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 Варламова Е. А.

Группа ИУ7-51Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

Поток, или поток выполнения, - базовая упорядоченная последовательность инструкций, которые могут быть переданы или обработаны одним ядром процессора. Вычисления разбиваются на части, за каждую из которых отвечает отдельный поток. Если в системе больше одного процессора, то использование многопоточности фактически позволяет выполнять несколько действий одновременно.

Многие алгоритмы (операции над матрицами, алгоритмы компьютерной графики) допускают многопоточную реализацию. Поэтому **целью** данной работы является сравнить производительность обычной и многопоточной реализаций алгоритма трассировки лучей. Для достижения поставленной цели необходимо решить следующие задачи.

1. Выбрать метод распараллеливания алгоритма трассировки лучей.
2. Реализовать однопоточный алгоритм трассировки лучей.
3. Реализовать многопоточный алгоритм трассировки лучей.
4. Протестировать реализованные алгоритмы.
5. Провести сравнительный анализ алгоритмов по времени выполнения в зависимости от количества потоков.

1 Аналитическая часть

1.1 Трассировка лучей

Трассировка лучей – метод синтеза компьютерных изображений. Основная идея, лежащая в основе этого метода заключается в том, что наблюдатель видит любой объект посредством испускаемого неким источником света, который падает на этот объект и затем доходит до наблюдателя. Если проследить за лучами, испущенными источником света, то лишь немногие из них дойдут до наблюдателя. Поэтому предлагается отслеживать лучи в обратном направлении, то есть от наблюдателя к объекту[1]. В алгоритме предполагается, что наблюдатель находится на оси z, а картинная плоскость (растр) перпендикулярна ей. Для каждого пиксела в итоговом изображении математический луч испускается из положения наблюдателя через растр в направлении к объектам сцены. Тогда для каждого луча необходимо найти все пересечения с объектами сцены и выбрать ближайшее к наблюдателю. Это пересечение и будет являться видимой наблюдателю частью объекта. Для учёта освещения, отражения, прозрачности, теней и блеска объектов используется тот же алгоритм трассировки, но лучи испускаются не из положения наблюдателя, а из точки на поверхности объектов.

Алгоритм трассировки лучей требует большого количества вычислений, поскольку он предполагает поиск пересечений всех объектов сцены со всеми лучами, количество которых равно размеру растра. Поэтому время синтеза изображения оказывается очень большим. Однако из описания алгоритма видно, что вычисление цвета каждого пиксела может быть выполнено параллельно. Такая оптимизация позволит значительно ускорить синтез изображения.

1.2 Вывод

В данном разделе был расссмотрен алгоритм трассировки лучей и выбран метод распараллеливания вычислений.

2 Конструкторская часть

2.1 Разработка алгоритмов

На рисунках 2.1 и 2.2 приведены схемы однопоточного и многопоточного синтеза изображения, в основе которого лежит алгоритм трассировки лучей. На рисунке 2.3 приведена схема алгоритма трассировки лучей.

