СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Сегодня, в век высоких технологий, роль компьютерной графики велика как никогда. Более того, потребность в реалистичном изображении существует во всех сферах жизни человека [1].

Во-первых, кинематограф занимает одно из центральных мест в индустрии развлечений, а реалистичные спецэффекты позволяют создавать самые удивительные и невообразимые сцены, что в свою очередь делает кино более зрелищным причем с минимальными затратами. В конечном итоге массовый потребитель имеет более широкий выбор фильмов для просмотра, а цена остается минимальной.

Во-вторых, в игровой индустрии графика, становясь все более реалистичной, обеспечивает полное погружение пользователя в игровой процесс и создает изображение, едва отличимое от реальности.

В-третьих, реалистичная компьютерная графика упрощает работу современным архитекторам и дизайнерам: можно в мельчайших деталях рассмотреть объекты, а также достичь более быстрой разработки проектов и обеспечить тесное взаимодействие между заказчиком и исполнителем.

В-четвертых, современные технологии компьютерной графики находят все большее применение в дополненной реальности. Например, в сложных хирургических операциях врачи используют специальные очки, показывающие капилляры с высокой долей точности, что позволяет избежать лишних кровопотерь.

Таким образом, задача создания реалистичного изображения актуальна и крайне востребована. Наиболее значимой составляющей этой задачи является корректный расчет освещения. В окружающем нас мире способны видеть различные вещи благодаря тому, что лучи света преломляются, излучаются, отражаются от объектов и попадают на нашу сетчатку глаза. Следовательно, необходимо подобрать методы, моделирующие тот же самый процесс, но уже в компьютерной визуализации.

Целью данной работы является проектирование программного обеспечения, реализующего построение реалистичного изображения с учетом оптических свойств объекта, на примере визуализации кубика льда, имеющего пузырьки воздуха внутри.

Для достижения указанной выше цели следует выполнить следующие задачи:

- рассмотреть существующие решения для моделирования;
- формально описать структуру объектов синтезируемой схемы;
- выбрать или модифицировать существуюшие алгоритмы трехмерной графики, необходимые для визуализации модели;
- привести схемы выбранных алгоритмов для решения поставленной задачи;
- описать используемые типы данных;
- описать структуру разрабатываемого ПО;
- реализовать выбранные алгоритмы визуализации;
- протестировать разработанное ПО;
- сделать выводы на основании проделанной работы.

1 Аналитическая часть

В этом разделе будут рассмотрены предметная область и основные алгоритмы, необходимые для создания реалистичного изображения, а также произведён и обоснован выбор алгоритмов для реализации в проекте.

1.1 Постановка задачи

Задачей программы является представление трехмерной визуализации кубика льда с пузырьками воздуха, которая позволила бы наблюдать процесс преломления, благодаря чему было бы проще изучить оптические свойства льда и основные законы движения световых лучей.

1.2 Предметная область

Лед - вода в твердом агрегатном состоянии. Льдом иногда называют некоторые вещества в твердом агрегатном состоянии, которым свойственно иметь жидкую или газообразную форму при комнатной температуре; в частности сухой лед, амиачный лед или метановый лед [2].

В настоящее время известны три аморфных разновидности и 17 кристаллических модификаций льда. В природных условиях Земли вода образует кристаллы одной кристаллической модификации — гексагональной сингонии. Лед встречается в природе в виде собственно льда (материкового, плавающего, подземного), а также в виде снега, инея, изморози. Под действием собственного веса лед приобретает пластические свойства и текучесть.

Природный лед обычно значительно чище, чем вода, так как при кристаллизации воды в первую очередь в решётку встают молекулы воды. Лед может содержать механические примеси — твёрдые частицы, капельки концентрированных растворов, пузырьки газа (см. рисунок 1.1). Наличием кристалликов соли и капелек рассола объясняется солоноватость морского льда.

Рисунок 1.1 – Фото естественных кубиков льда с пузырьками воздуха

1.3 Обзор существующих программных решений

Следует отметить, что существуют приложения, которые позволяют выполнить визуализацию льда. Более того, некоторые из также моделируют такие физические процессы, как таяние, трескание и кристаллизацию с течением времени.

Одним из самых популярных решений является Blender [3]. Данная программа позволяет выполнить высоко детализированную визуализацию с любыми условиями окружающей среды (см. рисунки 1.2–1.4).

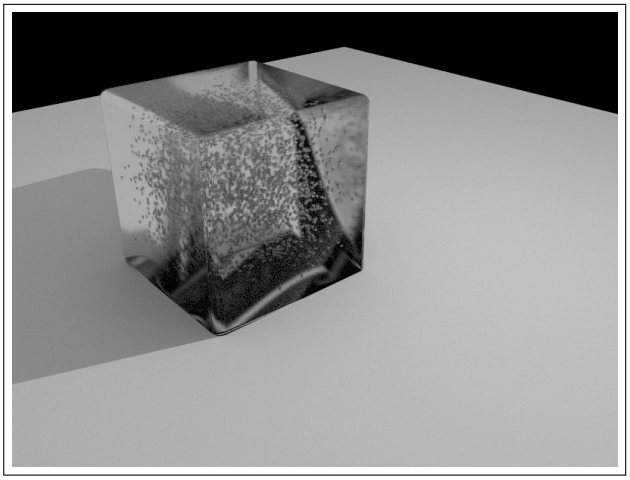

Рисунок 1.2 – Моделирование кубика льда с пузырьками воздуха в Blender

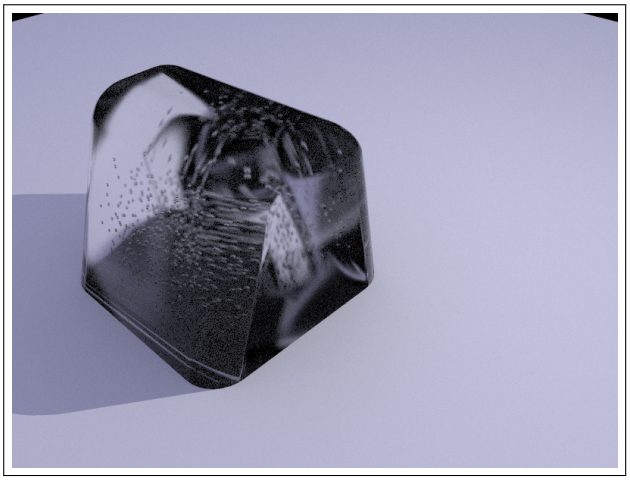

Рисунок 1.3 – Моделирование таяния льда в Blender

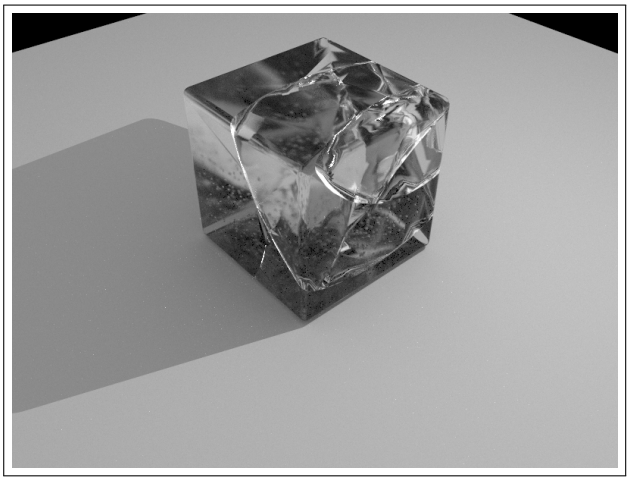

Рисунок 1.4 – Моделирование трескания льда в Blender

На рисунке 1.5 показан пример кубика льда из рекламных материалов, созданных в программе KeyShot. Данная утилита предназначенна для визуализации дизайнерских решений и не обладает широким функционалом настройки свойств окружения.

Также широко используется моделирование в программе AutoDesk Maya ввиду совместимости с другими продуктами компании и легкого импорта или экспорта модели (см. рисунок 1.6).

Список программ для визуализации не исчерпывается приведенными выше решениями. Их существует гораздо больше, каждое из которых имеет свои преимущества и недостатки в конкретной задаче.

