Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ НА ТЕМУ:

«Визуализация мыльных пузырей»

Студент Карпова Екатерина Олеговна
фамилия, имя, отчество

Студент ИУ7-52Б _____ Карпова Е. О.
группа *подпись, дата* *фамилия, и. о.*

Руководитель курсовой работы _____ Кострицкий А. С.
подпись, дата *фамилия, и. о.*

Москва — 2022 г.

Оглавление

Введение

В наши дни, в связи с повсеместной компьютеризацией, востребованность технологии компьютерного графического моделирования растёт [5]. Синтез изображения с использованием информационных технологий применяется в таких сферах, как:

— медицина;

— архитектура;

— кинематограф;

— разработка компьютерных игр.

Данный способ представления данных является более наглядным и позволяет получить полное визуальное представление какого-либо разрабатываемого проекта.

Построение трёхмерного изображения мыльных пузырей вещества может быть применимо в контексте изучения физических свойств мыльных пузырей и химических свойств вещества для повышения наглядности представления. Также, подобные изображения могут быть использованы при разработке компьютерных игр или в процессе постпроизводства отснятых кинолент.

Цель работы: разработка программы для построения трёхмерного изображения мыльных пузырей вещества.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1) формальное описание заданных объектов сцены;

2) изучение и сравнение алгоритмов построения реалистичных изображений, необходимых для визуализации мыльных пузырей вещества;

3) изучение оптических свойств мыльных пузырей вещества для синтеза реалистичного изображения;

4) разработка алгоритма, позволяющего построить реалистичное изображение мыльных пузырей вещества;

5) описание структуры разрабатываемого ПО;

6) выбор средств разработки, позволяющих реализовать выбранный алгоритм;

7) программная реализация данного алгоритма;

8) проведение исследования зависимости времени работы программы от количества выделяемых потоков.

1. Аналитическая часть

В данном разделе проводится анализ и формализация объектов синтезируемой сцены, исследование задачи построения реалистичного трёхмерного изображения сцены из данных объектов, и рассматриваются различные методы, решающие перечисленные задачи.

1.1. Описание способа определения модели трёхмерного объекта на сцене

Геометрическая модель тела — это машинное представление его формы и размера. Выделяется несколько видов геометрических моделей [7]:

— каркасная модель, которая предоставляет информацию о координатах вершин и рёбрах объекта, но не даёт представления о наличии отверстий, является простейшим видом геометрической модели тела;

— поверхностная модель, которая позволяет задавать положение и геометрию поверхностей аналитическим способом, но не даёт представления о том, с какой стороны относительно заданной поверхности находится материал;

— объёмная модель, которая предоставляет информацию о внутреннем объёме тела, является наиболее информативным видом модели и в сравнении с другими видами моделей наилучшим образом удовлетворяет задачам, в которых требуется производить нетривиальные вычисления.

К 3-D моделям применимы следующие требования [7]:

— модель не должна противоречить исходному объекту;

— модель должна допускать возможность конструирования тела целиком;

— модель должна позволять вычисление геометрических характеристик тел;

— модель должна позволять производить расчёты.

Для наглядности сравнения с учётом поставленных задач была составлена таблица 1.1 сопоставления приведённых методов определения модели на сцене. Принятые обозначения: КМ — каркасная модель, ПМ — поверхностная модель, ОМ — объёмная модель.

Таблица 1.1 — Таблица сопоставления приведённых методов определения модели на сцене

Свойство	Виды геометрических моделей		
	КМ	ПМ	ОМ
Полное соответствие исходному объекту	Нет	Нет	Да
Удовлетворение задачам, требующим проведения нетривиальных вычислений	Нет	Нет	Да
Удовлетворение задачам работы с текстурами и материалом	Нет	Нет	Да
Применимость к объектам, форма которых сферическая или близка к ней	Нет	Да	Да

По результатам сопоставления видов моделей и требований к моделям, перечисленных выше, принято решение использовать объёмную модель.

1.2. Описание способа задания геометрии трёхмерного объекта на сцене

Выделяется несколько основных способов задания геометрии трёхмерного объекта на сцене:

1) Аналитический способ. Данный способ подразумевает описание объекта математическими формулами: функцией вида $z = f(x, y)$ или неявным уравнением вида $f(x, y, z) = 0$. Зачастую используется параметрическая форма описания поверхности. Данному способы свойственны невысокая сложность расчёта координат каждой точки поверхности за счёт вычисления значения функции или параметров уравнения в данной точке, относительно невысокая теоретическая оценка потребления ресурсов, но трудоёмкость изменения геометрии объектов, и проведения динамических манипуляций [8].

2) Способ с использованием полигональной сетки. Полигональная сетка — это совокупность вершин, рёбер и граней, определяющих форму многогранного объекта в трёхмерной компьютерной графике. В качестве граней обычно применяют выпуклые многоугольники. Полигональные сетки могут быть представлены с помощью [9]:

— вершинного представления, при котором объект описывается как список вершин, соединённых с другими вершинами. При этом информация о гранях и рёбрах не выражена явно, поэтому для генерации граней необходимо обойти все данные, что замедляет обработку информации и усложняет выполнение операций с рёбрами и гранями, но обеспечивает теоретически более оптимальное потребление ресурсов, относительно нижеописанных представлений;

— списка граней, представляющим объект как совокупность множеств граней и вершин, что позволяет осуществлять явный поиск вершин грани, и граней окружающих вершину. Такое представление также делает менее трудозатратными обход или поиск данных и изменение геометрии, чем при вершинном представлении;

— «крылатого» представления, при котором объект описывается как совокупность списков вершин, рёбер и граней. Такое представление обеспечивает высочайшую гибкость в изменении геометрии сетки, потому что могут быть быстро выполнены операции разрыва и объединения. Также для него свойственны теоретически неоптимальное потребление ресурсов и усложнение данных из-за содержания множества индексов.

Для наглядности сравнения с учётом поставленных задач была составлена таблица 1.2 сопоставления приведённых методов определения модели на сцене. Принятые обозначения: АС — аналитический способ, ПС — полигональная сетка, n — количество полигонов сетки.

Таблица 1.2 — Таблица сопоставления приведённых методов задания геометрии модели на сцене

Свойство	Способы задания геометрии модели	
	АС	ПС
Трудоёмкость получения координат каждой точки поверхности	$O(1)$	$O(n)$
Теоретическая оценка потребления ресурсов	$O(1)$	$O(n)$
Трудоёмкость изменения геометрии объектов	Да	Нет

Оценки сложности в приведённой таблице сформированы на основе данных из источника [8]. В данной работе было принято решение использовать аналитический способ задания трёхмерного объекта на сцене, как основной. Способ с использованием полигональной сетки может быть использован, как дополнительный, при необходимости изменении

геометрии объекта. В качестве элементов полигональной сетки было принято решение использовать треугольные полигоны, как наиболее широко используемые и обеспечивающие большую гибкость и временную эффективность при проведении вычислений [4]. В работе было использовано представление в виде списка граней, характеризующееся большей наглядностью представления объекта и относительной эффективностью обработки данных в сравнении с другими перечисленными методами. Также оно является наиболее широко применимым среди остальных перечисленных представлений [9].

1.3. Формализация объектов синтезируемой сцены

Синтезируемая сцена состоит из следующих объектов:

1) Геометрический объект, заданный аналитически или представляемый в виде полигональной сетки из треугольных полигонов. Сетка реализована с использованием списка граней, то есть указания координат каждой вершины и связей каждой вершины с двумя соседними. Для описания геометрического объекта необходимы сведения о:

   — фоновом освещении;

   — диффузном освещении;

   — зеркальном освещении;

   — значении коэффициента фонового освещения;

   — значении коэффициента диффузного освещения;

   — значении коэффициента зеркального освещения;

   — значении коэффициента отражения;

   — значении коэффициента преломления;

   — значении показателя преломления;

   — значении степени, аппроксимирующей пространственное распределение зеркально отражённого света.

   Можно изменять геометрию объекта в процессе использования программы. На сцене может одновременно находиться несколько объектов.

2) Источник света, характеризуемый пространственным расположением, цветом и интенсивностью излучения. Точечный источник излучает свет равномерно во всех направлениях. Также источник света может быть расположен на бесконечности. Тогда он будет иметь выделенное направление освещения. Можно задавать произвольное количество источников света.

3) Камера, характеризующаяся положением в пространстве и направлением взгляда.

1.4. Анализ задачи синтеза реалистичного трёхмерного изображения сцены и методов её решения

Задача синтеза реалистичного трёхмерного изображения — это задача создания визуального представления объектов, имеющих формальное описание [7]. Таким образом, объектами данной задачи являются искусственно созданные изображения, а подзадачами — построение, генерация и преобразование изображения.

К основным этапам синтеза относятся [7]:

1) Разработка трехмерной математической модели синтезируемой визуальной обстановки.

2) Определение направления линии визирования , положения картинной плоскости, размеров окна обзора.

3) Формирование операторов, осуществляющих пространственное перемещение моделируемых объектов.

4) Преобразование модели, синтезируемой в пространстве, к координатам наблюдателя.

5) Отсечение объектов визуального пространства в пределах пирамиды видимости.

6) Вычисление двумерных перспективных проекций синтезируемых объектов видимости на картинную плоскость.

7) Исключение невидимых элементов синтезируемого пространства при данном положении наблюдателя, закрашивание и затенение видимых элементов объектов визуализации.

8) Вывод полутонового изображения синтезируемого визуального пространства на экран растрового дисплея.

Далее подробно рассмотрим 7 этап, как подразумевающий большее количество вариаций в зависимости от поставленных задач.

1.4.1. Задача удаления невидимых линий и поверхностей синтезируемого пространства для данного положения наблюдателя

— алгоритмы, работающие в объектном пространстве — это алгоритмы, которые работают в мировой системе координат. Такие алгоритмы гарантируют более точные результаты, но неэффективны для сложных сцен;

— алгоритмы, работающие в пространстве изображения — это алгоритмы, которые работают в системе координат экрана, на котором визуализируются объекты сцены, поэтому их точность ограничивается разрешающей способностью экрана.

Пример удаления, например, невидимых рёбер, проиллюстрирован на рис. 1.1.

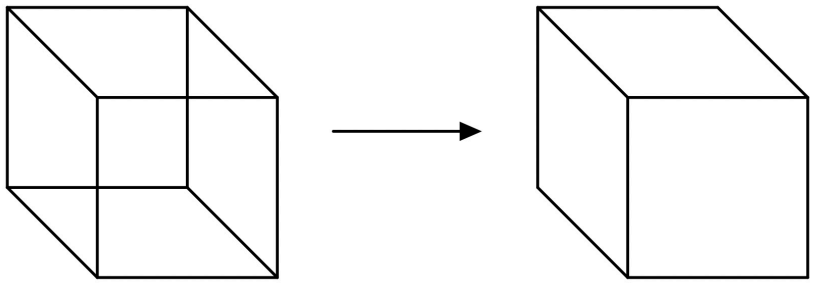

Рисунок 1.1 — Пример удаления невидимых рёбер

Далее будут рассмотрены некоторые алгоритмы удаления невидимых линий и поверхностей.

Алгоритм, использующий z-буфер

Данный алгоритм считается одним из простейших алгоритмов для удаления невидимых поверхностей [6]. Алгоритм, использующий z-буфер, работает в пространстве изображения. Этот алгоритм основан на идее о буфере кадра, который представляет из себя хранилище информации об интенсивности каждого пиксела в пространстве изображения. Также, в данном алгоритме используется z-буфер, хранящий координату z каждого видимого пиксела в пространстве изображения.

Основные этапы работы алгоритма можно описать так:

1) Буфер кадра заполняется значением, соответствующим фоновому цвету.

2) z-буфер заполняется минимальным значением координаты z.

3) Каждый многоугольник преобразуется в растровую форму, для каждого пиксела многоугольника вычисляется значение координаты z.

4) Если глубина текущего пиксела больше, чем глубина, записанная на соответствующей по координатам x, y позиции в z-буфере, то записать её на эту позицию и обновить информацию в буфере кадра для данной позиции. Иначе, никаких действий не производится.

Там, где это целесообразно, то есть для невыпуклых многогранников, предварительным шагом будет являться удаление нелицевых граней.

Уравнение плоскости имеет вид $ax+by+cz+d=0$. Тогда координату z можно выразить так:

$$z = \frac{-(ax+by+d)}{c} \neq 0. \tag{1.1}$$

Для текущей сканирующей строки значение y является постоянным. Тогда для пиксела с координатой $x^{'} = x + \delta x$ глубину можно выразить так:

$$z^{'} - z = -\frac{ax' + d}{c} + \frac{ax + d}{c} = \frac{a(x - x')}{c}. \tag{1.2}$$

Данная запись эквивалентна записи

$$z^{'} = z - \frac{a}{c}\delta x. \tag{1.3}$$

Учитывая, что пикселы рассматриваются последовательно по сканирующей строке, то есть $\delta x = 1$, формулу можно записать в виде:

$$z^{'} = z - \frac{a}{c}. \tag{1.4}$$

Пример работы алгоритма представлен на рис. 1.2.

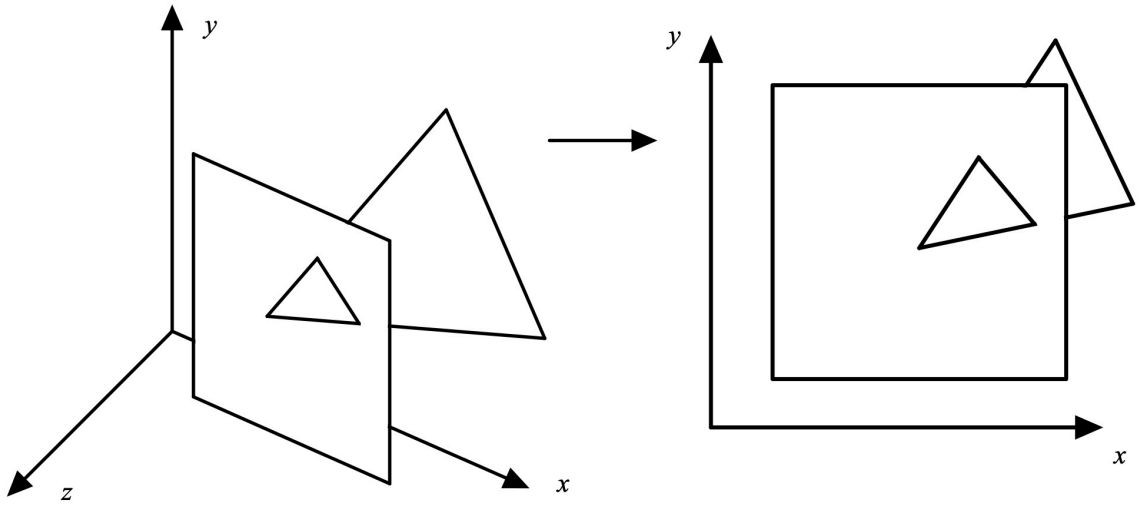

Рисунок 1.2 — Пример работы алгоритма, использующего z-буфер

Свойством данного алгоритма является простота решения им задач визуализации сложных пересечений и удаления невидимых поверхностей [6]: они не требуют особой обработки и рассматриваются аналогично со всеми остальными ситуациями. Алгоритм, использующий z-буфер, допускает построение сцен любой сложности, при этом не требуя предварительной сортировки элементов сцены и обеспечивая линейную оценочную трудоёмкость. К свойствам данного алгоритма также относятся большое потребление ресурсов в теоретической оценке и трудоёмкость устранения лестничного эффекта, реализации эффектов прозрачности и просвечивания. Попытка реализовать данные свойства зачастую приводит к некорректной работе алгоритма. Также, для невыпуклых тел приходится производить дополнительные вычисления, что замедляет работу алгоритма.

Алгоритмы построчного сканирования

Данный класс алгоритмов обрабатывает сцену в порядке прохождения по её пикселам сканирующей строкой [6] и оперируют в пространстве изображения. За счёт работы с растровой развёрткой данные алгоритмы позволяют рассматривать задачу удаления невидимых поверхностей как более тривиальную задачу удаления невидимых линий.