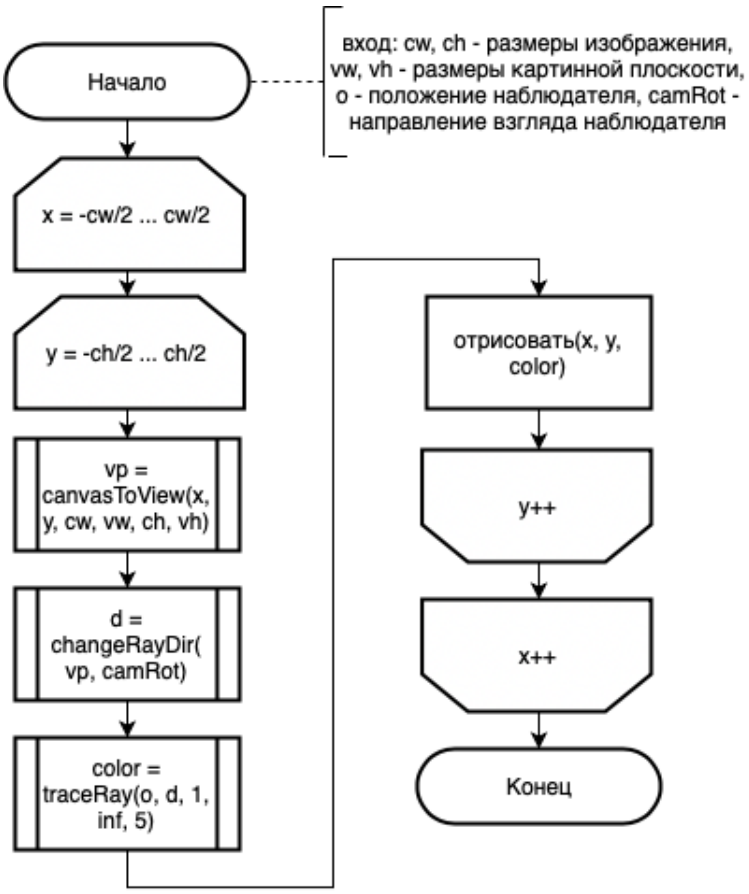

Рис. 2.1: Схема однопоточного синтеза изображения

2.2 Вывод

На основе анализа алгоритма трассировки лучей были построены схемы однопоточного и многопоточного синтеза изображения.

