ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовой проект
по дисциплине «Объектно-ориентированное программирование»
«Ray Casting алгоритм»

Пояснительная записка

Выполнил
студент гр.
3331506/10401 _____ Арсланов М. И.
(подпись)

Работу принял _____ Ананьевский М.С.
(подпись)

Санкт-Петербург
2024 г.

Оглавление

Техническое задание

Реализовать метод рендеринга компьютерной графики «бросание лучей» (*ray casting*). Для вывода изображения, использовать терминал xterm в Ubuntu 22.04.

1. Введение

Рендеринг или **отрисовка** — термин в компьютерной графике, обозначающий процесс получения изображения по модели с помощью компьютерной программы.

Методы рендеринга:

- Растеризация (*rasterization*) совместно с методом сканирования строк (Scanline rendering). Визуализация производится проецированием объектов сцены на экран без рассмотрения эффекта перспективы относительно наблюдателя.
- Ray casting (“*бросание лучей*”). Сцена рассматривается, как наблюдаемая из определённой точки. Из точки наблюдения на объекты сцены направляются лучи, с помощью которых определяется цвет пиксела на двумерном экране. При этом лучи прекращают своё распространение (в отличие от метода обратного трассирования), когда достигают любого объекта сцены либо её фона. Возможно использование каких-либо очень простых способов добавления оптических эффектов. Эффект перспективы получается естественным образом в случае, когда бросаемые лучи запускаются под углом, зависящим от положения пикселя на экране и максимального угла обзора камеры.
- Трассировка лучей (*ray tracing*) похожа на метод бросания лучей. Из точки наблюдения на объекты сцены направляются лучи, с помощью которых определяется цвет пиксела на двумерном экране. Но при этом луч не прекращает своё распространение, а разделяется на три луча-компонента, каждый из которых вносит свой вклад в цвет пикселя на двумерном экране: отражённый, теневой и преломлённый. Количество таких компонентов определяет глубину трассировки и влияет на качество и фотореалистичность изображения. Благодаря своим концептуальным особенностям, метод позволяет получить очень фотореалистичные изображения, однако из-за большой ресурсоёмкости процесс визуализации занимает значительное время.

Где может использоваться рейкастинг?

Рейкастинг можно использовать по-разному, особенно в трёхмерном пространстве. Я выделю три наиболее важных по моему мнению применения, часто встречающихся в игровых 2D-движках. Самой известной

игрой, использующей эту технику, является Wolfenstein 3D. Лучи в ней трассировались для определения ближайших объектов, а их расстояние от позиции игрока использовалось для правильного масштабирования.

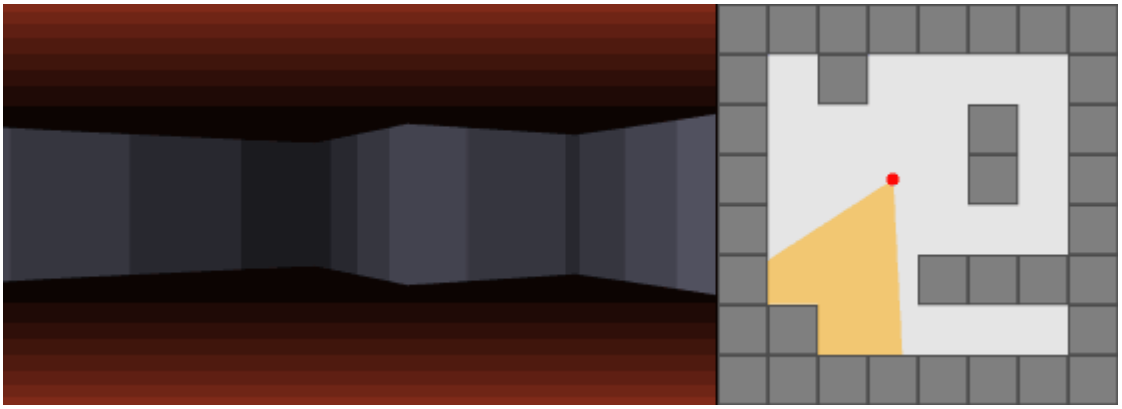


Рисунок 1. Простая иллюстрация рейкастинга в играх.

2. Ход работы

2.1. Работа с консолью.

Для начала, напишем класс Console, который будет отвечать за все операции с терминалом xterm.

Базовые функции, которые нам нужны:

- Установка четких размеров окна терминала.
- Очистка терминала
- Спрятать/показать курсор
- Установить курсор в определенное положение окна.

