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»

Отчёт по лабораторной работе №4
по дисциплине «Анализ алгоритмов»

Тема: Исследование многопоточности
Студент: Карпова Е. О.
Группа: ИУ7-52Б
Оценка (баллы): _____
Преподаватели: Волкова Л. Л., Строганов Ю. В.

Москва — 2022 г.

Оглавление

Введение

Поток выполнения — наименьшая единица обработки, исполнение которой назначает ядро операционной системы. Также поток можно определить как часть кода, которая может выполняться параллельно с другими частями кода.

Потоки разделяют адресное пространство процесса, а именно код, данные и его контекст. Фактический порядок выполнения потоков зависит от количества процессоров (ядер): если их несколько, то потоки будут действительно исполняться параллельно, а если нет — квазипараллельно, то есть будет происходить поочерёдное выделение квантов времени потокам.

К достоинствам многопоточности относятся:

- отзывчивость — обеспечение быстрой реакции одного потока во время блокировки или занятости других;
- разделение ресурсов — выполнение нескольких задач одновременно в одном адресном пространстве;
- экономичность — переключение контекста происходит быстрее, чем при создании нового процесса;
- масштабируемость — использование мультипроцессорной или однопроцессорной архитектуры.

Цель работы: получение навыков реализации многопоточности для программного продукта, сравнение быстродействия для однопоточной и многопоточной реализаций алгоритма обратной трассировки лучей.

Задачи работы:

1) изучение алгоритма обратной трассировки лучей;
2) разработать последовательную и параллельную версии алгоритма обратной трассировки лучей;
3) разработка схем данных алгоритмов;
4) программная реализация данного алгоритма с использованием одного потока;

5) программная реализация данного алгоритма с использованием вспомогательных потоков;

6) проведение замеров времени работы (в мс) данных реализаций алгоритма;

7) получение зависимости замеряемых величин от количества потоков;

8) проведение сравнительного анализа данных реализаций алгоритма на основе полученных зависимостей.

1. Аналитическая часть

В данном разделе будут рассмотрены теоретические основы алгоритма обратной трассировки лучей.

1.1. Алгоритм обратной трассировки лучей

Алгоритм обратной трассировки лучей работает в пространстве изображения [1]. Предполагается, что сцена уже преобразована в это пространство.

Наблюдатель находится на положительной полуоси OZ. Картинная плоскость, т.е. растр, перпендикулярна оси OZ. Каждый луч проходит через пиксель растра до сцены. Траектория каждого луча отслеживается, чтобы определить, какие именно объекты сцены, если таковые существуют, пересекаются с данным лучом. Необходимо проверить пересечение каждого объекта сцены с каждым лучом. Пересечение с максимальным значением z представляет видимую поверхность для данного пикселя.

Для дальнейшего определения цвета пикселя 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 (камеры) не найдено объектов, с которыми он пересекается, пиксел закрашивается цветом фона.

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

Метод прямой трассировки предполагает построение траекторий лучей от всех источников освещения ко всем точкам всех объектов сцены. Тогда достаточно часто будут выполняться расчеты для лучей, которые не попадут в камеру, результаты которых не будут учтены. Поэтому данный вариант алгоритма считается неэффективным.

Алгоритм 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. Это позволит ускорить синтез изображения.

1.2. Алгоритм обратной трассировки лучей в однопоточной реализации

При работе с одним потоком при реализации алгоритма трассировки лучей осуществляется обход всех пикселей растра изображения. В каждый пиксель испускается луч из положения наблюдателя, и соответствующий пиксель закрашивается рассчитанным цветом.

Однопоточная реализация алгоритма работает достаточно долго, так как требуется произвести большое количество вычислений: поиск пересечений со всеми объектами сцены луча наблюдателя на каждом пикселе, определение теней, рекурсивный расчёт траектории отражённого и преломлённого лучей и т.д.

1.3. Алгоритм обратной трассировки лучей в многопоточной реализации

При работе с одном потоком при реализации алгоритма трассировки лучей растр делится на участки (окна), размер которых равен:

$$window = \frac{size}{numThreads}, \tag{1.1}$$

где size — общее количество пикселов растра, а numThreads — количество выделяемых потоков.

Создаётся специальный массив, хранящий последний входящий в окно пиксел для каждого выделенного участка. Размер всех окон одинаков, кроме последнего, границей которого всегда является последний (крайний правый нижний) пиксел растра.

При проходе по окну все строки, кроме последней для данного окна, обрабатываются целиком, а последняя — до указанной во вспомогательном массиве крайней координаты обрабатываемых пикселов по ширине растра.

Каждое окно обрабатывается так же, как и весь растр при однопоточной реализации алгоритма, но за счёт параллельности вычислений алгоритм работает быстрее.

2. Конструкторская часть

В данном разделе будут представлены схемы реализации алгоритма обратной трассировки лучей, его однопоточной и многопоточной реализаций.

2.1. Разработка алгоритма обратной трассировки лучей

На рисунке 2.1 представлена схема реализации алгоритма обратной трассировки лучей.