Рисунок 1.5 – Визуализация ледяных коктелей в KeyShot

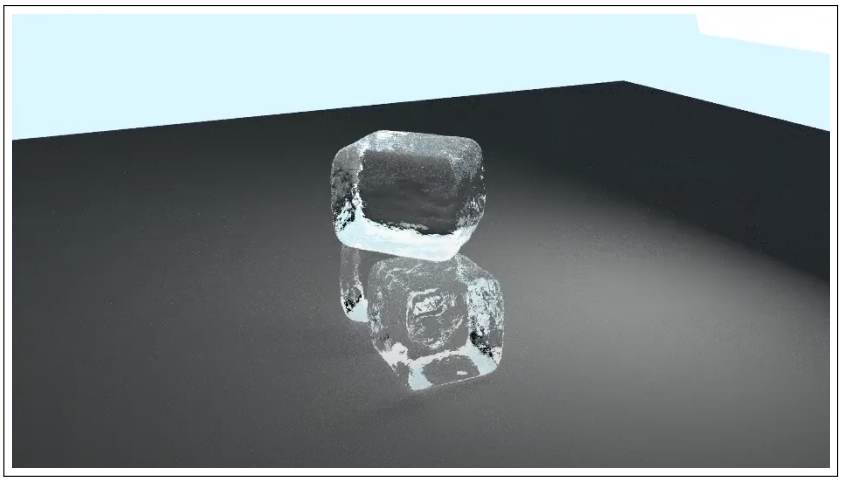

Рисунок 1.6 – Визуализация льда в Maya

1.4 Формализация объектов синтезируемой сцены

Сцена состоит из следующих объектов:

- трехмерная модель кубика льда, которая описывается положением в пространстве и оптическими свойствами;
- трехмерные пузырьки воздуха внутри кубика льда — в рамках поставленной задачи представляют собой сферические объекты. Каждый пузырек имеет свой размер;
- источниками света являются материальные точки, испускающие свет. Положение источника задается трехмерными координатами, а на-

правление распространения вектором.

1.5 Анализ способов задания трехмерных моделей

В компьютерной графике в основном используются 3 вида моделей трехмерных объектов [4]:

1. **Каркасная(проволочная) модель**. Это простейший вид моделей, содержащий минимум информации — о вершинах и рёбрах объектов. Главный недостаток — такая модель не всегда правильно передает представление об объекте (например, если в объекте есть отверстия).

2. **Поверхностная модель**. Отдельные участки задаются как участки поверхности того или иного вида. Эта модель решает проблему каркасной, но все еще имеет недостаток — нет информации о том, с какой стороны поверхности находится собственный материал.

3. **Объемная модель**. В отличии от поверхностной, содержит указание расположения материала (чаще всего указанием направления внутренней нормали).

Таким образом, каркасная модель не подойдет для решения поставленной задачи, так как она не позволяет построить реалистичное изображение из-за недостатка информации, а объемная модель будет избыточной, так как нам не нужна информация о том, с какой стороны находится материал, из которого изготовлен тот или иной объект. Следовательно, можно сделать вывод о том, что для решения поставленной задачи подойдет именно поверхностная модель объекта.

1.6 Анализ способов задания поверхностных моделей

Можно выделить следующие способы задания поверхностных моделей:

1. **Аналитический**.

   Этот способ подразумевает описание поверхности с помощью функции, неявного уравнения или параметрической формы. Для описания сложных поверхностей в этом случае можно использовать сплайны для аппроксимации их отдельных фрагментов. Чаще всего в трехмерной графике используют кубические сплайны, т.к. это наименьшая из степеней, позволяющая описать любую форму и обеспечить непрерывную первую производную при стыковке сплайнов, благодаря чему поверхность будет без изломов в местах стыка.

   К достоинствам аналитической модели можно отнести легкость расчета координат каждой точки поверхности, нормали, а также небольшой объем данных для описания достаточно сложных форм. Среди недостатков - сложность формул и используемых для их реализации функций, значительно снижающих скорость выполнения операций отображения, а также невозможность в большинстве случаев применить данный способ непосредственно для изображения поверхности - поверхность отображается как многогранник, координаты вершин и граней которого рассчитываются в процессе отображения, что уменьшает скорость в сравнении с полигональной моделью описания.

2. **Полигональная сетка**. Полигональная сетка представляет собой описание объекта совокупностью его вершин, ребер и граней (полигонов). Рассмотрим различные способы хранения информации о полигональной сетке:

   - Вершинное представление содержит информацию о соседних вершинах для каждой вершины объекта. При таком способе хранения нелегко выполняются операции с ребрами и гранями в связи с тем, что информация о них выражена неявно. Однако данный

метод не требует большого объема памяти и позволяет эффективно производить трансформации.

- Список граней описывает объект как множество граней и множество вершин. В таком случае, в отличие от вершинного представления, и грани, вершины объекта явно заданы, так что операции поиска соседних граней и вершин постоянны по времени. Однако ребра все еще не заданы явно, так что некоторые операции будут проводиться неэффективно по времени, например, поиск граней, окружающих заданную, а также разрыв или объединение грани.
- Крылатое представление уже хранит явно информацию и о вершинах, и о гранях, и о ребрах объекта. Такой метод решает проблему неэффективности операций разрыва и объединений, однако при этом требует большого объема памяти и достаточно сложен из-за содержания множества индексов.

Для визуализации кубика льда и пузырьков воздуха удобно использовать следующие геометрические примитивы: куб и сфера, которые в свою очередь легко описать аналитическими уравнениями. Таким образом, выбран аналитический способ задания поверхностей.

1.7 Анализ алгоритмов удаления невидимых ребер и поверхностей

Алгоритмы удаления невидимых линий и поверхностей служат для определения линий ребер, поверхностей, которые видимы или невидимы для наблюдателя, находящегося в заданной точке пространства.

Решать задачу можно в:

- объектном пространстве — используется мировая система координат, достигается высокая точность изображения. Обобщенный подход, основанный на анализе пространства объектов, предполагает попарное сравнение положения всех объектов по отношению к наблюдателю;

- пространстве изображений — используется экранная система координат, связанная с устройством, в котором отображается результат (графический дисплей).

Под экранированием подразумевается загораживание одного объекта другим. Под глубиной подразумевается значение координаты Z, направленной от зрителя, за плоскость экрана.

1.7.1 Алгоритм Робертса

Алгоритм Робертса решает задачу удаления невидимых линий. Работает в объектном пространстве. Данный алгоритм работает исключительно с выпуклыми телами. Если тело изначально является не выпуклым, то нужно его разбить на выпуклые составляющие. Алгоритм целиком основан на математических предпосылках [5].

Из-за сложности математических вычислений, используемых в данном алгоритме, и из-за дополнительных затрат ресурсов на вычисление матриц данный алгоритм является медленным.

1.7.2 Алгоритм Z-буфера

Алгоритм Z–буфера позволяет определить, какие пикселы граней сцены видимы, а какие заслонены гранями других объектов [6]. Z–буфер — это двухмерный массив, его размеры равны размерам окна, таким образом, каждому пикселу окна соответствует ячейка Z-буфера. В этой ячейке хранится значение глубины пиксела. Перед растеризацией сцены Z–буфер заполняется значением, соответствующим максимальной глубине. В случае, когда глубина характеризуется значением w, максимальной глубине соответствует нулевое значение. Анализ видимости происходит при растеризации граней, для каждого пиксела рассчитывается глубина и сравнивается со значением в Z–буфере, если рисуемый пиксел ближе (его w больше значения в Z–буфере), то пиксел рисуется, а значение в Z–буфере заменяется его глубиной. Если пиксел дальше, то пиксел не рисуется и Z–буфер

не изменяется, текущий пиксел дальше того, что нарисован ранее, а значит невидим.

Недостатками данного алгоритма являются довольно большой объем требуемой памяти, трудоемкость устранения лестничного эффекта и реализации эффектов прозрачности.

1.7.3 Алгоритм прямой трассировки лучей

Основная идея алгоритма прямой трассировки лучей состоит в том, что наблюдатель видит объекты благодаря световым лучам, испускаемым некоторым источником, которые падают на объект, отражаются, преломляются или проходят сквозь него и в результате достигают зрителя [7].

Основным недостатком алгоритма является излишне большое число рассматриваемых лучей, приводящее к существенным затратам вычислительных мощностей, так как лишь малая часть лучей достигает точки наблюдения. Данный алгоритм подходит для генерации статических сцен и моделирования зеркального отражения, а так же других оптических эффектов.

