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»

Курсовая работа по курсу "Компьютерная графика"

Тема Построение реалистичного изображения трехмерного объекта в виде замка

Студент Иммореева М.А.

Группа ИУ7-52Б

Оценка (баллы) _____

Преподаватель Вишневская Т.И.

Москва — 2023 г.

Оглавление

Введение

В настоящее время вопросы, связанные с отображением на экране дисплея разнообразных изображений, как никогда актуальны. Графика используется практически во всех областях деятельности человека, так или иначе связанных с использованием компьютера. Графически представленная информация является как удобным средством для взаимодействия с пользователем, так и неотъемлемой частью вычислительного комплекса, таких как: моделирование сложных процессов, природных явлений, реалистичной графики в трехмерных компьютерных играх.

До недавнего времени основным критерием выбора способа отображения трехмерных объектов являлась скорость вычислений, в силу того, что мощности компьютеров не хватало для полноценной реализации существующих алгоритмов. Однако сейчас технологии продвинулись далеко вперед и появилась потребность в отображении реалистичных изображений с отображением таких оптических эффектов, как отражение, преломление, блики света от воды и т.д.

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

Цель курсовой работы: разработать программу для проектирования изображения модели зàмка из композиции трехмерных геометрических объектов: куб, параллелепипед, цилиндр, сфера, конус. Предусмотреть возможность изменения пользователем через интерфейс программы:

- Цвета, текстуры объектов сцены;
- Координат и направления камеры;
- Интенсивности и цвета источника освещения.

Для достижения поставленной цели, требуется выполнить следующие задачи:

1. Провести анализ существующих алгоритмов для реализации и решения задачи создания конструктора из примитивных фигур и их реалистическое представление, выбрать оптимальные;
2. Реализовать выбранные алгоритмы, создать программный продукт для решения поставленной задачи проектирования изображения модели зàмка;
3. Реализовать интуитивный интерфейс для удобства пользователя;
4. Провести исследования скорости выполнения выбранных алгоритмов.

1 Аналитическая часть

В этом разделе будут приведены требования к программе, описание модели трехмерного объекта в сцене, рассмотрены формализация объектов сцены. Будут также рассмотрены алгоритмы построения трехмерного изображения, модели освещения, закраски, а также способы текстуризации.

1.1 Описание модели трехмерного объекта

В качестве метода представления трехмерного объекта в сцене был выбран метод полигональной сетки.

Полигональная сетка – совокупность вершин, рёбер и граней, которые определяют форму многогранного объекта в трехмерной компьютерной графике. Гранями обычно являются треугольники, четырехугольники или другие простые выпуклые многоугольники (полигоны), так как это упрощает рендеринг, но также может состоять из наиболее общих вогнутых многоугольников, или многоугольников с дырками. В данной работе было выбрано представление граней в качестве треугольников. Данный метод позволяет описывать многоугольники произвольной формы, и от количества полигонов зависит реалистичность модели, однако обработка большего количества граней влечет за собой увеличение требований к ресурсам системы, так что был выбран универсальный вариант.

1.2 Формализация объектов сцены

Сцена программы состоит из следующих объектов:

1. Геометрическая композиция – представляется в виде набора геометрический объектов, описанных в одном файле. Объекты описаны в виде полигональной сетки. В качестве элементов геометрической композиции допускаются куб, параллелепипед, цилиндр, сфера, конус. Для описания каждого геометрического объекта композиции требуется указать координаты вершин, ребра и грани между ними. Далее должно

быть описание группы объектов композиции с: входящими в группу объектами, цветом или текстурой объектов, свойствами поверхности (коэффициент отражения, коэффициент блеска).

2. Источник света – представлен в виде вектора направления и обладает рядом характеристик, таких как местоположение, цвет и интенсивность излучения.

3. Камера – описывается своими координатами расположения и направлением взгляда.

1.3 Требования к работе программы

Программа должна обеспечивать построение реалистического изображения. Однако, стоит заметить, что программа обязана без задержек иметь возможность добавлять и удалять геометрические композиции, изменять характеристики объектов и т.д. Следовательно, программа обязана иметь два режима. Первый режим характеризуется быстродействием для удобной работы пользователя и редактирования сцены, тогда как второй режим введен для рендера реалистичных характеристик сцены.

1.4 Анализ алгоритмов построения трёхмерного изображения

1.4.1 Алгоритм Z-буфера

Алгоритм Z-буфера (Z-buffer) – это алгоритм, использующийся в компьютерной графике для решения проблемы скрытия граней (перекрытие одной грани другой) при отображении 3D-сцен. Основная идея алгоритма заключается в том, что в каждом пикселе экрана хранится информация о глубине затравочной точки (z-координата), а также о цвете пикселя.

Для каждого пикселя на экране алгоритм Z-буфера хранит два значения: значение z–буфера (Z–buffer), которое соответствует наиболее удален-

ной (глубокой) точке изображения в текущем пикселе и соответствующее значение цвета.

При отрисовке каждого объекта в сцене для каждого пикселя на экране вычисляется его глубина (z–координата). Если z–координата текущего пикселя меньше значения z–буфера в данной точке, то значение z-координаты и цвета пикселя в z–буфере обновляются в соответствии с текущим объектом, иначе происходит игнорирование текущего пикселя. Таким образом, в конечном итоге на экране отображаются только наиболее удаленные точки изображения, то есть те, которые не скрыты другими объектами.