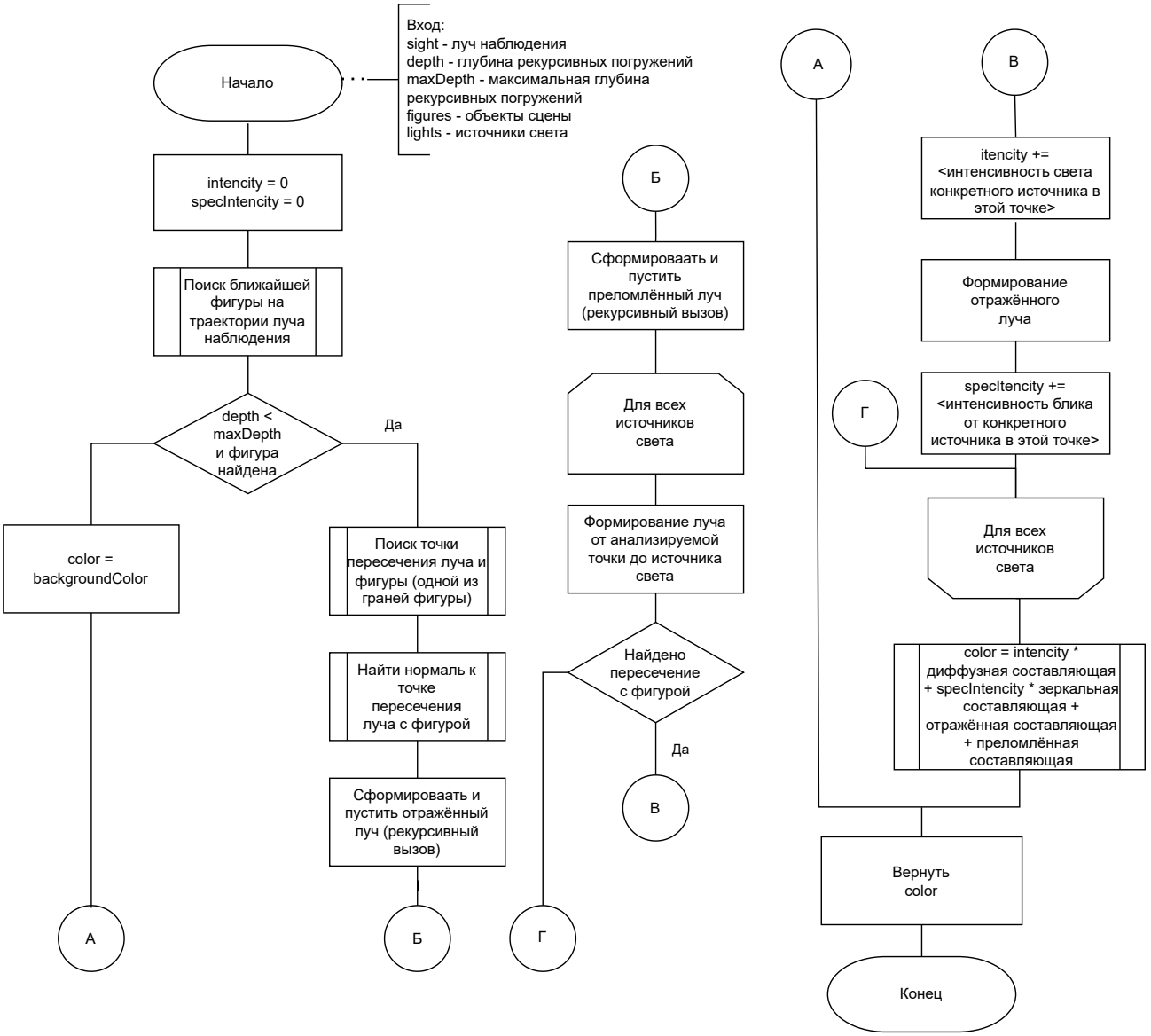

Рисунок 2.1 — Схема реализации алгоритма обратной трассировки лучей

2.2. Разработка последовательного алгоритма обратной трассировки лучей

На рисунке 2.2 представлена схема последовательного алгоритма обратной трассировки лучей.