1.7.4 Алгоритм обратной трассировки лучей

Алгоритм обратной трассировки лучей отслеживает лучи в обратном направлении (от наблюдателя к объекту)[7]. Такой подход призван повысить эффективность алгоритма в сравнении с алгоритмом прямой трассировки лучей. Обратная трассировка позволяет работать с несколькими источниками света, передавать множество разных оптических явлений.

Пример работы данного алгоритма приведен на рисунке 1.7.

Соответственно, порядок алгоритма трассировки такой: найти точку пересечения луча от наблюдателя с первым объектом, для определения степени освещенности объекта проверить его пересечения со всеми источниками света, рекурсивно вызвать процедуру в случае материала с полупрозрачной и/или отражающей поверхностью.

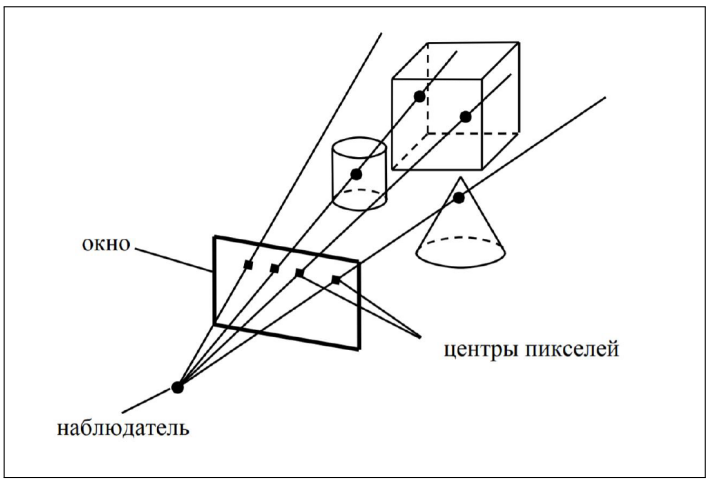

Рисунок 1.7 – Пример работы алгоритма обратной трассировки лучей

Считается, что наблюдатель расположен на положительной полуоси z в бесконечности, поэтому все световые лучи параллельны оси z. В ходе работы испускаются лучи от наблюдателя и ищутся пересечения луча и всех объектов сцены. В результате пересечение с максимальным значением z является видимой частью поверхности и атрибуты данного объекта используются для определения характеристик пикселя, через центр которого проходит данный световой луч.

Для расчета эффектов освещения сцены проводятся вторичные лучи от точек пересечения ко всем источникам света. Если на пути этих лучей встречается непрозрачное тело, значит, данная точка находится в тени.

Несмотря на более высокую эффективность алгоритма в сравнении с прямой трассировкой лучей, данный алгоритм считается достаточно медленным, так как в нем происходит точный расчет сложных аналитических выражений для нахождения пересечения с рассматриваемыми объектами.

1.7.5 Выводы

Оценив все изложенные выше алгоритмы, можно сделать вывод, что для данной работы лучше всего подходит алгоритм обратной трассировки лучей, так как он точно отражает суть физических явлений, таких как отражение и преломление лучей.

1.8 Анализ модели освещения

Для создания реалистичного изображения в компьютерной графике применяются различные алгоритмы освещения. Модель освещения предназначена для расчета интенсивности отраженного к наблюдателю света в каждой точке изображения.

Модель освещения может быть:

- локальной — в данной модели учитывается только свет от источников и ориентация поверхности;
- глобальной — в данной модели, помимо составляющих локальной, учитывается еще и свет, отраженный от других поверхностей или пропущенный через них.

Локальная модель включает 3 составляющих:

- диффузную составляющую отражения;
- отражающую составляющую отражения;
- рассеянное освещение.

Выбор алгоритма построения теней напрямую зависит от выбора алгоритма отсечения невидимых ребер и поверхностей, а также — от выбора модели освещения.

В алгоритме трассировки лучей тени получаются без дополнительных вычислений за счет выбранной модели освещения, а в алгоритме с Z буфером, можно получить тени, используя второй буфер, полученный подменой точки наблюдения на точку источника света.

Выбрана глобальная модель освещения, так как локальная дает неправильный результат в случае явления преломления света, которое широко используется в данной работе.

1.9 Выводы

В данном разделе была рассмотрена предметная область, аналоги разрабатываемому приложению, основные алгоритмы, необходимые для создания реалистичного изображения.

Подводя итог, следует отметить, что:

- для задания трехмерной моделей будет использоваться поверхностная модель;
- был выбран аналитической способ задания поверхностной модели;
- алгоритм обратной трассировки лучей является наиболее подходящим для удаления невидимых ребер и поверхностей;
- будет использоваться глобальная модель освещения.

2 Конструкторская часть

В данном разделе будут рассмотрены требования к программному обеспечению, а также схемы алгоритмов, выбранных для решения поставленной задачи. Также будут описаны пользовательские структуры данных и приведена структура реализуемого программного обеспечения.

2.1 Общий принцип работы программы

На рисунках 2.1–2.2 приведены IDEF0 диаграммы, которые представляют организацию работы программы.

