Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Программирование на языках высокого уровня

Тема: Метод трассировки луча

Выполнил

студент гр. 3331506/70401 Козлов Д. А.

Преподаватель Ананьевский М. С.

« »_____2020 г.

Санкт-Петербург

2020 г.

Оглавление

1. Введение

1.1 Формулировка задачи, которую решает алгоритм

В компьютерной графике задача отрисовки реалистичного изображения является основной.

Решить её можно, опираясь на реальный физический мир: свет исходит из источника света (солнца, лампочки и т. д.), отражается от нескольких объектов и наконец достигает наших глаз. Можно попробовать симулировать путь каждого фотона, испущенного из симулированных источников света, но это будет невероятно затратно по времени и к тому же наблюдателя достигала бы лишь малая часть этих фотонов.

Вместо этого трассируют лучи «в обратном порядке» - начинают с луча, находящегося на камере (наблюдателе), пропускают его через точку в окне просмотра и продвигают далее, пока он не столкнётся с каким-нибудь объектом в сцене. Цвет света, проходящего через точку в окне, будет являться цветом объекта.

В данной работе рассматривается метод трассировки луча, который используется для простейшей задачи вычислительной геометрии - локализация точки в выпуклом многоугольнике путем испускания луча и подсчета пересечений со сторонами многоугольника. Таким образом, реализуется метод трассировки лучей в его «первом приближении».

1.2 Словесное описание алгоритма

Пусть заданы выпуклый многоугольник P, состоящий из n вершин, и точка A, необходимо определить расположение точки (внутри или снаружи многоугольника).

Алгоритм:

1. Построить горизонтальный луч из точки A в положительном направлении оси Ox – отрезок, соединяющий точки с координатами (A.x; A.y) и (+∞; A.y).

2. Выбрать одну сторону многоугольника.

3. Проверить пересекает ли луч выбранную сторону многоугольника.

4. Повторить п. 2 и 3 для всех сторон многоугольника.

5. Если количество пересечений равно 1, то точка внутри многоугольника. Иначе точка снаружи многоугольника.

Пункт 3 реализуется, основываясь на утверждении, что два отрезка пересекаются тогда и только тогда, когда концы одного отрезка лежат по разные стороны другого и наоборот.

Определить положение конца одного отрезка относительно другого отрезка можно при помощи векторного произведения. Пусть заданы три точки A, B и C. Предположим, что мы смотрим из точки A в точку B. Определить положение точки C можно с помощью векторного произведения векторов $\mathbf{a}$ = AB и $\mathbf{b}$ = BC, точнее с помощью z-координаты такого произведения, которая вычисляется по простой формуле $z = \mathbf{a}_x\mathbf{b}_y - \mathbf{a}_y\mathbf{b}_x$. Если $z > 0$, то искомый поворот левый, если $z < 0$ — то правый. Если же $z = 0$, то три точки лежат на одной прямой.

2. Реализация алгоритма

Алгоритм был реализован на языке C++.

Заголовочный файл проекта изображен на рисунке 1. Здесь описываются структура точки, и класс многоугольника, который включает в себя указатель на массив вершин, количество вершин, конструктор, деструктор и методы, реализующие трассировку луча.

```
#ifndef RAYTRACING_H
#define RAYTRACING_H

#include <iostream>

struct Point
{
    int x;
    int y;
};

class Polygon
{
public:
    Point* polygon;
    int number_of_vertexes;
public:
    Polygon(Point* array_of_vertexes, int size);
    Polygon(int number_of_vertexes);
    ~Polygon();
    bool is_point_inside_polygon(Point point);
private:
    int rotate(Point a, Point b, Point c);
    bool intersect(Point a, Point b, Point c, Point d);
};

#endif
```

Рисунок 1 – Заголовочный файл *raytracing.h*

Реализации конструктора и деструктора изображены на рисунке 2.

```
Polygon::Polygon(Point* array_of_vertexes, int size)
{
    polygon = array_of_vertexes;
    number_of_vertexes = size;
}

Polygon::~Polygon()
{
    delete[] polygon;
}
```

Рисунок 2 – Реализации конструктора и деструктора

Реализация методов алгоритма изображена на рисунке 3.

```
int Polygon::rotate(Point a, Point b, Point c)
{
    int z_ab_bc = (b.x - a.x) * (c.y - b.y) - (b.y - a.y) * (c.x - b.x);
    if (z_ab_bc > 0) return 1;
    if (z_ab_bc < 0) return -1;
    return 0;
}

bool Polygon::intersect(Point a, Point b, Point c, Point d)
{
    return (((rotate(a, b, c) * rotate(a, b, d)) < 0) && ((rotate(c, d, a) * rotate(c, d, b)) < 0));
}

bool Polygon::is_point_inside_polygon(Point point)
{
    int number_of_intersections = 0;
    Point inf;
    inf.x = INFINITY;
    inf.y = point.y;
    for (int vertex_1 = 0, vertex_2 = 1; vertex_1 < number_of_vertexes; vertex_1++, vertex_2++)
    {
        if (vertex_1 == number_of_vertexes - 1) vertex_2 = 0;
        if (intersect(polygon[vertex_1], polygon[vertex_2], point, inf))
            number_of_intersections++;
    }
    if (number_of_intersections == 1) return true;
    else return false;
}
```

Рисунок 3 – Реализация методов алгоритма.