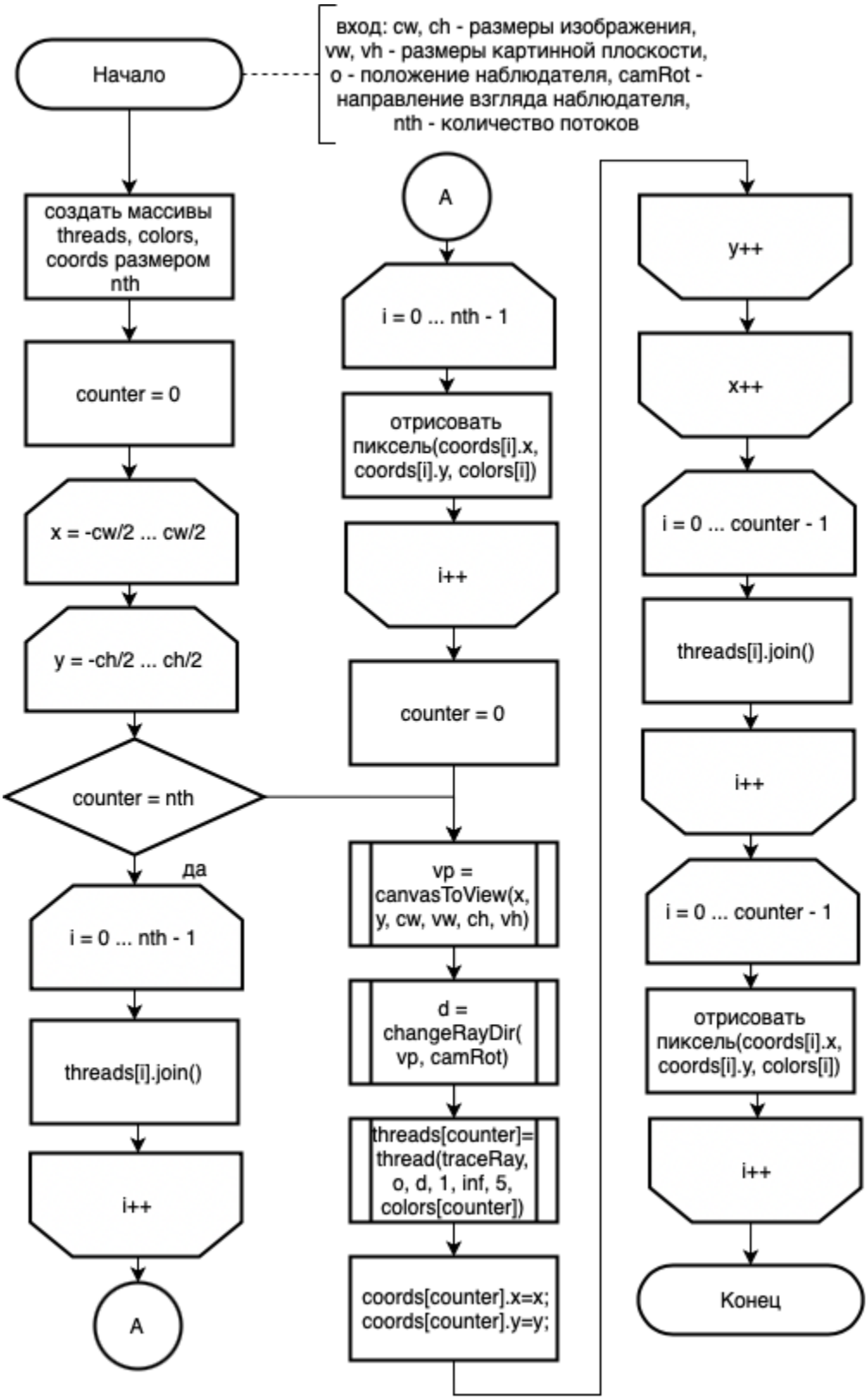

Рис. 2.2: Схема многопоточного синтеза изображения

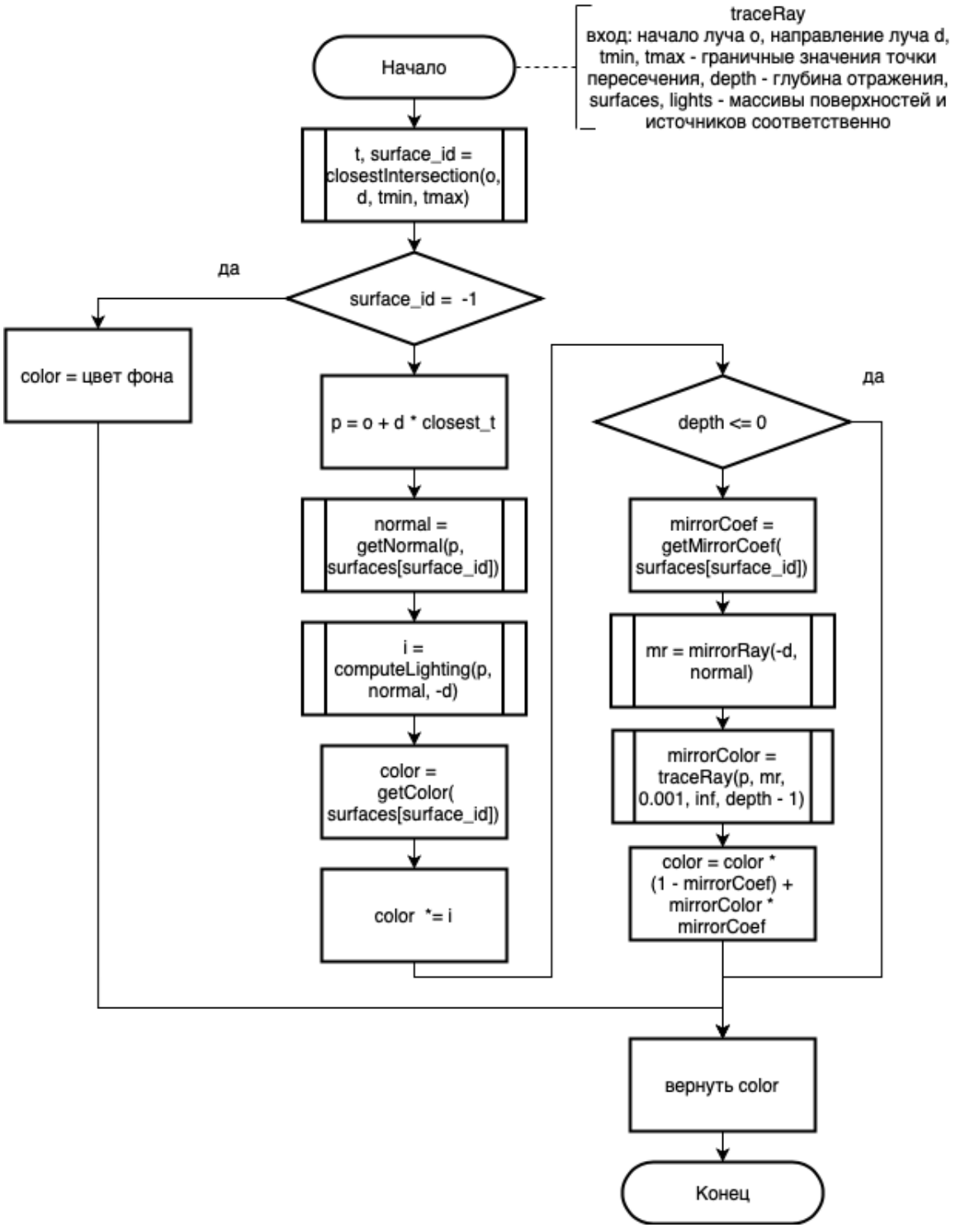

Рис. 2.3: Схема алгоритма трассировки лучей

3 | Технологическая часть

В данном разделе приведены средства реализации и листинги кода.

3.1 Средства реализации

Для реализации был выбран язык программирования C++. Данный выбор обусловлен наличием инструментов для создания и эффективной работы с потоками. В качестве среды разработки была выбрана среда CLion.

3.2 Реализация алгоритмов

В листингах 3.1, 3.2 и 3.3 приведены реализации однопоточного, многопоточного синтеза изображения и алгоритма трассировки лучей соответственно.

Листинг 3.1: Однопоточный синтез изображения

```
void View::drawScene(int cw, int ch, double vw, double vh, Point &o, Matrix<
    double> &camera_rotation)
{
    int planeZ = 1;
    scene->clear();
    for (int x = -cw / 2; x <= cw / 2; x++)
    {
        for (int y = -ch / 2; y <= ch / 2; y++)
        {
            Point viewportPoint(canvas_to_viewport(x, cw, vw),
                                canvas_to_viewport(y, ch, vh),
                                planeZ);
            Point d = changeRayDirection(viewportPoint, camera_rotation);
            QColor color;
            rt.traceRay(o, d, 1, INT_MAX, 1, color);
            scene->addRect(x, -y, 1, 1, QPen(color));
        }
    }
}
```

Листинг 3.2: Многопоточный синтез изображения

```