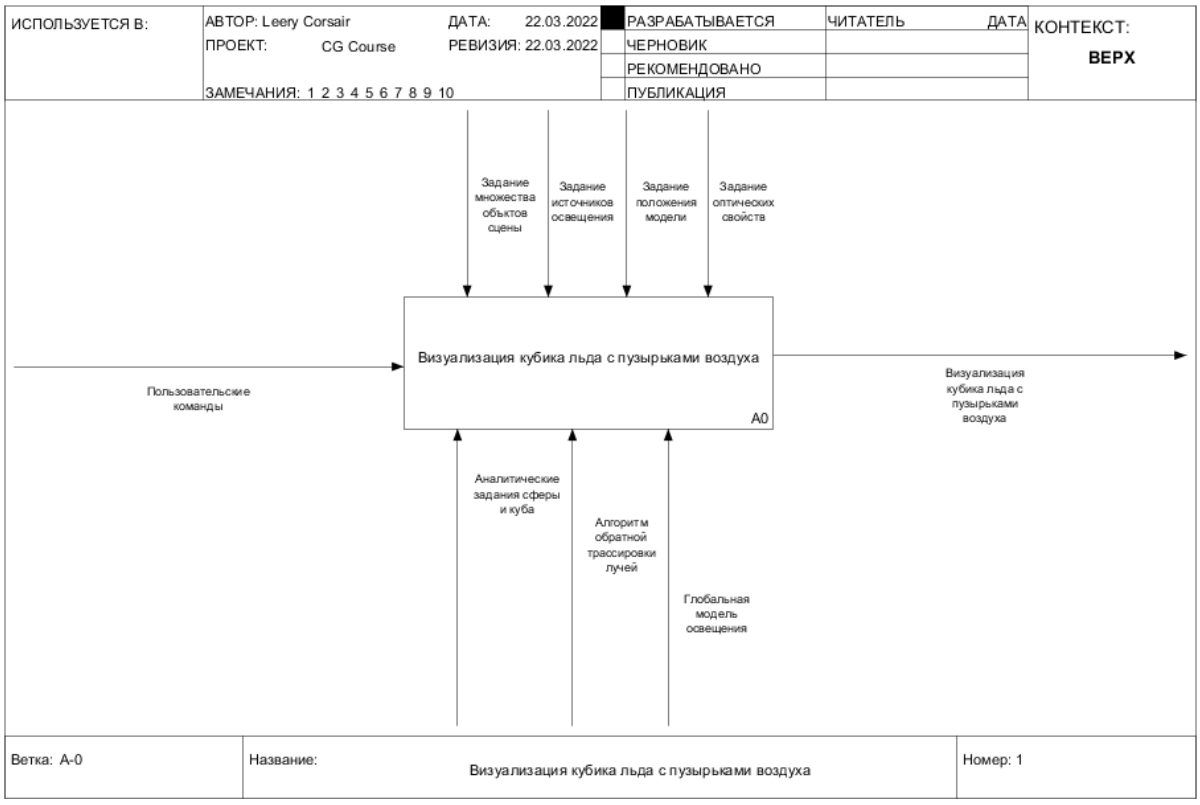

Рисунок 2.1 – IDEF0 диаграмма работы программы

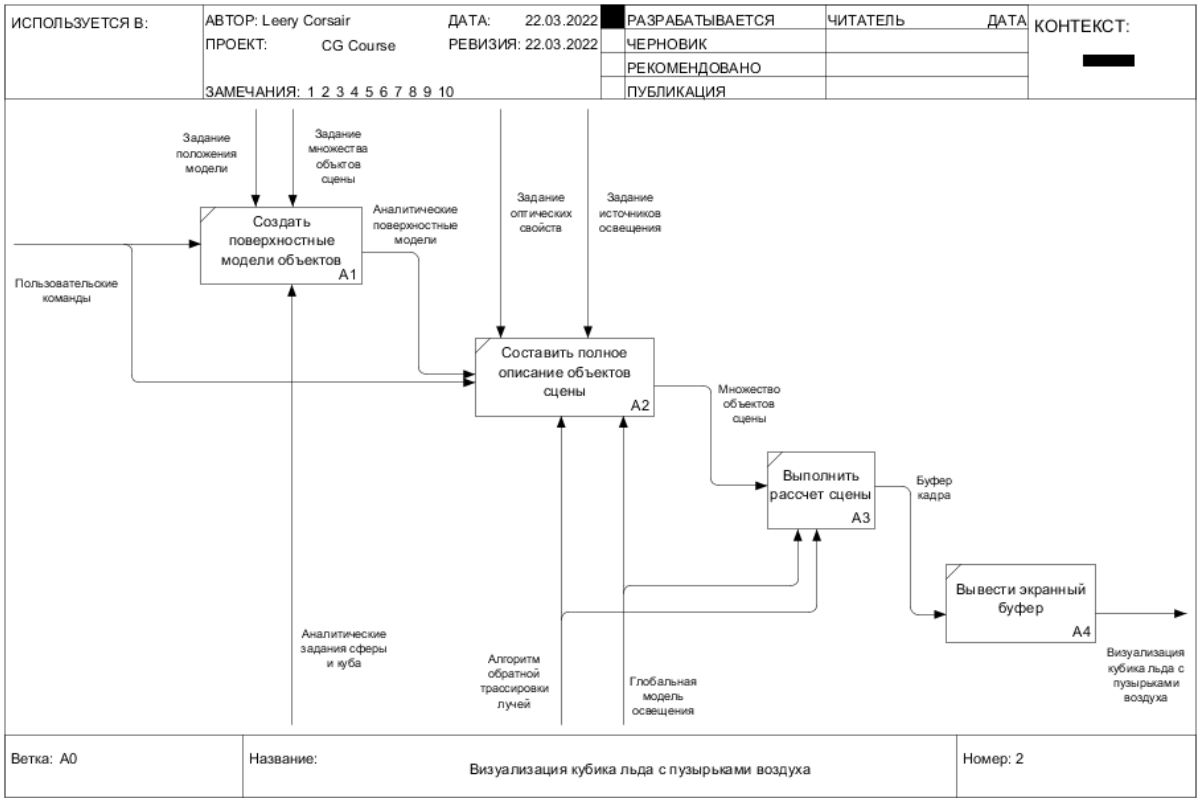

Рисунок 2.2 – Последовательность действий для визуализации кубика льда с пузырьками воздуха

2.2 Требования к ПО

Программа должна предоставлять доступ к следующему функционалу: перенос, масштабирование и поворот объектов сцены, а также возможность изменять параметры сцены — количество и характеристики пузырьков воздуха, количество и свойства источников освещения.

К ПО предъявляются следущие требования:

- программа должна давать отклик на действия пользователя за комфортное для человека время;
- программа должна корректно реагировать на любые действия пользователя.

2.3 Пересечение луча и сферы

Уравнение луча представлено ниже:

$$P = O + t\overrightarrow{D}, t \geq 0, \tag{2.1}$$

где P – точка лежащая на луче, O – начало луча, $\overrightarrow{D}$ – направление луча, t – произвольное положительное действительное число.

Сфера — это множество точек P, лежащих на постоянном расстоянии r от фиксированной точки C. Тогда можно записать уравнение, удовлетворяющее этому условию:

$$distance(P, C) = r \tag{2.2}$$

Запишем расстояние (2.2) между P и C как длину вектора из P в C.

$$|P - C| = r \tag{2.3}$$

Заменим на скалярное произведение вектора на себя:

$$\sqrt{\langle P - C\rangle, \langle P - C\rangle} = r \tag{2.4}$$

Избавимся от корня:

$$\langle P - C\rangle, \langle P - C\rangle = r^2 \tag{2.5}$$

В итоге есть два уравнения - уравнение луча и сферы. Найдем пересечение луча со сферой. Для этого подставим (2.1) в (2.5)

$$\langle O + t\overrightarrow{D} - C\rangle, \langle O + t\overrightarrow{D} - C\rangle = r^2 \tag{2.6}$$

Разложим скалярное произведение и преобразуем его. В результате получим:

$$t^2\langle\overrightarrow{D}, \overrightarrow{D}\rangle + 2t\langle\overrightarrow{OC}, \overrightarrow{D}\rangle + \langle\overrightarrow{OC}, \overrightarrow{OC}\rangle - r^2 = 0 \tag{2.7}$$

Представленное квадратное уравнение (2.7) имеет несколько возможных случаев решения. Если у уравнения одно решение, то луч касается сферы. Два решения – луч пересекает сферу. Нет решений – луч не пересекается со сферой.

2.4 Схемы алгоритмов

2.4.1 Алгоритм обратной трассировки лучей

На рисунке 2.3 представлена схема алгоритма обратной трассировки лучей.