Алгоритм Z-буфера является наиболее простым и эффективным методом решения проблемы скрытия по отношению ко всем другим методам. Он также обеспечивает эффективность отрисовки уровня сложности сцены и динамичности, которые могут изменяться на ходу.

На рисунке 1.1 представлена иллюстрация работы алгоритма Z-буфера

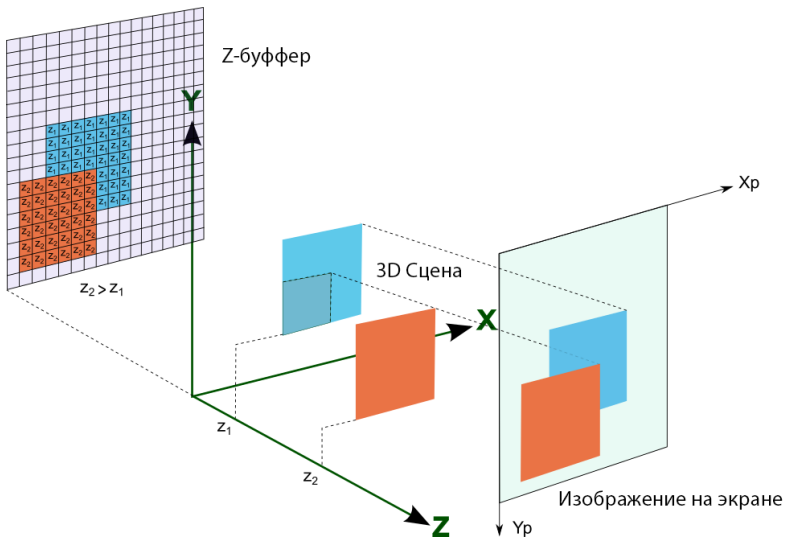

Рисунок 1.1 – Иллюстрация работы алгоритма Z-буфера

1.4.2 Алгоритм обратной трассировки лучей

Методы трассировки лучей на сегодняшний день считаются наиболее мощными и универсальными методами создания реалистических изображений. Известно много примеров реализации алгоритмов трассировки для

качественного отображения самых сложных трехмерных сцен. Можно отметить, что универсальность методов трассировки в значительной степени обусловлена тем, что в их основе лежат простые и ясные понятия, отражающие наш опыт восприятия окружающего мира.

Данный метод позволяет учесть все физические явления: отражение, преломление, воссоздание теней. У алгоритма существует недостаток: трассировка лучей каждый раз начинает заново процесс вычисления цвета пиксела, рассматривая каждый луч по отдельности. Временные затраты у алгоритма существенно большие, чем у алгоритма Z-буфера. 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.

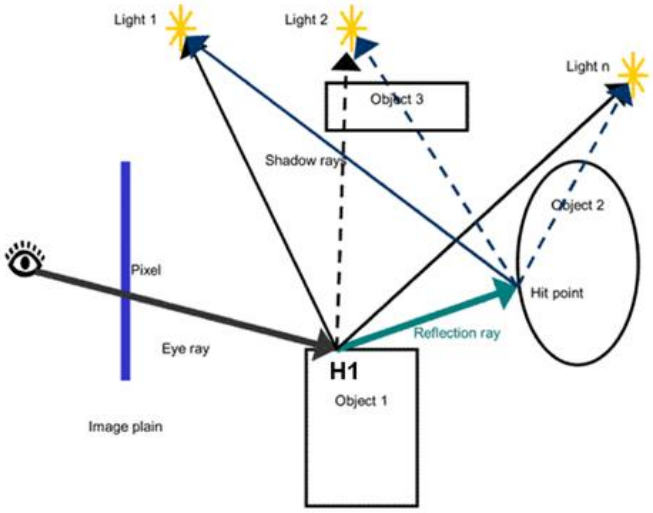

Рисунок 1.2 – Иллюстрация работы алгоритма обратной трассировки лучей

Для источника света определяется, видна ли для него эта точка. Чтобы это сделать, испускается теневой луч из точки сцены к источнику. Если луч пересек какой-либо объект сцены, то значит, что точка находится в тени, и её не надо освещать. В обратном случае требуется рассчитать степень освещенности точки. Затем алгоритм рассматривает отражающие свойства объекта: если они есть, то из точки H1 выпускается отраженный луч, и процедура повторяется рекурсивно. Тоже самое происходит при рассмотрении свойств преломления.

1.4.3 Вывод

Программа предоставляет два режима работы. В первом пользователь добавляет объекты и имеет возможность редактировать расположение конструкции, цвет или текстуру объектов, освещение. Во втором режиме предоставляется реалистическое изображение. Для первого режима был выбран Z-буфер так как важна скорость алгоритмов. Для второго режима был выбран алгоритм трассировки лучей так как важна реалистичность и учет оптических эффектов.

Характеристика	Z-буфер	Алгоритм трассировки лучей
Сложность реализации	Низкая	Высокая
Эффективность для сложных сцен	Низкая	Высокая
Точность	Высокая	Высокая
Использование ресурсов (память, процессорное время)	Высокое потребление	Низкое потребление
Поддержка прозрачности и отражений	Да	Да
Поддержка теней	Требует дополнительной обработки	Да

1.5 Анализ алгоритмов закраски

1.5.1 Метод закраски Гуро

Метод закраски, который основан на интерполяции интенсивности и известен как метод Гуро, позволяет устранить дискретность изменения интенсивности. Процесс закраски по методу Гуро осуществляется в четыре этапа:

1. Вычисляются нормали ко всем полигонам.