Для реализации данных методов, я использовал управляющие символы ANSI (*ANSI escape code*) - символы, встраиваемые в текст, для управления форматом, цветом и другими опциями вывода в текстовом терминале.

Ниже приведены некоторые команды, отвечающие за управления терминалом.

Код	Название
CSI *n* A	CUU — Cursor Up
CSI *n* B	CUD — Cursor Down
CSI *n* C	CUF — Cursor Forward
CSI *n* D	CUB — Cursor Back
CSI *n* E	CNL — Cursor Next Line
CSI *n* F	CPL — Cursor Previous Line
CSI *n* G	CHA — Cursor Horizontal Absolute
CSI *n* ; *m* H	CUP — Cursor Position
CSI *n* J	ED — Erase Data

Рисунок 2. Некоторые управляющие последовательности ANSI (неполный список).

Например, чтобы установить курсор в определенное место в терминале, в поток вывод необходимо записать вот такую команду:

«\033[col;rowH», где col и row — колонна и ряд соответственно.

```
class Console
{
private:
    size_t height{0};
    size_t width{0};

public:
    Console() = delete;
    Console(size_t height, size_t width) : height(height), width(width) {
        std::string command = "\e[8;";
        command += std::to_string(val: height) + ';';
        command += std::to_string(val: width) + 't';
        std::cout << command;
    }
    ~Console() = default;

    size_t get_height() { return height; }
    size_t get_width() { return width; }
    size_t size() { return height * width; }

    void clear() { std::cout << "\033[2J"; }

    void set_cursor(size_t col, size_t row) {
        std::string cmd = "\033[";
        cmd += std::to_string(val: col) + ';' + std::to_string(val: width) + 'H';
        std::cout << cmd;
    }

    void set_cursor_start() {
        std::cout << "\033[H";  // Установить курсор в начало консоли
    }

    void hide_cursor() { std::cout << "\033[?25l"; }

    void show_cursor() { std::cout << "\033[?25h"; }
};
```

Рисунок 3. Реализация класса Console.

2.2. Работа с векторами.

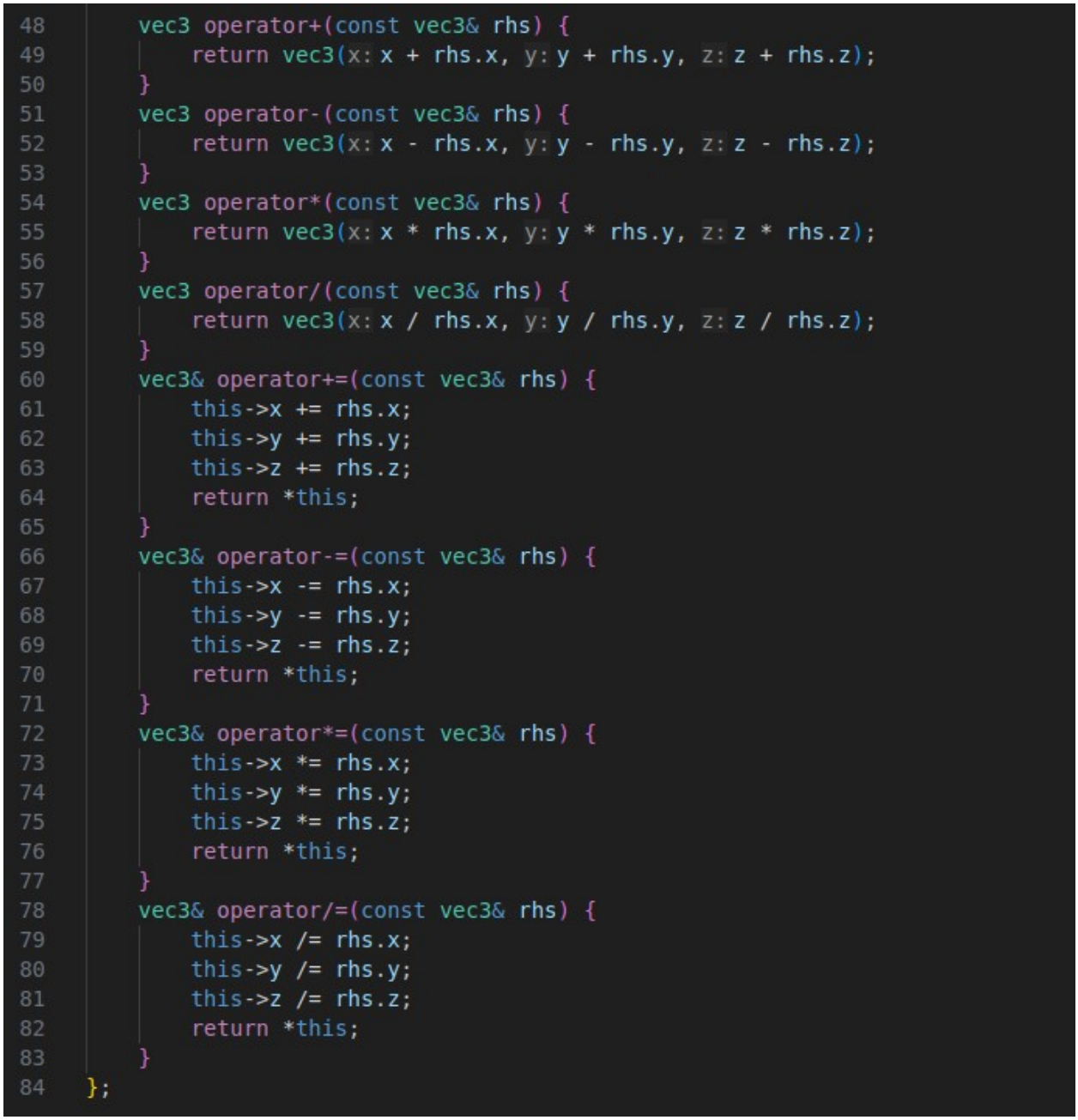
Основа данного проекта — умение работать с векторами в двух и трехмерных пространствах. Напишем соответсвующие классы для двух и трехмерного вектора соответсвенно. Определим базовые методы векторов:

- Длина вектора
- Нормализация вектора

- Скалярное произведение двух векторов.
- А также переопределение некоторых операторов.

Ниже представлена реализация трехмерного вектора.

```
4   class vec3
5   {
6   protected:
7       using data_type = double;
8
9       data_type x{0};
10      data_type y{0};
11      data_type z{0};
12
13  public:
14      vec3() = default;
15      vec3(data_type value) : x(value), y(value), z(value){};
16      vec3(data_type x, data_type y, data_type z) : x(x), y(y), z(z){};
17      vec3(data_type x, vec2 v2) : x(x), y(v2.get_x()), z(v2.get_y()){};
18      vec3(const vec3& other) : x(other.x), y(other.y), z(other.z) {}
19      ~vec3() = default;
20
21      data_type get_x() { return x; }
22      data_type get_y() { return y; }
23      data_type get_z() { return z; }
24      void set_x(data_type x) { this->x = x; }
25      void set_y(data_type y) { this->y = y; }
26      void set_z(data_type z) { this->z = z; }
27
28      double length() { return sqrt(x: x * x + y * y + z * z); }
29      vec3& norm() {
30          *this /= vec3(value: length());
31          return *this;
32      }
33      double dot_product(const vec3& other) {
34          return x * other.x + y * other.y + z * other.z;
35      }
36
37      vec3& operator=(const vec3& rhs) {
38          if (this == &rhs) {
39              return *this;
40          }
41          x = rhs.x;
42          y = rhs.y;
43          z = rhs.z;
44
45          return *this;
46      }
```



```
48      vec3 operator+(const vec3& rhs) {
49          return vec3(x: x + rhs.x, y: y + rhs.y, z: z + rhs.z);
50      }
51      vec3 operator-(const vec3& rhs) {
52          return vec3(x: x - rhs.x, y: y - rhs.y, z: z - rhs.z);
53      }
54      vec3 operator*(const vec3& rhs) {
55          return vec3(x: x * rhs.x, y: y * rhs.y, z: z * rhs.z);
56      }
57      vec3 operator/(const vec3& rhs) {
58          return vec3(x: x / rhs.x, y: y / rhs.y, z: z / rhs.z);
59      }
60      vec3& operator+=(const vec3& rhs) {
61          this->x += rhs.x;
62          this->y += rhs.y;
63          this->z += rhs.z;
64          return *this;
65      }
66      vec3& operator-=(const vec3& rhs) {
67          this->x -= rhs.x;
68          this->y -= rhs.y;
69          this->z -= rhs.z;
70          return *this;
71      }
72      vec3& operator*=(const vec3& rhs) {
73          this->x *= rhs.x;
74          this->y *= rhs.y;
75          this->z *= rhs.z;
76          return *this;
77      }
78      vec3& operator/=(const vec3& rhs) {
79          this->x /= rhs.x;
80          this->y /= rhs.y;
81          this->z /= rhs.z;
82          return *this;
83      }
84  };
```

Рисунок 4. Реализация трехмерного вектора

2.3 Работа с геометрическими фигурами.

