работы алгоритма обратной трассировки лучей от количества потоков программы

Особенностью алгоритма обратной трассировки лучей является то, что каждый луч работает независимо друг от друга. Следовательно, вычисления могут производиться параллельно. В программе это реализовано следующим образом: картинка делилась на n частей по диагонали, а затем каждый из n потоков работал на своей части.

В качестве сцены для тестирования использовалась следующая композиция объектов: сфера, цилиндр, конус, куб и точечный источник света. У всех объектов одинаковые параметры:

В таблице 4.3 приведены результаты тестирования работы алгоритма от количества потоков:

Таблица 4.3: Результаты тестов

Количество объектов	Время (в мс)
1	38771415
2	27341234
4	18204997
8	16129667
16	20010097

На рисунке 4.3 представлен график зависимости времени работы алгоритма от количества потоков:

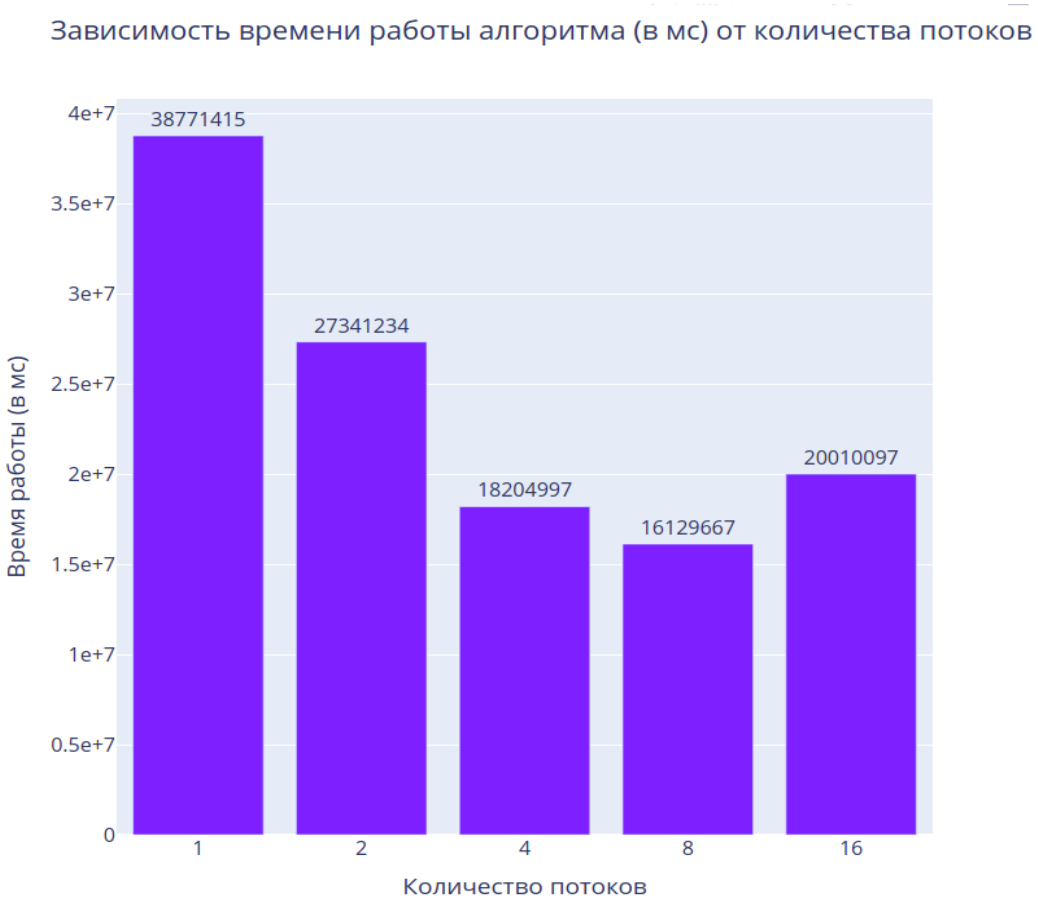

Рисунок. 4.3: График зависимости времени работы алгоритма от количества потоков

4.6 Зависимость времени работы алгоритма обратной трассировки лучей от глубины рекурсии

В алгоритме обратной трассировки лучей для каждого луча проверяется, достигнута ли глубина рекурсии. В случае, если не достигнута, рассчитывается направление и интенсивность отражённого и преломлённого лучей.

Чем меньше глубина рекурсии, тем меньше время работы программы, но полученное изображение менее реалистическое. В качестве сцены для тестирования использовалась следующая композиция объектов: сфера, цилиндр, конус, куб и точечный источник света.

В таблице 4.4 приведены результаты тестирования работы алгоритма от глубины рекурсии:

Таблица 4.4: Результаты тестов

Количество объектов	Время (в мс)
1	14030977
2	16341234
3	18204997
4	19129667
5	22010097

На рисунке 4.4 представлен график зависимости времени работы алгоритма от глубины рекурсии:

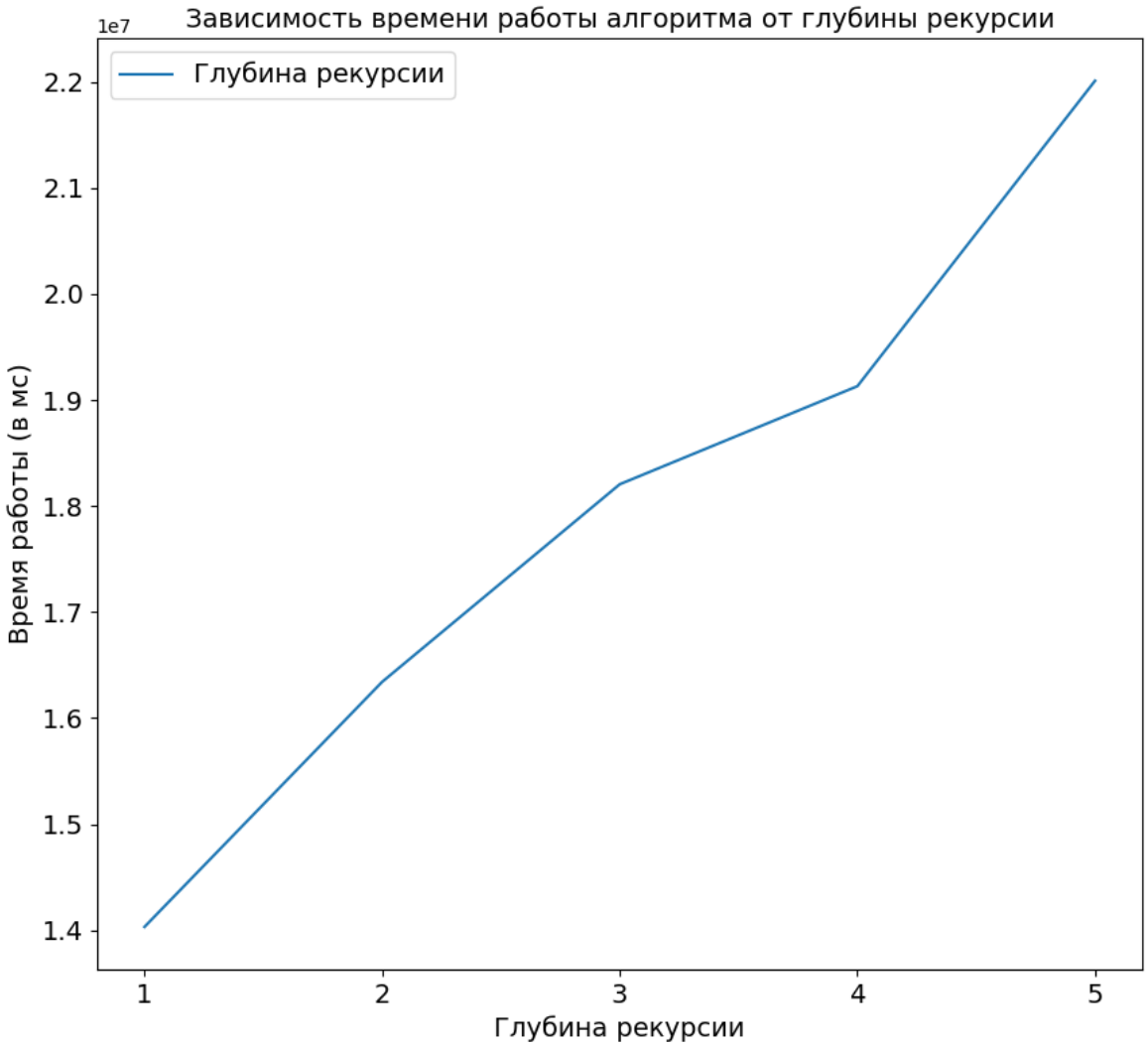

Рисунок. 4.4: График зависимости времени работы алгоритма от глубины рекурсии

4.7 Выводы

Время работы алгоритма обратной трассировки лучей зависит от его параметров и характеристик фигур, которые присутствуют в сцене. Алгоритм работает при 4 фигурах на 30% быстрее, если фигуры закрашиваются цветом, а не покрываются текстурами. Быстрее всего обратная трассировка лучей работает при $N = 8$ потоках, что соответствует количеству ядер в системе. Также выявлена прямая зависимость времени работы алгоритма от глубины рекурсии.

Заключение

В рамках курсового проекта было создано ПО для создания графических сцен из трёхмерных геометрических объектов и их визуализации с учетом выбранной текстуры или цвета, а также оптических эффектов отражения и преломления.

Были выполнены следующие задачи:

- проведён анализ существующих алгоритмов и выбран оптимальный путь для решения основной задачи.
- выбран подходящий язык программирования и среда разработки для выполнения работы.
- создан программный продукт для решения задачи, реализованы выбранные алгоритмы.
- реализован понятный интерфейс для клиента.
- проведено исследования на основе полученных результатов.

Литература

[1] Роджерс Д. Адамс Дж. Математические основы машинной графики. М.: Мир, 1989.

[2] Обратная трассировка лучей. Режим доступа: http://www.ray-tracing.ru/articles164.html (дата обращения: 22.12.2021).

[3] Алгоритмические основы современной компьютерной графики. Режим доступа: https://intuit.ru/studies/courses/70/70/info (дата обращения: 27.12.2021).

[4] Простые модели освещения. Режим доступа: http://grafika.me/node/344 (дата обращения: 27.12.2021).

[5] Creating an icosphere mesh in code. Режим доступа: http://blog.andreaskahler.com/2009/06/creating-icosphere-mesh-in-code.html (дата обращения: 27.12.2021).

[6] Ubuntu 20.04.3 LTS (Focal Fossa) [Электронный ресурс]. Режим доступа: https://releases.ubuntu.com/20.04/ (дата обращения: 27.12.2021).

[7] Процессор Intel® Core™ i5-1135G7. Режим доступа https://www.intel.ru/content/www/ru/ru/products/sku/208658/intel-core-i51135g7-processor-8m-cache-up-to-4-20-ghz/specifications.html (дата обращения: 15.10.2021).