void View::drawSceneMulti(int cw, int ch, double vw, double vh, Point &o,
    Matrix<double> &camera_rotation, int nth)
{
    int planeZ = 1;
    scene->clear();
    std::vector <std::thread > threads(nth);
    std::vector<QColor> colors(nth);
    std::vector<pair<int, int>> coords(nth);
    int counter = 0;
    for (int x = -cw / 2; x <= cw / 2; x++)
    {
        for (int y = -ch / 2; y <= ch / 2; y++)
        {
            if (counter == nth)
            {
                for (int i = 0; i < nth; ++i) {
                    threads[i].join();
                }
                for (int i = 0; i < nth; ++i)
                    scene->addRect(coords[i].first, coords[i].second, 1, 1,
                        QPen(colors[i]));
                counter = 0;
            }
            Point viewportPoint(canvas_to_viewport(x, cw, vw),
                                canvas_to_viewport(y, ch, vh),
                                planeZ);
            Point d = changeRayDirection(viewportPoint, camera_rotation);

            threads[counter] = std::thread(&RayTracer::traceRay, rt, o, d,
                1, INT_MAX, 1, std::ref(colors[counter]));
            coords[counter] = std::pair<int, int>(x, -y);
            counter++;
        }
    }
    for (int i = 0; i < counter; ++i) {
        threads[i].join();
    }
    for (int i = 0; i < counter; ++i)
        scene->addRect(coords[i].first, coords[i].second, 1, 1, QPen(colors[
            i]));
}
```