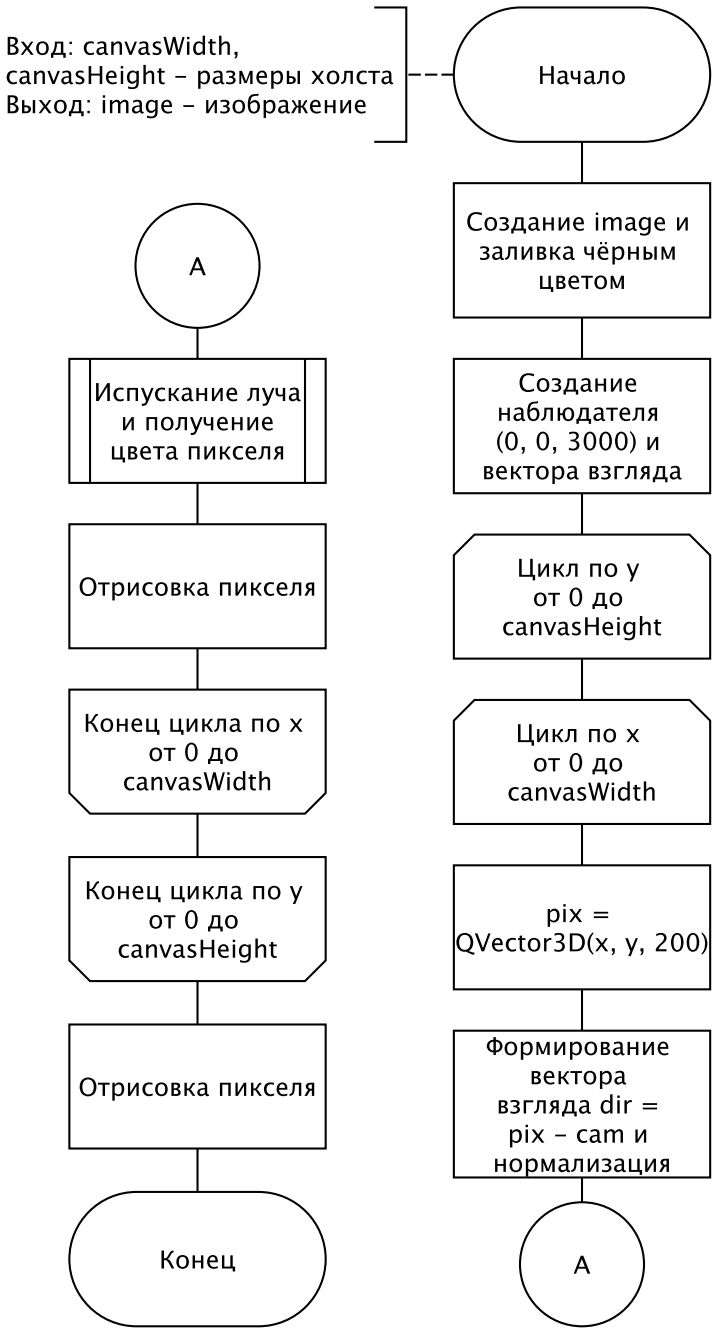

Рисунок 2.2 — Схема последовательного алгоритма обратной трассировки лучей

2.3. Разработка параллельного алгоритма обратной трассировки лучей

На рисунке 2.3 представлена схема параллельного алгоритма обратной трассировки лучей.

