

Министерство образования и науки Российской Федерации
Новосибирский государственный технический университет

Кафедра теоретической и прикладной информатики

Лабораторная работа №4
Трассировка лучей

Факультет:	ПМИ
Группа:	ПМ-54, ПМИ-51
Студент:	Кухтин В.А., Четвертакова Ю.С.
Преподаватель:	Задорожный А.Г.
Вариант:	2

Новосибирск
2018

1. Цель работы

Ознакомиться с основными аспектами метода трассировки лучей.

2. Задание

- Считывать из файла (в зависимости от варианта)
 - а) тип объекта;
 - б) координаты и размер объектов;
 - в) параметры материала объектов.
- Выполнить трассировку первичных лучей.
- Добавить зеркальную плоскость и учесть отраженные лучи.
- Предусмотреть возможность включения/исключения объектов.
- Предусмотреть возможность изменения положения источника света.

Вариант 2:

Объекты №1: сферы

Объекты №2: тетраэдры

3. Руководство пользователя

Управление

Мышь	Назначение
Движение курсора	Вращение камеры

Клавиша клавиатуры	Назначение
q (Q)	Включение/выключение режима вращения камеры мышью
w (W)	Движение камеры вперед
s (S)	Движение камеры назад
a (A)	Поворот камеры влево
d (D)	Поворот камеры вправо
t (T)	Включение/выключение трассировки лучей
c (C)	Включение/выключение режима включения/исключения фигур
e (E)	Включить/исключить фигуру в режиме включения/исключения фигур
Стрелка вверх	Переключать фигуры в порядке возрастания
Стрелка вниз	Переключать фигуры в порядке убывания
Стрелка влево	Сменить тип фигуры
7	Перемещение основного источника освещения в положительном направлении по оси X
4	Перемещение основного источника освещения в отрицательном направлении по оси X
8	Перемещение основного источника освещения в положительном направлении по оси Y

5	Перемещение основного источника освещения в отрицательном направлении по оси Y
9	Перемещение основного источника освещения в положительном направлении по оси Z
6	Перемещение основного источника освещения в отрицательном направлении по оси Z

4. Текст программы

camera.h

```

/*
Описание интерфейса класса камеры
*/

#include <math.h>
#include <windows.h>
#include "glut.h"
#include <vector>
#include <gl\gl.h>
#include "normals.h"

class Camera {
public:
    Vector3f Position; // Позиция камеры/положение точки наблюдателя.
    Vector3f View;      // Направление наблюдения.
    Vector3f UpVector;  // Вектор поворота сцены.

    Camera();
    void PositionCamera(float posX, float posY, float posZ,
                       float viewX, float viewY, float viewZ,
                       float upX, float upY, float upZ); // Установка позиции камеры.

    void SetViewByMouse(GLint width, GLint height); // Установка вида с помощью мыши.
    void MoveCamera(float speed); // Передвижение
    камеры вперед/назад.
    void RotateView(float angle, float x, float y, float z); // Вращение камеры вокруг заданной оси.
    void RotateAroundPoint(Vector3f vCenter, float angle, float x, float y, float z); // Вращение камеры вокруг наблюдателя.
};

```

camera.cpp

```

#include "camera.h"

/*
Реализация интерфейсов класса камеры.
*/
Camera::Camera() {
}

void Camera::PositionCamera(float posX, float posY, float posZ,
                           float viewX, float viewY, float viewZ,
                           float upX, float upY, float upZ) {
    // Установить позицию камеры.
    Vector3f _Position = Vector3f(
        posX,
        posY,
        posZ
    );
    Vector3f _View = Vector3f(
        viewX,
        viewY,
        viewZ
    );
    Vector3f _UpVector = Vector3f(
        upX,
        upY,
        upZ
    );

    Position = _Position;
    View = _View;
}

```

```

    UpVector = _UpVector;
}
void Camera::MoveCamera(float speed){
    Vector3f _View = View - Position; // Определить направление взгляда.
    // Передвижение камеры.
    Position.x += _View.x * speed; // Изменить положение.
    Position.z += _View.z * speed; // Камеры.

    View.x += _View.x * speed; // Изменить направление.
    View.z += _View.z * speed; // Взгляда камеры.
}
void Camera::RotateView(float angle, float x, float y, float z) {
    Vector3f _newView;
    Vector3f _View;

    // Определить направление взгляда.
    _View = View - Position;

    // Рассчитать синус и косинус переданного угла.
    float cosA = (float)cos(angle);
    float sinA = (float)sin(angle);

    /*      Пересчет координат по каким-то диким формулам
            новая координата X для вращаемой точки. */
    _newView.x = (cosA + (1 - cosA) * x * x) * _View.x;
    _newView.x += ((1 - cosA) * x * y - z * sinA) * _View.y;
    _newView.x += ((1 - cosA) * x * z + y * sinA) * _View.z;

    // Новая координата Y для вращаемой точки.
    _newView.y = ((1 - cosA) * x * y + z * sinA) * _View.x;
    _newView.y += (cosA + (1 - cosA) * y * y) * _View.y;
    _newView.y += ((1 - cosA) * y * z - x * sinA) * _View.z;

    // Новая координата Z для вращаемой точки.
    _newView.z = ((1 - cosA) * x * z - y * sinA) * _View.x;
    _newView.z += ((1 - cosA) * y * z + x * sinA) * _View.y;
    _newView.z += (cosA + (1 - cosA) * z * z) * _View.z;

    // Установить новый взгляд камеры.
    View.x = Position.x + _newView.x;
    View.y = Position.y + _newView.y;
    View.z = Position.z + _newView.z;
}
void Camera::SetViewByMouse(GLint width, GLint height) {
    POINT mousePos; // Позиция мыши.

    // Вычислить координаты центра окна.
    int middleX = width / 2.0f;
    int middleY = height / 2.0f;

    float angleY = 0.0f; // Направление взгляда вверх/вниз.
    float angleZ = 0.0f; // Значение, необходимое для вращения влево-вправо (по оси Y).
    static float currentRotX = 0.0f;

    // Получить текущие координаты мыши.
    GetCursorPos(&mousePos);

    /*      Если положение мыши не изменилось
            камеру вращать не нужно */
    if (mousePos.x == middleX && mousePos.y == middleY)
        return;

    // Вернуть координаты курсора в центр окна.
    SetCursorPos(middleX, middleY);

    // Определить, куда был сдвинут курсор.
    angleY = (float)((middleX - mousePos.x)) / 1000.0f;
    angleZ = (float)((middleY - mousePos.y)) / 1000.0f;

    /*      Сохраняем последний угол вращения
            и используем заново currentRotX */
    static float lastRotX = 0.0f;
    lastRotX = currentRotX;

    /*      Если поворот больше одного градуса, умевшим его
            чтобы уменьшить скорость вращения. */
    if (currentRotX > 1.0f) {
        currentRotX = 1.0f;
        if (lastRotX != 1.0f) {
            /*      Чтобы найти ось, вокруг которой нужно совершать вращение вверх и вниз, нужно
                    найти вектор, перпендикулярный вектору взгляда камеры и
                    вертикальному вектору */
            Vector3f vAxis = normal(View - Position, UpVector);

            // Нормализуем ось.
            vAxis = normalize(vAxis);

            // Вращаем камеру вокруг нашей оси на заданный угол.

```

```

        RotateView(1.0f - lastRotX, vAxis.x, vAxis.y, vAxis.z);
    }
}

// Если угол меньше -1.0f.
else if (currentRotX < -1.0f) {
    currentRotX = -1.0f;
    if (lastRotX != -1.0f) {
        // Вычисляем ось.
        Vector3f vAxis = normal(View - Position, UpVector);

        // Нормализуем ось.
        vAxis = normalize(vAxis);

        // Вращаем.
        RotateView(-1.0f - lastRotX, vAxis.x, vAxis.y, vAxis.z);
    }
} else { // Если в пределах 1.0f -1.0f - просто вращаем.
    Vector3f vAxis = normal(View - Position, UpVector);
    vAxis = normalize(vAxis);
    RotateView(angleZ, vAxis.x, vAxis.y, vAxis.z);
}

// Всегда вращаем камеру вокруг Y-оси.
RotateView(angleY, 0, 1, 0);
}

void Camera::RotateAroundPoint(Vector3f _Center, float angle, float x, float y, float z) {
    Vector3f _NewPosition;

    // Получим центр, вокруг которого нужно вращаться.
    Vector3f vPos = Position - _Center;

    // Вычислим синус и косинус угла.
    float cosA = (float)cos(angle);
    float sinA = (float)sin(angle);

    // Найдем значение X точки вращения.
    _NewPosition.x = (cosA + (1 - cosA) * x * x) * vPos.x;
    _NewPosition.x += ((1 - cosA) * x * y - z * sinA) * vPos.y;
    _NewPosition.x += ((1 - cosA) * x * z + y * sinA) * vPos.z;

    // Значение Y.
    _NewPosition.y = ((1 - cosA) * x * y + z * sinA) * vPos.x;
    _NewPosition.y += (cosA + (1 - cosA) * y * y) * vPos.y;
    _NewPosition.y += ((1 - cosA) * y * z - x * sinA) * vPos.z;

    // Значение Z.
    _NewPosition.z = ((1 - cosA) * x * z - y * sinA) * vPos.x;
    _NewPosition.z += ((1 - cosA) * y * z + x * sinA) * vPos.y;
    _NewPosition.z += (cosA + (1 - cosA) * z * z) * vPos.z;

    // Установить новую позицию камеры.
    Position = _Center + _NewPosition;
}

```

gauss.h

```

#pragma once
#include<math.h>

void solve_gauss(double **A, double *b, double *x, int n);
void direct_st(double **A, double *x, int n);
void transform(double **A, double *x, int i, int n);
void exchange(double **A, double *x, int first, int second, int n);

```

gauss.cpp

```

#include"gauss.h"

void solve_gauss(double **A, double *b, double *x, int n)
{
    int i, j;
    double s;
    direct_st(A, b, n);
    for (i = n - 1; i >= 0; i--)
    {
        s = 0;
        for (j = n - 1; j > i; j--)
            s += A[i][j] * x[j];
        x[i] = (b[i] - s) / A[i][i];
    }
}

void direct_st(double **A, double *x, int n)
{
    int i, j, k;
    double koeff;
    for (i = 0; i < n; i++)

```

```

    {
        transform(A, x, i, n);
        for (j = i + 1; j < n; j++)
        {
            koeff = -A[j][i] / A[i][i];
            for (k = i; k < n; k++)
                A[j][k] += A[i][k] * koeff;
            x[j] += x[i] * koeff;
        }
    }
}

void transform(double **A, double *x, int i, int n)
{
    double max = A[i][i];
    int gl = i;
    int j;
    for (j = i + 1; j < n; j++)
        if (fabs(A[j][i]) > fabs(max))
        {
            max = A[j][i];
            gl = j;
        }
    if (gl != i) exchange(A, x, i, gl, n);
}

void exchange(double **A, double *x, int first, int second, int n)
{
    int i;
    double mid;
    mid = x[first];
    x[first] = x[second];
    x[second] = mid;
    for (i = 0; i < n; i++)
    {
        mid = A[first][i];
        A[first][i] = A[second][i];
        A[second][i] = mid;
    }
}

```

helpTracer.h

```

/*
Описание вспомогательных классов.
*/

#include "camera.h"

// Интерфейс класса "луч".
class Ray {
public:
    Vector3f start;           // Стартовая точка луча.
    Vector3f dir;             // Направление луча.
    int recurseLevel;        // Текущий уровень рекурсии.

    Ray();
    void SetStart(Vector3f _start);    // Установить стартовую точку.
    void SetDir(Vector3f _dir);        // Установить направление.
};

// Интерфейс класса описывающего информацию о соударении.
class HitInfo {
public:
    double hitTime;           // Время соударения.
    int objectType;          // Тип объекта соударения.
    int objectNum;            // Номер объекта соударения.
    int surface;              // Поверхность соударения.
    bool isEntering;          // ?Луч входит или выходит из объекта.

    Vector3f hitPoint;        // Точка соударения.
    Vector3f hitNormal;        // Нормаль в точке соударения.

    HitInfo();
    void set(HitInfo hI);
};

// Интерфейс класса описывающего список соударений.
class Intersection {
public:
#define maxNumHits 8
    int numHits;              // Число соударений для положительных значений времени.
    HitInfo hit[maxNumHits];  // Список соударений.

    Intersection();
    void set(Intersection intr);
};