Листинг 3.3: Алгоритм трассировки лучей

```
QColor RayTracer::traceRay(Point o, Point d, double tmin, double tmax, int
    depth, QColor &tmp)
{
    int closest_surface;
    double closest_t;
    closestIntersection(o, d,  tmin, tmax, closest_surface, closest_t);
    if (closest_surface == -1) {
        tmp = QColor(0, 0, 0);
        return QColor(0, 0, 0);
    }
    Point p = o + d * closest_t;
    Point normal;
    surfaces[closest_surface]->getNormal(p, normal);

    double i = computeLighting(p, normal, -d, surfaces[closest_surface]->
       getShineCoef());
    QColor color = surfaces[closest_surface]->getColor();
    color = mulColor(color, i);

    double mirrorCoef = surfaces[closest_surface]->getMirrorCoef();
    if (!(depth <= 0 || mirrorCoef <= 0))
    {
        QColor mirrorColor = traceRay(p, mirrorRay(-d, normal), 0.001,
           INT_MAX, depth - 1, tmp);
        color = sumColor(mulColor(color, (1 - mirrorCoef)),
                         mulColor(mirrorColor, mirrorCoef));
    }

    tmp = color;
    return color;
}
```

3.3 Тестирование

При тестировании проверялась корректность изображений, полученных при работе программы с различными данными. Результаты представлены на 3.1 и 3.2.

Все тесты пройдены успешно.