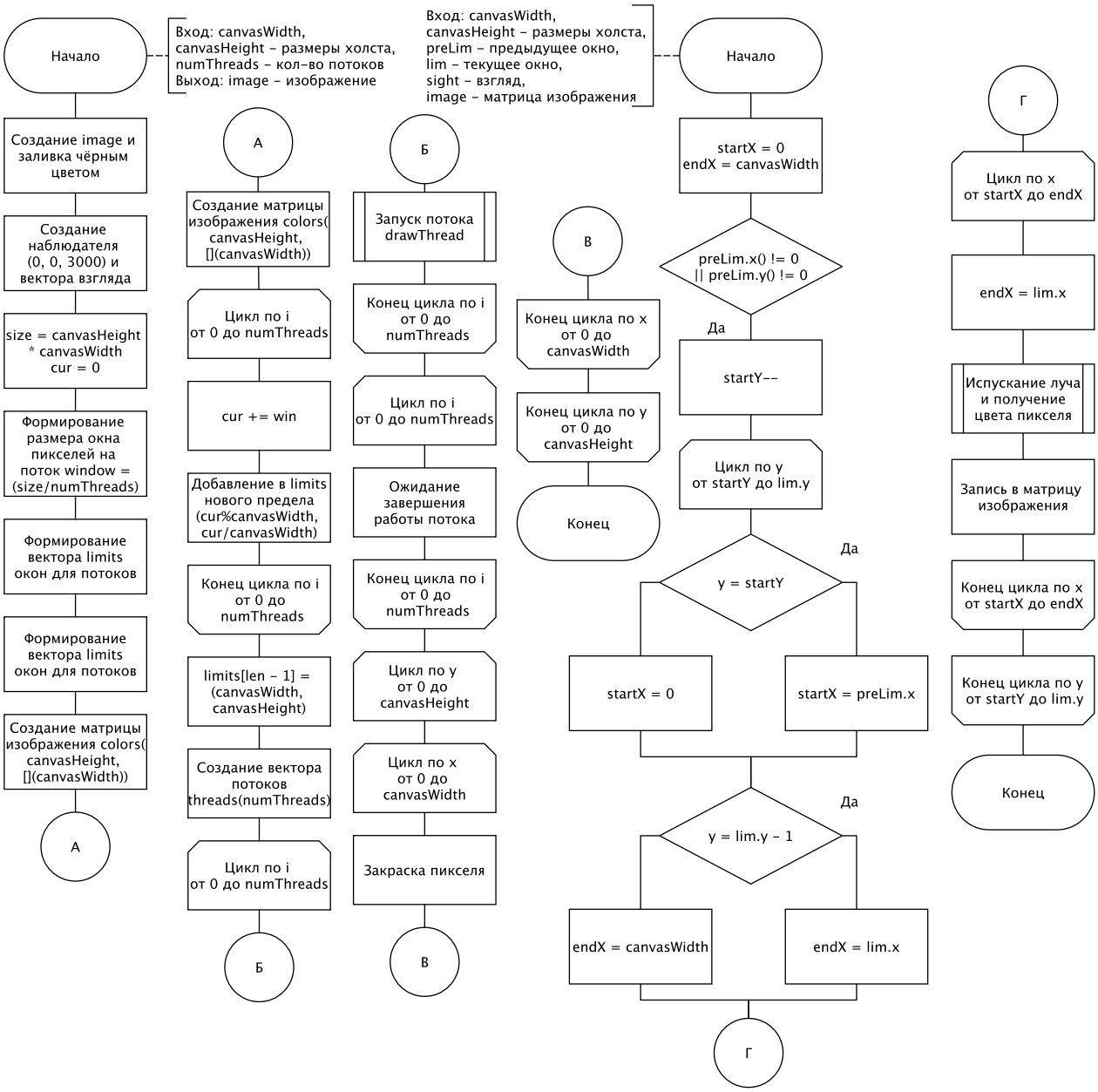

Рисунок 2.3 — Схема параллельного алгоритма обратной трассировки лучей

3. Технологическая часть

В данном разделе будет представлена реализация алгоритма обратной трассировки лучей. Также будут указаны требования к ПО, средства реализации алгоритмов и результаты проведённого тестирования программы.

3.1. Требования к ПО

Для автоматизации работы программы и использования сценария необходимо реализовать передачу в программу аргументов командной строки и их распознавание самой программой. Были предусмотрены следующие флаги:

- флаг «-d» - передача в программу значения глубины рекурсивных погружений, за флагом следует положительное целочисленное значение;
- флаги «-i», «-o» - передача в программу имени входного или выходного файла, за флагом следует строка с именем файла;
- флаг «-n» - передача в программу количества выделяемых потоков;
- флаг «-m» - указание программе о том, что проводятся замеры, при этом обязательно наличие параметров «-d» и «-i», а «-o» становится необязательным, так как полученное изображение в данном случае не используется;
- 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.

3.2. Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования $C++$ [3]. Данный выбор обусловлен наличием инструментов для реализации многопоточности. 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, obj, mtl, как наиболее широко применяемые в сфере компьютерной графики.

3.3. Реализация алгоритмов

В листингах 3.1 – 3.3 представлена реализация алгоритма обратной трассировки лучей. В листингах 3.4 – 3.7 представлены реализации однопоточного и многопоточного синтеза изображения.

Листинг 3.1 — Листинг функции реализации алгоритма обратной трассировки лучей (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 || !std::get<0>(res))
        return QColor(0, 0, 0);

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

    QVector3D reflectDir = reflect(dir, norm).normalized();
    sight_t sightRefl = {
        .cam = intersection,
        .dir = reflectDir
    };
    QColor reflectCol = castRay(sightRefl, depth + 1);

    QVector3D refractDir = refract(dir, norm,
    figures[closest]->getMaterial().getRefCoef()).normalized();
    sight_t sightRefr = {
        .cam = intersection,
        .dir = refractDir
    };
```

Листинг 3.2 — Листинг функции реализации алгоритма обратной трассировки лучей (продолжение листинга 3.1)

```
    QColor refractCol = castRay(sightRefr, depth + 1);

    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 = {
            .cam = 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);
            mirrored = mirrored.normalized();
            double angle = QVector3D::dotProduct(mirrored, dir * (-1));
            specIntensity += powf(std::max(0.0, angle),
            figures[closest]->getMaterial().getSpecCoef()) *
            lightSources[k].getIntensity();
        }
    }
```

Листинг 3.3 — Листинг функции реализации алгоритма обратной трассировки лучей (окончание листинга 3.2)

```
    QColor diffColor = figures[closest]->getMaterial().getDiffColor();
    QColor specColor = figures[closest]->getMaterial().getSpecColor();

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

    return color;
}
```

Листинг 3.4 — Листинг функции реализации однопоточного синтеза изображения

```
std::shared_ptr<QImage> Drawing::drawFigures()
{
    std::shared_ptr<QImage> image = std::make_shared<QImage>(canvasWidth,
    canvasHeight, QImage::Format_RGB32);
    image->fill(Qt::black);
    QVector3D cam = QVector3D(0, 0, 3000);
    sight_t sight = {
        .cam = cam
    };

    for (int y = 0; y < canvasHeight; y++)
        for (int x = 0; x < canvasWidth; x++) {
            QVector3D pix = QVector3D(x, y, 200);
            QVector3D dir = (pix - cam).normalized();
            sight.dir = dir;
            QColor refColor = castRay(sight, 0);
            image->setPixel(x, y, qRgb(refColor.red(), refColor.green(),
            refColor.blue()));
        }

    return image;
}
```

Листинг 3.5 — Листинг функции реализации многопоточного синтеза изображения (начало)