Сканирующая плоскость определяется точкой положения наблюдателя и сканирующей строкой. Пересечение заданной плоскости и сцены определяет окно, в пределах которого решается задача удаления невидимых линий. Пример расположения сканирующей плоскости изображён на рисунке 1.3.

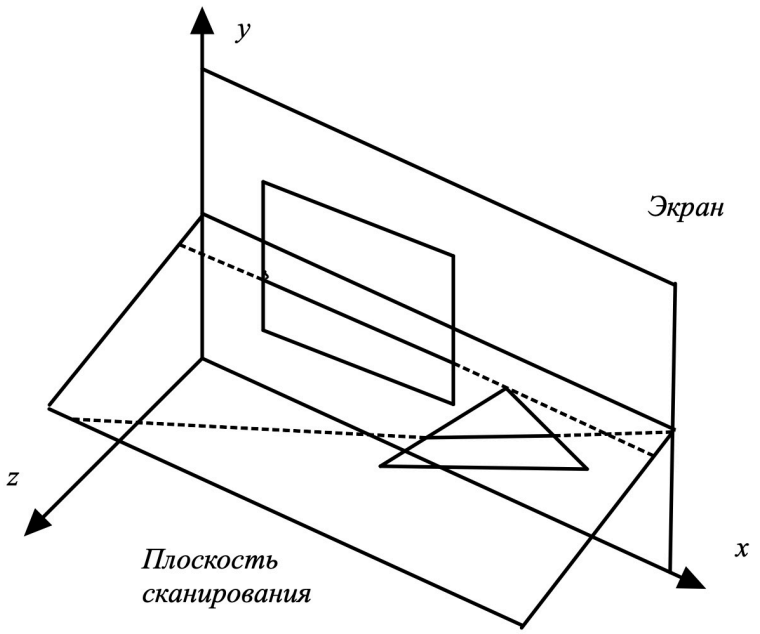

Рисунок 1.3 — Пример расположения сканирующей плоскости

К этому классу алгоритмов относится, например, алгоритм построчного сканирования, использующий z-буфер. В отличие от обычного алгоритма z-буфера, в этом алгоритма буферы хранятся исключительно для обрабатываемой строки, что улучшает теоретическую оценку потребления ресурсов. Основные этапы работы алгоритма можно описать так:

1) Для каждой сканирующей строки буфер кадра заполняется значением, соответствующим фоновому цвету, а z-буфер — минимальным значением координаты z.

2) Определяется пересечение строки с двумерными проекчиями объектов сцены. Пересечения образуют пары.

3) Глубина каждого пиксела в интервале пары сравневается с глубиной на текущей позиции в z-буфере. Если глубина этого пиксела больше, то данный отрезок видим, а соответствующая ему интенсивность заносится в буфер кадра.

4) Результаты сканирования по строке выводятся на экран слева направо.

Еще одним алгоритмом данного класса является интервальный алгоритм построчного сканирования. В отличие от алгоритма построчного сканирования, использующий z-буфер, где вычисление глубины многоугольника происходит на каждом пикселе сканирующей строки, в этом алгоритме строка делится на интервалы точками пересечения рёбер. Это позволяет дополнительно улучшить теоретическую оценку временных затрат.

Некоторые модификации данных алгоритмов позволят также учесть тени, эффект прозрачности и устранить ступенчатость, но не зеркальность поверхности. Также, данные алгоритмы имеют высокую теоретическую оценку по потреблению ресурсов.

Алгоритм Робертса

Алгоритм Робертса считается первым известным решением задачи удаления невидимых линий [6]. Данный алгоритм работает в объектном пространстве и только с выпуклыми телами. Для корректной работы алгоритма должна быть гарантирована возможность представить любое тело как набор пересекающихся плоскостей. В данном алгоритме используются эффективные математические методы, однако его вычислительная трудоёмкость без применения оптимизаций оценивается в $O(n^2)$, где n — количество объектов сцены, что снижает его «привлекательность».

Можно выделить четыре основных этапа работы алгоритма:

1) Подготовка данных. На этом этапе для каждого тела формируется матрица тела, которая далее будет обозначаться как V. Любая плоскость может быть задана урав-

нением вида $ax + by + cz + d = 0$. В матричном виде это можно записать так:

$$\begin{pmatrix} x & y & z & 1 \end{pmatrix} \begin{pmatrix} a & b & c & d \end{pmatrix}^T = 0. \tag{1.5}$$

Любое тело, по необходимым условиям корректной работы алгоритма, упомянутым выше, может быть представлено как набор пересекающихся плоскостей. Тогда любое тело может быть описано матрицей вида

$$V = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ b_1 & b_2 & \dots & b_n \\ c_1 & c_2 & \dots & c_n \\ d_1 & d_2 & \dots & d_n \end{pmatrix}. \tag{1.6}$$

Любая точка представима в однородных координатах как

$$S = \begin{pmatrix} x, & y, & z, & 1 \end{pmatrix}^T. \tag{1.7}$$

Если заданная точка принадлежит плоскости, то в результате умножения координат данной точки на вектор коэффициентов уравнения плоскости будет получено нулевое значение, как в (1.17). Иначе, будет получено ненулевое значение, а его знак будет определять, по какую сторону от данной плоскости находится точка. В алгоритме Робертса предполагается, что если точка находится внутри тела, то все произведения координат данной точки на вектора коэффициентов уравнений плоскостей, составляющих это тело, будут положительны. Поэтому зачастую требуется производить коррекцию полученной матрицы, например:

$$V = \begin{pmatrix} a_1 & a_2 \cdot (-1) & a_3 & a_4 \\ b_1 & b_2 \cdot (-1) & b_3 & b_4 \\ c_1 & c_2 \cdot (-1) & c_3 & c_4 \\ d_1 & d_2 \cdot (-1) & d_3 & d_4 \end{pmatrix}. \tag{1.8}$$

В случае, если для очередной грани условие положительности описанного произведения не выполняется, соответствующий столбец матрицы необходимо умножить на -1, что и происходит со вторым столбцом в (1.8). Для проведения проверки следует взять точку, координаты которой будут получены усреднением всех соответствующих координат вершин данного тела.

2) Удаление рёбер или граней, экранируемых самим телом. Полагается, что наблюдатель находится в бесконечности на положительной полуоси z, а его взгляд направлен в сторону отрицательной полуоси z. Тогда вектор взгляда наблюдателя можно задать как

$$E = \begin{pmatrix} 0, & 0, & -1, & 0 \end{pmatrix}^T. \tag{1.9}$$

Для определения невидимых граней тела, экранируемых самим телом, необходимо умножить вектор E на матрицу данного тела V. В результирующем векторе отрицательные компоненты будут соответствовать невидимым граням. Экранируемые рёбра определяются по следующему правилу: невидимые ребра образуются в результате пересечения невидимых граней. Если сцены включает в себя одно тело, работа алгоритма завершается, иначе выполняется следующий этап.

3) Удаление рёбер или граней, экранируемых другими телами сцены. Пусть наблюдатель находится в точке

$$g = \Big(0, \quad 0, \quad 1, \quad 0\Big)^T, \tag{1.10}$$

а каждое ребро тел можно задать параметрическим уравнением вида

$$p(t) = p_1 + (p_2 - p_1)t, \quad 0 \ \leq \ t \ \leq \ 1 \tag{1.11}$$

где p_1 и p_2 — начало и конец ребра соответственно, а t — параметр. Тогда можно записать уравнение для луча, соединяющего произвольную точку на ребре с точкой, в которой расположен наблюдатель, в виде

$$Q(t, \alpha) = p(t) + \alpha g, \tag{1.12}$$

где $p(t)$ — произвольная точка на данном ребре, а α — параметр, указывающий расположение точки на заданном луче, причём $\alpha \ \geq \ 0$, что указывает на то, что наблюдатель находится перед телом. Можно составить систему неравенств

$$h_j = QV, \tag{1.13}$$

где $h_j > 0, \ j = \overline{1, n}$, а n — количество граней тела. Невидимым точкам ребра p_1p_2 соответствуют такие значения параметров t и α, при которых выполняются все неравенства системы. Система неравенств задает область допустимых решений. Все точки расположенные внутри области определяют координаты невидимых точек отрезка. Минимальное и максимальное значения параметра t задают координаты начала и конца невидимой части отрезка, поэтому необходимо их вычислить.

4) Удаление линий пересечения тел, экранируемых самими телами, которым принадлежат эти линии, так как эти тела связаны отношением протыкания, и другими телами. В случае существования отношения протыкания появляются решения на границе $\alpha = 0$. Для этого необходимо запомнить все точки протыкания и добавить к сцене отрезки, образованные путём соединения каждой точки протыкания со всеми остальными точками протыкания для данной пары объектов. Затем проверяется экранирование этих отрезков данными и другими телами. Видимые отрезки образуют структуру протыкания.

На рисунке 1.4 представлены основные сущности, необходимые для понимания этапов работы алгоритма Робертса.

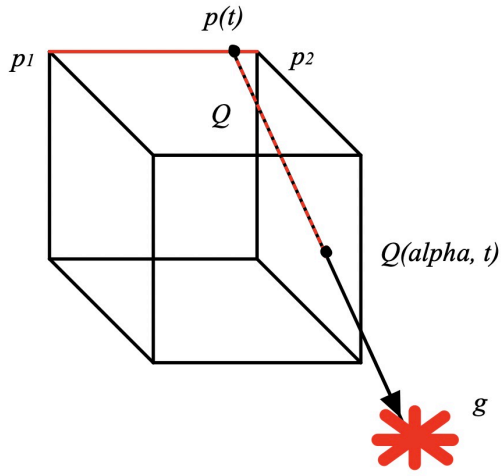

Рисунок 1.4 — Основные сущности, необходимые для понимания этапов работы алгоритма Робертса

Данный алгоритм предоставляет высокую точность вычислений, как и другие алгоритмы, работающие в объектном пространстве, но обладает высокой оценочной сложностью и не позволяет изображать зеркальные поверхности. Также, к недостаткам данного алгоритма можно отнести необходимость дополнительных проверок: на выпуклость тел и на корректность полученных матриц тел.

Было принято решение не рассматривать другие алгоритмы, работающие в объектном пространстве, как и алгоритм Робертса, так как они имеют схожие свойства и нецелесообразны к использованию для отрисовки сложных сцен.

Алгоритмы разбиения на окна

Данные алгоритмы основаны на идее о том, что большая часть времени обработки уходит на анализ областей с высоким информационным содержимым [6]. То есть производится попытка ускорить обработку сцены за счёт предположения об однородности отдельных достаточно крупных её участков.

Основные этапы можно обобщённо описать так:

1) Рассматривается окно размером в растр изображения.

2) Если в окне есть объекты и оно не является достаточно простым для визуализации, то окно разбивается на подокна.

3) Так происходит до тех пор, пока на данном шаге окно либо не окажется пустым, либо не окажется достаточно простым для визуализации, либо не достигнет предела разрешения, ограниченного точностью растрового дисплея.

4) В таком случае, информация в окне усредняется, и результат изображается с одинаковой интенсивностью.

Одним из алгоритмов этого класса является алгоритм Варнока. Данный алгоритм работает в пространстве изображения. Основные этапы можно описать так:

1) Окно, выделяемое по вышеописанному принципу, делится на 4 части, если оно непустое. Окно считается непустым, когда все многоугольники по отношению к нему являются внешними. Критерии расположения многоугольников описаны ниже.

2) Если все многоугольники сцены являются внешними: окно закрашивается цветом фона. Если внутри окна только один многоугольник: площадь окна вне многоугольника заполняется фоновым цветом, а сам многоугольник заполняется заданным для него цветом.Если окно охвачено ровно одним многоугольником, окно заполняется цветом многоугольника. Если с окном связано несколько многоугольников, но ближе всех прочих к наблюдателю расположен охватывающий, то окно заполняется цветом охватывающего многоугольника. В более сложных случаях проводится разбиение окна.

3) В случае достижения предела точности определяем глубину каждого из рассматриваемых многоугольников в этой точке. Пиксел закрашивается цветом охватывающего его многоугольника, наиболее близкого относительно центра данного пиксела, либо цветом фона, если для этого пиксела нет охватывающих многоугольников.

Для выпуклых многогранников перед началом работы требуется устранение нелицевых граней.

Способы расположения многоугольников относительно окна:

— внешний располагается целиком вне данного окна (a на рис. 1.5);

— внутренний располагается целиком внутри данного окна (b на рис. 1.5);

— пересекающий пересекает границу данного окна (c на рис. 1.5);

— охватывающий полностью заключает данное окно внутри себя (d на рис. 1.5);

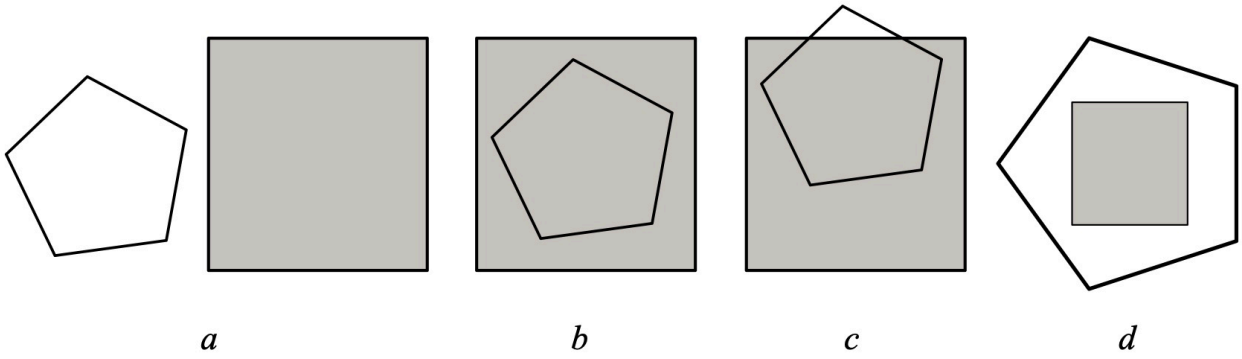

Рисунок 1.5 — Способы расположения многоугольников относительно окна

Алгоритм Вейлера-Азертона работает немного иначе, осуществляя сортировки по глубине, но так же подразумевает разбиение растра на более мелкие участки. Рассмотрение этого алгоритма, в связи с тем, что он работает в объектом пространстве и нецелесообразен к использованию на сложных сценах, было принято решение не проводить.

Алгоритмы разбиения на окна позволяют устранять ступенчатость при использовании некоторых модификаций, но требуют выполнения большого количества разбиений, что может замедлять работу реализации, и не позволяют учитывать определённые оптические эффекты. Также, некоторые из них, например, алгоритм Варнока, требует проведения дополнительных операций в частных, но достаточно распространённых случаях.

Алгоритм прямой трассировки лучей

В отличие от ранее рассмотренных алгоритмов, эффективность алгоритма прямой трассировки лучей не зависит от характеристик сцены, к которой он примняется [6]. Это связано с тем, что процесс его работы не учитывает характеристики объектов. Главная идея алгоритма может быть выражена в следующих соображениях:

1) Из заданных источников света испускаются лучи света.

2) Лучи света достигают поверхностей объектов и либо преломляются, либо проходят сквозь неё.

3) Свет доходит до наблюдателя и позволяет ему увидеть объект, через который прошёл или в котором преломился данный луч.

Было установлено, что такая последовательность работы алгоритма будет неэффективна в связи с тем, что немногие лучи света достигнут наблюдателя. Поэтому будет выполнено множество лишних вычислений, из-за чего данный алгоритм практически не используется [6]. В связи этим он будет исключён из дальнейшего рассмотрения.

Алгоритм обратной трассировки лучей

Алгоритм обратной трассировки лучей работает в пространстве изображения [6]. Картинная плоскость, т.е. растр, перпендикулярна оси OZ. Наблюдатель находится на положительной полуоси OZ. При простой трассировке полагается, что наблюдатель расположен в бесконечности, и, соответственно, все световые лучи параллельны оси z. При учёте перспективы полагается, что наблюдатель расположен не в бесконечности, и, соответственно, световые лучи необязательно параллельны оси z. На рис. 1.6 изображена простая трассировка луча, а на рис. 1.7 изображена трассировка луча с учётом перспективы.