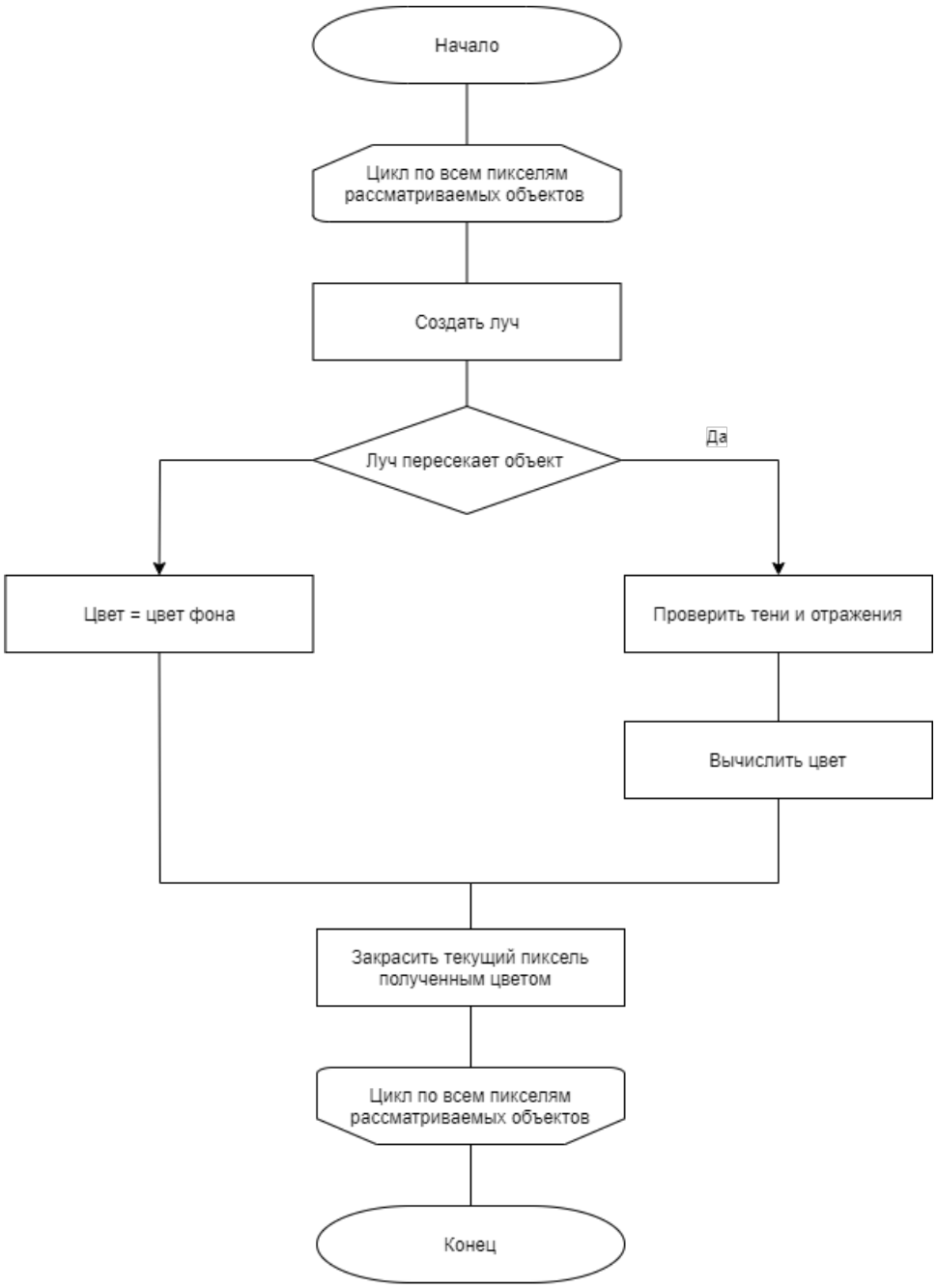

Рисунок 2.3 – Схема алгоритма обратной трассировки лучей

2.4.2 Алгоритм поиска пересечений луча и сферы

На рисунке 2.4 представлена схема алгоритма поиска пересечений луча и сферы.

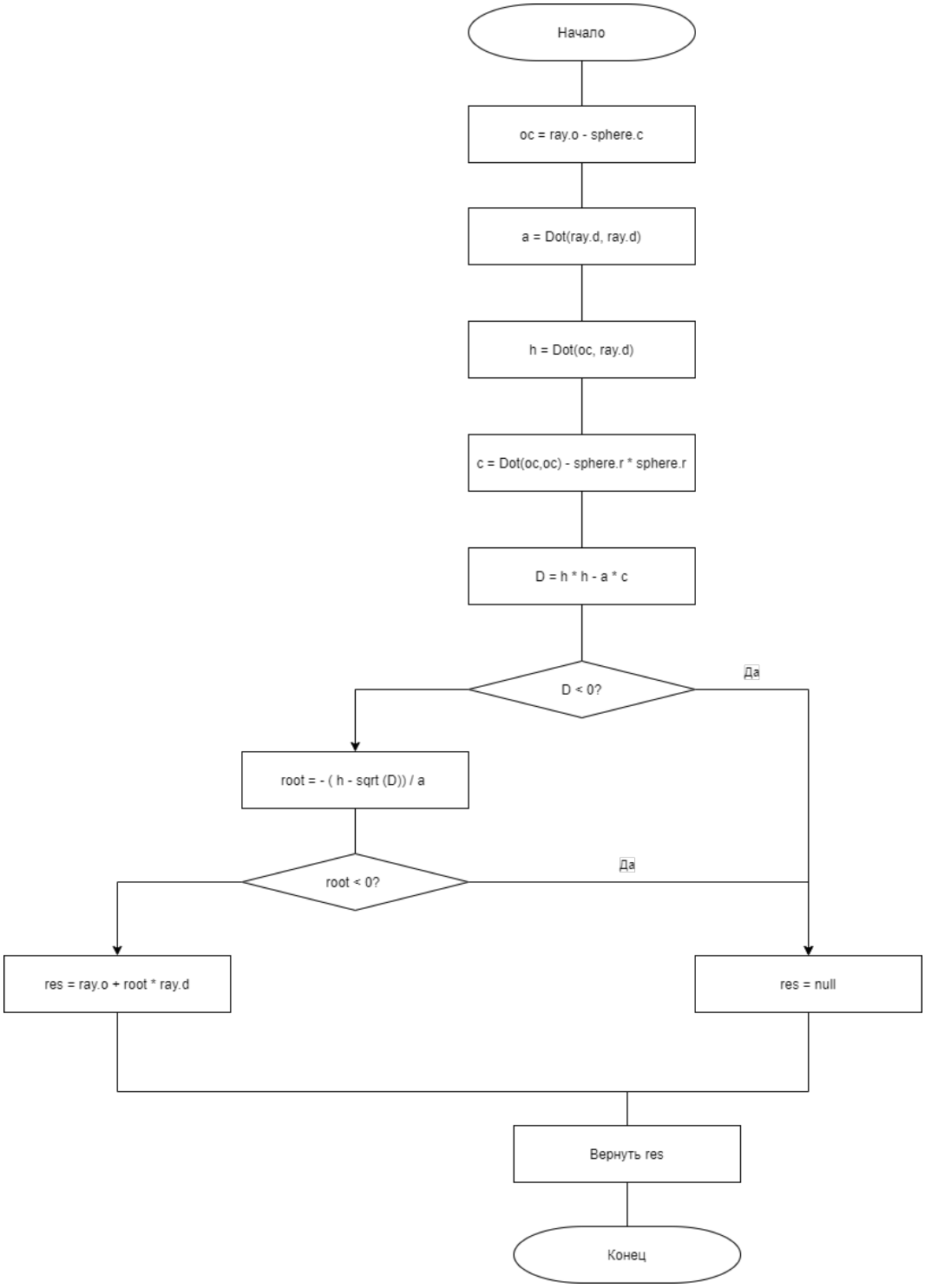

Рисунок 2.4 – Схема алгоритма поиска пересечений луча и сферы

2.4.3 Алгоритм поиска пересечений луча и куба

На рисунке 2.5 представлена схема алгоритма поиска пересечений луча и куба.

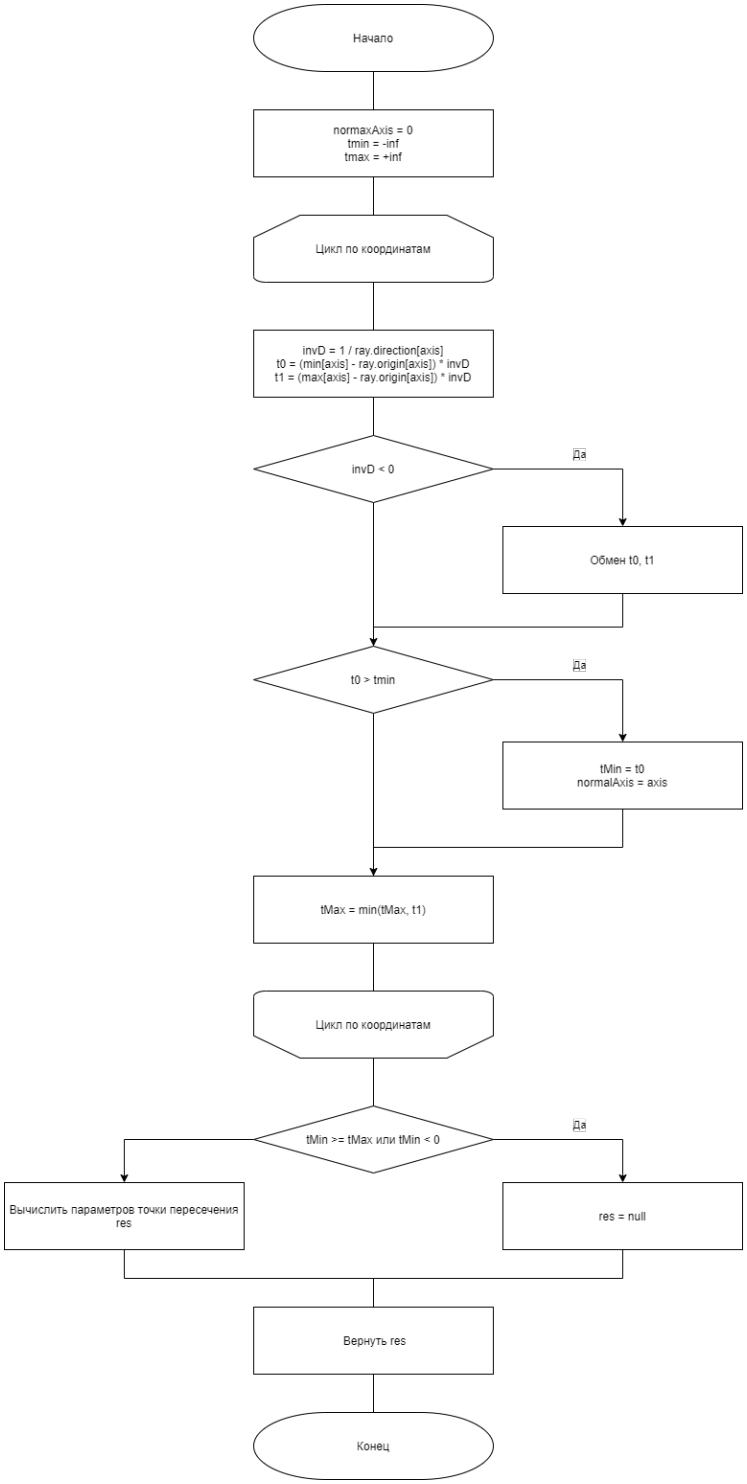

Рисунок 2.5 – Схема алгоритма поиска пересечений луча и куба

2.5 Описание общей структуры ПО

Для данной программы следует выделить следующие логические модули:

- *main* — точка входа в программу и загрузка модуля интерфейса приложения;
- *application* — домен реализации интерфейса программы;
- *geometry* — модуль реализации операций геометрических преобразований;
- *model* — модуль модели сцены;
- *objects* — модуль описания объектов сцены;
- *renderer* — модуль реализации отрисовщика сцены.