```
std::shared_ptr<QImage> Drawing::drawFigures(int numThreads)
{
    std::shared_ptr<QImage> image = std::make_shared<QImage>(canvasWidth,
    canvasHeight, QImage::Format_RGB32);
    image->fill(Qt::black);

    QVector3D cam = QVector3D(0, 0, 3000);
    sight_t sight = {
        .cam = cam
    };

    int size = canvasHeight * canvasWidth;
    int cur = 0;
    int win = (size / numThreads);
    std::vector<QVector3D> limits;
    limits.push_back(QVector3D(0,0,200));

    std::vector<std::vector<uint>> colors(canvasHeight,
    std::vector<uint>(canvasWidth, 0));

    if (numThreads == 0) {
        limits.push_back(QVector3D(canvasWidth, canvasHeight, 200));
        drawThread(limits[0], limits[1], sight, colors);
    } else {
        for (int i = 0; i < numThreads; i++) {
            cur += win;
            limits.push_back(QVector3D(cur % canvasWidth,
            cur / canvasWidth, 200));
        }
        limits[limits.size() - 1] = QVector3D(canvasWidth,
        canvasHeight, 200);
```

Листинг 3.6 — Листинг функции реализации многопоточного синтеза изображения (продолжение листинга 3.5)

```
        std::vector<std::thread> threads(numThreads);

        for (int i = 0; i < numThreads; i++) {
            threads[i] = std::thread(&Drawing::drawThread, this,
            std::ref(limits[i]), std::ref(limits[i+1]), sight,
            std::ref(colors));
        }
        for (int i = 0; i < numThreads; i++)
            threads[i].join();
    }

    for (int y = 0; y < canvasHeight; y++) {
        for (int x = 0; x < canvasWidth; x++) {
            image->setPixel(x, y, colors[y][x]);
        }
    }

    return image;
}
```

Листинг 3.7 — Листинг функции реализации многопоточного синтеза изображения (продолжение листинга 3.6)

```
void Drawing::drawThread(QVector3D &preLim, QVector3D &lim, sight_t sight,
std::vector<std::vector<uint>> &image)
{
    int startX = 0, endX = canvasWidth;

    int startY = preLim.y();
    if (preLim.x() != 0 || preLim.y() != 0) {
        startY--;
    }
```

Листинг 3.8 — Листинг функции реализации многопоточного синтеза изображения (продолжение листинга 3.7)

```
    for (int y = startY; y < lim.y(); y++)
    {
        if (y == startY) {
            startX = preLim.x();
        } else {
            startX = 0;
        }

        if (y == lim.y() - 1) {
            endX = lim.x();
        } else {
            endX = canvasWidth;
        }

        for (int x = startX; x < endX; x++) {
            QVector3D pix = QVector3D(x, y, 200);
            QVector3D dir = (pix - sight.cam).normalized();