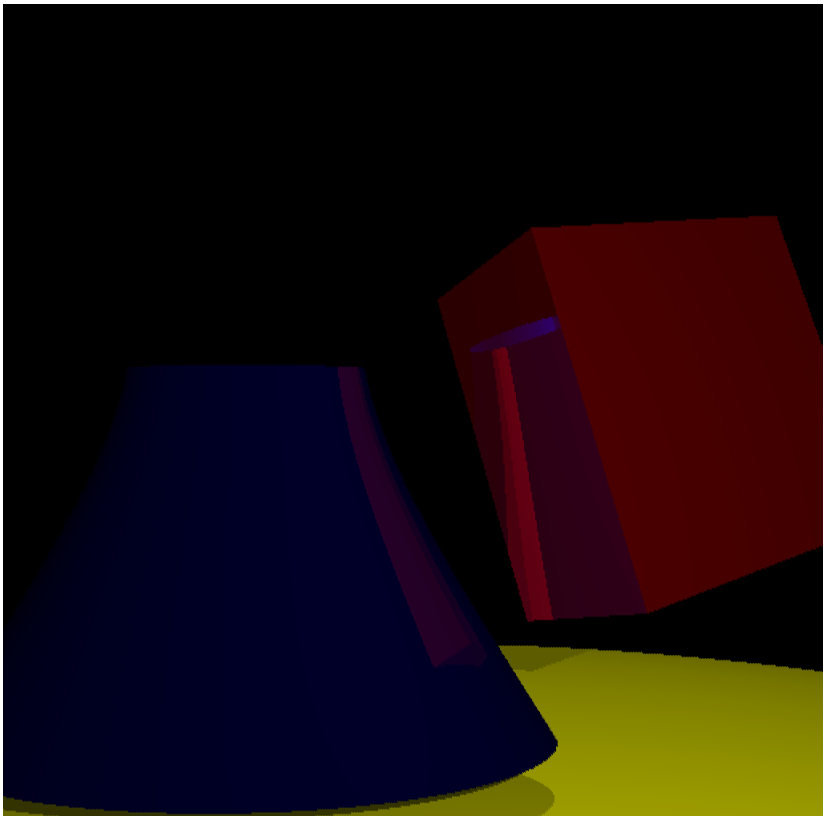

Рис. 3.1: Взаимное отражение

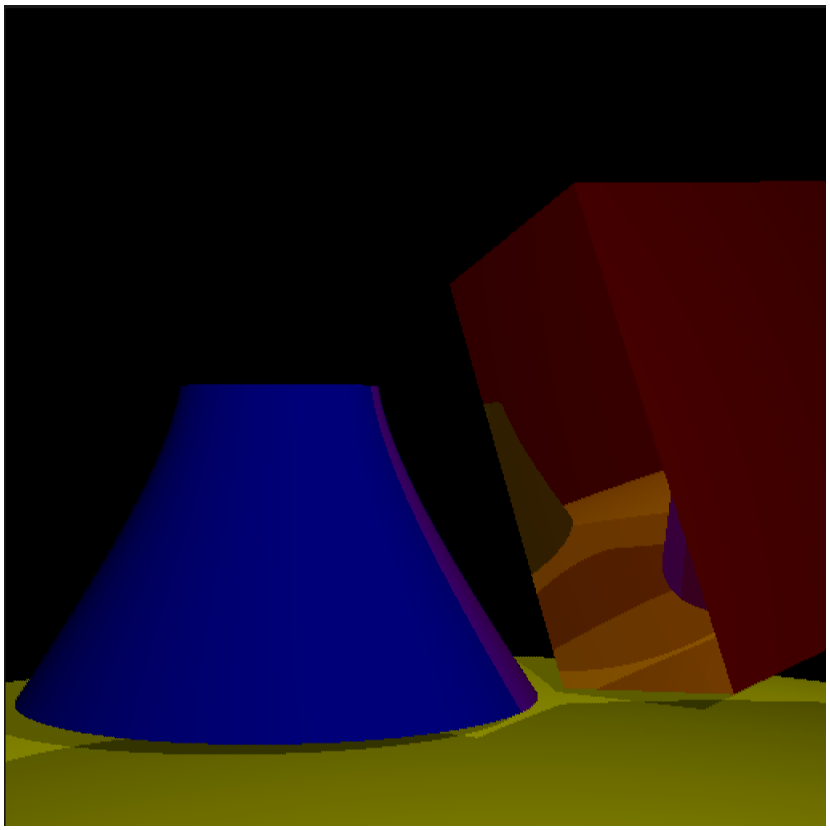

Рис. 3.2: Зелёная поверхность, закрытая синей, видна в отражении куба

3.4 Вывод

В данном разделе были разработаны исходные коды алгоритмов: однопоточный и многопоточный алгоритмы трассировки лучей.

4 | Исследовательская часть

4.1 Технические характеристики

Все нижепреведенные замеры времени проведены на процессоре: Intel Core i5, 1,4 GHz, четырёхъядерный, количество логических ядер - 8. Время работы алгоритмов было замерено с помощью time.h, функции clock, которая измеряет процессорное время [2].

4.2 Время выполнения алгоритма

В 4.2 приведено время выполнения алгоритма трассировки лучей при разных размерах изображения (от 100 * 100 до 500 * 500 пикселей) без распараллеливания и при разных количествах потоков (1, 2, 4, 8, 16, 32). На 4.1 приведён график, соответствующий данным в таблице. Время на графике и в таблице указано в мс.

Таблица 4.1: Таблица времени выполнения при разных размерах изображения и количествах потоков

Количество потоков	100 * 100	200 * 200	300 * 300	400 * 400	500 * 500
не расп-но	1614.33	6333.63	14180.2	25329.2	39221.8
1	1718.37	7531.45	16781.5	26634.1	40453.3
1	1614.33	6333.63	14180.2	25329.2	39221.8
2	1045.01	4143.79	9655.59	17172.8	25637.5
4	545.895	2226.87	5155.55	9462.02	15123.8
8	374.916	1617.05	3743.16	6907.58	10955.2
16	409.756	1752.07	4057.03	7311.09	11746.3
32	421.671	1805.8	4087.55	7360.05	11631.2