```

helpTracer.cpp

```
#include "helpTracer.h"

/*
Реализация интерфейса класса "луч".
*/
Ray::Ray(){
}
void Ray::SetStart(Vector3f _start) {
    start = Vector3f(
        _start.x,
        _start.y,
        _start.z
    );
}
void Ray::SetDir(Vector3f _dir) {
    dir = Vector3f(
        _dir.x,
        _dir.y,
        _dir.z
    );
}

/*
Реализация интерфейса класса хранящего информацию об ударе.
*/
HitInfo::HitInfo() {
    objectType = -1;
    objectNum = -1;
    hitTime = -1000;
    surface = -1;
    isEntering = false;
}
void HitInfo::set(HitInfo hI) {
    hitTime = hI.hitTime;
    objectType = hI.objectType;
    objectNum = hI.objectNum;
    surface = hI.surface;
    hitPoint = Vector3f(
        hI.hitPoint.x,
        hI.hitPoint.y,
        hI.hitPoint.z
    );
    hitNormal = Vector3f(
        hI.hitNormal.x,
        hI.hitNormal.y,
        hI.hitNormal.z
    );
}

/*
Реализация интерфейса класса хранящего список соударений.
*/
Intersection::Intersection() {
    numHits = 0;
}
void Intersection::set(Intersection intr) {
    numHits = intr.numHits;
    for (int i = 0; i < maxNumHits; i++)
        hit[i].set(intr.hit[i]);
}
```

Hook.h

```
#define _USE_MATH_DEFINES
#include <cmath>
#include <windows.h>
#include "glut.h"
#include <vector>
#include <gl\gl.h>
#include <cstdio>
#include "tracer.h"

#define kSpeed 0.03f //скорость перемещения камеры

using namespace std;

GLint width, height; //ширина и высота окна
GLfloat ratio; //соотношение ширины и высоты окна
Camera camera; //камера
Scene scene; //сцена
Raytracer raytracer; //трассировщик

bool rot = false; //включить вращение камеры с помощью мыши

bool raytracer_mode = false; //режим трассировки
```

normals.h

```
#pragma once
#define _USE_MATH_DEFINES
#include<math.h>

class Vector3f
{
public:
    float x, y, z;

    Vector3f() {};
    Vector3f(float x_0, float y_0, float z_0)    //конструктор по умолчанию
    {
        x = x_0;
        y = y_0;
        z = z_0;
    }

    Vector3f operator + (Vector3f vect)
    {
        return Vector3f(vect.x + x, vect.y + y, vect.z + z);
    }

    Vector3f operator - (Vector3f vect)
    {
        return Vector3f(x - vect.x, y - vect.y, z - vect.z);
    }

    Vector3f operator * (float n)
    {
        return Vector3f(x * n, y * n, z * n);
    }

    Vector3f operator / (float n)
    {
        return Vector3f(x / n, y / n, z / n);
    }
};

//нормаль двух векторов
Vector3f normal(Vector3f vect_1, Vector3f vect_2);

//норма вектора
float norma(Vector3f vect);

//нормализация вектора
Vector3f normalize(Vector3f vect);

//скалярное произведение двух векторов
float scalar(Vector3f vect_1, Vector3f vect_2);

//вектор между 2 точками
Vector3f count_vect(Vector3f point_1, Vector3f point_2);

//нормаль для полигона
Vector3f count_normal_for_polygon(Vector3f one, Vector3f two, Vector3f three);
```

normals.cpp

```
#include "normals.h"

//вычисляем нормаль двух векторов
Vector3f normal(Vector3f vect_1, Vector3f vect_2)
{
    Vector3f norm;

    //вычисление векторного произведения
    norm.x = ((vect_1.y*vect_2.z) - (vect_1.z*vect_2.y));
    norm.y = ((vect_1.z*vect_2.x) - (vect_1.x*vect_2.z));
    norm.z = ((vect_1.x*vect_2.y) - (vect_1.y*vect_2.x));

    return norm;
}

//вычисляем норму вектора
float norma(Vector3f vect)
{
    return (float)sqrt((vect.x*vect.x) + (vect.y*vect.y) + (vect.z*vect.z));
}

//нормализуем вектор
Vector3f normalize(Vector3f vect)
{
    //Вычислить норму вектора
    float norm = norma(vect);
```



```

        //нормализовать вектор
        vect = vect / norm;

    return vect;
}

//вычисляем скалярное произведение
float scalar(Vector3f vect_1, Vector3f vect_2)
{
    return vect_1.x*vect_2.x + vect_1.y*vect_2.y + vect_1.z*vect_2.z;
}

//вычисляем вектор между двумя точками
Vector3f count_vect(Vector3f point_1, Vector3f point_2)
{
    Vector3f vect;

    vect.x = point_1.x - point_2.x;
    vect.y = point_1.y - point_2.y;
    vect.z = point_1.z - point_2.z;
    return vect;
}

//вычисляем нормали полигона
Vector3f count_normal_for_polygon(Vector3f one, Vector3f two, Vector3f three)
{
    Vector3f vect_1 = count_vect(three, two);
    Vector3f vect_2 = count_vect(two, one);
    Vector3f norm = normal(vect_1, vect_2);
    norm = normalize(norm);
    return norm;
}

```

scene.h

```

#include <math.h>
#include <windows.h>
#include "glut.h"
#include <vector>
#include <gl\gl.h>
#include <stdio.h>
#include "helpTracer.h"
#include "gauss.h"

/*
Описание интерфейсов классов описывающих сцену.
*/
using namespace std;

//=====Для преобразования луча к базовому виду=====

// Определить матрицу трансформирования.
void makeTransformMatrix(double **TMatrix, Vector3f shift, GLfloat scaleX, GLfloat scaleY, GLfloat scaleZ);

// Перевод луча к базовому виду.
Ray transformRay(Ray _ray, Vector3f shift, GLfloat scaleX, GLfloat scaleY, GLfloat scaleZ);

// Скалярное произведение.
double scal(Vector3f vec1, Vector3f vec2);

// Определить позицию луча(точку соударения луча с объектом).
Vector3f rayPos(Ray ray, double t);

//=====

// Описание интерфейса класса, описывающего цвет.
class Color {
public:
    GLfloat red;
    GLfloat green;
    GLfloat blue;

    Color();
    Color(
        GLfloat _red,
        GLfloat _green,
        GLfloat _blue
    );
    void add(
        GLfloat _red,
        GLfloat _green,
        GLfloat _blue
    );
    void add(Color colr);
    void add(Color colr, Color ref1);
};

// Описание интерфейса класса, описывающего источник света.
class Light {

```

```

public:
    Vector3f position;           // Позиция источника света.
    Color color;                 // Цвет источника света.

    Light();
    void setPosition(Vector3f pos); // Установить позицию источника света.
    void setColor(Color col);      // Установить цвет источника света.
};

// Описание интерфейса класса, описывающего объект сцены: сфера.
class Sphere {
public:
    GLfloat radius;              // Радиус сферы.
    Vector3f center_coord;       // Координаты центра сферы.
    Color color;                 // Цвет сферы.
    Color FrameColor;           // Цвет каркаса сферы.
    bool display;               // true - если сфера отрисовывается.

    // Свойства материала сферы.
    GLfloat Ambient[4];          // Фоновое отражение.
    GLfloat Diffuse[4];          // Рассеянное отражение.
    GLfloat Specular[4];         // Зеркальное отражение.
    GLfloat Emission[4];         // Собственное излучение.
    GLfloat Shininess;           // Коэффициент блеска.

    Sphere();
    Sphere(
        GLfloat _radius,
        Vector3f _center,
        Color _color
    );
    void Draw();                 // Отрисовка сферы.
    void DrawFrame();           // Отрисовка
    каркаса сферы.

    bool hit(Ray ray, Intersection &inter); // Определение соударений луча со сферой.
    bool hit(Ray ray);           // Упрощенный метод hit (не строит запись о пересече-
    ниях, используется для определения тени).
};

// Описание интерфейса класса, описывающего объект сцены: тетраэдр
class Tetrahedron {
public:
    Vector3f coord[4];           // Координаты вершин.
    Vector3f center_coord;       // Координаты центра тетраэдра.
    Color color;                 // Цвет тетраэдра.
    Color FrameColor;           // Цвет каркаса тетраэдра.
    bool display;               // true - если тетраэдр отрисовывается.

    // Свойства материала тетраэдра.
    GLfloat Ambient[4];          // Фоновое отражение.
    GLfloat Diffuse[4];          // Рассеянное отражение.
    GLfloat Specular[4];         // Зеркальное отражение.
    GLfloat Emission[4];         // Собственное излучение.
    GLfloat Shininess;           // Коэффициент блеска.

    Tetrahedron();
    Tetrahedron(
        Vector3f _coord[],
        Vector3f _center,
        Color _color
    );
    void Draw();                 // Отрисовка тетраэдра.
    void DrawFrame();           // Отрисовка каркаса тетраэдра.

    bool hit(Ray ray, Intersection &inter); // Определение соударений луча с тетраэдром.
    bool hit(Ray ray);           // Упрощенный метод hit (не строит запись о пересече-
    ниях, используется для определения тени).

    bool hitPlane(Ray ray, int ver1, int ver2, int ver3, Vector3f &N, float &time); // Определение соударения с гранью
    тетраэдра.
};

// Описание интерфейса класса, описывающего объект сцены: квадрат
class Square {
public:
    Vector3f center_coord;       // Координаты центра плоскости.
    GLfloat scaleX;              // Расстояние по оси X.
    GLfloat scaleZ;              // Расстояние по оси Z.
    Color color;                 // Цвет квадрата.

    // Свойства материала квадрата.
    GLfloat Ambient[4];          // Фоновое отражение.
    GLfloat Diffuse[4];          // Рассеянное отражение.
    GLfloat Specular[4];         // Зеркальное отражение.
    GLfloat Emission[4];         // Собственное излучение.
    GLfloat Shininess;           // Коэффициент блеска.