            sight.dir = dir;
            QColor refColor = castRay(sight, 0);
            image[y][x] = qRgb(refColor.red(), refColor.green(),
            refColor.blue());
        }
    }
}
```

3.4. Тестирование

На рисунках 3.1 — 3.3 представлены тесты для алгоритма обратной трассировки лучей. Тестирование проводилось по проверке корректности получаемых изображений, по методологии белого ящика. Для передачи сведений об объектах сцены в программу использовались файлы с расширением *txt*, *obj*, *mtl*. Все тесты пройдены успешно.

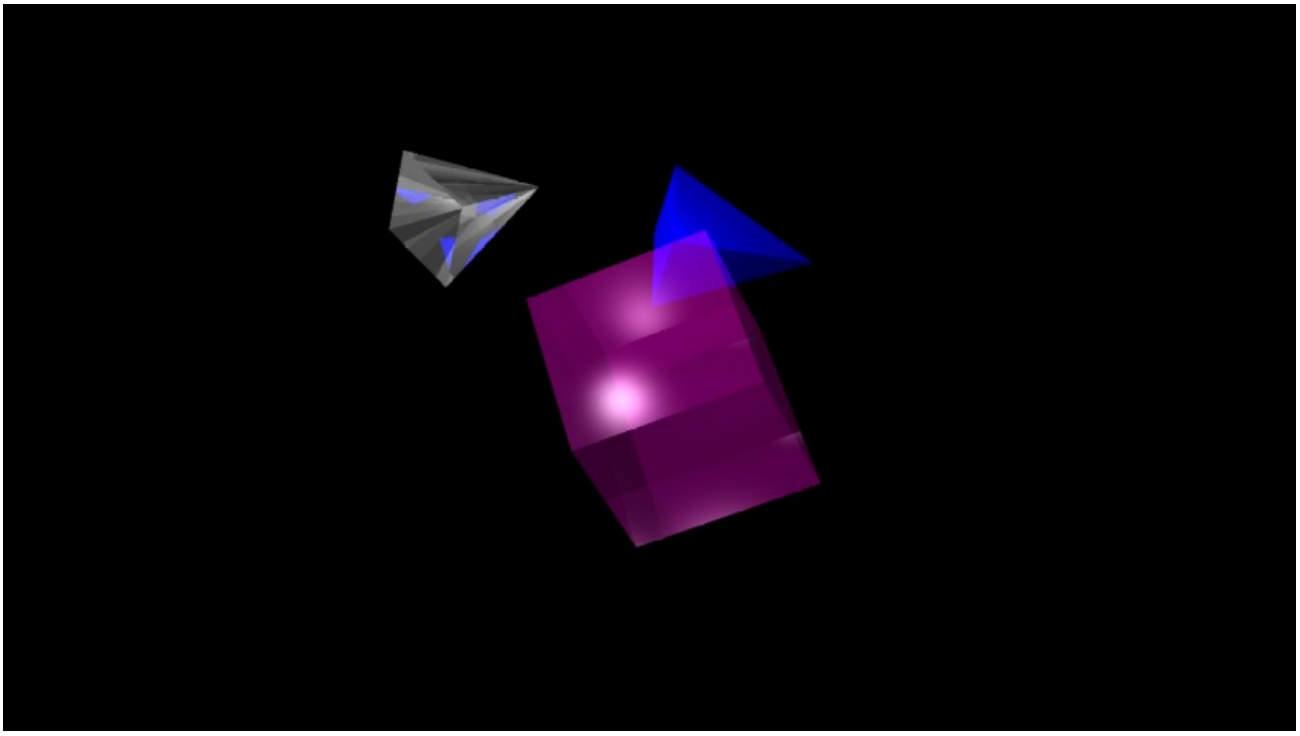

Рисунок 3.1 — Изображение-тест №1

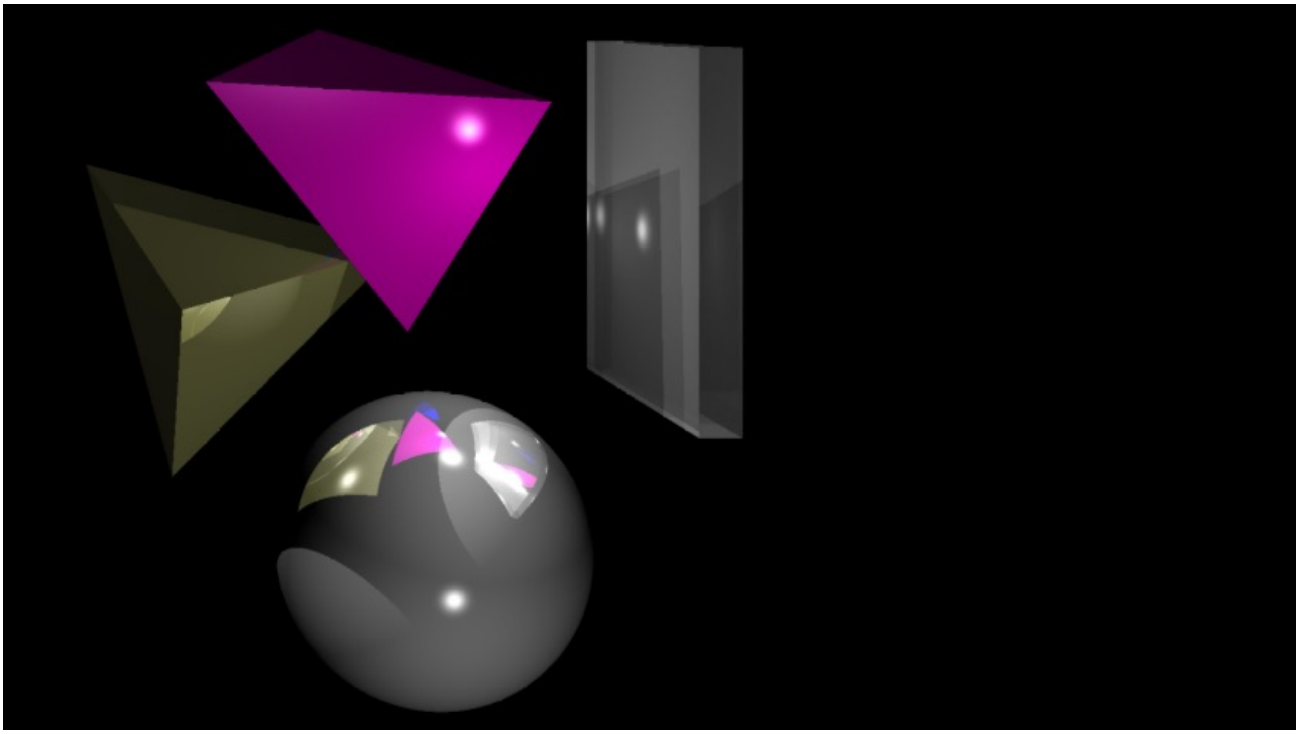

Рисунок 3.2 — Изображение-тест №2

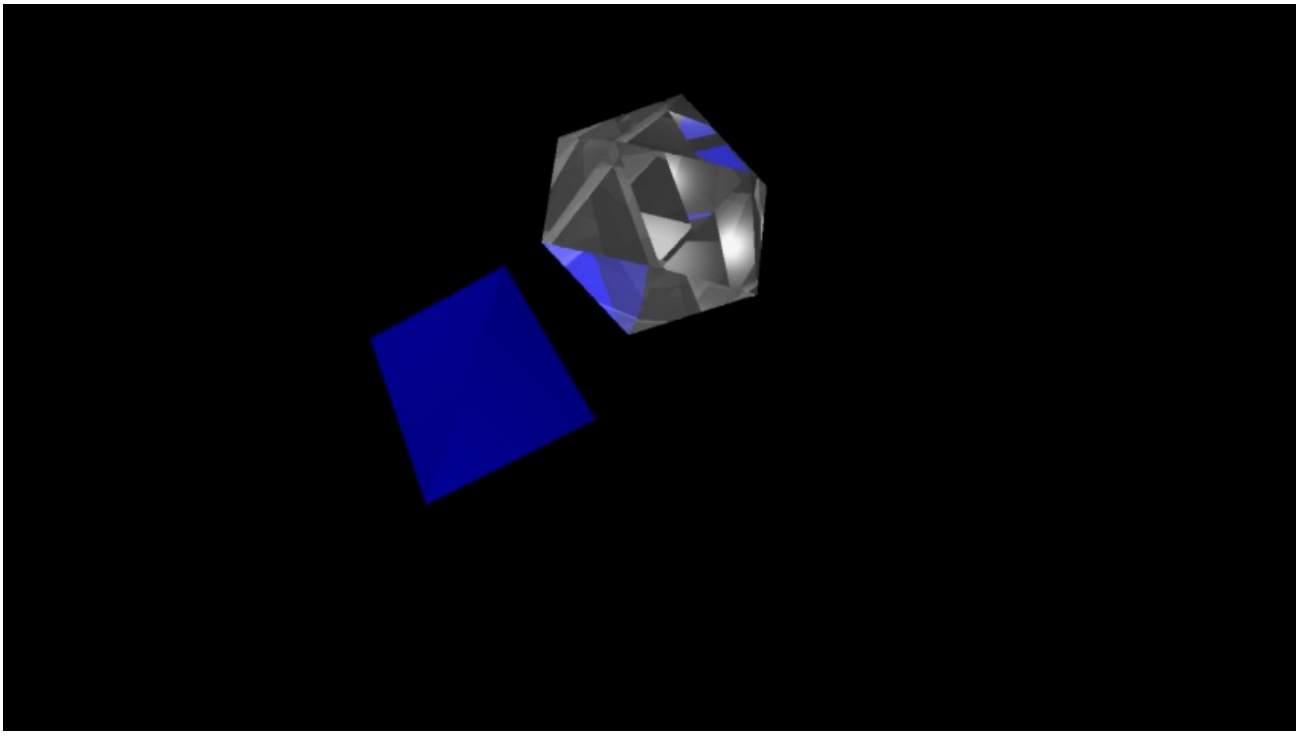

Рисунок 3.3 — Изображение-тест №3

4. Экспериментальная часть

В данном разделе описаны проведённые замеры и представлены результаты исследования. Также будут уточнены характеристики устройства, на котором проводились замеры.

4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени выполнения [4]:

- операционная система macOS Monterey 12.4;
- 8 ГБ оперативной памяти;
- процессор Apple M2 (базовая частота — 2400 МГц, но поддержка технологии Turbo Boost позволяет достигать частоты в 3500 МГц [5]).

4.2. Измерение времени выполнения реализаций алгоритма

Время работы алгоритма обратной трассировки лучей замерялось с помощью класса $std :: chrono :: system\_clock$ [6], который представляет реальное время. В исследуемых изображениях растр имеет одинаковое количество пикселей в обоих плоских измерениях.