3. Анализ алгоритма

3.1 Анализ сложности алгоритма

Очевидно, что в данном алгоритме отсутствуют наилучший и наихудший случаи.

Так как для локализации точки в выпуклом многоугольнике необходимо проверить пересечение луча со всеми сторонами многоугольника, то алгоритм является линейным по числу вершин многоугольника *n*.

Таким образом, сложность алгоритма можно оценить как O(*n*).

3.2 Численный анализ алгоритма

Проанализируем время выполнения алгоритма для многоугольников с разным количеством вершин *n*.

На рисунке 4 приведен график зависимости времени выполнения алгоритма от количества вершин многоугольника *n*.

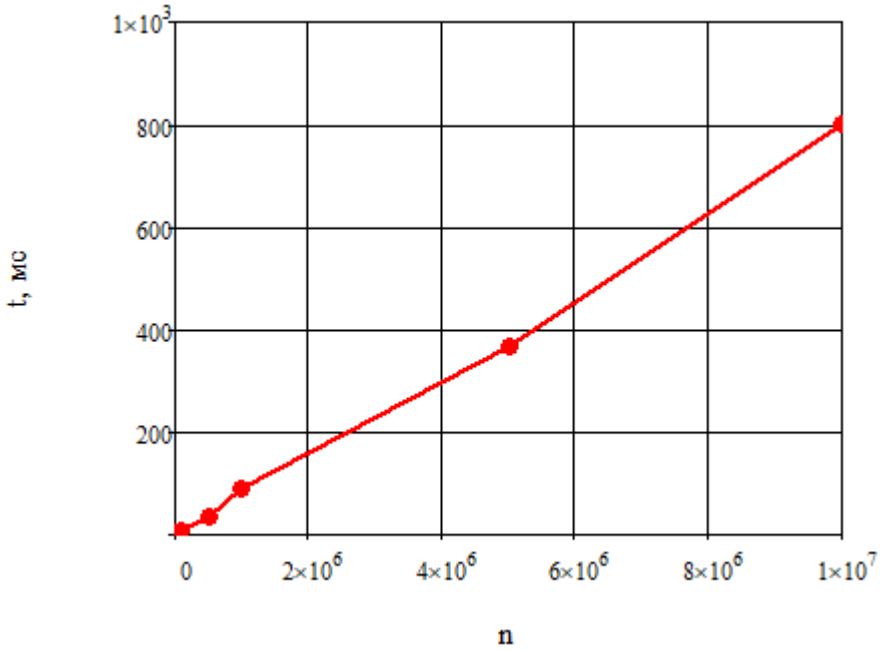

Рисунок 4 – Время выполнения алгоритма

Численные значения приведены в таблице 1.

Таблица 1 – Численный анализ

n	10	100	1000	10000	100000	500000	1000000	5000000	10000000
t, мс	≤1	≤1	≤1	≤1	8	35	91	369	801

Видно, что для n < 10000 время выполнения алгоритма незначительно, но в дальнейшем время возрастает пропорционально количеству вершин и при $n = 10000000$ алгоритм занимает уже ~ 1 секунду. Таким образом, основным недостатком данного алгоритма является низкая производительность.

4. Применение алгоритма

Как говорилось ранее, реализация алгоритма, представленная в данной работе, применяется в вычислительной геометрии для локализации точки. Но у данного алгоритма существует большое количество других более трудоемких реализаций, которые используются в различных областях:

— построение изображений трёхмерных моделей в компьютерных программах;

— трассировка лучей в компьютерных играх – создание реалистичного освещения, отражений и теней, обеспечивающее высокий уровень реализма.

5. Заключение

В ходе выполнения работы были рассмотрены принцип работы алгоритма трассировки луча, его реализация на языке программирования C++, проведен анализ сложности и численный анализ алгоритма, а также рассмотрено области его применения.

Таким образом, алгоритм трассировки лучей выводит реалистичность компьютерной графики на новый, отличающийся высоким качеством, уровень. Но серьёзным недостатком метода является низкая производительность.

Список литературы

1. Дмитрий Чеканов. Метод трассировки лучей против растеризации: новое поколение качества графики?. Tom's Hardware (7 сентября 2009).

2. Лев Дымченко. Проблемы трассировки лучей — из будущего в реальное время 6. Мир nVidia (13 декабря 2009).