```

```

Square();
Square(Vector3f _center, GLfloat _scaleX, GLfloat _scaleZ, Color _color);
void Draw();
// Отрисовка квадрата.
bool hit(Ray ray, Intersection &inter);
// Определение соударений луча с квадратом.
bool hit(Ray ray);
// Упрощенный метод hit (не строит запись о пересечениях, используется для определения тени).
};

// Описание интерфейса класс сцены.
class Scene {
public:
    Square square; // Плоскость.
    vector<Sphere> vector_Sphere; // Контейнер сфер.
    vector<Tetrahedron> vector_Tetrahedron; // Контейнер тетраэдров.
    vector<Light> vector_Light; // Контейнер источников освещения.

    Scene();
    void set_data(); // Считать из файла координаты фигур и их параметры.
    void switch_forward(); // Переключение между фигурами вперед.
    void switch_backward(); // Переключение между фигурами назад.

    void Draw(); // Отрисовка сцены.
    bool isInShadow(Ray ray); // Находится ли объект в тени другого объекта.

    bool add_del; // Режим включения/исключения объектов.
    bool sphere_mod; // Переключение между объектами сферами.
    bool tetrahedron_mod; // Переключение между объектами тетраэдрами.
    int active_Sphere; // Активная сфера.
    int active_Tetrahedron; // Активный тетраэдр.
};

```

scene.cpp

```

#include "scene.h"

/*
Реализация интерфейса класса "Цвет"
*/
Color::Color() {
    red = green = blue = 0;
}
Color::Color(GLfloat _red, GLfloat _green, GLfloat _blue) {
    red = _red;
    green = _green;
    blue = _blue;
}
void Color::add(GLfloat _red, GLfloat _green, GLfloat _blue) {
    red += _red;
    green += _green;
    blue += _blue;
}
void Color::add(Color colr, Color refl) {
    red += colr.red * refl.red;
    green += colr.green * refl.green;
    blue += colr.blue * refl.blue;
}
void Color::add(Color colr) {
    red += colr.red;
    green += colr.green;
    blue += colr.blue;
}

/*
Реализация интерфейса класса "Свет"
*/
Light::Light(){
}
void Light::setPosition(Vector3f pos){
    position = Vector3f(
        pos.x,
        pos.y,
        pos.z
    );
}
void Light::setColor(Color col){
    color = Color(
        col.red,
        col.green,
        col.blue
    );
}

/*
Реализация интерфейса класса "Сфера"
*/
Sphere::Sphere() {

```

```

}
Sphere::Sphere(GLfloat _radius, Vector3f _center, Color _color) {
    radius = _radius;
    center_coord = _center;
    color = _color;
    FrameColor = Color(255, 0, 0);

    display = true;

    // Дефолтные свойства материала
    Ambient[0] = 0.2f;
    Ambient[1] = 0.2f;
    Ambient[2] = 0.2f;
    Ambient[3] = 1.0f;
    Diffuse[0] = 0.8f;
    Diffuse[1] = 0.8f;
    Diffuse[2] = 0.8f;
    Diffuse[3] = 1.0f;
    Specular[0] = 0.0f;
    Specular[1] = 0.0f;
    Specular[2] = 0.0f;
    Specular[3] = 1.0f;
    Emission[0] = 0.0f;
    Emission[1] = 0.0f;
    Emission[2] = 0.0f;
    Emission[3] = 1.0f;
    Shininess = 0.0f;
}
void Sphere::Draw() {
    GLUQuadricObj *quadObj;

    // Создаем новый объект для создания сферы.
    quadObj = gluNewQuadric();

    // Сохраняем текущую матрицу.
    glPushMatrix();

    // Перемещаемся в центр фигуры.
    glTranslated(center_coord.x, center_coord.y, center_coord.z);

    glColor3f(color.red, color.green, color.blue);

    // Устанавливаем сплошной стиль объекта.
    gluQuadricDrawStyle(quadObj, GLU_FILL);

    // Рисуем сферу.
    gluSphere(quadObj, radius, 10, 10);

    // Восстанавливаем матрицу.
    glPopMatrix();

    // Удаляем объект.
    gluDeleteQuadric(quadObj);
}
void Sphere::DrawFrame() {
    GLUQuadricObj *quadObj;

    // Создаем новый объект для создания сферы.
    quadObj = gluNewQuadric();

    // Сохраняем текущую матрицу.
    glPushMatrix();

    // Перемещаемся в центр фигуры.
    glTranslated(center_coord.x, center_coord.y, center_coord.z);

    glColor3f(FrameColor.red, FrameColor.green, FrameColor.blue);

    // Устанавливаем сплошной стиль объекта.
    gluQuadricDrawStyle(quadObj, GLU_LINE);

    // Рисуем сферу.
    gluSphere(quadObj, radius * 1.01, 10, 10);

    // Восстанавливаем матрицу.
    glPopMatrix();

    // Удаляем объект.
    gluDeleteQuadric(quadObj);
}
bool Sphere::hit(Ray ray, Intersection &inter) {
    Ray genRay;

    // Создаем базовый луч.
    genRay = transformRay(ray, center_coord, radius, radius, radius);

    double A, B, C;
    A = scal(genRay.dir, genRay.dir);

```

```

B = scal(genRay.start, genRay.dir);
C = scal(genRay.start, genRay.start) - 1.0;

// Вычисляем дискриминант.
double discrim = B * B - A * C;

// Нет соударения с объектом.
if (discrim < 0)
    return false;

int num = 0; // Число соударений на данный момент.

double discRoot = sqrt(discrim);
double t1 = (-B - discRoot) / A; // Более раннее соударение.

// Если соударение находится впереди глаза.
if (t1 > 0.00001) {
    /* Формируем запись о соударении
       номер объекта будет установлен в вызывающем методе. */
    inter.hit[0].hitTime = t1;
    inter.hit[0].objectType = 0;
    inter.hit[0].surface = 0;
    inter.hit[0].isEntering = true;

    // Мировые координаты точки соударения.
    Vector3f P(rayPos(ray, t1));
    inter.hit[0].hitPoint = Vector3f(
        P.x,
        P.y,
        P.z
    );

    // Координаты нормали в точке соударения.
    P = Vector3f(rayPos(genRay, t1));
    inter.hit[0].hitNormal = Vector3f(
        P.x,
        P.y,
        P.z
    );
    num = 1;
}

double t2 = (-B + discRoot) / A; // Более позднее соударение.
if (t2 > 0.00001) {
    /* Формируем запись о соударении
       номер объекта будет установлен в вызывающем методе. */
    inter.hit[num].hitTime = t2;
    inter.hit[num].objectType = 0;
    inter.hit[num].surface = 0;
    inter.hit[num].isEntering = false;

    // Мировые координаты точки соударения.
    Vector3f P(rayPos(ray, t2));
    inter.hit[num].hitPoint = Vector3f(
        P.x,
        P.y,
        P.z
    );

    // Координаты нормали в точке соударения.
    P = Vector3f(rayPos(genRay, t2));
    inter.hit[num].hitNormal = Vector3f(
        P.x,
        P.y,
        P.z
    );
    num++;
}

inter.numHits = num;
return (num > 0);
}

bool Sphere::hit(Ray ray) {
    Ray genRay;

    // Создаем базовый луч.
    genRay = transformRay(ray, center_coord, radius, radius, radius);

    double A, B, C;
    A = scal(genRay.dir, genRay.dir);
    B = scal(genRay.start, genRay.dir);
    C = scal(genRay.start, genRay.start) - 1.0;

    // Вычисляем дискриминант.
    double discrim = B * B - A * C;

    // Нет соударения с объектом.
    if (discrim < 0)

```

```

        return false;

double discRoot = sqrt(discrim);

double t1 = (-B - discRoot) / A; // Более раннее соударение.
if (t1 >= 0.00001 && t1 <= 1.0)
    return true;

double t2 = (-B + discRoot) / A; // Более позднее соударение.
if (t2 >= 0.00001 && t2 <= 1.0)
    return true;

return false;
}

/*
Реализация интерфейса класса "Тетраэдр"
*/
Tetrahedron::Tetrahedron() {
}
Tetrahedron::Tetrahedron(Vector3f _coord[], Vector3f _center, Color _color) {
    coord[0] = Vector3f(_coord[0].x, _coord[0].y, _coord[0].z);
    coord[1] = Vector3f(_coord[1].x, _coord[1].y, _coord[1].z);
    coord[2] = Vector3f(_coord[2].x, _coord[2].y, _coord[2].z);
    coord[3] = Vector3f(_coord[3].x, _coord[3].y, _coord[3].z);
    center_coord = _center;
    color = _color;
    FrameColor = Color(255, 0, 0);

    display = true;

    // Свойства материала по умолчанию.
    Ambient[0] = 0.2f;
    Ambient[1] = 0.2f;
    Ambient[2] = 0.2f;
    Ambient[3] = 1.0f;
    Diffuse[0] = 0.8f;
    Diffuse[1] = 0.8f;
    Diffuse[2] = 0.8f;
    Diffuse[3] = 1.0f;
    Specular[0] = 0.0f;
    Specular[1] = 0.0f;
    Specular[2] = 0.0f;
    Specular[3] = 1.0f;
    Emission[0] = 0.0f;
    Emission[1] = 0.0f;
    Emission[2] = 0.0f;
    Emission[3] = 1.0f;
    Shininess = 0.0f;
}
void Tetrahedron::Draw() {
    // Сохраняем текущую матрицу.
    glPushMatrix();

    // Перемещаемся в центр фигуры.
    glTranslated(center_coord.x, center_coord.y, center_coord.z);

    // Рисуем тетраэдр.
    glBegin(GL_POLYGON);
    glColor3f(color.red, color.green, color.blue);
    glVertex3f(coord[0].x, coord[0].y, coord[0].z);
    glVertex3f(coord[1].x, coord[1].y, coord[1].z);
    glVertex3f(coord[2].x, coord[2].y, coord[2].z);
    glEnd();

    glBegin(GL_POLYGON);
    glColor3f(color.red, color.green, color.blue);
    glVertex3f(coord[1].x, coord[1].y, coord[1].z);
    glVertex3f(coord[3].x, coord[3].y, coord[3].z);
    glVertex3f(coord[2].x, coord[2].y, coord[2].z);
    glEnd();

    glBegin(GL_POLYGON);
    glColor3f(color.red, color.green, color.blue);
    glVertex3f(coord[3].x, coord[3].y, coord[3].z);
    glVertex3f(coord[0].x, coord[0].y, coord[0].z);
    glVertex3f(coord[2].x, coord[2].y, coord[2].z);
    glEnd();

    glBegin(GL_POLYGON);
    glColor3f(color.red, color.green, color.blue);
    glVertex3f(coord[1].x, coord[1].y, coord[1].z);
    glVertex3f(coord[0].x, coord[0].y, coord[0].z);
    glVertex3f(coord[3].x, coord[3].y, coord[3].z);
    glEnd();

    // Восстанавливаем матрицу.
    glPopMatrix();
}

```

```

}
void Tetrahedron::DrawFrame() {
    // Сохраняем текущую матрицу.
    glPushMatrix();

    // Перемещаемся в центр фигуры.
    glTranslated(center_coord.x, center_coord.y, center_coord.z);

    // Рисуем тетраэдр.
    glBegin(GL_LINE_LOOP);
    glLineWidth(5);
    glColor3f(FrameColor.red, FrameColor.green, FrameColor.blue);
    glVertex3f(coord[0].x, coord[0].y, coord[0].z);
    glVertex3f(coord[1].x, coord[1].y, coord[1].z);
    glVertex3f(coord[2].x, coord[2].y, coord[2].z);
    glEnd();

    glBegin(GL_LINE_LOOP);
    glColor3f(FrameColor.red, FrameColor.green, FrameColor.blue);
    glVertex3f(coord[1].x, coord[1].y, coord[1].z);
    glVertex3f(coord[3].x, coord[3].y, coord[3].z);
    glVertex3f(coord[2].x, coord[2].y, coord[2].z);
    glEnd();

    glBegin(GL_LINE_LOOP);
    glColor3f(FrameColor.red, FrameColor.green, FrameColor.blue);
    glVertex3f(coord[3].x, coord[3].y, coord[3].z);
    glVertex3f(coord[0].x, coord[0].y, coord[0].z);
    glVertex3f(coord[2].x, coord[2].y, coord[2].z);
    glEnd();

    glBegin(GL_LINE_LOOP);
    glColor3f(FrameColor.red, FrameColor.green, FrameColor.blue);
    glVertex3f(coord[1].x, coord[1].y, coord[1].z);
    glVertex3f(coord[0].x, coord[0].y, coord[0].z);
    glVertex3f(coord[3].x, coord[3].y, coord[3].z);
    glEnd();

    // Восстанавливаем матрицу.
    glPopMatrix();
}

bool Tetrahedron::hit(Ray ray, Intersection &inter) {
    float timeIn[4]; // Время соударения с каждой плоскостью.
    Vector3f N[4];   // Нормаль к каждой плоскости.

    // Число пересечений равно 0.
    inter.numHits = 0;

    // Луч в базовых координатах.
    Ray genRay = transformRay(ray, center_coord, 1, 1, 1);

    // Если нет пересечения с первой плоскостью (0 1 2).
    if (!hitPlane(genRay, 0, 1, 2, N[0], timeIn[0]))
        timeIn[0] = -10000.0;
    else
        inter.numHits++;

    // Если нет пересечения со второй плоскостью (1 3 2).
    if (!hitPlane(genRay, 2, 3, 1, N[1], timeIn[1]))
        timeIn[1] = -10000.0;
    else
        inter.numHits++;

    // Если нет пересечения с третьей плоскостью (3 0 2).
    if (!hitPlane(genRay, 3, 0, 2, N[2], timeIn[2]))
        timeIn[2] = -10000.0;
    else
        inter.numHits++;

    // Если нет пересечения с четвертой плоскостью (1 0 3).
    if (!hitPlane(genRay, 3, 0, 1, N[3], timeIn[3]))
        timeIn[3] = -10000.0;
    else
        inter.numHits++;

    /* Случай, что точка обзора(начало луча) находится в тетраэдре не рассматривается
    если соударений меньше двух, соударений нет. */
    if (inter.numHits < 2)
        return false;
    else {
        // Определить точку входа и точку выхода из фигуры.
        int max;
        int min;

        max = 0;

        for (int i = 1; i < 4; i++)
            if (timeIn[i] > timeIn[max])