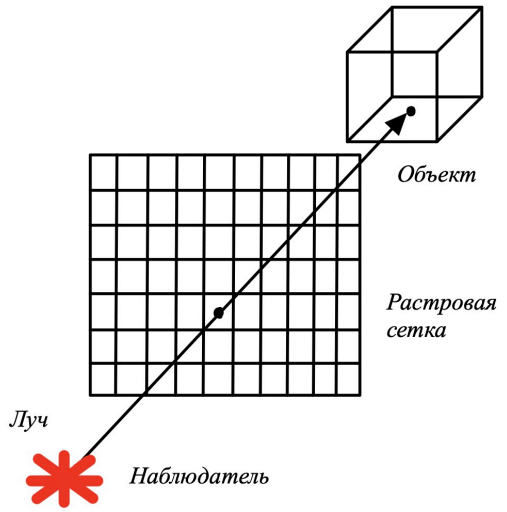

Рисунок 1.6 — Простая трассировка луча

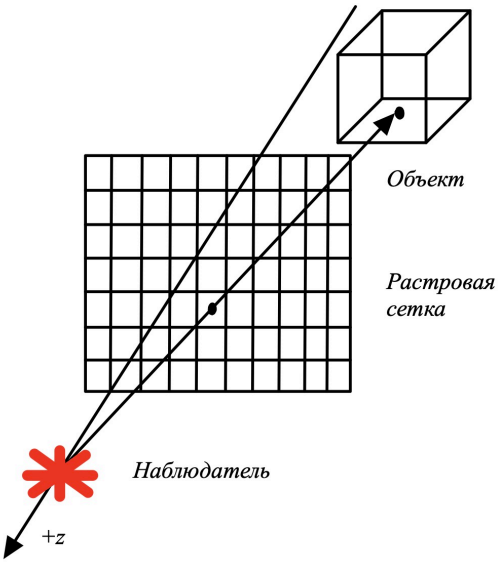

Рисунок 1.7 — Трассировка луча с учётом перспективы

Основные этапы работы алгоритма обратной трассировки лучей:

1) В каждый пиксел растра последовательно испускаются лучи из положения наблюдателя.

2) Осуществляется поиск ближайшего к наблюдателю пересечения луча взгляда с объектом сцены. Необходимо проверить пересечение каждого объекта сцены с каждым лучом. Пересечение с максимальным значением z представляет видимую поверхность для данного пиксела.

3) Для дальнейшего определения цвета пиксела рассматриваются лучи от точки пересечения луча наблюдения с объектом к каждому источнику света. Если на пути к источнику света луч пересекает иной объект, то свет от того источника не учитывается в расчёте цвета данного пиксела. Если для луча от наблюдателя не найдено объектов, с которыми он пересекается, пиксел закрашивается цветом фона.

4) Для расчёта отражений и преломлений луча, встретившего на своей траектории объект, используются физические законы, например, равенство угла падения и отражения, закон Снеллиуса, рассчитывается направление луча отраженного и преломлённого. Найденная точка пересечения теперь считается точкой наблюдения, и описанный алгоритм испускания луча повторяется столько раз, сколько составляет максимальная глубина рекурсивных погружений.

Алгоритм обратной трассировки лучей подразумевает множество вычислений, поэтому синтез изображения происходит долго. Возможной является многопоточная реализация алгоритма, при которой вычисление цвета каждого пиксела может быть выполнено параллельно. К свойствам данного алгоритма также относятся независимость метода от специфики обрабатываемого объекта, относительная простота встраивания учёта эффектов отражения одного объекта от другого, преломления, прозрачности, затенения и устранения лестничного эффекта. С этим связана высокая реалистичность получаемого изображения.

Сравнение рассмотренных алгоритмов удаления невидимых линий и поверхностей

Для наглядности сравнения рассмотренных алгоритмов удаления невидимых линий и поверхностей с учётом поставленных задач была составлена таблица 1.3. Принятые обозначения: ЗБ — алгоритм, использующий z-буфер, ПС — алгоритмы построчного сканирования, Р — алгоритм Робертса, РО — алгоритмы разбиения на окна, ОТ — алгоритм обратной трассировки лучей, w — ширина растра, h — высота растра, n — количество объектов сцены.

Таблица 1.3 — Таблица сопоставления рассмотренных алгоритмов удаления невидимых линий и поверхностей

Свойство	Алгоритм или класс алгоритмов				
	ЗБ	ПС	Р	РО	ОТ
Теоретическая оценка потребления ресурсов	$O(w \cdot h)$	$O(w)$	$O(n)$	$O(\log(w \cdot h))$	$O(1)$
Необходимость объёмных дополнительных или «лишних» вычислений в частных случаях	Нет	Нет	Да	Да	Нет
Возможность учёта всех оптических эффектов (отражение света, преломление света, тени и т.д.)	Нет	Нет	Нет	Нет	Да
Применимость к сложным сценам	Да	Да	Нет	Да	Да
Теоретическая оценка эффективности по времени	$O(w \cdot h \cdot n)$	$O(w \cdot h \cdot n)$	$O(n^2)$	$O(w \cdot h \cdot n)$	$O(w \cdot h \cdot n)$

Оценки сложности в приведённой таблице сформированы на основе данных из источников [6] и [7].Таким образом, так как алгоритм обратной трассировки лучей наилучшим образом удовлетворяет поставленным задачам, именно этот алгоритм был выбран для реализации программы.

1.4.2. Задача учёта теней на синтезируемом пространстве для заданного положения наблюдателя

Для заданного положения наблюдателя на видимой части изображения появляются тени в случае, когда положение наблюдателя не совпадает с положением единственного источника света. Иначе, теней не видно. Тень состоит из двух частей — полной тени и полутени [6]. В машинной графике в основном рассматриваются только точечные источники света, создающие только полную тень, так как в ином случае вычислительные затра-

ты резко возрастут. Если источник светы находится в бесконечности, тени определяются при помощи ортогонального проецирования, иначе — с помощью перспективной проекции. Процедура определения теней состоит в том, чтобы условно второй раз произвести действие по определению невидимых поверхностей, но уже не относительно наблюдателя, а относительно каждого источника света. Обобщённая схема процесса определения теней проилюстрирована рис. 1.8.

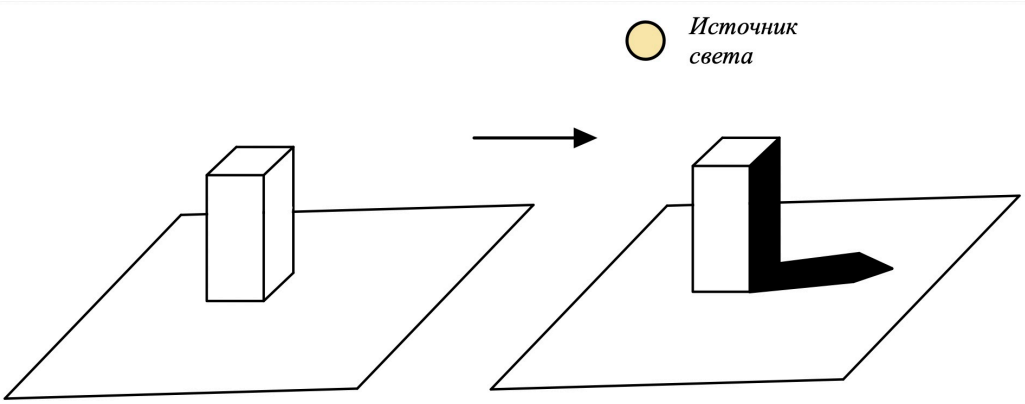

Рисунок 1.8 — Обобщённая схема процесса определения теней

В качестве алгоритма удаления невидимых поверхностей был выбран алгоритм обратной трассировки лучей. В этом алгоритме определение теней происходит в процессе работы алгоритма. Если луч, испущенный из точки пересечения луча наблюдения с каким-либо объектом к источнику света, встречает на своём пути иные объекты, то свет от этого источника не доходит до данного пиксела, поэтому он находится в тени относительно этого источника света. Иначе, свет от этого источника будет учтён при отрисовке данного пиксела. Трассировка лучей с построением теней проиллюстрирована рис. 1.9.

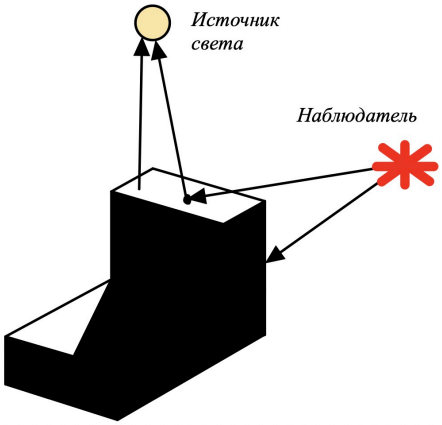

Рисунок 1.9 — Трассировка лучей с построением теней

1.4.3. Задача учёта освещения на синтезируемом пространстве для заданного положения наблюдателя

Для учёта освещенности на видимой части изображения необходимо выбрать используемую модель освещения. Она может быть локальной, что означает учёт только света от источников и ориентации поверхности, или глобальной, что означает также учёт света, отражённого от других объектов или прошедшего сквозь них. Общая схема работы глобальной модели освещения изображена на рис. 1.10, причём подразумевается, что на сцене есть тела с зеркальной и прозрачной поверхностями.

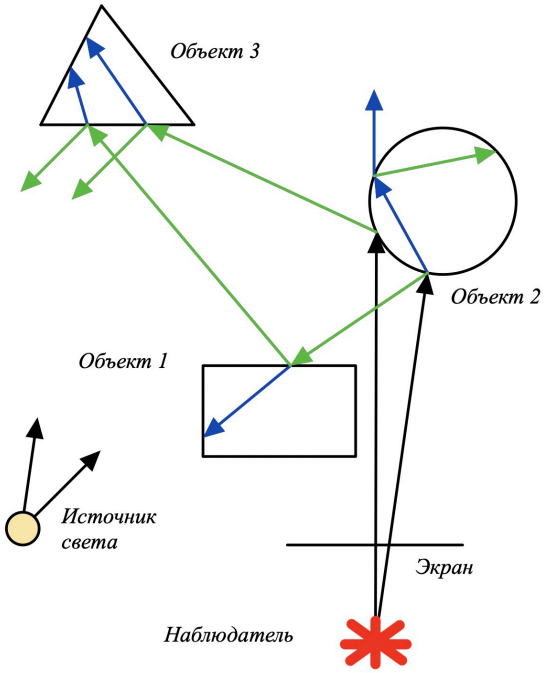

Рисунок 1.10 — Общая схема работы глобальной модели освещения

Используя глобальную модель освещения, можно добиться высокой реалистичности изображения, что в большей степени соответствует поставленным задачам. Только использование алгоритма обратной трассировки лучей позволяет применить глобальную модель освещения в связи с тем, что не исключает из рассмотрения невидимые для наблюдателя грани и позволяет учесть их при отражении и преломлении световых лучей. Таким образом, в данной работе будет использоваться глобальная модель освещения.

При использовании глобальной модели освещения для каждого пиксела изображения определяется его интенсивность. Рассчитываются несколько составляющих интенсивности:

— *ambient* — фоновое освещение. Постоянно, учитывается для любого участка сцены и не зависит от пространственных координат освещаемой точки и источника;

— $diffuse$ — рассеянный свет. Рассчитывается, как в модели Ламберта по закону косинусов (закону Ламберта):

$$I_d = k_d \cos(L, N) i_d = k_d (L \cdot N) i_d, \tag{1.14}$$

где I_d — рассеянная составляющая освещенности в точке, k_d — свойство материала воспринимать рассеянное освещение, L — направление из точки на источник, N — вектор нормали в точке, i_d — мощность рассеянного освещения;

— $specular$ — зеркальная составляющая освещения. Зависит от того, насколько близко (насколько мал угол) находятся вектор отражённого луча и вектор к наблюдателю. Учёт этой составляющей представляет собой работу с моделью Фонга — комбинацию модели Ламберта и зеркальной составляющей. Так, кроме равномерного освещения на материале при определённых условиях будет появляться блик. Расположение блика на объекте определяется из закона равенства углов падения и отражения. Падающий луч, отраженный луч и нормаль к отражающей поверхности в точке падения лежат в одной плоскости. Нормаль делит угол между лучами на две равные части. Таким образом, отраженная составляющая освещенности в данной точке зависит от величины угла между направлением на наблюдателя и отраженным лучом. Эту зависимость можно выразить следующей формулой:

$$I_s = k_s \cos^{\alpha}(R, V) i_s = k_s (R \cdot V)^{\alpha} i_s, \tag{1.15}$$

где I_s — зеркальная составляющая освещенности в точке,k_s — коэффициент зеркального отражения,R — направление отраженного луча,V — направление на наблюдателя,i_s — мощность зеркального освещения, α — коэффициент блеска, свойство материала;

— $refract$ — составляющая интенсивности от преломлённого луча.

Обобщая все эти составляющие и учитывая наличие на сцене других тел, можно записать формулу для глобальной модели освещения:

$$I = k_a I_a + k_d \sum_j I_j (N \cdot L_j) + k_s \sum_j I_j (V \cdot R_j)^{\alpha} + k_s I_s + k_r I_r, \tag{1.16}$$

где

— k_a – коэффициент фонового освещения;

— k_d – коэффициент диффузного отражения;

— k_s – коэффициент зеркального отражения;

— k_r – коэффициент пропускания;

— N – единичный вектор нормали к поверхности в данной точке;

— L_j – единичный вектор, направленный к j-му источнику света;

— V – единичный вектор, направленный на наблюдателя;

— R_j – вектор направления отражённого луча L_j;

— I_a – интенсивность фонового освещения;

— I_j – интенсивность j-го источника света;

— I_s – интенсивность от зеркально отражённого луча;

— I_r – интенсивность от преломлённого луча.

В глобальной модели освещения трассировка луча не прекращается при первом найденном пересечении с каким-либо объектом: необходима дальнейшая трассировка отражённого и преломлённого лучей, ход которых рассчитывается по законам геометрической оптики. Процесс продолжается до тех пор, пока очередные лучи не перестанут пересекаться с какими-либо объектами сцены. Такой процесс может стать бесконечным, если задана сложная сцена, поэтому необходимо задать максимальную глубину рекурсии для трассировки.

1.5. Физические основы синтеза изображения мыльного пузыря

В данной работе необходимо получить реалистичное изображение мыльного пузыря. На поверхности мыльного пузыря обычно появляются радужные пятна. Это связано с явлением интерференции света.

Пузырь представляет из себя два мыльных слоя, между которыми заключён слой воды, а внутри находится воздух. Тогда при попадании луча света на поверхность пузыря часть света отражается от первого мыльного слоя, а часть проходит внутрь плёнки, преломляясь, и впоследствии вновь отражаясь от внутреннего мыльного слоя, и снова выходит наружу. При каждом проходе световая волна смещается на определенное значение, пропорциональное длине волны и зависящее от толщины мыльной пленки. Этот процесс показан на рисунке 1.11.

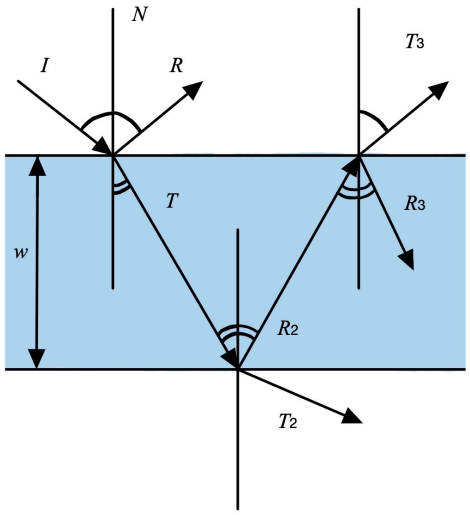

Рисунок 1.11 — Преломление и отражение света в мыльной пленке

Важным свойством мыльной плёнки является то, что вновь вылетевший из плёнки луч T_3 параллелен отражённому лучу R. Таким образом, свет, движущийся по R, будет взаимодействовать со светом на T_3. Так как свет — это волна, его итоговая интенсивность будет зависеть от длины волны и фазы каждой составляющей волны. Пусть свет, поступающий вдоль I имеет длину волны λ. Пленка имеет толщину w, а показатель преломления пленки равен η. Угол падения между I и нормалью N к поверхности равен θ_i, а угол преломления — θ_t. Свет по лучу T_3 будет сдвинут по фазе относительно света по лучу R, а разность фаз будет определять, усиливают или гасят эти волны друг друга. Чтобы найти разность фаз, нужно сначала найти оптическую разность хода. Теперь будет рассматриваться рисунок 1.12.