2. Определяются нормали в вершинах путем усреднения нормалей по всем полигональным граням, которым принадлежит вершина.

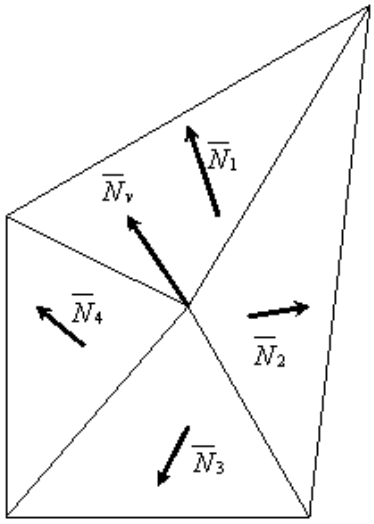

Рисунок 1.3 – Нормали к вершинам: v=(1 + 2 + 3 + 4 + v)/4

3. Используя нормали в вершинах и применяя произвольный метод закраски, вычисляются значения интенсивности в вершинах.

4. Каждый многоугольник закрашивается путем линейной интерполяции значений интенсивностей в вершинах сначала вдоль каждого ребра, а затем и между ребрами вдоль каждой сканирующей строки.

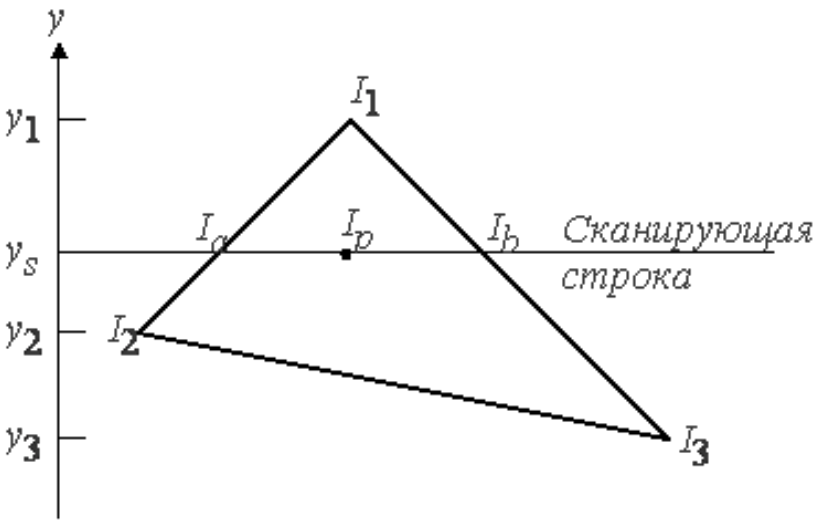

Рисунок 1.4 – Интерполяция интенсивностей

Интерполяция вдоль ребер легко объединяется с алгоритмом удаления скрытых поверхностей, построенным на принципе построчного сканирования. Для всех ребер запоминается начальная интенсивность, а также изменение интенсивности при каждом единичном шаге по координате y, Заполнение видимого интервала на сканирующей строке производится путем интерполяции между значениями интенсивности на двух ребрах, ограничивающих интервал.

$$
\begin{aligned}
I_a &= I_1 \frac{y_s - y_2}{y_1 - y_2} + I_2 \frac{y_1 - y_s}{y_1 - y_2} \\
I_b &= I_1 \frac{y_s - y_3}{y_1 - y_3} + I_3 \frac{y_1 - y_s}{y_1 - y_3} \\
I_p &= I_a \frac{x_b - x_y}{x_b - x_a} + I_b \frac{x_y - x_a}{x_b - x_a}
\end{aligned}
\tag{1.1}
$$

Для цветных объектов отдельно интерполируется каждая из компонент цвета.

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 неестественным освещением.

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 реалистического изображения будет использоваться алгоритм обратной трассировки лучей, который не требует какого-то дополнительного алгоритма закраски.

Гуро имеет более низкую сложность реализации и потребляет меньше ресурсов, но менее точен и не поддерживает прозрачность, отражения и объемные объекты. Алгоритм обратной трассировки лучей более эффективен для сложных сцен, более точен и поддерживает прозрачность, отра-

жения и объемные объекты.

Характеристика	Гуро	Алгоритм трассировки лучей
Сложность реализации	Низкая	Высокая
Эффективность для сложных сцен	Низкая	Высокая
Точность	Низкая	Высокая
Использование ресурсов (память, процессорное время)	Низкое потребление	Низкое потребление
Поддержка прозрачности и отражений	Нет	Да
Поддержка теней	Требует дополнительной обработки	Да

1.6 Анализ алгоритмов моделирования освещения

Моделирование освещения – основополагающий элемент фотореализма.

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. Свет при попадании на поверхность равномерно рассеивается во все стороны. При расчёте учитывается только ориентация нормали поверхности (нормаль $\vec{N}$) и направление на источник света (вектор $\vec{L}$). 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:

$$I_d = K_d(\vec{L}, \vec{N})I_0 \tag{1.2}$$

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.