```

```

        max = i;

    min = max;
    for (int i = 0; i < 4; i++) {
        if (timeIn[i] < timeIn[min] && timeIn[i] > 0.0)
            min = i;
    }

    // Если максимальное время отрицательно - соударения нет.
    if (timeIn[max] < 0.0)
        return false;

    // Формируем записи о соударениях.
    inter.hit[0].objectType = 1;
    inter.hit[0].hitTime = timeIn[min];
    inter.hit[0].isEntering = true;
    inter.hit[0].surface = min;

    // Мировые координаты точки соударения.
    Vector3f P(rayPos(ray, timeIn[min]));
    inter.hit[0].hitPoint = Vector3f(P.x, P.y, P.z);

    // Нормаль к точке соударения.
    inter.hit[0].hitNormal = N[min];

    inter.hit[1].objectType = 1;
    inter.hit[1].hitTime = timeIn[max];
    inter.hit[1].isEntering = false;
    inter.hit[1].surface = max;

    // Мировые координаты точки соударения.
    Vector3f C(rayPos(ray, timeIn[max]));
    inter.hit[1].hitPoint = Vector3f(C.x, C.y, C.z);

    // Нормаль к точке соударения.
    inter.hit[1].hitNormal = N[max];

    return true;
}
}
bool Tetrahedron::hit(Ray ray) {
    float timeIn; // Время соударения с каждой плоскостью.
    Vector3f N[4]; // Нормаль к каждой плоскости.

    // Число пересечений равно 0.
    int num = 0;

    // Луч в базовых координатах.
    Ray genRay = transformRay(ray, center_coord, 1, 1, 1);

    // Если пересечение с первой плоскостью (0 1 2).
    if (hitPlane(genRay, 0, 1, 2, N[0], timeIn))
        if (timeIn >= 0.0 && timeIn <= 1.0)
            return true;

    // Если пересечение со второй плоскостью (2 3 1).
    if (hitPlane(genRay, 2, 3, 1, N[1], timeIn))
        if (timeIn >= 0.0 && timeIn <= 1.0)
            return true;

    // Если пересечение с третьей плоскостью (3 0 2).
    if (hitPlane(genRay, 3, 0, 2, N[2], timeIn))
        if (timeIn >= 0.0 && timeIn <= 1.0)
            return true;

    // Если пересечение с четвертой плоскостью (3 0 1).
    if (hitPlane(genRay, 3, 0, 1, N[3], timeIn))
        if (timeIn >= 0.0 && timeIn <= 1.0)
            return true;

    return false;
}
bool Tetrahedron::hitPlane(Ray ray, int ver1, int ver2, int ver3, Vector3f &N, float &time) {
    Vector3f P0; // Начальная точка луча.
    Vector3f Dir; // Направление луча.
    Vector3f A, B, C; // Вершины треугольника.
    Vector3f N; // Нормаль треугольника.

    Vector3f O; // Точка пересечения.

    P0 = ray.start;
    Dir = ray.dir;
    A = coord[ver1];
    B = coord[ver2];
    C = coord[ver3];
    N = count_normal_for_polygon(A, B, C);

    float vn = scalar(Dir, N);

```



```

// Находим точку соударения.
Vector3f V = P0 - A;
float t = -scalar(V, N) / vn;
V = Dir * t;
O = P0 + V;

// Определить принадлежность треугольнику.
// Выбираем плоскость для проекции YZ.
if (fabs(N.x) >= fabs(N.y) && fabs(N.x) >= fabs(N.z)) {
    if ((O.z - A.z) * (B.y - A.y) - (O.y - A.y) * (B.z - A.z) < 0)
        return false;
    if ((O.z - B.z) * (C.y - B.y) - (O.y - B.y) * (C.z - B.z) < 0)
        return false;
    if ((O.z - C.z) * (A.y - C.y) - (O.y - C.y) * (A.z - C.z) < 0)
        return false;
} else if (fabs(N.z) >= fabs(N.y)) { // Выбираем плоскость для проекции XY.
    if ((O.y - A.y) * (B.x - A.x) - (O.x - A.x) * (B.y - A.y) < 0)
        return false;
    if ((O.y - B.y) * (C.x - B.x) - (O.x - B.x) * (C.y - B.y) < 0)
        return false;
    if ((O.y - C.y) * (A.x - C.x) - (O.x - C.x) * (A.y - C.y) < 0)
        return false;
} else { // Выбираем плоскость для проекции XZ.
    if ((O.z - A.z) * (B.x - A.x) - (O.x - A.x) * (B.z - A.z) < 0)
        return false;
    if ((O.z - B.z) * (C.x - B.x) - (O.x - B.x) * (C.z - B.z) < 0)
        return false;
    if ((O.z - C.z) * (A.x - C.x) - (O.x - C.x) * (A.z - C.z) < 0)
        return false;
}
_N = N;
time = t;
return true;
}

/*
Реализация интерфейса класса "Куб"
*/
Square::Square() {
}
Square::Square(Vector3f _center, GLfloat _scaleX, GLfloat _scaleZ, Color _color) {
    center_coord = _center;
    scaleX = _scaleX;
    scaleZ = _scaleZ;
    color = _color;

    // Свойства материала по умолчанию.
    Ambient[0] = 0.2f;
    Ambient[1] = 0.2f;
    Ambient[2] = 0.2f;
    Ambient[3] = 1.0f;
    Diffuse[0] = 0.8f;
    Diffuse[1] = 0.8f;
    Diffuse[2] = 0.8f;
    Diffuse[3] = 1.0f;
    Specular[0] = 0.2f;
    Specular[1] = 0.2f;
    Specular[2] = 0.2f;
    Specular[3] = 1.0f;
    Emission[0] = 0.0f;
    Emission[1] = 0.0f;
    Emission[2] = 0.0f;
    Emission[3] = 1.0f;
    Shininess = 0.6f;
}
void Square::Draw() {
    // Сохраняем текущую матрицу.
    glPushMatrix();

    // Масштабируем квадрат.
    glScalef(scaleX, 1, scaleZ);

    // Перемещаемся в центр фигуры.
    glTranslated(center_coord.x, center_coord.y, center_coord.z);

    // Рисуем квадрат.
    glBegin(GL_POLYGON);
    glColor3f(color.red, color.green, color.blue);
    glVertex3f(-1, 0, -1);
    glVertex3f(-1, 0, 1);
    glVertex3f(1, 0, 1);
    glVertex3f(1, 0, -1);
    glEnd();

    // Восстанавливаем матрицу.
    glPopMatrix();
}

```

```

bool Square::hit(Ray ray, Intersection &inter) {
    Ray genRay;

    // Переход к базовому лучу.
    genRay = transformRay(ray, center_coord, scaleX, 1, scaleZ);

    inter.numHits = 0;

    double denom = genRay.dir.y;

    // Луч параллелен плоскости.
    if (fabs(denom) < 0.0001)
        return false;

    double time = -genRay.start.y / denom; // Время соударения.

    // Квадрат лежит позади взгляда.
    if (time <= 0.0)
        return false;

    double hx = genRay.start.x + genRay.dir.x * time;
    double hz = genRay.start.z + genRay.dir.z * time;

    // Проходит мимо в направлении x.
    if (hx > 1.0 || hx < -1.0)
        return false;

    // Проходит мимо в направлении y.
    if (hz > 1.0 || hz < -1.0)
        return false;

    // Есть соударение.
    inter.numHits = 1;
    inter.hit[0].objectType = 10;
    inter.hit[0].hitTime = time;
    inter.hit[0].isEntering = true;
    inter.hit[0].surface = 0;

    // Точка соударения в мировых координатах.
    Vector3f P(rayPos(ray, time));
    inter.hit[0].hitPoint = Vector3f(P.x, P.y, P.z);

    // Нормаль в базовых.
    inter.hit[0].hitNormal = Vector3f(0, 1, 0);

    return true;
}
bool Square::hit(Ray ray) {
    Ray genRay;

    // Переход к базовому лучу.
    genRay = transformRay(ray, center_coord, scaleX, 1, scaleZ);

    double denom = genRay.dir.y;

    // Луч параллелен плоскости.
    if (fabs(denom) < 0.0001)
        return false;

    double time = -genRay.start.y / denom; // Время соударения.

    // Квадрат лежит позади
    взгляда.
    if (time <= 0.0)
        return false;

    double hx = genRay.start.x + genRay.dir.x * time;
    double hz = genRay.start.z + genRay.dir.z * time;

    // Проходит мимо в направлении x.
    if (hx > 1.0 || hx < -1.0)
        return false;

    // Проходит мимо в направлении y.
    if (hz > 1.0 || hz < -1.0)
        return false;

    if (time >= 0 && time <= 1)
        return true;

    return false;
}

/*
Реализация интерфейса класса "Сцена"
*/
Scene::Scene() {
    set_data();
}

```

```

    add_del = false;
    sphere_mod = false;
    tetrahedron_mod = false;
    active_Sphere = 0;
    active_Tetrahedron = 0;
}
void Scene::set_data() {
    FILE *in, *in_mat;
    Sphere _Sphere;
    Tetrahedron _Tetrahedron;
    int type;

    in = fopen("scene_objects.txt", "r");
    in_mat = fopen("scene_materials.txt", "r");

    while (!feof(in)) {
        fscanf(in, "%d", &type);
        switch (type) {
            case 0: { // Сфера.
                float _radius;
                Vector3f _center;
                Color _color;

                fscanf(in, "%f", &_radius);
                fscanf(in, "%f%f%f", &_center.x, &_center.y, &_center.z);
                fscanf(in, "%f%f%f", &_color.red, &_color.green, &_color.blue);

                _Sphere = Sphere(_radius, _center, _color);

                int is;

                fscanf(in_mat, "%d", &is);
                // Если указано фоновое отражение.
                if (is)
                    for (int i = 0; i < 4; i++)
                        fscanf(in_mat, "%f", &_Sphere.Ambient[i]);

                fscanf(in_mat, "%d", &is);
                // Если указано рассеянное отражение.
                if (is)
                    for (int i = 0; i < 4; i++)
                        fscanf(in_mat, "%f", &_Sphere.Diffuse[i]);

                fscanf(in_mat, "%d", &is);
                // Если указано зеркальное отражение.
                if (is)
                    for (int i = 0; i < 4; i++)
                        fscanf(in_mat, "%f", &_Sphere.Specular[i]);

                fscanf(in_mat, "%d", &is);
                // Если указано собственное излучение.
                if (is)
                    for (int i = 0; i < 4; i++)
                        fscanf(in_mat, "%f", &_Sphere.Emission[i]);

                fscanf(in_mat, "%d", &is);
                // Если указан коэффициент блеска.
                if (is)
                    fscanf(in_mat, "%f", &_Sphere.Shininess);

                vector_Sphere.push_back(_Sphere);
            }
            break;
            case 1: { // Тетраэдр.
                Vector3f _coord[4];
                Vector3f _center;
                Color _color;

                for (int i = 0; i < 4; i++)
                    fscanf(in, "%f%f%f", &_coord[i].x, &_coord[i].y, &_coord[i].z);
                fscanf(in, "%f%f%f", &_center.x, &_center.y, &_center.z);
                fscanf(in, "%f%f%f", &_color.red, &_color.green, &_color.blue);

                _Tetrahedron = Tetrahedron(_coord, _center, _color);

                int is;

                fscanf(in_mat, "%d", &is);
                // Если указано фоновое отражение.
                if (is)
                    for (int i = 0; i < 4; i++)
                        fscanf(in_mat, "%f", &_Tetrahedron.Ambient[i]);

                fscanf(in_mat, "%d", &is);
                // Если указано рассеянное отражение.
                if (is)
                    for (int i = 0; i < 4; i++)