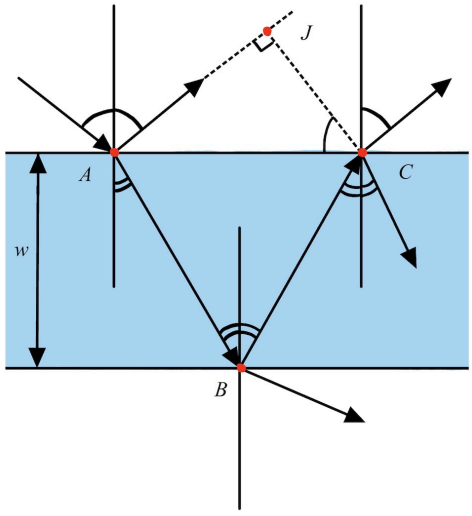

Рисунок 1.12 — Разность фаз волн света в мыльной пленке

Геометрическая разница в расстоянии составляет $AB + BC - AJ$. Свет всегда распространяется медленнее в более плотной среде, поэтому расстояние, пройденное светом в мыльной плёнке, нужно умножить на показатель преломления мыльной воды. Таким образом, эта часть разности хода составляет

$$\Delta = \eta(AB + BC) - AJ. \tag{1.17}$$

По рисунку видно, что:

$$\cos(\theta_t) = \frac{w}{AB}, \tag{1.18}$$

$$AB = BC. \tag{1.19}$$

Тогда

$$AB = BC = \frac{w}{\cos(\theta_t)}. \tag{1.20}$$

После подставления результата в формулу (1.17) получается

$$\Delta = 2 \cdot \eta \cdot \frac{w}{\cos(\theta_t)} - AJ. \tag{1.21}$$

Так как $\angle ACJ = \theta_i$, то $AJ = AC \cdot \sin(\theta_i)$. По закону Снеллиуса $AJ = AC \cdot \eta \cdot \sin(\theta_t)$. Также видно, что $\tan(\theta_t) = \frac{\frac{AC}{2}}{w}$, поэтому $AC = 2 \cdot w \; cdot \tan(\theta_t)$.

После подставления результата в формулу (1.17) получается

$$\Delta = 2 \cdot \eta \cdot \frac{w}{\cos(\theta_t)} - 2 \cdot w \cdot \tan(\theta_t) \cdot \eta \cdot \sin(\theta_t). \tag{1.22}$$

Если упростить данное выражение, то получится

$$\Delta = 2 \cdot w \cdot \eta \cdot \cos(\theta_t). \tag{1.23}$$

Также известно, что, когда свет переходит из одной среды в среду с более высоким показателем преломления, он претерпевает фазовый сдвиг, равный половине длины его волны, то есть $\frac{\lambda}{2}$.

Тогда итоговая разность хода

$$\Delta_{res} = 2 \cdot w \cdot \eta \cdot \cos(\theta_t) - \frac{\lambda}{2}. \tag{1.24}$$

Если в разность хода укладывается чётное число полуволн, то в точке падения луча будет максимум интенсивности света. Если в разность хода укладывается нечётное число полуволн, то в точке падения луча будет минимум интенсивности света.

По данной формуле рассчитывается разность фаз:

$$\delta = \frac{2 \cdot \pi \cdot \Delta_{res}}{\lambda}. \tag{1.25}$$

Для волн с одинаковой интенсивностью итоговая интенсивность в результате интерференции будет рассчитываться по формуле

$$I = 2 \cdot I_0 \cdot (1 + \cos(\delta)). \tag{1.26}$$

1.6. Выводы по аналитической части

В данном разделе были проведены анализ и формализация объектов синтезируемой сцены, исследование задачи построения реалистичного трёхмерного изображения сцены из данных объектов, и рассмотрены различные методы, решающие перечисленные задачи.

2. Конструкторская часть

В данном разделе будут представлены математические основы для реализации алгоритма обратной трассировки лучей, схемы данного алгоритма и необходимых подпрограмм и обоснование используемых типов и структур данных.

2.1. Математические основы для реализации алгоритма обратной трассировки лучей

2.1.1. Поиск пересечения луча со сферой

Использован геометрический подход. Основные обозначения представлены для наглядности на рис. 2.1.

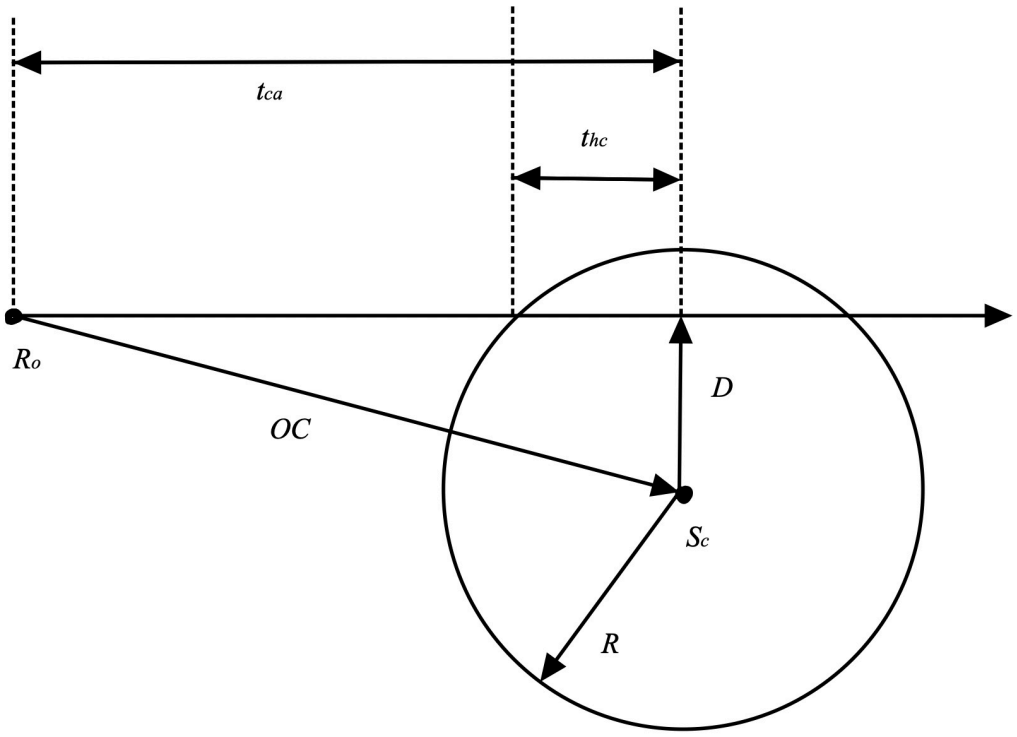

Рисунок 2.1 — Геометрический подход при поиске пересечения луча со сферой

Нужно произвести следующие действия (все обозначения соответствуют представленным на рис. 2.1, индекс i используется для значений, связанных с точкой пересечения луча и сферы, c — с точкой центра сферы):

1) Проверка, лежит ли начало луча внутри сферы. Для этого вычисляется длина вектора OC: $OC = S_c - R_0$. Если $OC < R$, то начало луча лежит внутри сферы.

2) Нахождение ближайшей к центру сферы точки луча, расстояние до которой от центра обозначается как D. Расстояние до этой точки от начала луча обозначается как t_{ca}. Тогда $t_{ca} = OC \cdot dir$ (равно проекции OC на направление луча), где dir — это директриса луча.
   Теперь если $t_{ca} < 0$ и R_0 вне сферы, то луч не пересекает данную сферу. Иначе выполняются следующие действия.

3) Нахождение расстояния от ближайшей к центру точки луча до точки пересечения со сферой. Вычисляется $D^2 = OC^2 - t_{ca}^2$ по теореме Пифагора. Тогда $t_{hc}^2 = R^2 - D^2 = R^2 - (OC^2 - t_{ca}^2)$ также по теореме Пифагора.
   Теперь если $t_{hc}^2 < 0$, то луч не пересекает данную сферу. Иначе выполняются следующие действия.

4) Вычисление расстояния от начала луча до точки пересечения. Если R_0 внутри сферы, то используется формула $t = t_{ca} + t_{hc}$, иначе используется формула $t = t_{ca} - t_{hc}$.

5) Вычисление координат точки пересечения луча со сферой. Используется формула $R(t) = R_0 + R_d t$ (в параметрической форме) или $(x_i, y_i, z_i) = (x_0 + x_d t, y_0 + y_d t, z_0 + z_d t)$ (в координатах).

Также, удобно сразу вычислить нормаль к точке пересечения при помощи формулы:

$$n = \left(\frac{x_i - x_c}{R}, \quad \frac{y_i - y_c}{R}, \quad \frac{z_i - z_c}{R} \right)^T. \tag{2.1}$$

2.1.2. Поиск пересечения луча с многогранником

Для поиска пересечения луча с многогранником необходимо осуществить поиск пересечения луча с каждой его гранью (многоугольником). Данная задача рассмотрена ниже.

Однако, есть ситуации, когда можно сразу понять, пересекается ли многогранник заданным лучом, выполнив менее трудозатратные вычисления. Для этого перед проверкой всех полигонов многогранника проверяется пересечение лучом сферической оболочки, в которую вписан многогранник, по вышеописанному алгоритму. На рис. 2.2 изображена сферическая оболочка для четырёхугольной пирамиды.

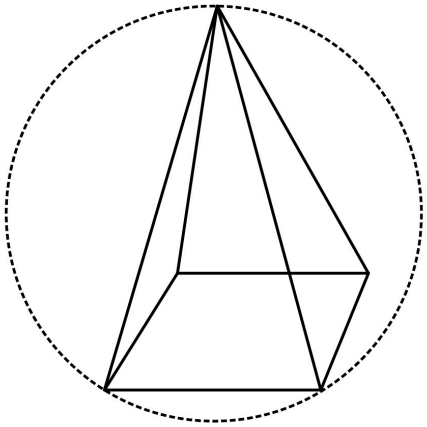

Рисунок 2.2 — Сферическая оболочка для четырёхугольной пирамиды

2.1.3. Поиск пересечения луча с полигоном (многоугольником)

Любой луч представим параметрическим уравнением вида $R(t) = R_0 + R_d t$. Любая плоскость представима в виде $ax + by + cz + d = 0$. Для данной плоскости:

— $a^2 + b^2 + c^2 = 1$;

— $n = \begin{pmatrix} a, & b, & c \end{pmatrix}^T$;

— расстояние от данной плоскости до точки $(0, 0, 0)$ равняется $-d$.

Уравнение луча можно представить в виде плоскости и разрешить относительно параметра t:

$$t = \frac{-(ax_0 + by_0 + cz_0 + d)}{ax_d + by_d + cz_d}. \tag{2.2}$$

Тогда если $ax_d + by_d + cz_d = 0$, то луч параллелен плоскости, и пересечения точно нет. Если $t < 0$, то пересечение происходит с прямой, на которой лежит луч, а не с самим лучом. Иначе, существует пересечение луча с заданной плоскостью в точке $(x_i, y_i, z_i) = (x_0 + x_d t, y_0 + y_d t, z_0 + z_d t)$.

Теперь необходимо проверить, принадлежит ли точка пересечения с плоскостью многоугольнику. Для этого применяется самый распространённый тест: проверяются последовательно знаки векторных произведений каждой стороны полигона на вектор, соединяющий начало текущей стороны и точку пересечения. Если какое-либо векторное произведение нулевое, значит точка лежит на стороне полигона. Если знаки всех векторных произведений совпадают, то точка принадлежит полигону, иначе она находится вне полигона.

На рис. 2.3 изображены возможные ситуации при данной проверке.

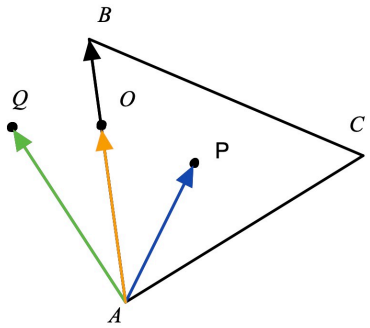

Рисунок 2.3 — Возможные ситуации при проверке принадлежности точки полигону

Векторные произведения $AQ \times AB$ и $BQ \times BC$, например, будут иметь разные знаки, так как точка Q лежит вне полигона. Векторные произведения $AP \times AB$ и $BP \times BC$ будут иметь одинаковые знаки (как и $AP \times AC$), так как точка P лежит внутри полигона. При рассмторении точки O одно из векторных произведений будет нулевым, так как она лежит на стороне полигона.

2.1.4. Поиск коэффициентов уравнения плоскости полигона и вектора нормали

Если известны 3 точки, принадлежащие плоскости, то можно найти коэффициенты уравнения плоскости, приведя следующую запись к виду уравнения плоскости:

$$\begin{vmatrix} x - x_0 & x_1 - x_0 & x_2 - x_0 \\ y - y_0 & y_1 - y_0 & y_2 - y_0 \\ z - z_0 & z_1 - z_0 & z_2 - z_0 \end{vmatrix} = 0. \tag{2.3}$$

Будут получены следующие уравнения для коэффициентов:

$$A = (y_1 - y_0)(z_2 - z_0) - (y_2 - y_0)(z_1 - z_0), \tag{2.4}$$

$$B = (x_1 - x_0)(z_2 - z_0) - (x_2 - x_0)(z_1 - z_0), \tag{2.5}$$

$$C = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0), \tag{2.6}$$

$$D = -Ax_0 + By_0 - Cz_0. \tag{2.7}$$

Далее следует подставить в уравнение с полученными коэффициентами точку, координаты которой соответствуют усреднённому значению координат всех вершин данного полигона. Если значение положительное, то коэффициенты следует домножить на -1, так как осуществляется поиск внешней нормали. Формально используются координаты векторов, составляющих стороны полигона. Иллюстрация представлена на рис. 2.4.

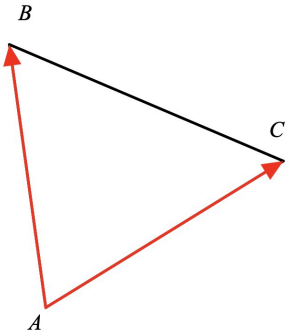

Рисунок 2.4 — Выбор векторов для поиска вектора нормали к полигону

Тогда вектор нормали к данному полигону можно представить как

$$n = \begin{pmatrix} A, & B, & C \end{pmatrix}^T, \tag{2.8}$$

где A, B, C — коэффициенты уравнения плоскости заданного полигона.

2.1.5. Определение направлений отражённого и преломлённого лучей

Для алгоритма обратной трассировки лучей необходимо вычислять направления отражённого и преломлённого лучей. Пусть A — направление падающего луча, B — направление отражённого луча, C — направление преломлённого луча, а N — нормаль к поверхности. Луч A можно разбить на два слагаемых: A_p — вектор, перпендикулярный нормали, и A_n — вектор, параллельный нормали. На рис. 2.5 изображены направления данных лучей и приведены используемые ниже обозначения, причём все вектора нормализованы.

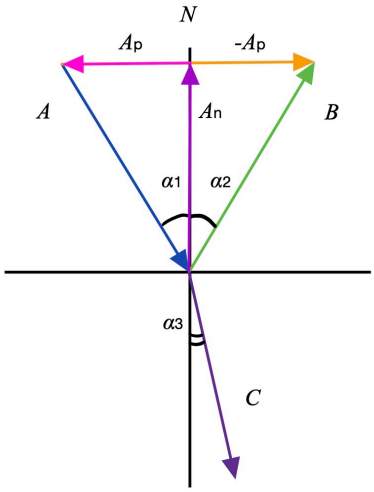

Рисунок 2.5 — Направления отражённого и преломлённого лучей

По свойству скалярного произведения $A_n = N(N \cdot (-A))$ и $A_p = -A - A_n = -A + N(N \cdot A)$. Отражённый луч можно выразить через разность этих векторов $B = A - 2N(N \cdot A)$.