1.6.2 Модель Фонга

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

- Фоновое освещение (ambient) – оно присутствует на любом участке сцены, не зависит от источников света, потому считается константой;
- Рассеянный свет (diffffuse) – 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:

$\vec{N}$ – вектор нормали к поверхности в точке,

$\vec{L}$ – падающий луч,

$\vec{R}$ – отражённый луч,

$\vec{V}$ – 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:

$$I_a = K_a * I_a + K_d(\underline{L}, \underline{N}) + K_s(\underline{R}, \underline{V})^p \tag{1.3}$$

На рисунке 1.5 приведён пример работы модели Фонга:

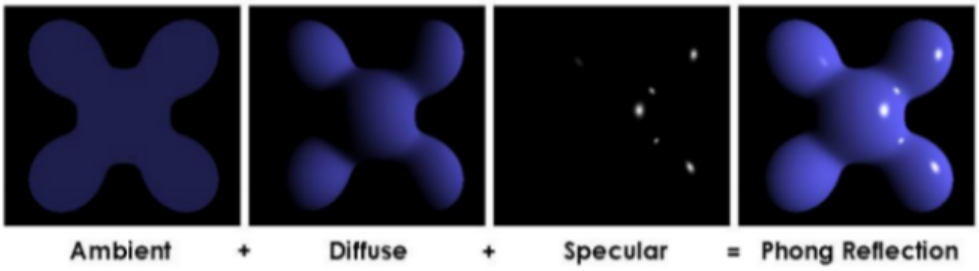

Рисунок 1.5 – Иллюстрация работы модели Фонга

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.

Помимо направления взгляда, отражённого луча, источника света, также рассматривается и составляющая преломления. 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:

$$I = K_a I_a C + K_d I_d C + K_s I_s + K_r I_r + K_t I_t \tag{1.4}$$

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 показывает высокую эффективность в комбинации с обратной трассировкой лучей.

Характеристика	Модель Ламберта	Модель Уиттеда	Модель Фонга
Расчет освещенности	Используется только диффузное отражение	Используется диффузное и зеркальное отражение	Используется диффузное, зеркальное преломленное отражение
Сложность реализации	Низкая	Средняя	Высокая
Реалистичность	Низкая	Средняя	Высокая
Использование ресурсов (память, процессорное время)	Низкое потребление	Среднее потребление	Высокое потребление
Поддержка теней	Требует дополнительной обработки	требует дополнительной обработки	Да

1.7 Текстурирование объектов трехмерной 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.

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:

$$u = \frac{(u/z)}{1/z}, v = \frac{(u/z)}{1/z} \tag{1.5}$$

1.8 Алгоритм заполнения треугольника с использованием барицентрических координат

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

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

$$\begin{gathered} p = b_o v_0 + b_1 v_1 + b_2 v_2 \\ b_0 + b_1 + b_2 = 1 \end{gathered} \tag{1.6}$$

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:

$$T = T_0 b_0 + T_1 b_1 + T_2 b_2 \tag{1.7}$$

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 текстуризации 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 трехмерных объектов

Так как пользователь должен иметь возможность составить модель для просмотра и может в любой момент изменить параметры объектов, 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:

$$\begin{aligned} X &= X_0 * R\cos(a * i) \\ Y &= Y_0 \\ Z &= Z_0 * R\sin(a * i) \end{aligned} \tag{2.1}$$

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. 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 следующий:

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

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:

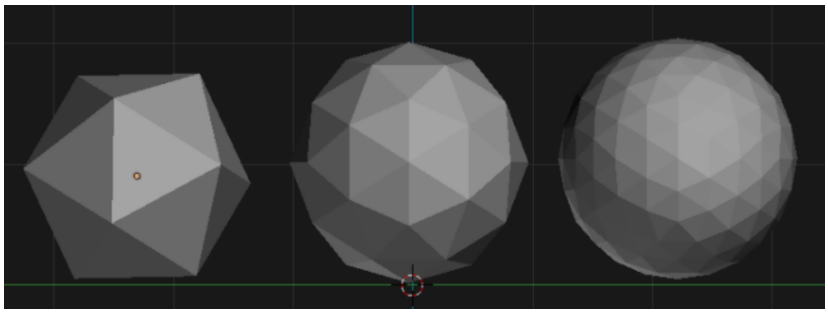

Рисунок 2.1 – Процесс итеративного приближения икосаэдра к сфере

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:

$$\begin{aligned} L &= \sqrt{X_0^2 + Y_0^2 + Z_0^2} \\ X_1 &= \frac{X_0}{L} \\ Y_1 &= \frac{Y_0}{L} \\ Z_1 &= \frac{Z0}{L} \end{aligned} \tag{2.2}$$

2.2 Описание трехмерных преобразований

2.2.1 Способ хранения декартовых координат

Для хранения координат точек будет применяться вектор-столбец, состоящий из четырех координат: x, y, z, w, причем w по умолчанию равна 1. Это сделано для комфортного умножения вектора на матрицы трансформации, которые имеют размерность 4x4.

2.2.2 Преобразование трехмерных координат в двухмерное пространство

Экран располагает только двумя координатами, потому необходимо выбрать способ, с помощью которого нужно переносить трехмерные объекты на двухмерное пространство. Любой пиксель имеет определенный цвет, и за счет этого необходимо разрешить проблему передачи объемности 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 трехмерной сцены в пространство камеры

Чтобы привести трехмерную сцену к пространству камеры, нужно выполнить операцию умножения для каждой вершины всех полигональных моделей на матрицу камеры.

Как правило, матрица камеры определяется с помощью параметров, которые зависят от трехмерного положения камеры. Т.е параметров камеры, таких как ее положение и ориентации. Положение камеры задается точкой в трехмерном пространстве, а ориентация определяется матрицей поворота.