```

```

        fscanf(in_mat, "%f", &_Tetrahedron.Diffuse[i]);

        fscanf(in_mat, "%d", &is);
        // Если указано зеркальное отражение.
        if (is)
            for (int i = 0; i < 4; i++)
                fscanf(in_mat, "%f", &_Tetrahedron.Specular[i]);

        fscanf(in_mat, "%d", &is);
        // Если указано собственное излучение.
        if (is)
            for (int i = 0; i < 4; i++)
                fscanf(in_mat, "%f", &_Tetrahedron.Emission[i]);

        fscanf(in_mat, "%d", &is);
        // Если указан коэффициент блеска.
        if (is)
            fscanf(in_mat, "%f", &_Tetrahedron.Shininess);

        vector_Tetrahedron.push_back(_Tetrahedron);
    }
    break;
}
fclose(in);
fclose(in_mat);

// Плоскость.
square = Square(Vector3f(0, -2, 0), 10, 10, Color(255, 0, 0));

// Источники освещения
Light _Light;
_Light.position = Vector3f(-5, 2, -5);
_Light.color = Color(0.4, 0.4, 0.4);

vector_Light.push_back(_Light);

_Light.position = Vector3f(-13.0, 1.0, 8.0);
_Light.color = Color(0.1, 0.2, 0.5);
vector_Light.push_back(_Light);
}
void Scene::switch_forward() {
    // Работа со сферами.
    if (sphere_mod) {
        active_Sphere++;
        if (active_Sphere == vector_Sphere.size())
            active_Sphere = 0;
    }

    // Работа с тетраэдрами.
    if (tetrahedron_mod) {
        active_Tetrahedron++;
        if (active_Tetrahedron == vector_Tetrahedron.size())
            active_Tetrahedron = 0;
    }
}
void Scene::switch_backward() {
    // Работа со сферами.
    if (sphere_mod) {
        active_Sphere--;
        if (active_Sphere == -1)
            active_Sphere = vector_Sphere.size() - 1;
    }

    // Работа с тетраэдрами.
    if (tetrahedron_mod) {
        active_Tetrahedron--;
        if (active_Tetrahedron == -1)
            active_Tetrahedron = vector_Tetrahedron.size() - 1;
    }
}
void Scene::Draw() {
    for (int i = 0; i < vector_Sphere.size(); i++) {
        if (vector_Sphere[i].display)
            vector_Sphere[i].Draw();

        if (add_del && sphere_mod && i == active_Sphere)
            vector_Sphere[i].DrawFrame();
    }

    for (int i = 0; i < vector_Tetrahedron.size(); i++) {
        if (vector_Tetrahedron[i].display)
            vector_Tetrahedron[i].Draw();

        if (add_del && tetrahedron_mod && i == active_Tetrahedron)
            vector_Tetrahedron[i].DrawFrame();
    }
}

```

```

        square.Draw();
    }
    bool Scene::isInShadow(Ray ray) {
        for (int i = 0; i < vector_Sphere.size(); i++)
            if (vector_Sphere[i].display)
                if (vector_Sphere[i].hit(ray))
                    return true;
        for (int i = 0; i < vector_Tetrahedron.size(); i++)
            if (vector_Tetrahedron[i].display)
                if (vector_Tetrahedron[i].hit(ray))
                    return true;
        if (square.hit(ray))
            return true;
        return false;
    }

    //=====Для преобразования луча к базовому виду=====
    // Определение матрицы трансформации.
    void makeTransformMatrix(double **TMatrix, Vector3f shift, GLfloat scaleX, GLfloat scaleY, GLfloat scaleZ) {
        TMatrix[0][0] = scaleX;
        TMatrix[0][1] = 0;
        TMatrix[0][2] = 0;
        TMatrix[0][3] = shift.x;
        TMatrix[1][0] = 0;
        TMatrix[1][1] = scaleY;
        TMatrix[1][2] = 0;
        TMatrix[1][3] = shift.y;
        TMatrix[2][0] = 0;
        TMatrix[2][1] = 0;
        TMatrix[2][2] = scaleZ;
        TMatrix[2][3] = shift.z;
        TMatrix[3][0] = 0;
        TMatrix[3][1] = 0;
        TMatrix[3][2] = 0;
        TMatrix[3][3] = 1;
    }

    // Приведение луча к базовому виду.
    Ray transformRay(Ray _ray, Vector3f shift, GLfloat scaleX, GLfloat scaleY, GLfloat scaleZ) {
        double **TMatrix;
        TMatrix = new double *[4];
        for (int i = 0; i < 4; i++)
            TMatrix[i] = new double[4];

        // Определить матрицу трансформации.
        makeTransformMatrix(TMatrix, shift, scaleX, scaleY, scaleZ);

        // Определить начало луча и направление в однородных координатах.
        double point[4] = { _ray.start.x, _ray.start.y, _ray.start.z, 1.0f };
        double dir[4] = { _ray.dir.x, _ray.dir.y, _ray.dir.z, 0.0f };

        double sol_point[4];
        double sol_dir[4];

        // Найти базовые координаты начала луча и его направления.
        solve_gauss(TMatrix, point, sol_point, 4);
        solve_gauss(TMatrix, dir, sol_dir, 4);

        // Перевести в обычные координаты.
        Vector3f newPoint = Vector3f(sol_point[0], sol_point[1], sol_point[2]);
        Vector3f newDir = Vector3f(sol_dir[0], sol_dir[1], sol_dir[2]);

        // Создать новый луч.
        Ray newRay;
        newRay.SetStart(newPoint);
        newRay.SetDir(newDir);

        for (int i = 0; i < 4; i++)
            delete[] TMatrix[i];
        delete[] TMatrix;

        // Вернуть новый луч.
        return newRay;
    }

    // Скалярное произведение.
    double scal(Vector3f vec1, Vector3f vec2) {
        return vec1.x * vec2.x + vec1.y * vec2.y + vec1.z * vec2.z;
    }

    // Определить позицию луча.
    Vector3f rayPos(Ray ray, double t) {
        Vector3f vec;
        vec = Vector3f(ray.start.x + ray.dir.x * t,
            ray.start.y + ray.dir.y * t,
            ray.start.z + ray.dir.z * t);
        return vec;
    }
}

```

tracer.h

```
/*
Интерфейс класса, отвечающего за трассировку лучей.
*/

#include "scene.h"

class Raytracer {
public:
    Vector3f eye;        // Точка выхода лучей/положение камеры.
    Scene scene;         // Сцена.
    Camera camera;       // Камера.
    float aspect;
    int blockSize;       // Размер блока пикселей.
    int nCols;
    int nRows;

    Raytracer();
    Raytracer(
        Vector3f _eye,
        Scene _scene,
        Camera _camera,
        float _aspect,
        int _nCols,
        int _nRows,
        int _blockSize
    );

    void Raytrace();      // Трассировка.
    Color Shade(Ray ray); // Ядро трассировки.
    Intersection getFirstHit(Ray ray); // Метод, для нахождения первого препятствия на пути следования луча.
};
```

tracer.cpp

```
#include "tracer.h"

/*
Реализация интерфейса класса описанного в файле "tracer.h".
*/

Raytracer::Raytracer() {
}

Raytracer::Raytracer(
    Vector3f _eye,
    Scene _scene,
    Camera _camera,
    float _aspect,
    int _nCols,
    int _nRows,
    int _blockSize
) {
    eye = _eye;
    scene = _scene;
    camera = _camera;
    aspect = _aspect;
    nCols = _nCols;
    nRows = _nRows;
    blockSize = _blockSize;
}

void Raytracer::Raytrace() {
    Ray ray;        // Луч.
    Color color;     // Цвет.
    float tetha = 60 * (M_PI / 180.); // Угол охвата.
    float N = 0.1;   // Расстояние от точки взгляда, до ближней плоскости.
    Vector3f n = camera.Position - camera.View;
    Vector3f u = normal(camera.UpVector, n);
    Vector3f v = normal(n, u);

    // Нормализация с.к., связанной с камерой.
    n = normalize(n);
    u = normalize(u);
    v = normalize(v);

    float H = N * tan(tetha / 2.0);
    float W = H * aspect;

    // Установить координаты начала луча.
    ray.SetStart(eye);

    // Текущий уровень рекурсии.
    ray.recurseLevel = 0;
    nCols = 250;
    nRows = 250;
    // Установки OpenGL для 2D-рисования.
```

```

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0, nCols, 0, nRows);
glDisable(GL_LIGHTING);

for (int row = 0; row < nRows; row += blockSize) {
    for (int col = 0; col < nCols; col += blockSize) {
        float x = -W + (col * 2 * W) / (float)nCols;
        float y = -H + (row * 2 * H) / (float)nRows;

        Vector3f direction;
        float coeff = 0.0001;          // Коэффициент отклонения луча.

                                                // Начальный цвет черный.
        color = Color(0, 0, 0);

        // Посылаем 4 луча из одной точки с небольшими отклонениями.
        // Для сглаживания изображения.
        for (int i = 0; i < 2; i++) {
            direction = Vector3f(
                -N * n.x + x * u.x + y * v.x + coeff,
                -N * n.y + x * u.y + y * v.y - coeff,
                -N * n.z + x * u.z + y * v.z - coeff
            );
            direction = normalize(direction);
            ray.SetDir(direction);
            color.add(Shade(ray));

            direction = Vector3f(
                -N * n.x + x * u.x + y * v.x + coeff,
                -N * n.y + x * u.y + y * v.y + coeff,
                -N * n.z + x * u.z + y * v.z - coeff
            );
            direction = normalize(direction);
            ray.SetDir(direction);
            color.add(Shade(ray));

            coeff = -coeff;
        }

        // Вычисляем усредненный цвет.
        color = Color(
            color.red / float(4),
            color.green / float(4),
            color.blue / float(4)
        );

        // Закрашиваем блок пикселей полученным цветом.
        glColor3f(
            color.red,
            color.green,
            color.blue
        );
        glRecti(
            col,
            row,
            col + blockSize,
            row + blockSize
        );
    }
}

Intersection Raytracer::getFirstHit(Ray ray) {
    Intersection inter;          // Запись пересечений.
    Intersection best;

    best.numHits = 0;

    // Цикл по всем сферам.
    for (int i = 0; i < scene.vector_Sphere.size(); i++) {
        // Если сфера должна отображаться.
        if (scene.vector_Sphere[i].display) {
            // Если нет соударения.
            if (!scene.vector_Sphere[i].hit(ray, inter))
                continue;

            // Если соударение было.
            // Записываем номер объекта соударения.
            for (int j = 0; j < inter.numHits; j++)
                inter.hit[j].objectNum = i;

            /* Если в best еще нет соударений или полученное соударение лучше,
               чем соударение в best. */
            if (best.numHits == 0 || inter.hit[0].hitTime < best.hit[0].hitTime) // Копируем inter в best.
                best.set(inter);
        }
    }
}