Падающий, преломлённый луч и нормаль к поверхности лежат в одной плоскости. Пусть μ_i – показатели преломления сред, причём $i = \overline{1,2}$. Применяя закон Снеллиуса, параметры преломлённого луча можно вычислить по формуле

$$C = \frac{\mu_1}{\mu_2} A + \left(\frac{\mu_1}{\mu_2} \cos(\alpha_1) - \cos(\alpha_3)\right) N, \tag{2.9}$$

где $cos(\alpha_3) = \sqrt{1 - (\frac{\mu_1}{\mu_2})^2 (1 - cos(\alpha_1))^2}$.

2.2. Разработка алгоритма обратной трассировки лучей

На рисунке 2.6 представлена схема алгоритма обратной трассировки лучей.

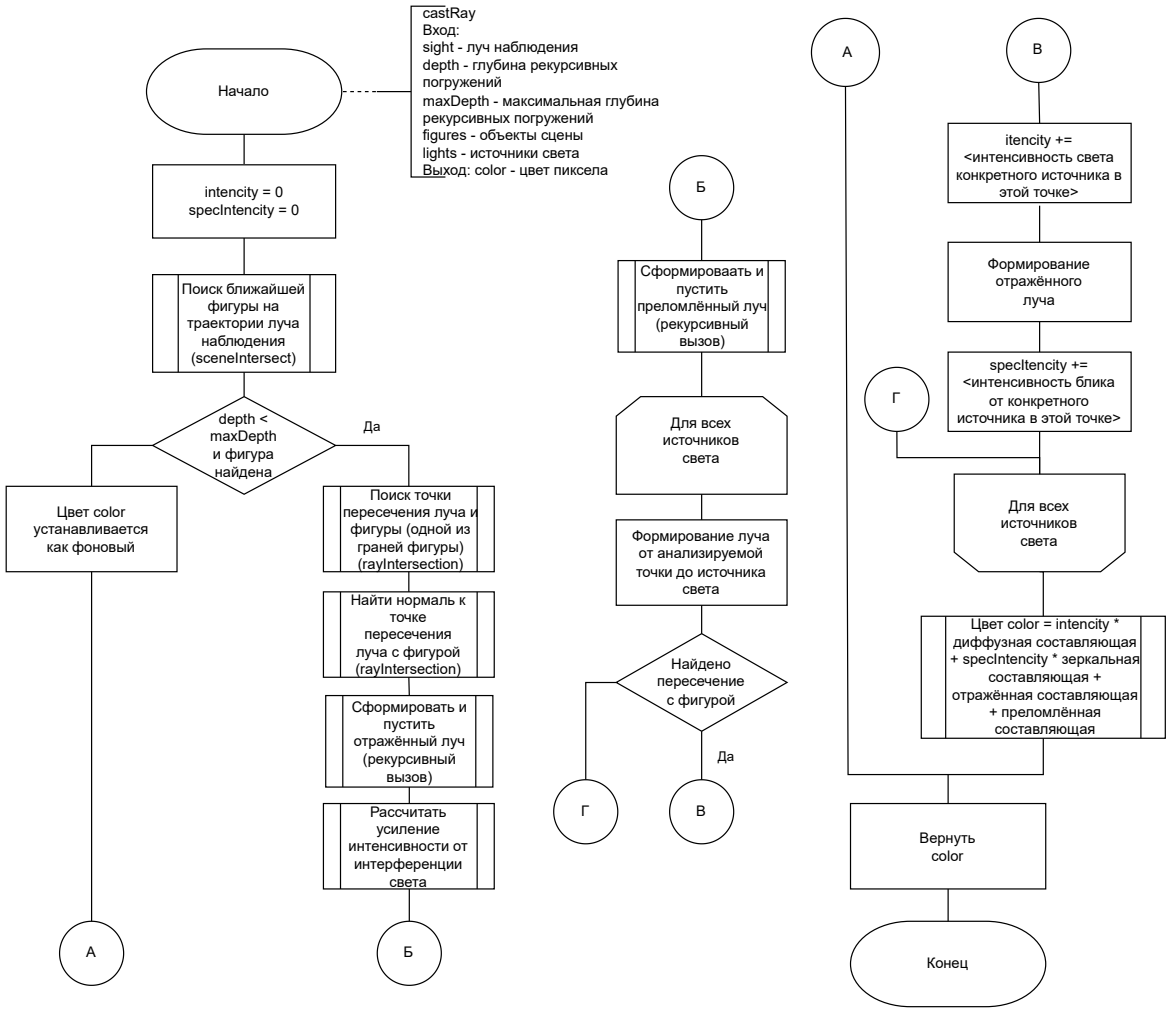

Рисунок 2.6 — Схема алгоритма обратной трассировки лучей

На рисунке 2.7 представлена схема подпрограммы поиска ближайшего пересечения луча с объектом сцены.

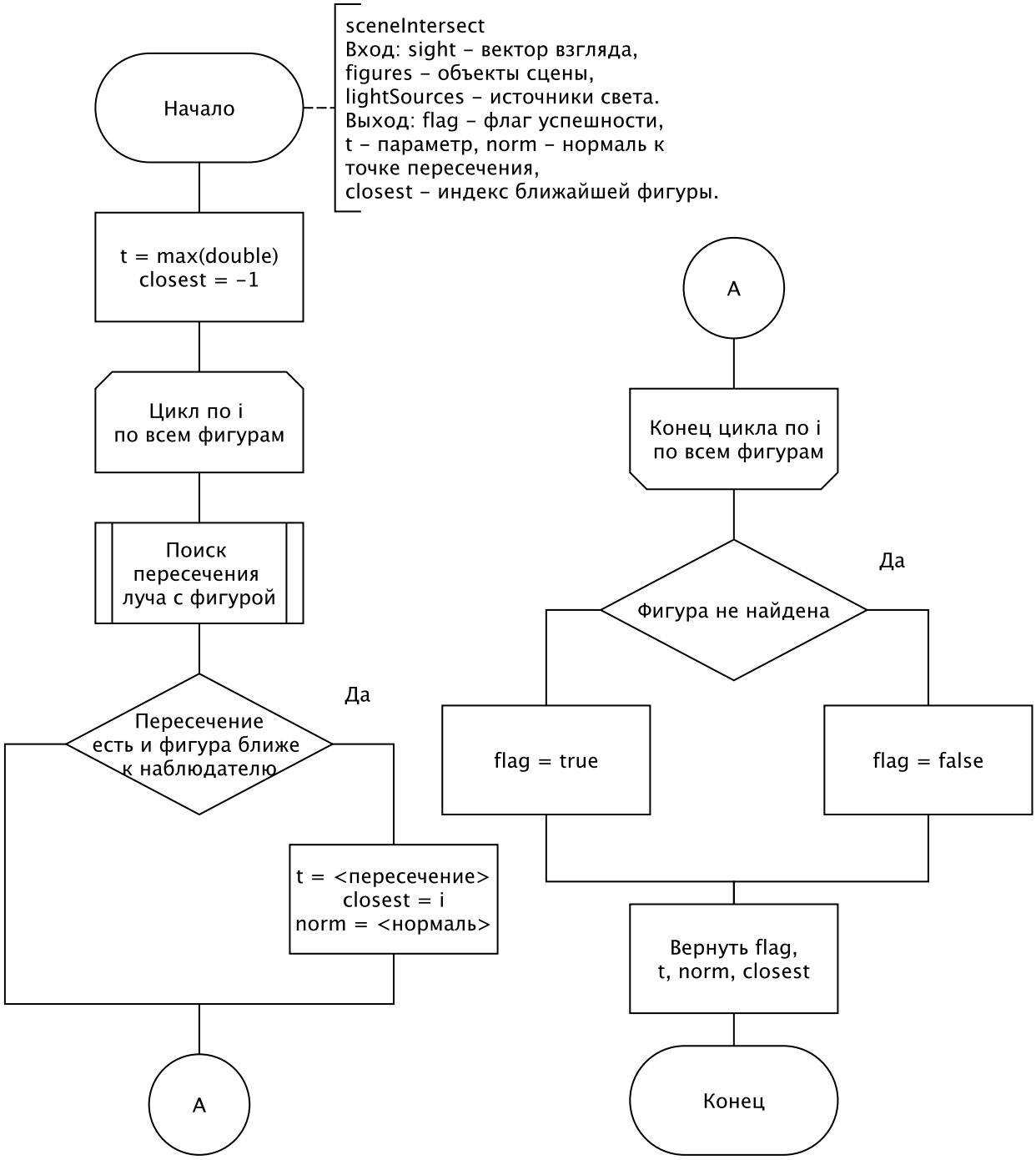

Рисунок 2.7 — Схема подпрограммы поиска ближайшего пересечения луча с объектом сцены

На рисунке 2.8 представлена схема подпрограммы поиска точки пересечения луча с многогранником. На рисунке 2.9 представлена схема подпрограммы поиска точки пересечения луча с треугольным полигоном.

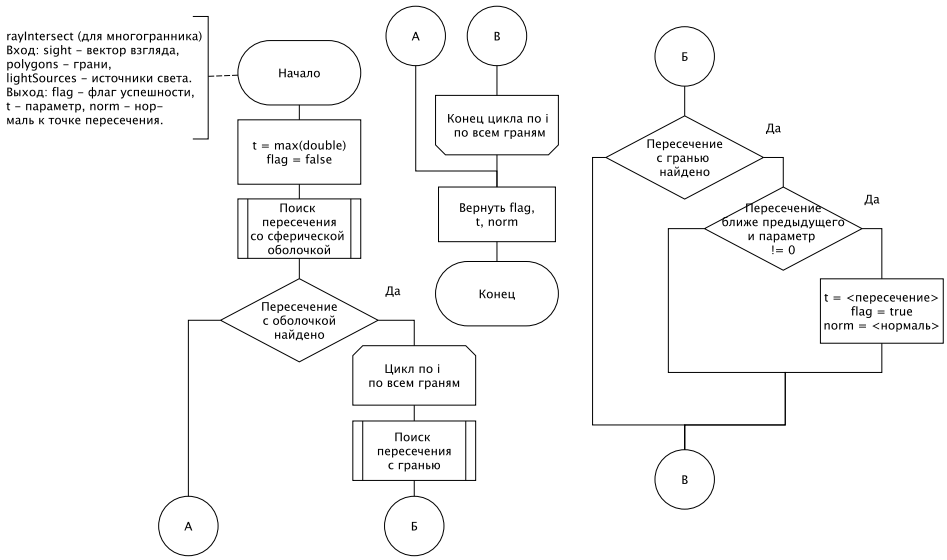

Рисунок 2.8 — Схема подпрограммы поиска точки пересечения луча с многогранником

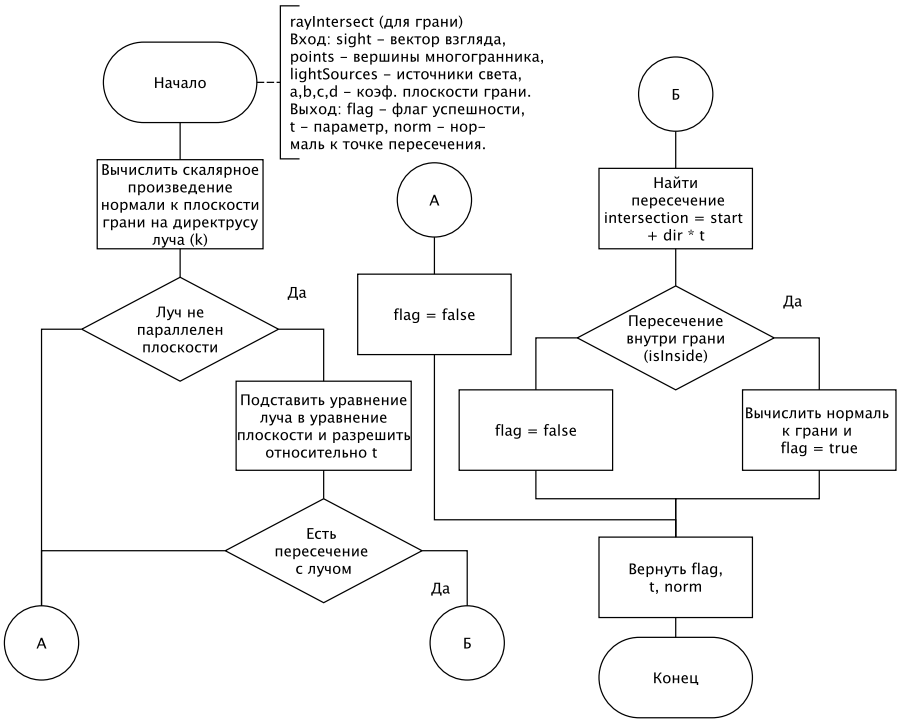

Рисунок 2.9 — Схема подпрограммы поиска точки пересечения луча с треугольным полигоном

На рисунке 2.10 представлена схема подпрограммы поиска точки пересечения луча со сферой.

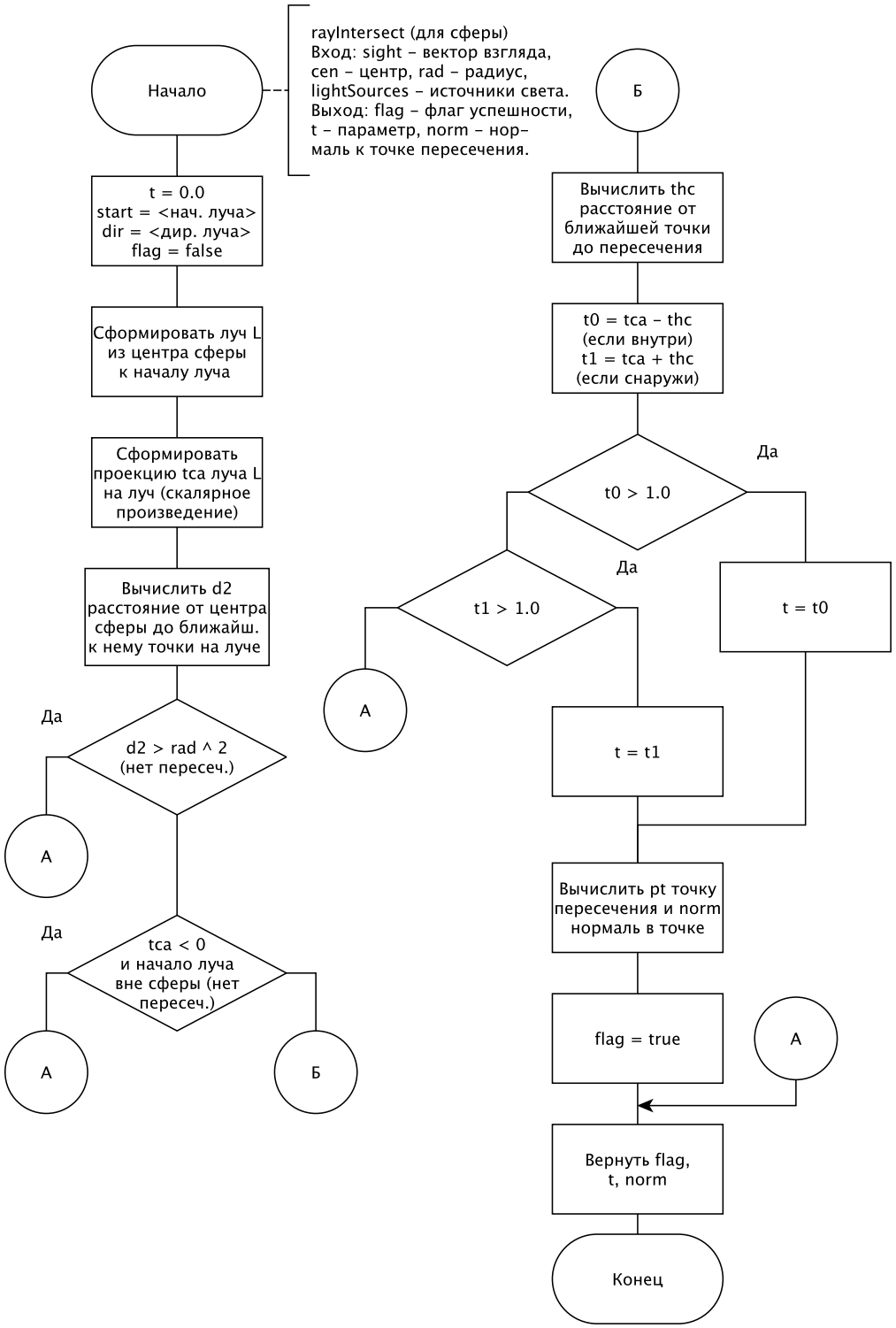

Рисунок 2.10 — Схема подпрограммы поиска точки пересечения луча со сферой

На рисунке 2.11 представлена схема подпрограммы проверки принадлежности точки грани.