После того, как матрица камеры определена, каждая вершина в сцене умножается на эту матрицу, чтобы получить ее координаты в пространстве камеры. Это позволяет отобразить трехмерную сцену на двумерном экране.

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:

$$A = \begin{pmatrix} \alpha_x & \beta_x & \Psi_x & 0 \\ \alpha_y & \beta_y & \Psi_y & 0 \\ \alpha_z & \beta_z & \Psi_z & 0 \\ -(P * \alpha) & -(P * \beta) & -(P * \Psi) & 1 \end{pmatrix} \tag{2.3}$$

2.2.4 Матрица перспективной проекции

Матрица проекции используется для преобразования координат вершин из пространства камеры в пространство отсечения. Пространство отсечения является усеченной пирамидой, которая представляет область пространства, которая может быть видна из камеры. Диапазон усеченной пи-

рамиды зависит от параметров камеры, таких как угол обзора и соотношение сторон экрана.

После преобразования координат вершин в пространство отсечения, значения координат x, y и z будут находиться в диапазоне от -w до w, где w - это компонента, которая отвечает за перспективное искажение. Значение w для каждой вершины зависит от ее расстояния от камеры. Чем дальше от камеры находится вершина, тем больше будет значение w, что приведет к ее сжатию вдоль оси z в пространстве отсечения. Все вершины, находящиеся вне диапазона [-w, w] будут отсечены.

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:

$$A = \begin{pmatrix} \frac{\cot\frac{\alpha}{2}}{AR} & 0 & 0 & 0 \\ 0 & \cot\frac{\alpha}{2} & 0 & 0 \\ 0 & 0 & \frac{Z_f \times Z_n}{Z_f - Z_n} & 1 \\ 0 & 0 & \frac{Z_f}{Z_f - Z_n} & 0 \end{pmatrix} \tag{2.4}$$

Следующий этап – спроецировать все координаты на одну плоскость, разде- лив 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

2.3 Алгоритм Z-буфера

Алгоритм Z-буфера используется в первом режиме работы программы. Совместно с алгоритмом Z-буфера применяется для закраски метод Гуро, модель Ламберта используется в качестве модели освещения. На рисунке изображена схема алгоритма Z-буфера с применением метода Гуро для закраски:

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.

В рамках алгоритма обратной трассировки лучей, для каждого пикселя на экране, создается луч, направленный вдоль зрительного луча, проходящего через этот пиксель. Затем, для каждого луча рассчитывается пересечение со сценой и определяется точка, в которой луч попадает на поверхность объекта.

Далее, в зависимости от свойств поверхности, на которую был направлен луч, рассчитывается вклад в светосильность от отражения и рассеяния света. Формулы Уиттеда для расчета коэффициентов отражения и рассеяния определяются в зависимости от типа поверхности.

Когда вклад от освещения рассчитан, луч продолжает свой путь, отражаясь или преломляясь в зависимости от свойств материала, пока он не попадет на источник света. Таким образом, алгоритм обратной трассировки лучей учитывает сложное взаимодействие света и объектов, что позволяет получить реалистичное изображение трехмерной сцены.

Отражённый луч можно найти, зная направление падающего луча и нормаль к поверхности. Пусть:

$\vec{L}$ – направление луча,

$\vec{n}$ – нормаль к поверхности.

Луч можно разбить на две части: , $\vec{L}_p$ которая перпендикулярна нормали, и $\vec{L}_n$ – параллельна нормали.

Представленная ситуация изображена на рисунке 2.2

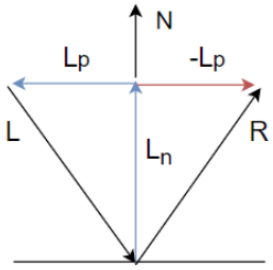

Рисунок 2.2 – Рассматриваемые векторы для расчёта отражённого луча

Учитывая свойства скалярного произведения $\vec{L}_n = \vec{n} * (\vec{n}, \vec{L})$ и $\vec{L}_p = \vec{L} - \vec{n} * (\vec{n}, \vec{L})$ так как отражённый луч выражается через разность этих векторов, то отражённый луч выражается по формуле:

$$R = 2 * \vec{n} * (\vec{n}, \vec{L}) - \vec{L} \tag{2.5}$$

По закону преломления падающий, преломлённый луч и нормаль к поверхности лежат в одной плоскости. Пусть:

μ_i – показатели преломления сред,

η_i – углы падения и отражения света.

Применяя закон Снеллиуса, параметры преломлённого луча можно вычислить по формуле:

$$\begin{gathered} R = \frac{\mu_1}{\mu_2}\vec{L} + (\frac{\mu_1}{\mu_2}\cos(\eta_1) - \cos(\eta_2))\vec{n} \\ \cos(\eta_2) = \sqrt{1 - (\frac{\mu_1}{\mu_2})^2 * (1 - \cos(\eta_1))^2} \end{gathered} \tag{2.6}$$

На рисунке приведена схема алгоритма обратной трассировки лучей:

2.5 Алгоритм пересечения луча с параллелепипедом

Алгоритм пересечения луча с параллелепипедом - это алгоритм, используемый в компьютерной графике для определения того, пересекает ли луч, исходящий из камеры, параллелепипед на своем пути.

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.

Алгоритм пересечения луча с параллелепипедом работает следующим образом: 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. Происходит проверка, пересекает ли луч плоскость параллелепипеда. Для этого используется формула пересечения луча с плоскостью, позволяющая определить расстояние от начала луча до точки пересечения.