На сцене мы будет отрисовывать простейшие геометрические фигуры. Начнем со сферы. Класс фигуры будет содержать в себе координаты, габариты (в случае сферы — ее радиус) и функцию, определяющую пересечения луча со сферой, в зависимости от начального положения луча и его направления.

Луч определяется:

- Началом (точка): $\overrightarrow{R_0} = (x_0, y_0, z_0)$
- Направлением (вектор единичной длины): $\overrightarrow{R_d} = (x_d, y_d, z_d)$

Тогда луч в параметрической форме записи – это множество точек:

$$\vec{R}(t) = \overrightarrow{R_0} + \overrightarrow{R_d} \cdot t,\ t > 0\,.$$

Сфера определяется:

- Центром (точка): $\overrightarrow{S_c} = (x_c, y_c, z_c)$
- Радиусом (вещественное число): r

Поверхность сферы состоит из множества точек $\vec{S} = (x_s, y_s, z_s)$, удовлетворяющих:

$$(x_s - x_c)^2 + (y_s - y_c)^2 + (z_s - z_c)^2 = r^2$$

Подставим точку луча в уравнение сферы и решим относительно t:

$$t^2 + 2t \cdot \left(\overrightarrow{R_d}, \left(\overrightarrow{R_0} - \overrightarrow{S_c}\right)\right) + \left(\overrightarrow{R_0} - \overrightarrow{S_c}\right)^2 - r^2 = 0\,.$$

Обозначим:

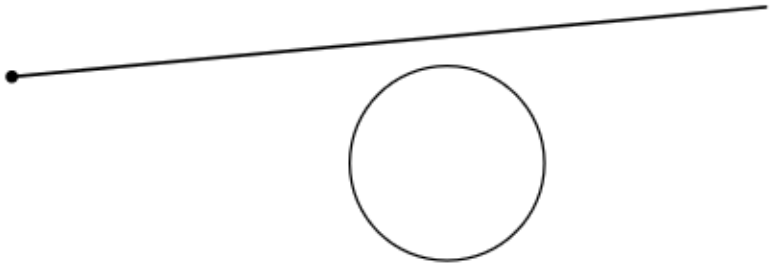
$$B = \left(\overrightarrow{R_d}, \left(\overrightarrow{R_0} - \overrightarrow{S_c}\right)\right)$$

$$C = \left(\overrightarrow{R_0} - \overrightarrow{S_c}\right)^2 - r^2$$

И получим:

$$t^2 + 2Bt + C = 0\,.$$

Если дискриминант $D = B^2 - C$ отрицательный, то пересечений нет:



Меньший положительный корень (если существует) определяет ближайшую точку пересечения. Некоторая экономия времени может достигаться, если вычислять сначала меньший корень, а затем при необходимости больший:

$$t_0 = -B - \sqrt{B^2 - C},\ t_1 = -B + \sqrt{B^2 - C}\ .$$

```
7   enum class Shape_type { SPHERE, PLANE, CUBE };
8
9   class Abstract_shape
10  {
11  protected:
12      vec3 coordinates;
13
14  public:
15      Abstract_shape(const vec3& coordinates) : coordinates(coordinates){};
16      virtual vec2 ray_interspect(vec3 ro, vec3 rd) = 0;
17  };
18
19  class Sphere : public Abstract_shape
20  {
21  public:
22      static const Shape_type shape_type = Shape_type::SPHERE;
23
24  private:
25      double radius{0};
26
27  public:
28      Sphere() = delete;
29      Sphere(const vec3& coordinates, double radius)
30              : Abstract_shape(coordinates), radius(radius) {}
31      ~Sphere() = default;
32
33      vec2 ray_interspect(vec3 ro, vec3 rd) {
34          vec3 dist = ro - coordinates;  // vector form ro to center of sphere
35          double b = rd.dot_product(other: dist);
36          double c = dist.dot_product(other: dist) - radius * radius;
37          double d = b * b - c;
38          if (d < 0.0) return vec2(value: -1.0);
39          d = sqrt(x: d);
40          return vec2(x: -b - d, y: -b + d);
41      }
42  };
```

Рисунок 5. Реализация класса Sphere

2.4. Работа со сценой.