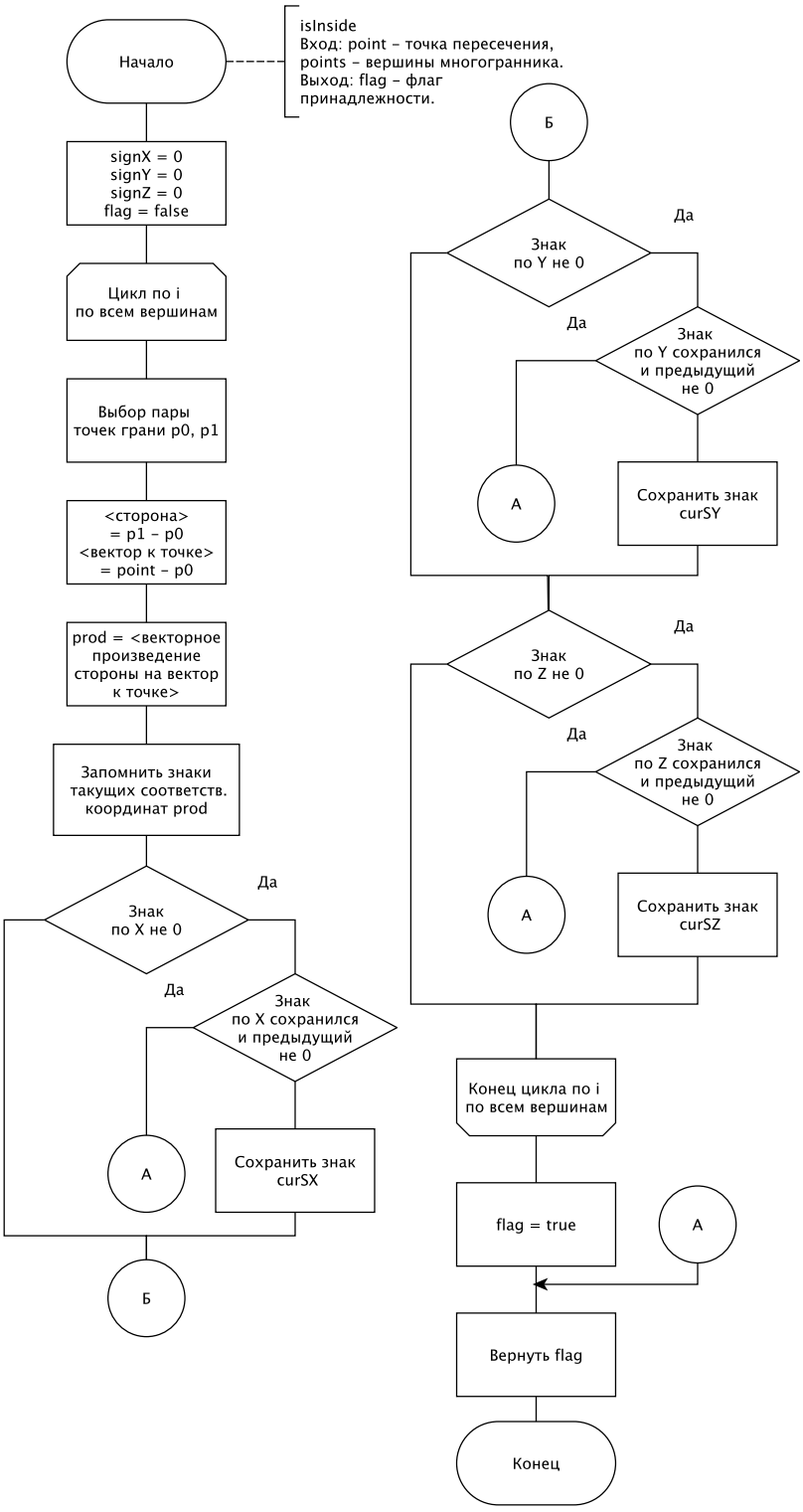

Рисунок 2.11 — Схема подпрограммы проверки принадлежности точки грани

2.3. Обоснование используемых типов и структур данных

При реализации алгоритма обратной трассировки лучей для синтеза реалистичного изображения были использованы следующие типы и структуры данных:

1) Сцена (*drawing*) представляет из себя структуру со следующими полями:

   — массив объектов сцены;

   — камера (наблюдатель);

   — массив источников света на сцене.

   Объекты сцены хранятся в массивах, для обеспечения доступа к отдельному объекту со сложностью $O(1)$ с помощью индексации и работы с итераторами. Объекты разного типа хранятся в разных массивах во избежание усложнённого выбора элементов.

2) Объект-фигура (*figure*) представляет из себя структуру, содержащую поле с информацией о свойствах материала объекта типа *material*, описанного ниже.

3) Многогранник (*polyhedron*) представляет из себя структуру со следующими полями:

   — массив вершин;

   — массив связей вершин (полигонов);

   — сферическую оболочку типа *sphere*, описанного ниже.

   Элементы многогранника хранятся в массивах, для обеспечения доступа к отдельному объекту со сложностью $O(1)$ с помощью индексации и работы с итераторами. Сферическая оболочка необходима для ускорения проверки на пересечение многогранника лучом.

4) Полигон (*polygon*) представляет из себя структуру со следующими полями: вектор вершин и набор коэффициентов задающей грань плоскости (для точности вычислений используется числовой тип двойной точности). Элементы полигона хранятся в массивах, для обеспечения доступа к отдельному объекту со сложностью $O(1)$ с помощью индексации и работы с итераторами.

5) Сфера (*sphere*) представляет из себя структуру, содержащую поля с информацией о векторе радиуса и точке центра (координаты типа вектор). Это позволяет задавать сферу аналитическим образом для работы с программой.

6) Источник света (*lightSource*) представляет из себя структуру, содержащую информацию о расположении источника в пространстве (координаты типа вектор) и числовое значение его интенсивности (для точности вычисления цвета используется числовой тип двойной точности). Структура источника света, если расположен в бесконечности, также хранит информацию о направлении распространения света (координаты типа вектор). Предусмотрен только белый цвет излучения.

7) Камера (*camera*) представляет из себя структуру, содержащую поля с информацией о расположении наблюдателя в пространстве и направлении вектора его взгляда (координаты типа вектор).

8) Материал (*material*) представляет из себя структуру со следующими полями:

   — цвет фонового освещения (три целочисленных переменных в модели RGB);
   — цвет диффузного освещения (три целочисленных переменных в модели RGB);
   — цвет зеркального освещения (три целочисленных переменных в модели RGB);
   — коэффициент фонового освещения (вещественная переменная двойной точности);
   — коэффициент диффузного освещения (вещественная переменная двойной точности);
   — коэффициент зеркального освещения (вещественная переменная двойной точности);
   — степень, аппроксимирующая пространственное распределение зеркально отражённого света (целочисленная переменная);
   — коэффициент отражения (вещественная переменная двойной точности);
   — коэффициент преломления (вещественная переменная двойной точности);
   — показатель преломления (вещественная переменная двойной точности).

   Все данные, влияющие на цвет закраски пикселов изображения, хранятся с использованием типов с двойной точностью как наиболее важные для решения поставленных задач, требующих повышенную точность вычислений.

2.4. Выводы по конструкторской части

В данном разделе были представлены математические основы для реализации алгоритма обратной трассировки лучей, схемы данного алгоритма и необходимых подпрограмм и обоснование используемых типов и структур данных.

3. Технологическая часть

В данном разделе будет представлена реализация решения поставленной задачи в в формате листинга кода. Также будут указаны требования к ПО и средства реализации, описан пользовательский интерфейс и результаты проведённого тестирования программы.

3.1. Требования к ПО

Были учтены следующие требования к программе:

1) Программа выводит сообщение об ошибке и генерирует ненулевой код возврата в случае возникновения исключения в процессе выполнения.

2) Программа выполняет построение изображения сцены из трёхмерных объектов, которые могут представлять из себя многогранники, сферы, или комбинацию из двух объектов одного типа — пузырь.

3) Программа выполняет обработку только многогранников, геометрия которых описана с помощью *obj* файла, и только сферы, геометрия которых описана аналитическим способом (с помощью указания основных параметров канонического уравнения сферы).

4) Программа принимает на вход файлы, составленные по правилам, которые описаны ниже.

5) Программа может сохранять полученное изображение в *png* файл.

6) Программа предоставляет возможность замера времени выполнения при вызове из командной строки.

7) Программа предоставляет возможность проведения модульного тестирования при вызове из командной строки.

8) Программа предоставляет возможность изменять положение как отдельного объекта сцены, так и всех объектов сцены одновременно, а именно перемещать объекты, масштабировать их и вращать относительно собственного центра;

9) Программа предоставляет возможность указать, какое количество потоков должно быть выделено для обработки сцены, и какой будет глубина рекурсивных погружений при работе алгоритма обратной трассировки лучей.

На рисунках 3.1 – 3.2 представлена функциональная модель программы в нотации $IDEF0$.

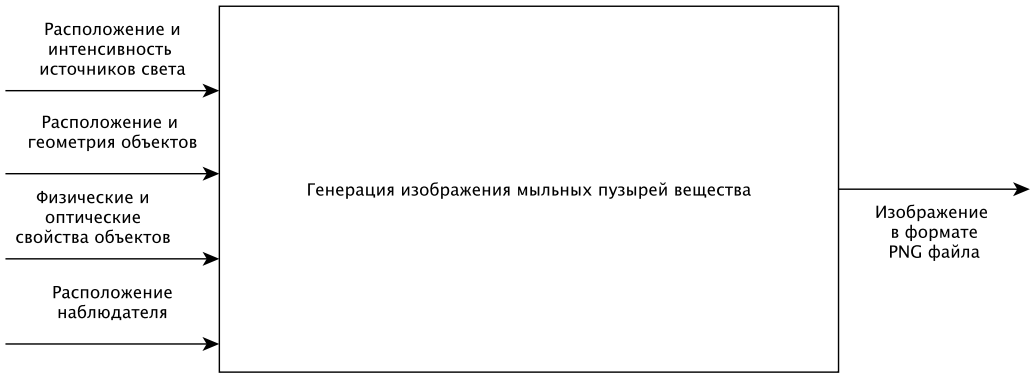

Рисунок 3.1 — Функциональная модель программы (начало)

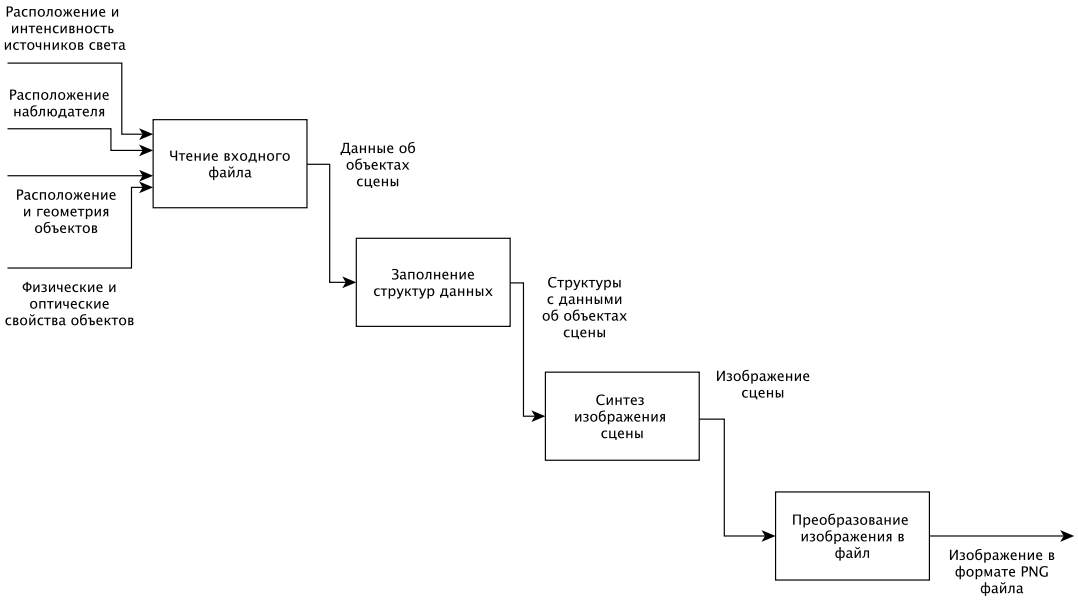

Рисунок 3.2 — Функциональная модель программы (продолжение рисунка 3.1)

3.2. Требования к входным данным

Требования ко входному файлу:

1) На вход принимаются только *txt* файлы.

2) Для описания сферы, заданной аналитически, перед указанием непосредственно параметров фигуры, указывается параметр A.

3) Для описания многогранника, заданного при помощи полигональной сетки, перед указанием имени файла с параметрами фигуры, указывается параметр O.

4) Для описания пузыря указывается параметр B, после него следует один из вышеуказанных параметров в зависимости от способа описания слоёв.

5) Для описания источника света указывается параметр L, после которого следует информация об источнике света.

Параметры фигур:

1) Для сферы, заданной аналитически, необходимо указать координаты точки центра, радиус, параметры материала — коэффициенты рассеянного, диффузного и зеркального отражения, коэффициенты пропускания и преломления, оптическая плотность, диффузный цвет и цвет бликов.

2) Для многогранника, заданного при помощи полигональной сетки, необходимо указать имя *obj* файла, в котором хранится информация о данном многограннике. В одном файле может хранится информация о нескольких многогранниках.

3) Для пузыря необходимо привести описания двух его слоёв в соответствии с указанными выше правилами. Пузырь может состоять только из двух слоёв одного типа.

3.3. Реализация управления из командной строки

Для автоматизации работы программы необходимо реализовать передачу в программу аргументов командной строки и их распознавание самой программой. Для этого была использована функция *getopt* [11], которая последовательно перебирает переданные параметры в программу в соответствии с заданной строкой параметров, содержащей имена параметров и признак наличия передаваемого значения (символ двоеточие).

Были предусмотрены следующие флаги:

— флаг «$-d$» определяет значение глубины рекурсивных погружений, за именем флага следует положительное целочисленное значение;

— флаги «$-i$», «$-o$» определяют имена входного и выходного файлов соответственно, за именем флага следует строка с именем файла;

— флаг «$-n$» определяет количество выделяемых потоков, за именем флага следует положительное целочисленное значение;

— флаг «$-r$» определяет угол поворота сцены вокруг выбранной оси, за именем флага следует целочисленное значение;

— флаг «$-m$» — указание программе о том, что проводятся замеры времени работы, при этом обязательно наличие параметров «$-d$» и «$-i$», а «$-o$» становится необязательным, так как полученное изображение в данном случае не используется;

— флаг «$-t$» — указание программе о том, что проводится модульное тестирование, при этом все остальные параметры необязательны.

При стандартном запуске программы, когда важно полученное изображение, передача параметров «$-d$», «$-i$» и «$-o$» является обязательной, в случае отсутствия обязательных параметров программой генерируется ненулевой код возврата.

3.4. Средства реализации

В качестве языка программирования для разработки ПО был выбран язык программирования $C++$ [2]. Данный выбор обусловлен наличием необходимого функционала для работы с наборами объектов и проведения исследования по замеру времени выполнения. Также был выбран фреймворк Qt [1] в связи с наличием необходимых для работы с компьютерной графикой библиотек и встроенных средств. Для передачи сведений об объектах сцены в программу использовались файлы с расширением txt, как имеющие наибольшую гибкость для формализации ввода.