Затем происходит проверка, находится ли эта точка пересечения внутри параллелепипеда. Для этого необходимо проверить, удовлетворяют ли координаты точки пересечения условиям нахождения внутри параллелепипеда. Если точка находится внутри параллелепипеда, то луч пересекает его, и требуемая информация может быть получена для дальнейшей обработки.

Если же точка находится вне параллелепипеда, то необходимо проверить пересечение с другими сторонами параллелепипеда. Это делается путем повторения процедуры пересечения луча с каждой из сторон, пока не будет найдена точка пересечения внутри параллелепипеда или не будут просмотрены все стороны.

Рассмотрим пару плоскостей, параллельных плоскости yz: X = x1 и X = x2. Пусть:

D – вектор направления луча.

Если координата x вектора D = 0, то заданный луч параллелен этим плоскостям, и, если x0 < x1 или x0 > x1, то он не пересекает рассматриваемый прямоугольный параллелепипед. Если же Dx не равно 0, то вычисляются отношения:

$$\begin{aligned} t_1x &= \frac{x_1 - x_0}{D_x} \\ t_2x &= \frac{x_2 - x_0}{D_x} \end{aligned} \tag{2.7}$$

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:

$$\begin{aligned} t_1x &= \frac{y_1 - y_0}{D_y} \\ t_2x &= \frac{y_2 - y_0}{D_y} \end{aligned} \tag{2.8}$$

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:

$$\begin{aligned} t_1z &= \frac{z_1 - z_0}{D_z} \\ t_2z &= \frac{z_2 - z_0}{D_z} \end{aligned} \tag{2.9}$$

и повторяются сравнения. 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)
2. Текстурные координаты (u, v)
3. Координаты нормалей (x, y, z)
4. Определение поверхности задаётся в формате i1/i2/i3, где i1 – индекс координаты вершины, i2 – индекс координаты текстуры, i3 – индекс координаты нормали

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, 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 данном 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

В программе реализованы 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 трассировке 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;

3.3 Листинг кода

На листинге 3.1 представлен код отрисовки модели в первом режиме работы программы. На листинге 3.2 представлен код закраски треугольника в первом режиме работы программы. На листинге 3.3 представлен код алгоритма трассировки одного луча.

Листинг 3.1 – Отрисовка модели в первом режиме работы программы

```
1     void Scene::rasterize(Model& _model)
2     {
3         auto camera = camers[curr_camera];
4         auto projectMatrix = camera.projectionMatrix;
5         auto viewMatrix = camera.viewMatrix();
6
7         auto rotation_matrix = model.rotation_matrix;
8         auto objToWorld = model.objToWorld();
9
10        for (auto& face: model.faces)
11        {
12            auto a = vertex_shader->shade(face.a,
                  rotation_matrix, objToWorld, camera);
13            auto b = vertex_shader->shade(face.b,
                  rotation_matrix, objToWorld, camera);
14            auto c = vertex_shader->shade(face.c,
                  rotation_matrix, objToWorld, camera);
15
16            a = geom_shader->shade(a, projectMatrix, viewMatrix);
17            b = geom_shader->shade(b, projectMatrix, viewMatrix);
18            c = geom_shader->shade(c, projectMatrix, viewMatrix);
19            rasterBarTriangle(a, b, c);
20        }
21    }
```

Листинг 3.2 – Закраски треугольника

```
1 #define Min(val1, val2) std::min(val1, val2)
2 #define Max(val1, val2) std::max(val1, val2)
3 void SceneManager::rasterBarTriangle(Vertex p1_, Vertex p2_,
     Vertex p3_)
4 {
5     if (!clip(p1_) && !clip(p2_) && !clip(p3_))
6     {
7         return;
8     }
9     denormolize(width, height, p1_);
10    denormolize(width, height, p2_);
11    denormolize(width, height, p3_);
12
13    auto p1 = p1_.pos;
```

```
14      auto p2 = p2_.pos;
15      auto p3 = p3_.pos;
16
17      float sx = std::floor(Min(Min(p1.x, p2.x), p3.x));
18      float ex = std::ceil(Max(Max(p1.x, p2.x), p3.x));
19
20      float sy = std::floor(Min(Min(p1.y, p2.y), p3.y));
21      float ey = std::ceil(Max(Max(p1.y, p2.y), p3.y));
22
23      for (int y = static_cast<int>(sy); y < static_cast<int>(ey);
            y++)
24      {
25          for (int x = static_cast<int>(sx); x <
                static_cast<int>(ex); x++)
26          {
27              Vec3f bary = toBarycentric(p1, p2, p3,
                    Vec3f(static_cast<float>(x),
                    static_cast<float>(y)));
28              if ( (bary.x > 0.0f || fabs(bary.x) < eps) && (bary.x
                    < 1.0f || fabs(bary.x - 1.0f) < eps) &&
29              (bary.y > 0.0f || fabs(bary.y) < eps) && (bary.y <
                    1.0f || fabs(bary.y - 1.0f) < eps) &&
30              (bary.z > 0.0f || fabs(bary.z) < eps) && (bary.z <
                    1.0f || fabs(bary.z - 1.0f) < eps))
31              {
32                  auto interpolated = baryCentricInterpolation(p1,
                        p2, p3, bary);
33                  interpolated.x = x;
34                  interpolated.y = y;
35                  if (testAndSet(interpolated))
36                  {
37                      auto pixel_color = pixel_shader->shade(p1_,
                            p2_, p3_, bary) * 255.f;
38                      img.setPixelColor(x, y, qRgb(pixel_color.x,
                            pixel_color.y, pixel_color.z));
39                  }
40              }
41          }
42      }
43
44  }
```