```

```

    }
}

// Цикл по всем тетраэдрам.
for (int i = 0; i < scene.vector_Tetrahedron.size(); i++) {
    // Аналогично для тетраэдров.
    if (scene.vector_Tetrahedron[i].display) {

        // Если нет соударения.
        if (!scene.vector_Tetrahedron[i].hit(ray, inter))
            continue;

        //если соударение было
        //записываем номер объекта соударения
        for (int j = 0; j < inter.numHits; j++) inter.hit[j].objectNum = i;

        /*      Если в best еще нет соударений или полученное соударение лучше,
        чем соударение в best. */
        if (best.numHits == 0 || inter.hit[0].hitTime < best.hit[0].hitTime) //копируем inter в best
            best.set(inter);
    }
}

// Для плоскости.
if (scene.square.hit(ray, inter)) {
    /*      Если в best еще нет соударений или полученное соударение лучше,
    чем соударение в best. */
    if (best.numHits == 0 || inter.hit[0].hitTime < best.hit[0].hitTime) //копируем inter в best
        best.set(inter);
}

return best;
}

Color Raytracer::Shade(Ray ray) {
    Color color; // Возвращаемый цвет данного луча.
    Intersection best; // Наилучшее соударение.
    int MAX = 4; // Максимальный уровень рекурсии.

    // Получить данные о лучшем соударении.
    best = getFirstHit(ray);

    // Копия данных о первом соударении.
    HitInfo h = best.hit[0];

    // Направление на наблюдателя.
    Vector3f v = Vector3f(
        -ray.dir.x,
        -ray.dir.y,
        -ray.dir.z
    );
    v = normalize(v);

    int typeObj = best.hit[0].objectType; // Тип объекта.
    int numObj = best.hit[0].objectNum; // Номер объекта.

    /*      Если луч прошел мимо всех объектов, вернуть фоновый цвет
    по умолчанию цвет - черный. */
    if (best.numHits == 0)
        return color;

    // Соударение со сферой.
    if (typeObj == 0) {
        // Установить эмиссионный цвет объекта.
        color.red = scene.vector_Sphere[numObj].Emission[0];
        color.green = scene.vector_Sphere[numObj].Emission[1];
        color.blue = scene.vector_Sphere[numObj].Emission[2];

        // Нормаль в точке соударения.
        Vector3f normal;
        normal = h.hitNormal;
        normal = normalize(normal);

        float eps = 0.0001; // Малое число.
        Ray feeler; // Щуп теней.

        feeler.start = h.hitPoint - ray.dir*eps; // Стартовая точка щупа.
        feeler.recurseLevel = 1;

        // Цикл по всем источникам освещения.
        for (int i = 0; i < scene.vector_Light.size(); i++) {
            // Добавить фоновое освещение.
            Color ambientCol = Color(
                scene.vector_Sphere[numObj].Ambient[0] * scene.vector_Light[i].color.red,
                scene.vector_Sphere[numObj].Ambient[1] * scene.vector_Light[i].color.green,
                scene.vector_Sphere[numObj].Ambient[2] * scene.vector_Light[i].color.blue
            );

```



```

        color.add(ambientCol.red, ambientCol.green, ambientCol.blue);

        // Обработка тени.
        feeler.dir = scene.vector_Light[i].position - h.hitPoint;           // Направление щупа.

        // Если точка в тени, диффузная и зеркальная компоненты не учитываются.
        if (scene.isInShadow(feeler)) continue;

        // Вектор от точки соударения до источника.
        Vector3f s = scene.vector_Light[i].position - h.hitPoint;
        s = normalize(s);

        // Член Ламберта.
        float mDotS = scal(s, normal);

        // Если точка соударения повернута к свету.
        if (mDotS > 0.0) {
            Color diffuseColor = Color(
                mDotS*scene.vector_Sphere[numObj].Diffuse[0] * scene.vector_Light[i].color.red,
                mDotS*scene.vector_Sphere[numObj].Diffuse[1] *
scene.vector_Light[i].color.green,
                mDotS*scene.vector_Sphere[numObj].Diffuse[2] * scene.vector_Light[i].color.blue
            );

            // Добавить диффузную составляющую.
            color.add(
                diffuseColor.red,
                diffuseColor.green,
                diffuseColor.blue
            );
        }

        Vector3f _h = v + s;
        _h = normalize(_h);

        // Член Фонга.
        float mDotH = scal(_h, normal);
        if (mDotH > 0) {
            float phong = pow(mDotH, scene.vector_Sphere[numObj].Shininess);
            Color specColor = Color(
                phong*scene.vector_Sphere[numObj].Specular[0] * scene.vector_Light[i].color.red,
                phong*scene.vector_Sphere[numObj].Specular[1] *
scene.vector_Light[i].color.green,
                phong*scene.vector_Sphere[numObj].Specular[2] * scene.vector_Light[i].color.blue
            );

            // Добавить зеркальную составляющую.
            color.add(
                specColor.red,
                specColor.green,
                specColor.blue
            );
        }

        // Если достигнут максимальный уровень рекурсии.
        if (ray.recurseLevel == MAX)
            return color;

        // Если объект достаточно блестящий.
        if (scene.vector_Sphere[numObj].Shininess > 0.5) {
            Ray reflectedRay; // Отраженный луч.
            Vector3f reflDir = ray.dir - normal * scalar(ray.dir, normal) * 2;           // Направление
отраженного луча.
            reflectedRay.start = h.hitPoint - ray.dir*eps;
            // Стартовая точка отраженного луча.
            reflDir = normalize(reflDir);
            reflectedRay.SetDir(reflDir);
            reflectedRay.recurseLevel = ray.recurseLevel + 1;
            // Увеличить уровень рекурсии.
            Color c = Color(
                scene.vector_Sphere[numObj].Specular[0],
                scene.vector_Sphere[numObj].Specular[1],
                scene.vector_Sphere[numObj].Specular[2]
            );

            // Добавить отраженный свет.
            color.add(Shade(reflectedRay), c);
        }
    }
}

// Соударение с тетраэдром.
if (typeObj == 1) {
    // Установить эмиссионный цвет объекта.
    color.red = scene.vector_Tetrahedron[numObj].Emission[0];
    color.green = scene.vector_Tetrahedron[numObj].Emission[1];
    color.blue = scene.vector_Tetrahedron[numObj].Emission[2];
}

```

```

// Нормаль в точке соударения.
Vector3f normal;
normal = h.hitNormal;
normal = normalize(normal);

float eps = 0.0001;           // Малое число.
Ray feeler;                   // Щуп теней.

feeler.start = h.hitPoint - ray.dir*eps;    // Стартовая точка щупа.
feeler.recurseLevel = 1;

// Цикл по всем источникам освещения.
for (int i = 0; i < scene.vector_Light.size(); i++) {
    // Добавить фоновое освещение.
    Color ambientCol = Color(
        scene.vector_Tetrahedron[numObj].Ambient[0] * scene.vector_Light[i].color.red,
        scene.vector_Tetrahedron[numObj].Ambient[1] * scene.vector_Light[i].color.green,
        scene.vector_Tetrahedron[numObj].Ambient[2] * scene.vector_Light[i].color.blue
    );

    color.add(
        ambientCol.red,
        ambientCol.green,
        ambientCol.blue
    );

    // Обработка тени.
    feeler.dir = scene.vector_Light[i].position - h.hitPoint;    // Направление щупа.

    // Если точка в тени, диффузная и зеркальная компоненты не учитываются.
    if (scene.isInShadow(feeler)) continue;

    // Вектор от точки соударения до источника.
    Vector3f s = scene.vector_Light[i].position - h.hitPoint;
    s = normalize(s);

    // Член Ламберта.
    float mDotS = scal(s, normal);

    // Если точка соударения повернута к свету.
    if (mDotS > 0.0) {
        Color diffuseColor = Color(
            mDotS*scene.vector_Tetrahedron[numObj].Diffuse[0] *
scene.vector_Light[i].color.red,
            mDotS*scene.vector_Tetrahedron[numObj].Diffuse[1] *
scene.vector_Light[i].color.green,
            mDotS*scene.vector_Tetrahedron[numObj].Diffuse[2] *
scene.vector_Light[i].color.blue
        );

        // Добавить диффузную составляющую.
        color.add(diffuseColor.red, diffuseColor.green, diffuseColor.blue);
    }

    Vector3f _h = v + s;
    _h = normalize(_h);

    // Член Фонга.
    float mDotH = scal(_h, normal);
    if (mDotH > 0) {
        float phong = pow(mDotH, scene.vector_Tetrahedron[numObj].Shininess);
        Color specColor = Color(
            phong*scene.vector_Tetrahedron[numObj].Specular[0] *
scene.vector_Light[i].color.red,
            phong*scene.vector_Tetrahedron[numObj].Specular[1] *
scene.vector_Light[i].color.green,
            phong*scene.vector_Tetrahedron[numObj].Specular[2] *
scene.vector_Light[i].color.blue
        );

        // Добавить зеркальную составляющую.
        color.add(
            specColor.red,
            specColor.green,
            specColor.blue
        );
    }

    // Если достигнут максимальный уровень рекурсии.
    if (ray.recurseLevel == MAX)
        return color;

    // Если объект достаточно блестящий.
    if (scene.vector_Tetrahedron[numObj].Shininess > 0.5) {
        Ray reflectedRay;
        Vector3f reflDir = ray.dir - normal * scalar(ray.dir, normal)*2.0;    // Направление
отраженного луча.
    }
}

```

```

        reflectedRay.start = h.hitPoint - ray.dir*eps;
// Стартовая точка отраженного луча.
        reflDir = normalize(reflDir);
        reflectedRay.SetDir(reflDir);
        reflectedRay.recurseLevel = ray.recurseLevel + 1;
        Color c = Color(
            scene.vector_Tetrahedron[numObj].Specular[0],
            scene.vector_Tetrahedron[numObj].Specular[1],
            scene.vector_Tetrahedron[numObj].Specular[2]
        );

        // Добавить отраженную составляющую.
        color.add(Shade(reflectedRay), c);
    }
}

// Соударение с плоскостью.
if (typeObj == 10) {
    // Установить эмиссионный цвет объекта.
    color.red = scene.square.Emission[0];
    color.green = scene.square.Emission[1];
    color.blue = scene.square.Emission[2];

    // Нормаль в точке соударения.
    Vector3f normal;
    normal = h.hitNormal;
    normal = normalize(normal);

    float eps = 0.0001; // Малое число.
    Ray feeler; // Щуп теней.

    feeler.start = h.hitPoint - ray.dir*eps; // Стартовая точка щупа.
    feeler.recurseLevel = 1;

    // Цикл по всем источникам освещения.
    for (int i = 0; i < scene.vector_Light.size(); i++) {
        // Добавить фоновое освещение.
        Color ambientCol = Color(
            scene.square.Ambient[0] * scene.vector_Light[i].color.red,
            scene.square.Ambient[1] * scene.vector_Light[i].color.green,
            scene.square.Ambient[2] * scene.vector_Light[i].color.blue
        );

        color.add(ambientCol.red, ambientCol.green, ambientCol.blue);

        // Обработка тени.
        feeler.dir = scene.vector_Light[i].position - h.hitPoint; // Направление щупа.

        // Если точка в тени, диффузная и зеркальная компоненты не учитываются.
        if (scene.isInShadow(feeler)) continue;

        // Вектор от точки соударения до источника.
        Vector3f s = scene.vector_Light[i].position - h.hitPoint;
        s = normalize(s);

        // Член Ламберта.
        float mDotS = scal(s, normal);

        // Если точка соударения повернута к свету.
        if (mDotS > 0.0) {
            Color diffuseColor = Color(
                mDotS*scene.square.Diffuse[0] * scene.vector_Light[i].color.red,
                mDotS*scene.square.Diffuse[1] * scene.vector_Light[i].color.green,
                mDotS*scene.square.Diffuse[2] * scene.vector_Light[i].color.blue
            );

            // Добавить диффузную составляющую.
            color.add(
                diffuseColor.red,
                diffuseColor.green,
                diffuseColor.blue
            );
        }

        Vector3f _h = v + s;
        _h = normalize(_h);

        // Член Фонга.
        float mDotH = scal(_h, normal);
        if (mDotH <= 0) continue;

        float phong = pow(mDotH, scene.square.Shininess);
        Color specColor = Color(
            phong*scene.square.Specular[0] * scene.vector_Light[i].color.red,
            phong*scene.square.Specular[1] * scene.vector_Light[i].color.green,
            phong*scene.square.Specular[2] * scene.vector_Light[i].color.blue
        );
    }
}