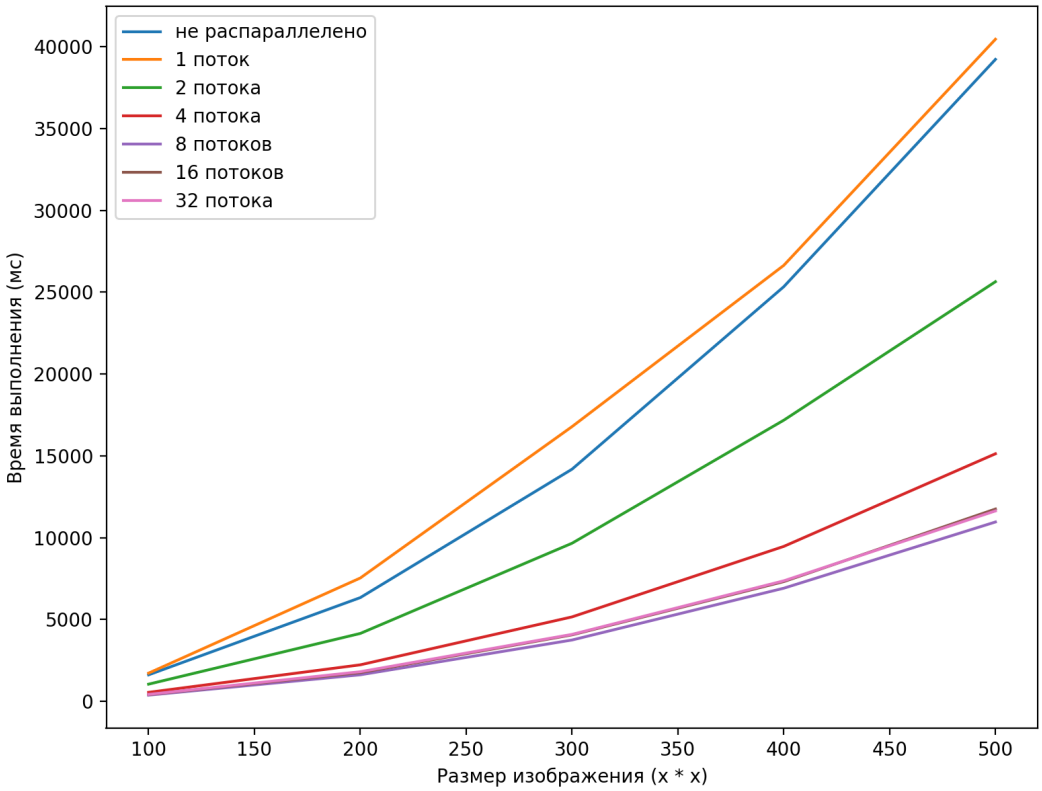

Рис. 4.1: Зависимость времени выполнения от размера изображения при разном числе потоков

4.3 Вывод

Наилучшее время параллельные алгоритмы показали при 8 потоках, что соответствует количеству логических ядер компьютера, на котором проводилось тестирование. На изображениях размером 500 на 500 пикселей, параллельный алгоритм с 8 потоками работает примерно в 3.6 раз быстрее однопоточной реализации. При количестве потоков большем восьми время выполнения ухудшается по сравнению с реализацией с восемью потоками. Таким образом, рекомендуется использовать число потоков, равное числу логических ядер.

Не распараллеленная реализация работает немного быстрее однопоточной, поскольку в однопоточной уходит время на создание потока.

Заключение

В рамках данной лабораторной работы были решены следующие задачи.

- Выбран метод распараллеливания алгоритма трассировки лучей.
- Реализован однопоточный алгоритм трассировки лучей.
- Реализован многопоточный алгоритм трассировки лучей.
- Протестированы реализованные алгоритмы.
- Проведён сравнительный анализ алгоритмов по времени выполнения в зависимости от количества потоков.

В результате исследования было выяснено, что параллельные реализации алгоритма трассировки лучей работают быстрее однопоточной реализации. Наиболее эффективны данные алгоритмы при количестве потоков, совпадающем с количеством логических ядер компьютера. Так, например, на изображениях размером 500 на 500, удалось улучшить время выполнения алгоритма в 3.6 раз (в сравнении с однопоточной реализацией).

Поставленная цель была достигнута.

Литература

1. Роджерс Д., Адамс Дж. Математические основы машинной графики. М.: Мир, 2001. 604 с.

2. Стандартная функция clock(), измеряющая процессорное время [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/chrono/c/clock. Дата обращения: 19.10.2021