На рисунке 2.6 представлена диаграмма классов, отражающая разбиение и взаимодействие объектов сцены.

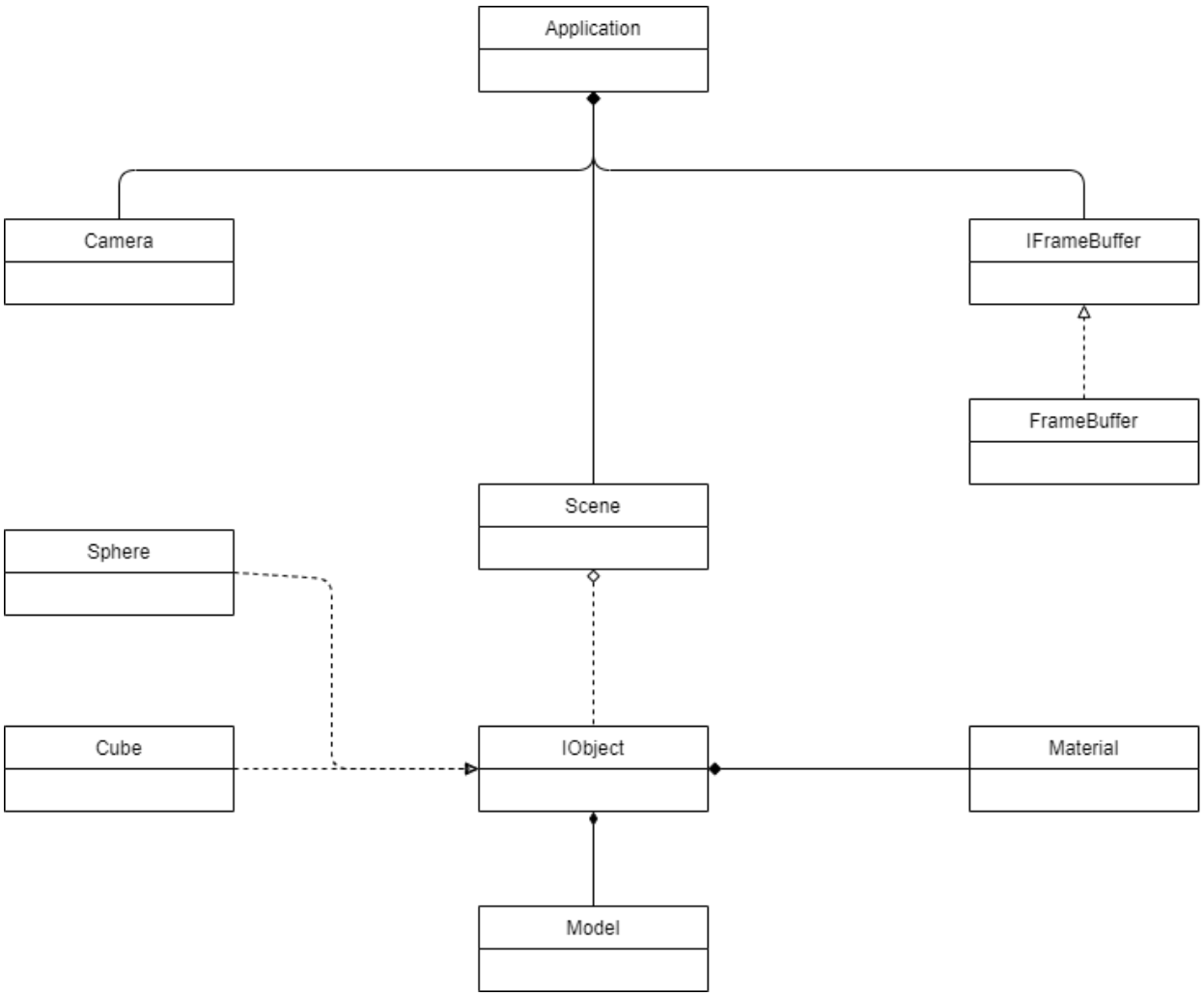

Рисунок 2.6 – Диаграмма классов разрабатываемого ПО

2.6 Выводы

В данном разделе были рассмотрены требования к программе, общий алгоритм ее работы, представлены схемы для реализации выбранных алгоритмов и математические формулы.

3 Технологическая часть

В данном разделе описаны средства реализации, приведены листинги кода и рассмотрен пользовательский интерфейс.

3.1 Средства реализации

Для решения поставленной задачи был выбран язык программирования C++ ввиду ряда причин:

- поддержка объектно-ориентированного подхода к программированию дает возможность создавать четко структурированные и легко модифицируемые программы;
- строгая типизация и стандартизация позволяет избежать множество ошибок, перекладывая ответственность за контролем типов компилятору;
- будучи представителем компилируемых языков, обеспечивает высокую скорость исполнения, что особенно важно в трудоемких алгоритмах трехмерной компьютерной графики.

3.2 Листинг кода

В листингах 3.1–3.2 приведена общая реализация алгоритма обратной трассировки лучей.

Листинг 3.1 – Алгоритм трассировки лучей (часть 1)

```
1 void Render(IFramebuffer* framebuffer, const Camera& camera, const Scene& scene, int
      depth) {
2     int width = framebuffer->width();
3     int height = framebuffer->height();
4 #pragma omp parallel for
5     for (int y = 0; y < height; y++) {
6         for (int x = 0; x < width; x++) {
7             float s = (float)x / (width - 1);
8             float t = (float)y / (height - 1);
```

Листинг 3.2 – Алгоритм трассировки лучей (часть 2)

```
            Ray ray = camera.generateRay(s, t);
            framebuffer->setPixel(x, y, castRay(ray, scene, depth));
        }
    }
}
```

В листинге 3.3 приведена реализация алгоритма пуска луча.

Листинг 3.3 – Алгоритм пуска луча

```
static Vec3 castRay(const Ray& ray, const Scene& scene, int depth) {
    if (depth <= 0) return std::min(1.0f, 2 * scene.getAmbient()) * background(ray);

    if (std::optional<HitRecord> record = scene.hit(ray)) {
        Material& material = record->material;
        Vec3 reflectDir = Reflect(ray.direction, record->normal);
        Vec3 refractDir = Refract(ray.direction, record->normal, material.refractive);
        Vec3 reflected = castRay(Ray(record->position, reflectDir), scene, depth - 1);
        Vec3 refracted = castRay(Ray(record->position, refractDir), scene, depth - 1);

        float diffuse = scene.getAmbient();
        float specular = 0.0f;
        for (const Vec3& light : scene.lights()) {
            Vec3 source = Normalize(light - record->position);
            diffuse += std::max(0.0f, Dot(source, record->normal));
            Vec3 r = -Reflect(-source, record->normal);
            specular += std::pow(std::max(0.0f, Dot(r, Normalize(ray.direction))),
                material.shininess);
        }
        return material.diffuseAlbedo * material.diffuse * std::min(1.0f, diffuse) +
               material.specularAlbedo * material.specular * std::min(1.0f, specular) +
               material.reflectAlbedo * reflected +
               material.refractAlbedo * refracted;
    }
    return std::min(1.0f, 2 * scene.getAmbient()) * background(ray);
}
```

В листингах 3.4–3.5 приведена реализация алгоритма поиска пересечения луча и сферы.