```

```

        // Добавить зеркальную составляющую.
        color.add(specColor.red, specColor.green, specColor.blue);

        // Если достигнут максимальный уровень рекурсии.
        if (ray.recurseLevel == MAX)
            return color;

        // Если объект достаточно блестящий.
        if (scene.square.Shininess > 0.5) {
            Ray reflectedRay;
            Vector3f reflDir = ray.dir - normal * scalar(ray.dir, normal) * 2;
            reflectedRay.start = h.hitPoint - ray.dir*eps;
            reflDir = normalize(reflDir);
            reflectedRay.SetDir(reflDir);
            reflectedRay.recurseLevel = ray.recurseLevel + 1;
            Color c = Color(
                scene.square.Specular[0],
                scene.square.Specular[1],
                scene.square.Specular[2]
            );

            // Добавить отраженную составляющую.
            color.add(Shade(reflectedRay), c);
        }
    }
    return color;
}

```

MAIN.cpp

```

#include "Hook.h"

void reshape(GLint w, GLint h)
{
    if (!raytracer_mode)
    {
        //изменяем размеры окна
        width = w;
        height = h;

        /* вычисляем соотношение между шириной и высотой */

        //предотвращаем деление на 0
        if (height == 0)
            height = 1;
        ratio = 1. * width / height;

        //устанавливаем матрицу проекции (определяет объем сцены)
        glMatrixMode(GL_PROJECTION);

        //загружаем единичную матрицу
        glLoadIdentity();

        //определяем окно просмотра
        glViewport(0, 0, width, height);

        //перспективная проекция
        gluPerspective(60, ratio, 0.1f, 100.0f);

        //возвращаемся к матрице модели
        glMatrixMode(GL_MODELVIEW);
    }
    else
    {
        //меняем размеры окна
        width = w;
        height = h;

        if (height == 0)
            height = 1;
        ratio = 1. * width / height;

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glViewport(0, 0, width, height);
        gluOrtho2D(0.0, width, 0.0, height);
    }
}

//отрисовка сетки
void draw_grid()
{
    glColor3ub(0, 255, 0);
    for (float i = -50; i <= 50; i += 1)
    {
        glBegin(GL_LINES);

```

```

        //ось X
        glVertex3f(-50, 0, i);
        glVertex3f(50, 0, i);

        //ось Z
        glVertex3f(i, 0, -50);
        glVertex3f(i, 0, 50);
        glEnd();
    }
}

//отрисовка на экран
void display(void)
{
    //очистка буфера цвета и глубины
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //включаем буфер глубины
    glEnable(GL_DEPTH_TEST);

    //обнуляем трансформации (т.е. загружаем единичную матрицу)
    glLoadIdentity();

    //устанавливаем вид камеры
    gluLookAt(camera.Position.x, camera.Position.y, camera.Position.z,
              camera.View.x, camera.View.y, camera.View.z,
              camera.UpVector.x, camera.UpVector.y, camera.UpVector.z);

    //отрисовка сетки
    draw_grid();

    //отрисовка сцены
    scene.Draw();

    //если включен режим трассировки
    if (raytracer_mode)
    {
        //устанавливаем параметры трассировщика
        raytracer = Raytracer(camera.Position, scene, camera, ratio, width, height, 1);

        //запустить трассировку
        raytracer.Raytrace();
    }

    glutPostRedisplay();

    //сменить буфер
    glutSwapBuffers();
}

//реакция на движение мыши
void mouse_move(int x, int y)
{
    if (rot)
        camera.SetViewByMouse(width, height);
}

//реакция на нажатие клавиш клавиатуры
void press_key(unsigned char key, int x, int y)
{
    //включение/выключение вращения камеры мышью
    if (key == 'q' || key == 'Q')
    {
        rot = !rot;
        ShowCursor(!rot);
    }

    //включение/выключение трассировки
    if (key == 't' || key == 'T')
    {
        raytracer_mode = !raytracer_mode;
        if (!raytracer_mode)
            reshape(width, height);
    }

    //включение/выключение режима включения/исключения фигур
    if (key == 'c' || key == 'C')
    {
        //переключить режим
        scene.add_del = !scene.add_del;

        //если режим включен
        if (scene.add_del)
        {
            //задать начальные установки
            scene.sphere_mod = true;
            scene.tetrahedron_mod = false;
            scene.active_Sphere = 0;
        }
    }
}

```

```

        scene.active_Tetrahedron = 0;
    }
    else
    {
        scene.sphere_mod = false;
        scene.tetrahedron_mod = false;
    }
}

//включить/исключить фигуру
if (key == 'e' || key == 'E')
{
    //если включен режим включения/исключения фигур
    if (scene.add_del)
    {
        if (scene.sphere_mod)
            scene.vector_Sphere[scene.active_Sphere].display = !scene.vector_Sphere[scene.active_Sphere].display;

        if (scene.tetrahedron_mod)
            scene.vector_Tetrahedron[scene.active_Tetrahedron].display = !scene.vector_Tetrahedron[scene.active_Tetrahedron].display;
    }
}

//движение камеры
if (key == 'w' || key == 'W')
{
    camera.MoveCamera(kSpeed);
}
if (key == 's' || key == 'S')
{
    camera.MoveCamera(-kSpeed);
}
if (key == 'a' || key == 'A')
{
    camera.RotateAroundPoint(camera.View, -kSpeed * 2.0f, 0.0f, 1.0f, 0.0f);
}
if (key == 'd' || key == 'D')
{
    camera.RotateAroundPoint(camera.View, kSpeed*2.0f, 0.0f, 1.0f, 0.0f);
}

//перемещение источника света
if (key == '8')
{
    scene.vector_Light[0].position.y++;
    printf("%.0f %.0f %.0f\n", scene.vector_Light[0].position.x, scene.vector_Light[0].position.y, scene.vector_Light[0].position.z);
}
if (key == '5')
{
    scene.vector_Light[0].position.y--;
    printf("%.0f %.0f %.0f\n", scene.vector_Light[0].position.x, scene.vector_Light[0].position.y, scene.vector_Light[0].position.z);
}
if (key == '7')
{
    scene.vector_Light[0].position.x++;
    printf("%.0f %.0f %.0f\n", scene.vector_Light[0].position.x, scene.vector_Light[0].position.y, scene.vector_Light[0].position.z);
}
if (key == '4')
{
    scene.vector_Light[0].position.x--;
    printf("%.0f %.0f %.0f\n", scene.vector_Light[0].position.x, scene.vector_Light[0].position.y, scene.vector_Light[0].position.z);
}
if (key == '9')
{
    scene.vector_Light[0].position.z++;
    printf("%.0f %.0f %.0f\n", scene.vector_Light[0].position.x, scene.vector_Light[0].position.y, scene.vector_Light[0].position.z);
}
if (key == '6')
{
    scene.vector_Light[0].position.z--;
    printf("%.0f %.0f %.0f\n", scene.vector_Light[0].position.x, scene.vector_Light[0].position.y, scene.vector_Light[0].position.z);
}
}

void press_special_key(int key, int x, int y)
{
    //переключать фигуры вперед
    if (key == GLUT_KEY_UP)