Все наши геометрические фигуры будут располагаться на сцене. Создадим массив указателей на абстрактный класс для хранения разных геометрических фигур на нашей сцене. Еще нам понадобиться градиент символов для эмуляции света. Символ с наибольшим кол-вом символов самый яркий. Символ с наименьшим количеством света самый темный. Также определим на сцене расположение камеры и направление источника света.

Класс сцена содержит в себе функции:

- Добавление новой геометрической фигруы.
- Обновление отрисовки. По сути это основная функция, которая в зависимости от текущего набора геометрических фигур, направления света и направление камеры, отрисовывает картинку в терминале.

```cpp
class Scene
{
private:
    static constexpr char gradient[] = {" .:!/r(l1Z4H9W8$@"};

private:
    vec3 dimension{value: 0};
    vec3 camera{value: 0};
    char* projection;
    vec3 light_dir{value: 0};

    std::vector<Abstract_shape*> shapes;

public:
    Scene() = delete;
    Scene(const vec3& dimension, const vec3& camera);
    ~Scene();

    double window_aspect() {
        return static_cast<double>(dimension.get_y()) /
               static_cast<double>(dimension.get_z());
    }
    double pixel_aspect() { return 11.0 / 24.0; }  // pixels are not square
    double aspect() { return window_aspect() * pixel_aspect(); }

    size_t gradient_size() { return sizeof(gradient) - 1; }
    size_t size2d() { return dimension.get_y() * dimension.get_z(); }

    void set_light_dir(const vec3& light_dir);

    void add_shape(Abstract_shape* shape);

    void update();

    friend std::ostream& operator<<(std::ostream& os, Scene& scene) {
        if (!scene.size2d()) {
            os << "Scene is empty";
            return os;
        }
        os << scene.projection;
        return os;
    };
```

Рисунок 6. Scene.hpp

```cpp
Scene::Scene(const vec3& dimension, const vec3& camera)
        : dimension(dimension), camera(camera) {
    size_t width = this->dimension.get_y();
    size_t height = this->dimension.get_z();
    projection = new char[height * width + 1];
    projection[height * width] = '\0';
}

Scene::~Scene() { delete[] projection; }

void Scene::set_light_dir(const vec3& light_dir) {
    this->light_dir = light_dir;
    this->light_dir.norm();
}

void Scene::add_shape(Abstract_shape* shape) { shapes.push_back(x: shape); }

void Scene::update() {
    for (size_t idx = 0; idx < size2d(); ++idx) {
        vec2 uv{};
        size_t height = static_cast<size_t>(dimension.get_z());
        size_t width = static_cast<size_t>(dimension.get_y());
        uv.set_x(range_scale(x: idx % width, a: 0.0, b: width, c: -1.0, d: 1.0));
        uv.set_y(range_scale(x: static_cast<double>(idx) / width, a: 0.0, b: height,
                             c: -1.0, d: 1.0));
        uv *= vec2(x: aspect(), y: 1);
        vec3 temp_dir = vec3(x: 1, v2: uv);
        vec3 camera_dir = temp_dir.norm();
        vec2 interspection = shapes[0]->ray_interspect(ro: camera, rd: camera_dir);

        int color = 0;
        if (interspection.get_x() > 0) {
            vec3 it_point = camera + camera_dir * interspection.get_x();
            vec3 it_point_norm = it_point.norm();
            double diff = it_point_norm.dot_product(other: light_dir);
            diff = range_scale(x: diff, a: -1, b: 1, c: 0, d: gradient_size());
            color = limit(value: diff, min: 0, max: gradient_size());
        }
        char pixel = ' ';
        pixel = gradient[color];
        projection[idx] = pixel;
    }
}
```

Рисунок 7. Scene.cpp

Заключение

В ходе выполнения курсовой работы были получены знания о методах рендеринга компьютерной графики. В частности, реализован метод ray_casting. Также получены навыки работы с трехмерными векторами и работы с терминалом xterm.

Список литературы

1. https://invisible-island.net/xterm/ctlseqs/ctlseqs.html
2. http://nsucgcourse.github.io/lectures/Lecture13/Slide_13_Valeev_Rays.pdf
3. https://habr.com/ru/articles/436790/
4. http://ray-tracing.ru/articles245.html