Листинг 3.4 – Алгоритм поиска пересечения луча и сферы (часть 1)

```
std::optional<HitRecord> Sphere::hit(const Ray &ray) const
{
    Vec3 oc = ray.origin - center;
    float a = Dot(ray.direction, ray.direction);
    float h = Dot(oc, ray.direction);
```

Листинг 3.5 – Алгоритм поиска пересечения луча и сферы (часть 2)

```
    float c = Dot(oc, oc) - radius * radius;

    float discriminant = h * h - a * c;
    if (discriminant < 0)
        return std::nullopt;

    float root = (-h - std::sqrt(discriminant)) / a;
    if (root < 0.001f)
        return std::nullopt;

    HitRecord record;
    record.position = ray.at(root);
```

В листингах 3.6–3.7 приведена реализация алгоритма поиска пересечения луча и куба.

Листинг 3.6 – Алгоритм поиска пересечения луча и куба (часть 1)

```
std::optional<HitRecord> Cube::hit(const Ray &ray) const
{
    int normalAxis = 0;
    float tMin = std::numeric_limits<float>::min();
    float tMax = std::numeric_limits<float>::max();
    for (int axis = 0; axis < 3; axis++)
    {
        float invD = 1.0f / ray.direction[axis];
        float t0 = (min[axis] - ray.origin[axis]) * invD;
        float t1 = (max[axis] - ray.origin[axis]) * invD;
        if (invD < 0.0f)
            std::swap(t0, t1);
        if (t0 > tMin)
        {
            tMin = t0;
            normalAxis = axis;
        }
        tMax = std::min(tMax, t1);
    }
    if (tMin >= tMax)
        return std::nullopt;
    if (tMin < 0.001f)
        return std::nullopt;

    Vec3 point = ray.at(tMin);
    Vec3 normal;
    normal[normalAxis] = Sign((point - center)[normalAxis]);
```

Листинг 3.7 – Алгоритм поиска пересечения луча и куба (часть 2)

```
    HitRecord record;
    record.parameter = tMin;
    record.position = point;
    record.normal = normal;
    record.material = material;
    return record;
}
```

3.3 Описание интерфейса программы

На рисунке 3.1 представлен интерфейс программы.
Можно выделить следующие возможности интерфейса:

- выбор объектов сцены и перемещение объектов сцены;
- масштабирование объектов сцены;
- задание коэффициентов диффузного, зеркального, рефракционного отражений, блеска и рефракции;
- удалять и создавать новые пузырьки;
- удалять, создавать и перемещать источники освещения;
- регулировать максимальную глубину рекурсии;
- изменять позицию, направление камеры, ее угол обзора.

3.4 Примеры работы программы

На рисунках 3.2–3.4 приведены результаты работы разработанного ПО, визуализирующего кубик льда с пузырьками воздуха. В каждом из примеров использовались различные оптические параметры, описывающие модель, максимальная глубина рекурсии алгоритма обратной трассировки лучей составляла 5.

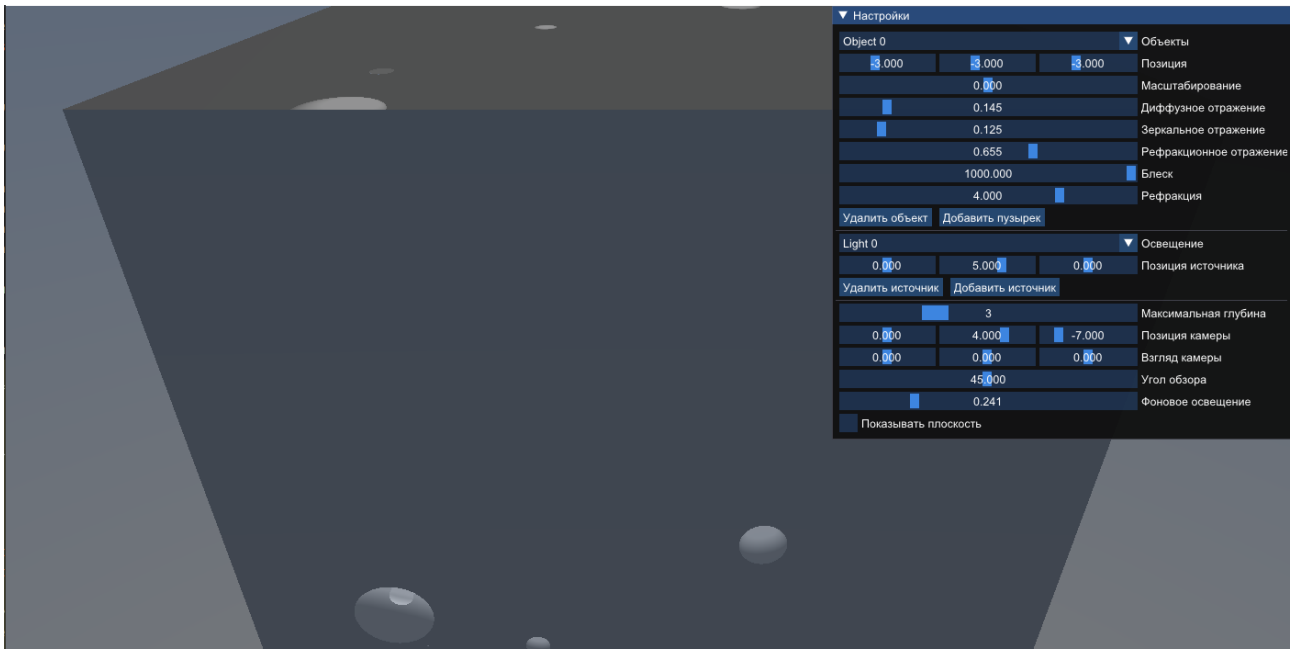

Рисунок 3.1 – Интерфейс программы

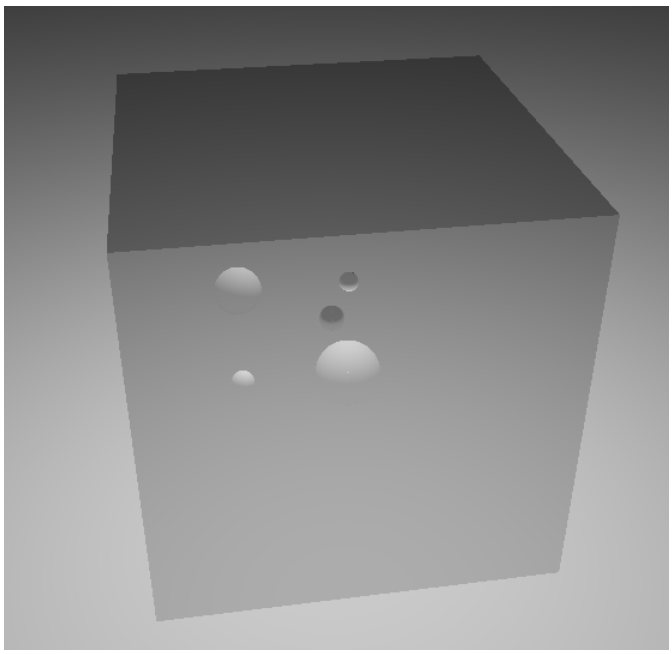

Рисунок 3.2 – Пример работы программы (1)

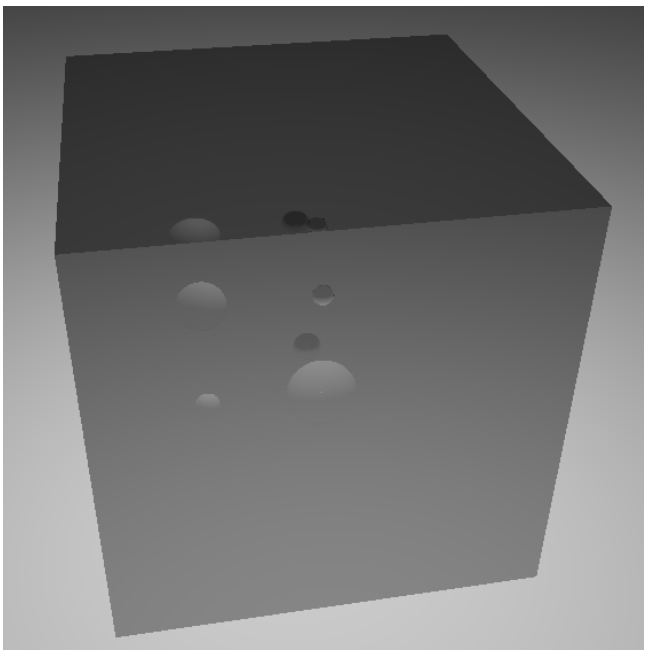

Рисунок 3.3 – Пример работы программы (2)