```

```

{
    if (scene.add_del)
        scene.switch_forward();
}

//переключать фигуры назад
if (key == GLUT_KEY_DOWN)
{
    if (scene.add_del)
        scene.switch_backward();
}

//сменить активный контейнер фигур
if (key == GLUT_KEY_LEFT)
{
    if (scene.add_del)
        if (scene.sphere_mod)
        {
            if (scene.vector_Tetrahedron.size()>0)
            {
                scene.sphere_mod = false;
                scene.tetrahedron_mod = true;
            }
        }
        else
        {
            if (scene.vector_Sphere.size()>0)
            {
                scene.sphere_mod = true;
                scene.tetrahedron_mod = false;
            }
        }
    }
    glutPostRedisplay();
}

//инициализация
void init()
{
    ratio = 1. * width / height;
    camera.PositionCamera(-12.0f, 4.0f, -3.0f, -5.5f, 1.0f, -1.5f, 0.0f, 1.0f, 0.0f); //установка начальной позиции ка-
меры
}

int main(int argc, char *argv[])
{
    width = 500;
    height = 500;
    glutInit(&argc, argv);

    //включить буфер глубины/двойную буферизацию
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowSize(width, height);
    glutCreateWindow("pure.zlo");
    init(); //начальные установки

    glutKeyboardFunc(key_press); //обработка клавиш с кодами ascii
    glutSpecialFunc(key_special_press); //обработка не-ascii клавиш

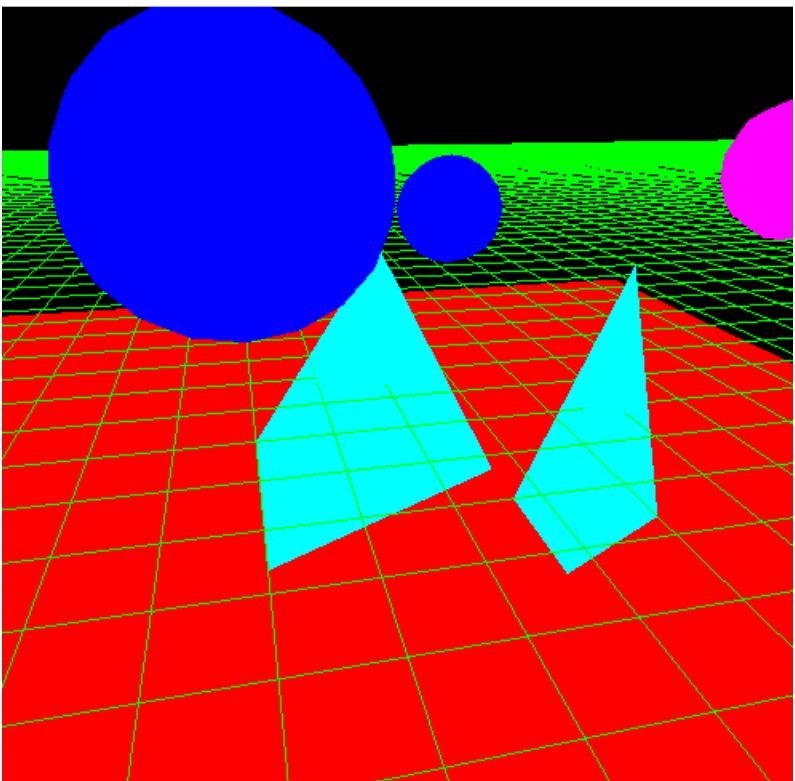
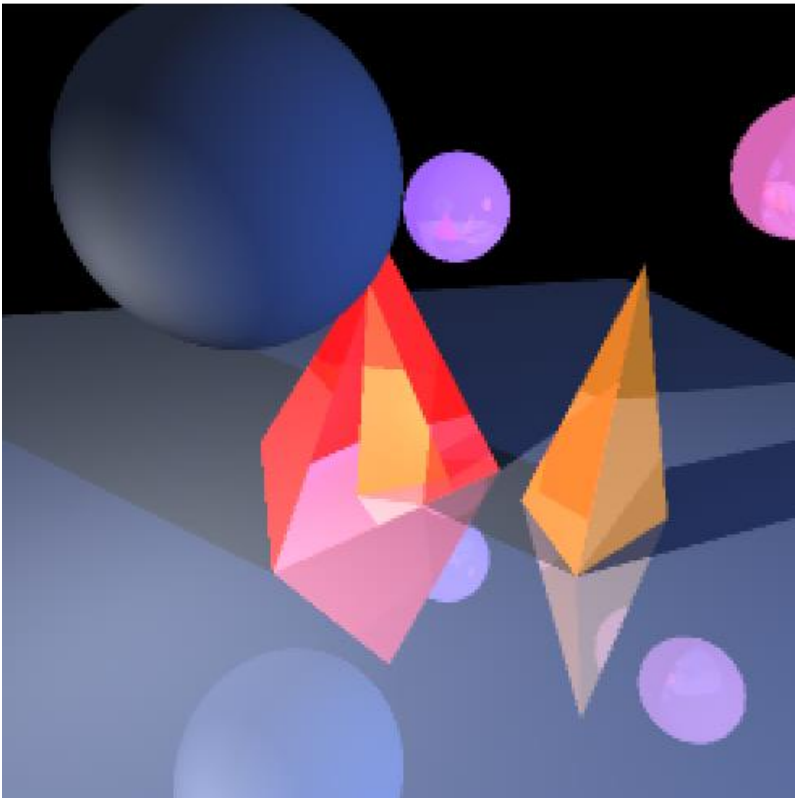
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);

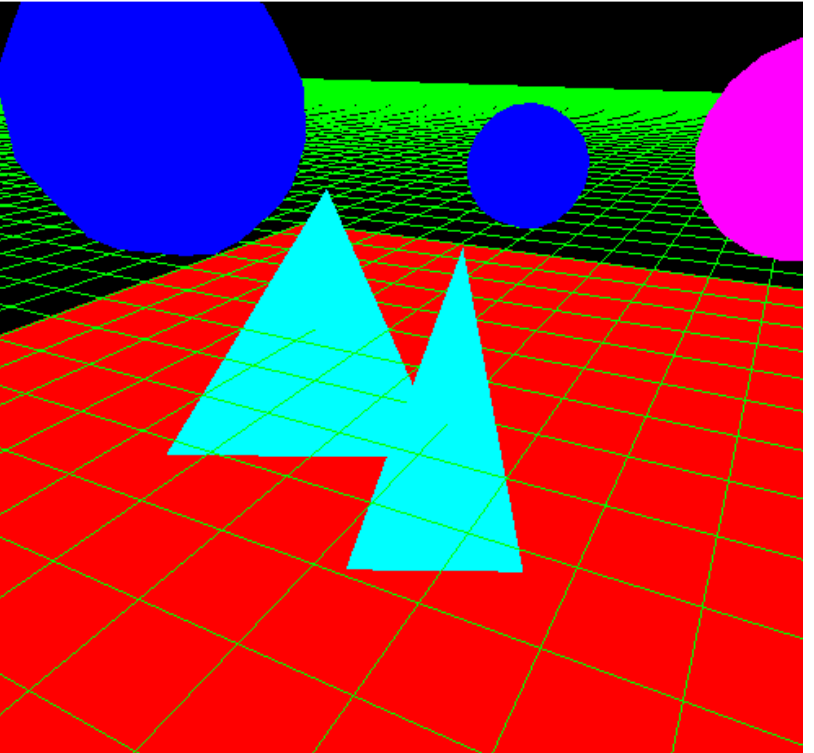
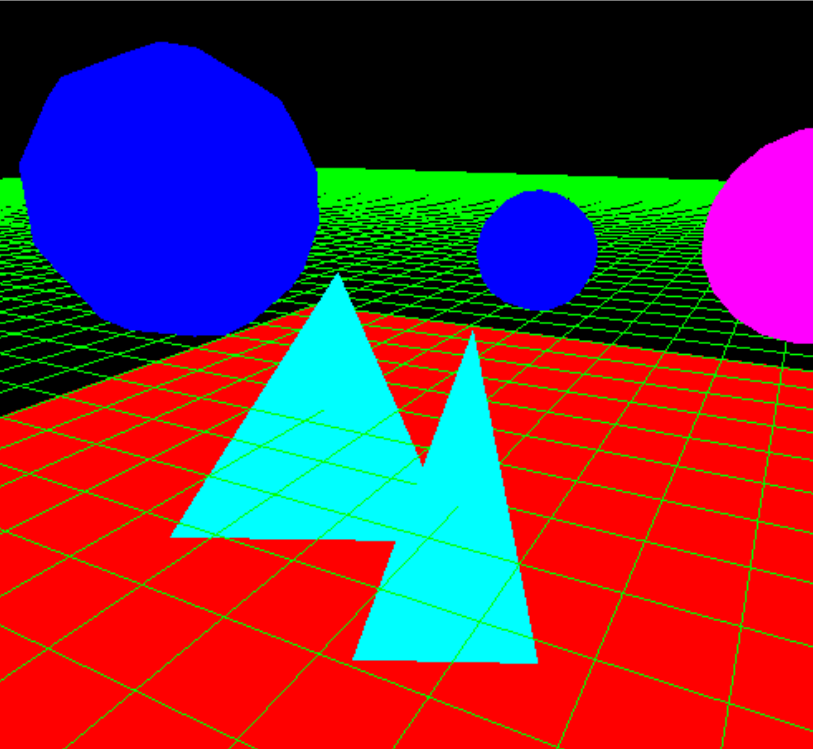
    glutPassiveMotionFunc(mouse_move);

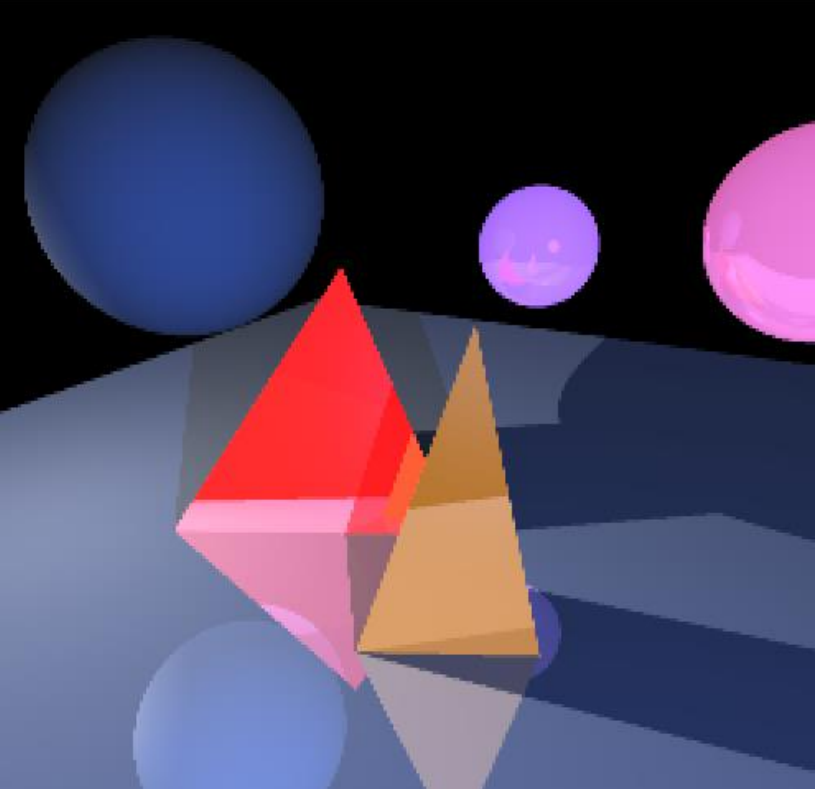
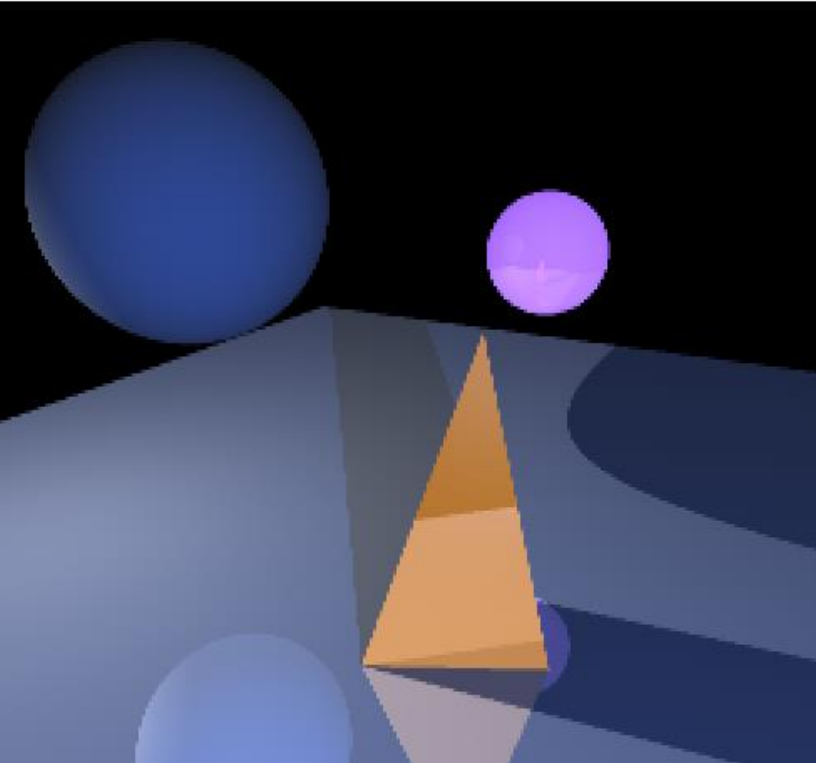
    glutMainLoop();
}

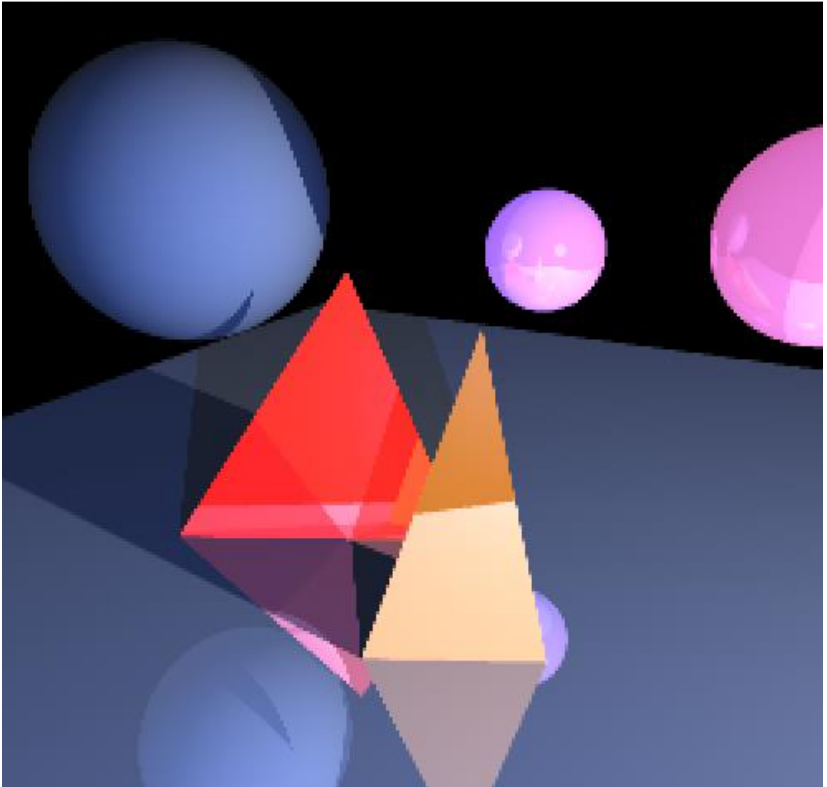
```

5. Тесты

№	Действие	Результат
1	Отображение сцены с объектами	
2	Включение режима трассировки	

3	<p>Выключение режим трассировки и поворот сцены</p>	
4	<p>Включение режима поворота камеры с помощью мыши и вращение камеры</p>	

5	Включение режима трассировки	
6	Включение режима включения/исключения фигур и исключение двух фигур	

7	Возвращение всех фигур и изменение положения камеры	
8	Изменение положения камеры	