Результаты замеров времени выполнения (в мс) приведены в таблице 4.1. замеры проводились до значения количества выделяемых потоков равного 32, так как уже на этом значении происходит замедление работы алгоритма. 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 новый поток. На рисунке 4.1 приведён график, отображающий зависимость времени работы реализации алгоритма от количества выделяемых потоков. Выполнялась визуализация сцены из 3 объектов, представленной на рис. 3.2. В замерах использовались изображения размером от 128×128 пикселов до 512×512 пикселов.

Таблица 4.1 — Таблица времени выполнения реализации алгоритма (мс)

Размер изображения (в пикселях)	Количество потоков						
	0	1	2	4	8	18	32
128	187.727	189.169	96.128	54.11	33.404	35.011	63.605
256	763.037	774.614	390.659	203.641	122.011	124.56	231.414
384	1732.54	1728.67	886.806	466.111	298.466	299.978	508.959
512	8140.94	8145.62	4146.27	3424.31	2958.07	2023.53	2375.9

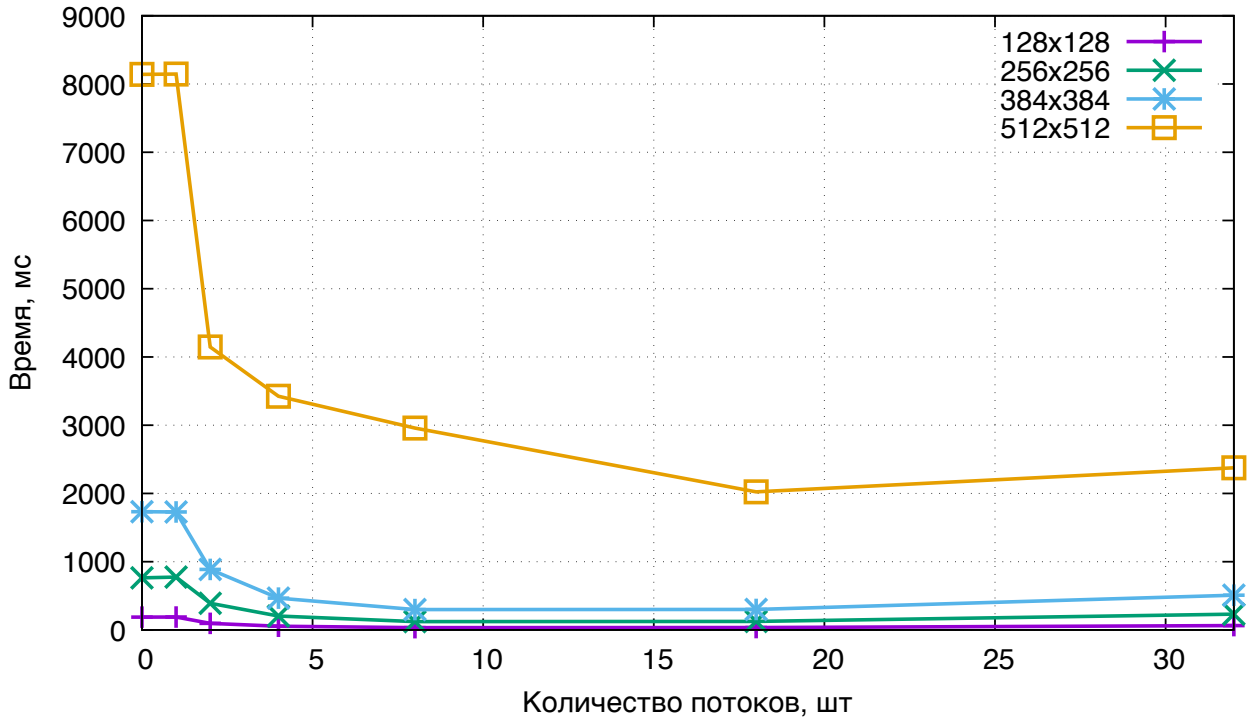

Рисунок 4.1 — Зависимость времени работы алгоритма от количества выделяемых потоков

Заключение

В результате проведения замеров были сформулированы нижеперечисленные выводы.

Наибольшую эффективность работы алгоритм достигает при выделении 8 потоков, так как это соответствует числу логических ядер процессора устройства, на котором проводились замеры (8 ядер). Если потоков выделялось больше, в большинстве случаев алгоритм работал медленнее, чем на 8 ядрах. Например, при 32 потоках при размере изображения 512×512 алгоритм работал в 1.17 раз медленнее, чем при 18 потоках. Также все алгоритмы работают дольше при выделении 1 потока, чем при отсутствии вспомогательных потоков, так как требуется время на выделение такового. В среднем разница составляла до 10 мс.

При увеличении размера изображения для рендера время работы алгоритма росло. Например, на 4 потоках при изображении размером 128×128 алгоритм работал в 8,6 раз быстрее, чем на размере изображения 384×384.

В среднем для любого размера изображения алгоритма работал на 8 потоках в 5 – 6 раз быстрее, чем при однопоточной реализации. Время работы реализации алгоритма постепенно снижалось при повышении числа потоков до 8.