Для хранения информации о многогранниках, заданных при помощи полигональной сетки, были использованы obj файлы. Данный выбор обусловлен тем, что obj является одним из самых популярных форматов передачи трёхмерной компьютерной геометрии [12], позволяет хранить информацию о текстурах в отдельном mtl файле в отличие от некоторых аналогов, например, stl файлов, и имеет широкую поддержку экспорта и импорта программного обеспечения САПР. Для хранения полученного изображения был выбран формат png, так как среднеквадратическая ошибка — среднее различие в квадрате между

идеальным и фактическим пиксельными значениями — для, например, *jpeg* файлов на больших изображениях гораздо больше, чем у *png* файлов [13], и качество изображения формата *png* не меняется при любой степени сжатия.

3.5. Реализация разработанного ПО

В листингах 3.1 – 3.4 представлена реализация алгоритма обратной трассировки лучей с учётом интерференции света, в листинге 3.5 представлена реализация поиска первого пересечения луча с объектом сцены, в листинге 3.6 представлена реализация поиска пересечения луча с многогранником, в листинге 3.7 представлена реализация поиска пересечения луча с многоугольником, в листинге 3.8 представлена реализация поиска пересечения луча со сферой, в листингах 3.9 – 3.10 представлена реализация проверки принадлежности точки многоугольнику.

Листинг 3.1 — Листинг функции, реализующей алгоритм обратной трассировки лучей с учётом интерференции света (начало)

```
QColor Drawing::castRay(const sight_t& sight, const int& depth)
{
    double intensity = 0.0;
    double specIntensity = 0.0;

    std::tuple<bool, double, QVector3D, int> res = sceneIntersect(sight);
    if (depth > this->depth)
        return Qt::black;
    if (!std::get<0>(res))
            return bgc;

    double t = std::get<1>(res);
    QVector3D norm = std::get<2>(res);
    int closest = std::get<3>(res);
    QVector3D cam = sight.start;
    QVector3D dir = sight.dir;
    QVector3D intersection = cam + dir * t;
    double reflectK = 0.0;
```

Листинг 3.2 — Листинг функции, реализующей алгоритм обратной трассировки лучей с учётом интерференции света (продолжение листинга 3.1)

```
    QColor reflectCol = Qt::black;
    if (!qFuzzyIsNull(figures[closest]->getMaterial().getAlbedo().z())) {
        QVector3D reflectDir = reflect(dir, norm).normalized();
        sight_t sightRefl = {
            .start = intersection,
            .dir = reflectDir,
            .color = sight.color,
            .len = sight.len,
            .n = sight.n,
        };
        reflectCol = castRay(sightRefl, depth + 1);
        double rC = getInterferenceColorElem(sight.color,
        figures[closest]->getMaterial());

        reflectK = getInterferenceStrengthCoef(figures[closest]->getT(),
        rC, dir, norm, sight.len);
    }
    QColor refractCol = Qt::black;
    if (!qFuzzyIsNull(figures[closest]->getMaterial().getAlbedo().w())) {
        double rC = getInterferenceColorElem(sight.color,
        figures[closest]->getMaterial());

        QVector3D refractDir = refract(dir, norm, rC, sight.n).normalized();
        sight_t sightRefr = {
            .start = intersection,
            .dir = refractDir,
            .color = sight.color,
            .len = sight.len,
            .n = rC,
        };
        refractCol = castRay(sightRefr, depth + 1);
    }
```

Листинг 3.3 — Листинг функции, реализующей алгоритм обратной трассировки лучей с учётом интерференции света (продолжение листинга 3.2)

```
    for (size_t k = 0; k < lightSources.size(); k++) {
        QVector3D light = intersection - lightSources[k].getPos();
        light = light.normalized();
        double t1 = 0.0;
        bool flag = false;
        sight_t sightLight = {
            .start = lightSources[k].getPos(),
            .dir = light
        };

        std::tuple<bool, double, QVector3D> res1 =
        figures[closest]->rayIntersection(sightLight, lightSources);

        t1 = std::get<1>(res1);
        std::tuple<bool, double, QVector3D, int> res2 =
        sceneIntersect(sightLight);
        if (std::get<1>(res2) < t1)
            flag = true;

        if (!flag) {
            intensity += lightSources[k].getIntensity() * std::max(0.f,
            QVector3D::dotProduct(norm, (-1) * light));

            QVector3D mirrored = reflect(light, norm).normalized();
            double angle = QVector3D::dotProduct(mirrored, dir * (-1));
            specIntensity += powf(std::max(0.0, angle),
            figures[closest]->getMaterial().getSpecCoef()) *
            lightSources[k].getIntensity();
        }
    }
    QColor diffColor = figures[closest]->getMaterial().getDiffColor();
    QColor specColor = figures[closest]->getMaterial().getSpecColor();
}
```

Листинг 3.4 — Листинг функции, реализующей алгоритм обратной трассировки лучей с учётом интерференции света (окончание листинга 3.3)

```
    QColor color = getColor(intensity, specIntensity,
    figures[closest]->getMaterial().getAlbedo(), diffColor, specColor,
    reflectCol, refractCol, reflectK, depth, this->depth);

    return color;
}
```

Листинг 3.5 — Листинг функции, выполняющей поиск первого пересечения луча с объектом сцены

```
std::tuple<bool, double, QVector3D, int> Drawing::sceneIntersect(const
sight_t& sight)
{
    double t = std::numeric_limits<double>::max();
    int closest = -1;
    QVector3D norm;

    for (size_t i = 0; i < figures.size(); i++) {
        std::tuple<bool, double, QVector3D> res =
        figures[i]->rayIntersection(sight, lightSources);
        double resT = std::get<1>(res);
        if (std::get<0>(res) && resT < t) {
            t = resT;
            closest = i;
            norm = std::get<2>(res);
        }
    }

    if (closest == -1)
        return std::tuple<bool, double, QVector3D, int>(false, 0,
        QVector3D(0, 0, 0), 0);

    return std::tuple<bool, double, QVector3D, int>(true, t, norm, closest);
}
```

Листинг 3.6 — Листинг функции, выполняющей поиск точки пересечения луча с многогранником

```
std::tuple<bool, double, QVector3D> Polyhedron::rayIntersection(const
sight_t& sight, const std::vector<LightSource>& lights)
{
    bool flag = false;
    double t = std::numeric_limits<double>::max();
    QVector3D norm;

    std::tuple<bool, double, QVector3D> res = sphere->rayIntersection(sight,
    lights);
    if (!std::get<0>(res))
        return std::tuple<bool, double, QVector3D>(flag, t, norm);

    for (size_t i = 0; i < polygons.size(); i++) {
        std::tuple<bool, double, QVector3D> res =
        polygons[i].rayIntersection(sight, points, lights,
        material.getSpecCoef());
        if (std::get<0>(res)) {
            double t1 = std::get<1>(res);
            if (t1 < t && !fuzzyIsZero(t1)) {
                t = t1;
                norm = std::get<2>(res);
                flag = true;
            }
        }
    }
    return std::tuple<bool, double, QVector3D>(flag, t, norm);
}
```

Листинг 3.7 — Листинг функции, выполняющей поиск точки пересечения луча с многоугольником

```
std::tuple<bool, double, QVector3D> Polygon::rayIntersection(const sight_t&
sight, const std::vector<QVector3D>& points, const std::vector<LightSource>&
lights, const double& specCoef)
{
    QVector3D dir = sight.dir;
    QVector3D start = sight.start;
    dir = dir.normalized();

    if (vertices.size() < 3)
        return std::tuple<bool, double, QVector3D>(false, 0.0, QVector3D(0,
        0, 0));

    double k = a * dir.x() + b * dir.y() + c * dir.z();
    if (qFuzzyIsNull(k))
        return std::tuple<bool, double, QVector3D>(false, 0.0, QVector3D(0,
        0, 0));

    double t = -(a * start.x() + b * start.y() + c * start.z() + d) / k;
    if (t < 0)
        return std::tuple<bool, double, QVector3D>(false, 0.0, QVector3D(0,
        0, 0));

    QVector3D intersection = start + dir * t;
    if (!isInside(intersection, points))
        return std::tuple<bool, double, QVector3D>(false, 0.0, QVector3D(0,
        0, 0));

    QVector3D n = QVector3D(a, b, c).normalized();

    return std::tuple<bool, double, QVector3D>(true, t, n);
}
```

Листинг 3.8 — Листинг функции, выполняющей поиск пересечения луча со сферой

```
std::tuple<bool, double, QVector3D> Sphere::rayIntersection(const sight_t&
sight, const std::vector<LightSource>& lights)
{
    double t = 0.0;
    QVector3D start = sight.start;
    QVector3D dir = sight.dir;
    QVector3D norm = QVector3D(0, 0, 0);
    QVector3D pt = QVector3D(0, 0, 0);

    QVector3D L = cen - start;
    double tca = QVector3D::dotProduct(L, dir);
    double d2 = QVector3D::dotProduct(L, L) - tca * tca;

    double rad2 = rad * rad;
    if (d2 > rad2)
        return std::tuple<bool, double, QVector3D>(false, t, norm);

    if (tca < 0 && QVector3D::dotProduct(L, L) < rad2)
        return std::tuple<bool, double, QVector3D>(false, t, norm);
    double thc = sqrt(rad2 - d2);

    double t0 = tca - thc, t1 = tca + thc;
    if (t0 > 1.0)
        t = t0;
    else if (t1 > 1.0)
        t = t1;
    else
        return std::tuple<bool, double, QVector3D>(false, t, norm);

    pt = start + dir * t;
    norm = (pt + (cen * (-1))).normalized();

    return std::tuple<bool, double, QVector3D>(true, t, norm);
}
```

Листинг 3.9 — Листинг функции, выполняющей проверку принадлежности точки многоугольнику (начало)

```
bool Polygon::isInside(QVector3D& point,
const std::vector<QVector3D>& points)
{
    int signX = 0;
    int signY = 0;
    int signZ = 0;

    if (!qFuzzyIsNull(a))
    {
        for (size_t i = 0; i < vertices.size(); i++)
        {
            QVector3D p0 = points[vertices[i]];
            QVector3D p1 = points[vertices[(i + 1) % vertices.size()]];

            QVector3D side = p1 - p0;
            QVector3D toPoint = point - p0;

            double prod = crossProduct(side.y(), side.z(), toPoint.y(),
            toPoint.z());

            int curSignX = sign(prod);

            if (curSignX != 0) {
                if (curSignX != signX && signX != 0)
                    return false;
                signX = curSignX;
            }

        }
    }
    else if (!qFuzzyIsNull(b))
    {
```

Листинг 3.10 — Листинг функции, выполняющей проверку принадлежности точки многоугольнику (окончание листинга 3.9)

```
        for (size_t i = 0; i < vertices.size(); i++) {
            QVector3D p0 = points[vertices[i]];
            QVector3D p1 = points[vertices[(i + 1) % vertices.size()]];
            QVector3D side = p1 - p0;
            QVector3D toPoint = point - p0;
            double prod = crossProduct(side.x(), side.z(), toPoint.x(),
            toPoint.z());
            int curSignY = sign(prod);
            if (curSignY != 0) {
                if (curSignY != signY && signY != 0)
                    return false;
                signY = curSignY;
            }
        }
    }
    else {
        for (size_t i = 0; i < vertices.size(); i++) {
            QVector3D p0 = points[vertices[i]];
            QVector3D p1 = points[vertices[(i + 1) % vertices.size()]];
            QVector3D side = p1 - p0;
            QVector3D toPoint = point - p0;
            double prod = crossProduct(side.x(), side.y(), toPoint.x(),
            toPoint.y());
            int curSignZ = sign(prod);
            if (curSignZ != 0) {
                if (curSignZ != signZ && signZ != 0)
                    return false;
                signZ = curSignZ;
            }
        }
    }
    return true;
}
```

3.6. Описание интерфейса программы

Программа запускается с помощью среды разработки *QtCreator* при нажатии на кнопку сборки и запуска в левом нижнем углу.

На рисунке 3.3 представлен пользовательский интерфейс разработанной программы:

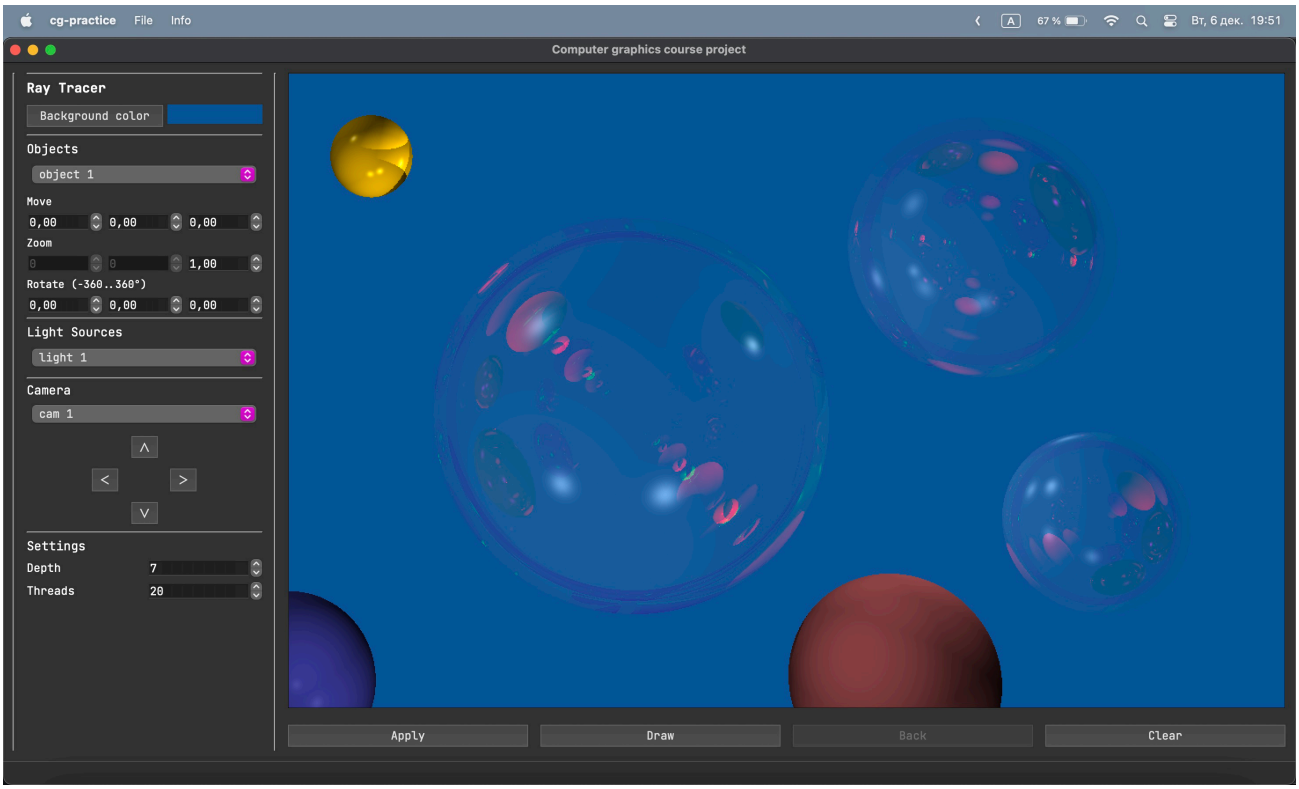

Рисунок 3.3 — Интерфейс ПО

Опции верхнего меню:

1) *File* — предоставляет дополнительные опции для работы с файлами: загрузка и сохранение.

2) *Info* — предоставляет краткие сведения о программе и авторе программы.

Опции нижнего меню:

1) *Apply* — позволяет применить действия по преобразованию объектов, выбранные пользователем в левом меню, к объектам сцены.

2) *Draw* — позволяет синтезировать заданную пользователем сцену.

3) *Clear* — позволяет очистить экран, удалить информацию о сцене и сбросить все настройки.

Опции левого меню:

1) Пользователь может изменять цвет фона для сцены. Миниатюра с цветом расположена рядом с кнопкой выбора цвета. После выбора цвета фона автоматически начинается синтез сцены.