Листинг 3.3 – Алгоритм трассировки

```
1  Vec3f RayThread::cast_ray(const Ray &ray, int depth)
2  {
3
4      InterSectionData data;
5      if (depth > 4 || !sceneIntersect(ray, data))
6      return Vec3f{0.f, 0, 0};
7
8      float di = 1 - data.model.specular;
9
10     float distance = 0.f;
11
12     float occlusion = 1e-4f;
13
14     Vec3f ambient, diffuse = {0.f, 0.f, 0.f}, spec = {0.f, 0.f,
           0.f}, lightDir = {0.f, 0.f, 0.f},
15     reflect_color = {0.f, 0.f, 0.f}, refract_color = {0.f, 0.f,
           0.f};
16
17     if (fabs(data.model.refractive) > 1e-5)
18     {
19         Vec3f refract_dir = refract(ray.direction, data.normal,
               power_ref, data.model.refractive).normalize();
20         Vec3f refract_orig = Vec3f::dot(refract_dir, data.normal)
               < 0 ? data.point - data.normal * 1e-3f : data.point +
               data.normal * 1e3f;
21         refract_color = cast_ray(Ray(refract_orig, refract_dir),
               depth + 1);
22     }
23
24     if (fabs(data.model.reflective) > 1e-5)
25     {
26         Vec3f reflect_dir = reflect(ray.direction,
               data.normal).normalize();
27         Vec3f reflect_orig = Vec3f::dot(reflect_dir, data.normal)
               < 0 ? data.point - data.normal * 1e-3f : data.point +
               data.normal * 1e-3f;
28         reflect_color = cast_ray(Ray(reflect_orig, reflect_dir),
               depth + 1);
29     }
30
```

```
31      for (auto &model: models)
32      {
33          if (model->isObject()) continue;
34          Light* light = dynamic_cast<Light*>(model);
35          if (light->t == Light::light_type::ambient)
36          ambient = light->color_intensity;
37          else
38          {
39              if (light->t == Light::light_type::point)
40              {
41                  lightDir = (light->position - data.point);
42                  distance = lightDir.len();
43                  lightDir = lightDir.normalize();
44              } else{
45                  lightDir = light->getDirection();
46                  distance = std::numeric_limits<float>::infinity();
47              }
48
49              auto tDot = Vec3f::dot(lightDir, data.normal);
50
51              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]
52              InterSectionData tmpData;
53              if (sceneIntersect(Ray(shadow_orig, lightDir),
                    tmpData))
54              if ((tmpData.point - shadow_orig).len() < distance)
55              continue;
56
57              diffuse += (light->color_intensity * std::max(0.f,
                    Vec3f::dot(data.normal, lightDir)) * di);
58              if (fabs(data.model.specular) < 1e-5) continue;
59              auto r = reflect(lightDir, data.normal);
60              auto r_dot = Vec3f::dot(r, ray.direction);
61              auto power = powf(std::max(0.f, r_dot), data.model.n);
62              spec += light->color_intensity * power *
                    data.model.specular;
63          }
64
65      }
```

```
66
67      return data.color.hadamard(ambient +
68      diffuse +
69      spec +
70      reflect_color * data.model.reflective +
71      refract_color * data.model.refractive).saturate();
72  }
```