Таким образом, рекомендуется использовать число потоков, равное числу логических процессоров (ядер).

В ходе выполнения лабораторной работы была достигнута поставленная цель: были получены навыки реализации многопоточности для программного продукта, было проведено сравнение быстродействия для однопоточной и многопоточной реализаций алгоритма обратной трассировки лучей.

В процессе выполнения лабораторной работы были также реализованы все поставленные задачи, а именно:

1) был изучен алгоритм обратной трассировки лучей;

2) были разработаны последовательная и параллельная версии алгоритма обратной трассировки лучей;

3) была проведена разработка схем данных алгоритмов;

4) была выполнена программная реализация данного алгоритма с использованием одного потока;

5) была выполнена программная реализация данного алгоритма с использованием вспомогательных потоков;

6) был проведён замер времени работы (в мс) данных реализаций алгоритма;

7) была получена зависимость замеряемых величин от количества потоков;

8) был проведён сравнительный анализ данных реализаций алгоритма на основе полученных зависимостей.

Список использованных источников

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

[2] Трассировка лучей в реальном времени [Электронный ресурс]. Режим доступа: *https://www.ixbt.com/3dv/directx-raytracing.html* (дата обращения: 20.09.2022).

[3] 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).

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

[5] *AppleM2* [Электронный ресурс]. Режим доступа: *https://www.notebookcheck.net/Apple-M2-Processor-Benchmarks-and-Specs.632312.0.html* (дата обращения: 10.10.2022).

[6] 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).