2) Пользователь может изменять положение объектов на сцене. Одновременно можно взаимодействовать как с одним выбранным объектом, так и сразу со всеми объектами сцены.

3) Пользователь может изменять положение точки наблюдения в помощью кнопок перемещения.

4) Пользователь может указать количество выделяемых для синтеза сцены потоков и максимальную глубину рекурсивных погружений. В целях обеспечения быстродействия программы, параметр глубины ограничен от 1 до 50, а количество выделяемых потоков не может превышать 64.

3.7. Тестирование

Было произведено тестирование нескольких модулей программы с помощью библиотеки *testlib* и фреймворка *QtTest*. В листинге 3.11 показано подключение библиотеки *testlib* для тестирования.

Листинг 3.11 — Подключение библиотеки для тестирования

```
QT += testlib
```

Были протестированы основные модули программы: *bubble*, *polyhedron*, *polygon*, *sphere*, *camera*, $transform$. Для примера тестирования была выбрана функция *rotateAxe* из модуля $transform$. Создан класс $TestTransform$, наследуемый от класса *QObject*. Название класса формируется как $Test <$ название тестируемого модуля $>$. В публичном слоте находится конструктор класса, а в приватном — непосредственно создаваемые тесты. В листинге 3.12 показана реализация класса $TestTransform$ для тестирования модуля $transform$.

Листинг 3.12 — Листинг реализации класса TestTransform для тестирования модуля transform

```
class TestTransform : public QObject {
    Q_OBJECT
public:
    explicit TestTransform(QObject* parent = nullptr);

private slots:
    void test_rotate_01();
};
```

В листинге 3.13 приведен листинг реализации конкретного теста поворота точки на 90 градусов для функции *rotateAxe*, которая осуществляет поворот заданной точки на данный угол.

Листинг 3.13 — Листинг реализации конкретного теста поворота точки на 90 градусов для функции rotateAxe

```
void TestTransform::test_rotate_01()
{
    double x = 1.0;
    double y = 2.0;

    rotateAxe(x, y, 90.0);

    QCOMPARE(qFuzzyCompare(x, 2.0), true);
    QCOMPARE(qFuzzyCompare(y, -1.0), true);
}
```

Проверка корректности полученного результата производится с помощью макросов фреймворка *QtTest*. В данном примере используется макрос *QCOMPARE*, сравнивающий ожидаемое значение с полученным. Функция *qFuzzyCompare* используется для корректного сравнения чисел с плавающей точкой, так как *QCOMPARE* сравнивает числа с помощью оператора «==». Такое сравнение не используется для чисел с плавающей точкой, но подходит для сравнения значений типа *bool*, что и показано в приведенном примере.

3.8. Примеры работы программы

На рисунках 3.4 — 3.5 представлены примеры работы реализованной программы.

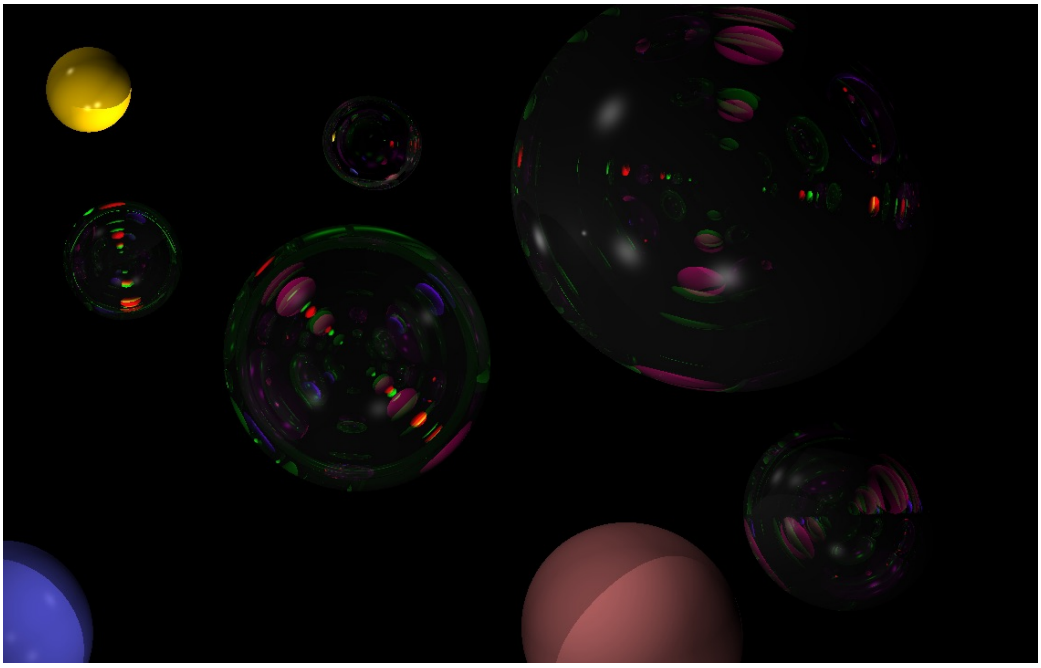

Рисунок 3.4 — Пример работы программы №1

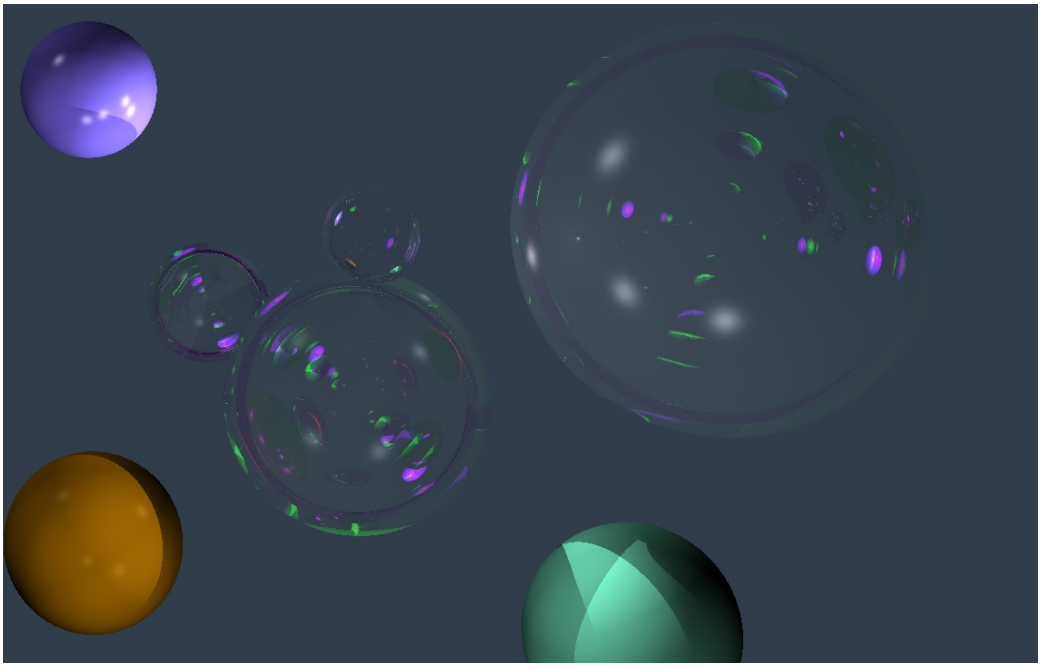

Рисунок 3.5 — Пример работы программы №2

3.9. Выводы по технологической части

В данном разделе была представлена реализация решения поставленной задачи в в формате листинга кода, были указаны требования к ПО и средства реализации, описан пользовательский интерфейс и результаты проведённого тестирования программы.

4. Исследовательская часть

В данном разделе описано проведённое исследование и представлены его результаты. Также приведены характеристики устройства, на котором проводились замеры.

4.1. Технические характеристики устройства

Технические характеристики устройства, на котором выполнялось исследование [3]:

— операционная система *macOS Monterey* 12.4;

— 8 ГБ оперативной памяти;

— процессор *Apple M*2 (базовая частота — 2400 МГц, но поддержка технологии *Turbo Boost* позволяет достигать частоты в 3500 МГц).

4.2. Постановка исследования

Целью исследования является определение зависимости времени генерации изображения сцены, содержащей изображение мыльных пузырей, от количества потоков, выделяемых для выполнения программы. Исследование проводилось для значений количества выделяемых потоков, не превышающих 25, так как уже на этом значении происходит замедление работы алгоритма, в связи с превышением затрат времени на выделение потока максимально возможных для поддержания ускорения работы программы. Во время проведения исследования устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений окружения, окружением и системой тестирования. Оптимизации компилятора были отключены. По результатам исследования составляется сравнительная таблица и строится график зависимости.

4.3. Средства исследования

Время синтеза сцены замерялось с помощью класса $std :: chrono :: system\_clock$ [10], который представляет реальное время. В изображениях для исследования растр имеет одинаковое количество пикселов в обоих плоских измерениях, на самом изображении находится один объект-пузырь и 1 источник света.

4.4. Результаты исследования

Результаты замеров времени выполнения (в мс) приведены в таблице 4.1. Количество потоков равное нулю означает, что замер проведён для однопоточной реализации, что отличается от столбца, соответствующего одному потоку, так как для последнего подразумевается именно переключение на дополнительный поток. В замерах использовались изображения размером от 128×128 пикселов до 512×512 пикселов.

Таблица 4.1 — Таблица времени выполнения программы (мс)

Размер изображения (в пикселах)	Количество потоков						
	0	1	2	4	8	16	25
128	1199.96	1202.21	610.509	329.765	215.551	200.37	213.304
256	4910.94	4916.96	2425.05	1337.52	1074.85	983.585	995.868
384	10 805.7	10 808.8	5432.39	3160.25	2498.38	2431.28	2633.66
512	17 347	17 414.2	9592.75	5332.85	3932.14	3802.21	3807.14

На рисунке 4.1 приведён график, отображающие зависимость времени работы программы от количества выделяемых потоков.

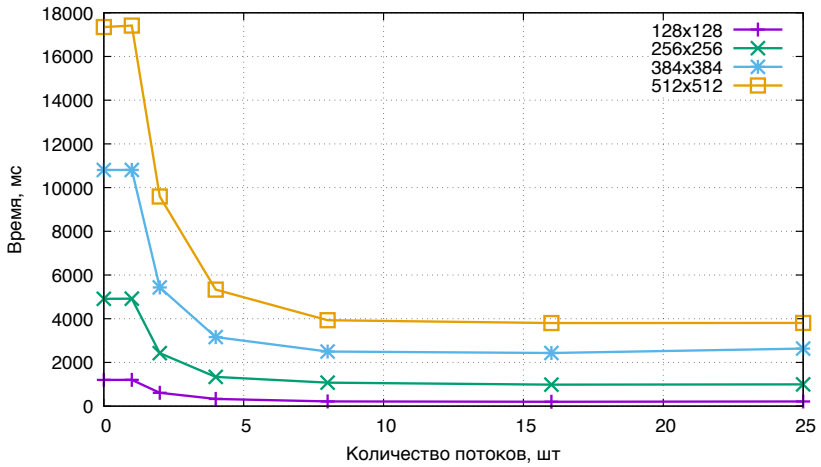

Рисунок 4.1 — Зависимость времени работы программы от количества выделяемых потоков

4.5. Выводы по исследовательской части

В данном разделе было описано проведённое исследование и представлены его результаты. Также были приведены характеристики устройства, на котором проводилось исследование. Результаты исследования показывают, что наибольшая эффективность достигается при использовании потоков, количество которых не более чем в 2 раза превышает количество ядер процессора. При большем количестве потоков скорость работы программы снижается. Это связано с тем, что программа содержит участки кода, которые выполняются последовательно, поэтому к ним неприменимо распараллеливание. Тогда определённая доля суммарного времени выполнения всегда будет оставаться постоянной и ускорение достигнет своего предела. Данный вывод согласуется с законом Амдала [14]. Также, при увеличении числа потоков увеличивается время, затрачиваемое на их выделение, что тоже ограничивает возможное ускорение. Например, при 25 потоках при размере изображения 512×512 программа работала в 1.005 раз медленнее, чем при 16 потоках. Также, программа работает немного дольше при выделении 1 потока, чем при отсутствии дополнительных потоков, так как требуется время на его выделение. В среднем разница составляла до 20 мс. При увеличении размера изображения время работы программы росло. Например, на 4 потоках при изображении размером 128×128 программа работал в 19.08 раз быстрее, чем при размере изображения 384×384. В среднем для любого размера изображения программа работала на 8 потоках в $5 - 6$ раз быстрее, чем без выделения дополнительных потоков. Скорость работы постепенно увеличивалась при повышении числа потоков до 16.

Заключение

В ходе выполнения курсовой работы была достигнута поставленная цель: была разработана программа для построения трёхмерного изображения мальных пузырей вещества.

Также реализованы все поставленные задачи, а именно:

1) было выполнено формальное описание заданных объектов сцены;
2) были изучены алгоритмы построения реалистичных изображений, необходимые для визуализации мыльных пузырей вещества, и проведено их сравнение;
3) были изучены оптические свойства мыльных пузырей вещества для синтеза реалистичного изображения;
4) был разработан алгоритм, позволяющий построить реалистичное изображение мыльных пузырей вещества;
5) была описана структура разрабатываемого ПО;
6) были выбраны средства разработки, позволяющие реализовать выбранный алгоритм;
7) была выполнена программная реализация выбранного алгоритма;
8) было проведено исследование зависимости времени работы программы от количества выделяемых потоков.

В будущем разработанную программу можно будет усовершенствовать, реализовав анимацию разрушения пузыря и движения жидкости между мыльными стенками. Также, можно добавить возможность визуализации мыльной пены и изменения формы пузырей.

Список использованных источников

1. Документация библиотеки Qt [Электронный ресурс]. Режим доступа: *https://doc.qt.io* (дата обращения: 20.10.2022).

2. Справочник по языку C++ [Электронный ресурс]. Режим доступа: *https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-language-reference?view=msvc-170* (дата обращения: 20.09.2022).

3. Техническая спецификация ноутбука MacBook Air [Электронный ресурс]. Режим доступа: *https://support.apple.com/kb/SP869* (дата обращения: 20.09.2022).

4. Mario Botsch, Mark Pauly, Leif Kobbelt, Pierre Alliez, Bruno Lévy, et al.. Geometric Modeling Based on Polygonal Meshes. 2007.

5. 3D-моделирование [Электронный ресурс]. Режим доступа: *http://amti.esrae.ru/pdf/2017/2(3)/129.pdf* (дата обращения: 10.10.2022).

6. Роджерс Д., Алгоритмические основы машинной графики — М.: Книга по Требованию, 2013. — 512с., ил.

7. Куров А. В. Конспект курса лекций по компьютерной графике, 2022.

8. Polygon Mesh Modelling for Computer Graphics [Электронный ресурс]. Режим доступа: *https://prism.ucalgary.ca/handle/1880/23924* (дата обращения: 10.10.2022).

9. Способы представления и размещения трёхмерных моделей для прототипирования ювелирных изделий : Материалы VI Международной научно-практической конференции (школы-семинара) молодых ученых. Тольятти, 23–25 апреля 2020 года. / Орг.: Министерство науки и высшего образования Российской Федерации ; Тольяттинский государственный университет. — Тольятти : Тольяттинский государственный университет. — 840-845с.

10. International Standard ISO/IEC 14882:2020(E) – Programming Language C++ [Электронный ресурс]. Режим доступа: *https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4868.pdf* (дата обращения: 10.10.2022).

11. getopt(3) — Linux manual page [Электронный ресурс]. Режим доступа: *https://man7.org/linux/man-pages/man3/getopt.3.html* (дата обращения: 10.10.2022).

12. An Overview of 3D Data Content, File Formats and Viewers [Электронный ресурс]. Режим доступа: *http://isda.ncsa.illinois.edu/ peter/publications/techreports/2008/NCSA-ISDA-2008-002.pdf* (дата обращения: 10.10.2022).

13. Comparison of different image formats using LSB Steganography [Электронный ресурс]. Режим доступа: *https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8269657* (дата обращения: 10.10.2022).

14. Hill M. D., Marty M. R. Amdahl's law in the multicore era //Computer. – 2008. – Т. 41. – №. 7. – С. 33-38.