3.4 Описание интерфейса

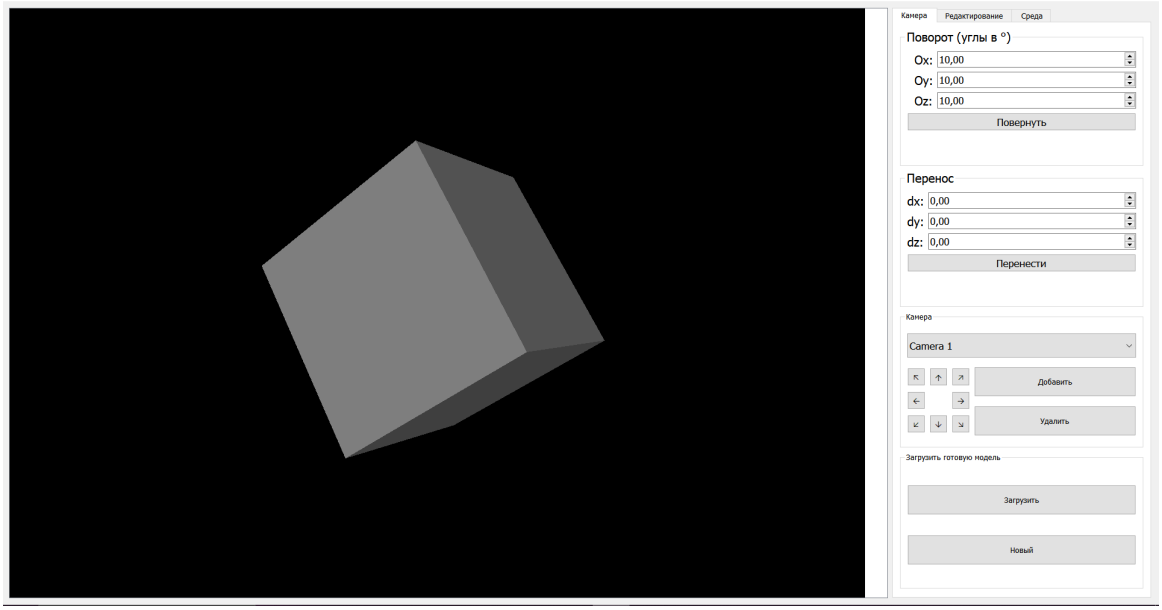

Рисунок 3.1 – Интерфейс программы, страница 1

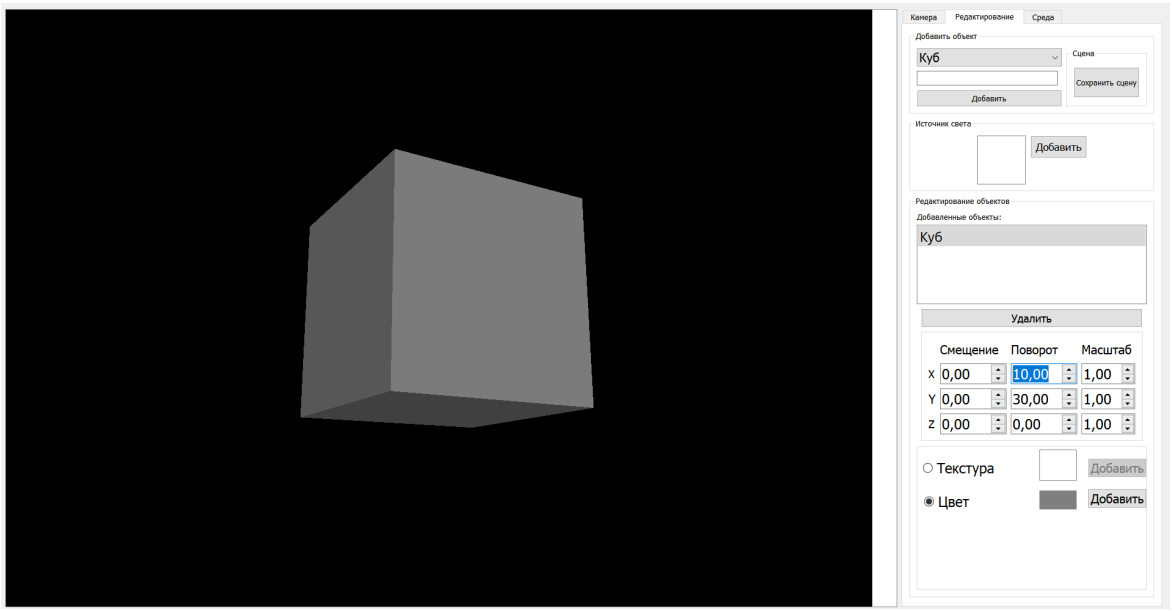

Рисунок 3.2 – Интерфейс программы, страница 2

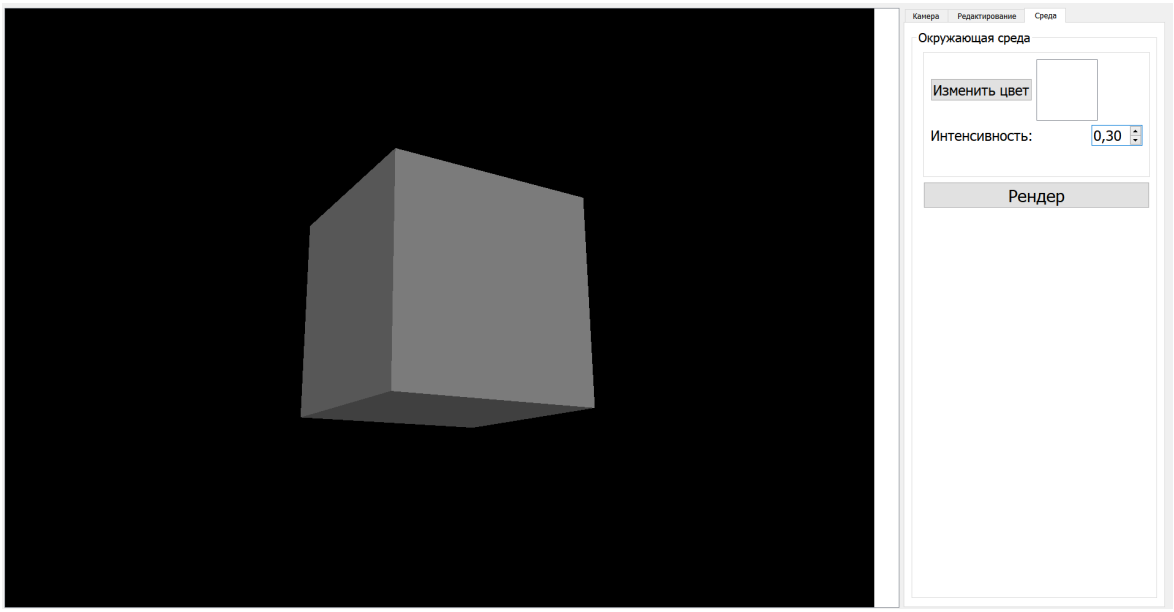

Рисунок 3.3 – Интерфейс программы, страница 3

3.5 Вывод

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

Список использованных источников

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

[2] Creating an icosphere mesh in code. Режим доступа: http://blog.andreaskahler.com/2009/06/creating-icosphere-mesh-in-code.html (дата обращения: 25.09.2023)