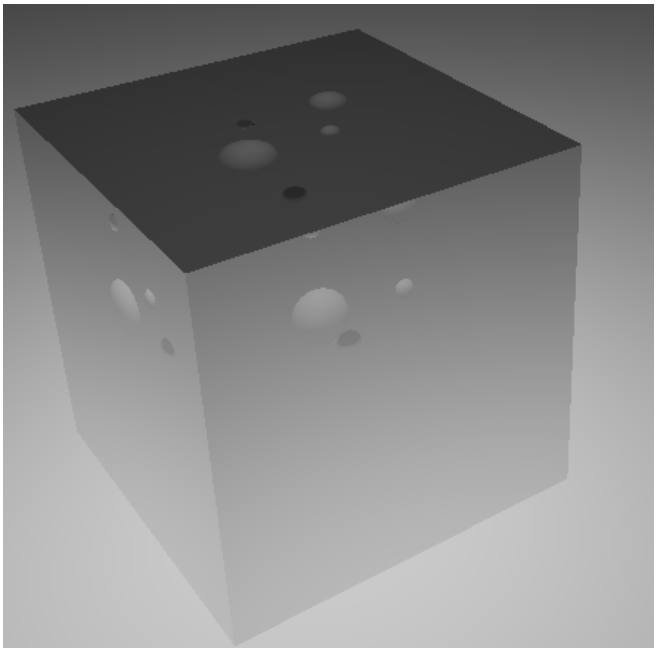

Рисунок 3.4 – Пример работы программы (3)

3.5 Выводы

В данном разделе были описаны средства реализации, приведены листинги кода и рассмотрен пользовательский интерфейс.

4 Исследовательская часть

В данном разделе приведено описание планирования эксперимента и их результаты.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Windows 10 64-bit [8].
- Память: 16 GB.
- Процессор: AMD Ryzen 5 4600H [9] @ 3.00 GHz.

4.2 Постановка эксперимента

Целью эксперимента является определение зависимости времени генерации изображения алгоритмом обратной трассировки лучей от количества используемых потоков.

Во время тестирования устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений окружения, окружением и системой тестирования. Оптимизация компилятора была отключена.

В качестве бенчмарка была выбрана сцена состоящая из кубика льда, пяти пузырьков воздуха и одного источника света (максимальная глубина рекурсии 5).

По результатам эксперимента составляются сравнительные таблицы, а также строятся графики зависимостей.

4.3 Результаты эксперимента

В таблице 4.1 приведены результаты эксперимента.

Таблица 4.1 – Время работы программы

Количество потоков	Время обработки сцены
1	953
2	453
4	247
8	121
12	107
16	117
24	109
32	113

На рисунке 4.1 представлены результаты эксперимента в графическом виде.

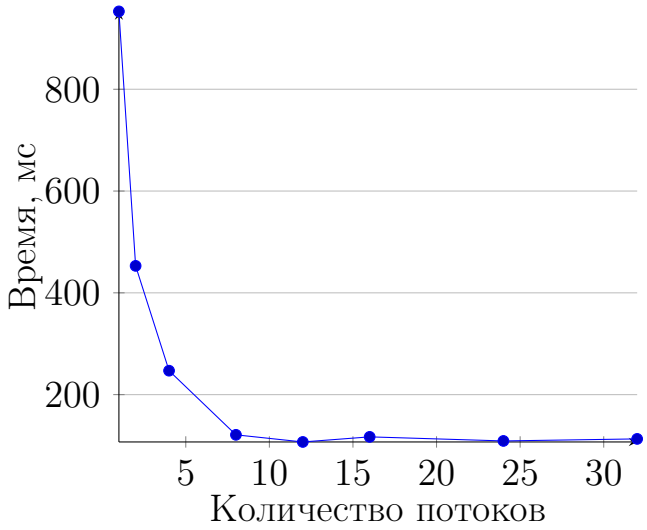

Рисунок 4.1 – Время работы программы

4.4 Выводы

По результатам эксперимента можно сделать вывод о том, что время вычисления алгоритма обратной трассировки лучей уменьшается линейно

с ростом числа потоков. Однако следует отметить, что количество физических ядер ограничивает рост быстродействия. Это подверждается тем, что в современных графических ускорителях десятки тысяч ядер для вычисления кадра изображения в режиме реального времени.

На основе этих данных с целью дальнейшего улучшения продукта можно также провести анализ, какое количество пузырьков является оптимальным, чтобы изображение было наиболее комфортным для человеческого глаза и при этом было достаточно маленькое время отклика приложения на действия пользователя.

ЗАКЛЮЧЕНИЕ

Цель курсовой работы достигнута — было реализовано программное обеспечение для визуализации модели кубика льда с пузырьками воздуха. Пользователь может динамически просматривать сцену: масштабировать, переносить и поворачивать модель, а также изменять параметры сцены — количество и характеристики пузырьков воздуха, количество и свойтва источников освещения.

В ходе выполнения работы были проанализированны существующие алгоритмы компьютерной графики — были рассмотрены методы задания моделей, способы описания поверхностой модели, способы удаления невидимых ребер и поверхностей, модели освещения, а также были выявлены их преимущества и недостатки.

Также над программным продуктом был проведен эксперимент, определяющий зависимость времени генерации изображения алгоритмом обратной трассировки лучей от количества используемых потоков. результатам эксперимента можно сделать вывод о том, что время вычисления алгоритма обратной трассировки лучей уменьшается линейно с ростом числа потоков. Однако следует отметить, что количество физических ядер выступает ограничивает рост быстродействия.

В дальнейшем этот продукт можно улучшить — использовать реалистичные текстуры для изображения поверхностей льда и воздушных пузырьков, повысить быстродействие использованием механизмов распараллеливания вычислений алгоритма обратной трассировки лучей, добавить возможность возможность просмотра изменения состояния льда под действием различных физических факторов и т.д.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Область применения компьютерной графики. [Электронный ресурс]. Режим доступа: https://edusar.soiro.ru/mod/page/view.php?id=6099 (дата обращения: 11.11.2021).

[2] Freezing water at constant volume. [Электронный ресурс]. Режим доступа: https://www.nature.com/articles/s42005-020-0303-9 (дата обращения: 12.11.2021).

[3] Blender Ice Models. [Электронный ресурс]. Режим доступа: https://www.turbosquid.com/Search/3D-Models/ice/blend (дата обращения: 10.11.2021).

[4] Методы трехмерного моделирования. [Электронный ресурс]. Режим доступа: https://studopedia.ru/19_307536_metodi-trehmernogo-modelirovaniya-karkasnoe-modelirovanie-poverhnostnoe-tverdotelnoe-modelirovanie-tipi-poverhnostey-chto-predstavlyayut-s-soboy-trehmernie-ob-ekti.html (дата обращения: 10.11.2021).

[5] Алгоритм Робертса. [Электронный ресурс]. Режим доступа: http://compgraph.tpu.ru/roberts.htm (дата обращения: 10.11.2021).

[6] Z-buffering. [Электронный ресурс]. Режим доступа: https://www.computerhope.com/jargon/z/zbuffering.htm (дата обращения: 10.11.2021).

[7] Ray Tracing in One Weekend. [Электронный ресурс]. Режим доступа: https://raytracing.github.io/books/RayTracingInOneWeekend.html (дата обращения: 10.11.2021).

[8] Explore Windows 10. [Электронный ресурс]. Режим доступа: https://www.microsoft.com/en-us/windows/ (дата обращения: 02.10.2021).

[9] AMD Processors [Электронный ресурс]. Режим доступа: https://www.amd.com/en/products/apu/amd-ryzen-5-4600h (дата обращения: 02.10.2021).