|  |  |  |
| --- | --- | --- |
| Министерство образования и науки Российской Федерации | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования | | |
| «Новосибирский государственный технический университет» | | |
| path817.png | | |
| Кафедра прикладной математики | | |
|  | | |
| Лабораторная работа № 4 | | |
| по дисциплине «Компьютерная графика» | | |
|  | | |
|  | | |
|  | | |
| эмблема_светлая.png | Факультет: | ПМИ |
| Группа: | ПМИ-62 |
| Бригада: | 4 |
| Студенты: | Мамонова Е. В., Ершов П. К. |
| Преподаватель: | Задорожный А. Г. |
|  | | |
| Новосибирск | | |
| 2019 | | |

1. Цель работы

Ознакомиться с основными аспектами метода трассировки лучей.

1. **Задание**
   1. Считывать из файла (в зависимости от варианта) а) тип объекта; б) координаты и размер объектов; в) параметры материала объектов.
   2. Выполнить трассировку первичных лучей.
   3. Добавить зеркальную плоскость и учесть отраженные лучи.
   4. Предусмотреть возможность включения/исключения объектов.
   5. Предусмотреть возможность изменения положения источника света.
2. **Описание параметров фигур**

Параметры освещения загружаются из файлов mat1S.txt для сферы и mat1T.txt для тетраэдера.

Параметры организованы следующим образом.

1 1 1 1 1

1 1 1 1 1

1 1 1 1 1

1 50

Где:

|  |  |  |  |
| --- | --- | --- | --- |
| Флаг учёта параметров (1 – да, 0 – нет) | RGB составляющая | Параметр прозрачности | Тип отражения |
| 1 | 1 1 1 | 1 | Фоновое |
| 1 | 1 1 1 | 1 | Рассеянное |
| 1 | 1 1 1 | 1 | Зеркальное |
| 1 | 50 - Коэффициент блеска | |  |

Параметры объектов скачиваются из файла scene\_objects.txt.

Для сферы:

0 – тип фигуры (0 – сфера).

2 – радиус.

3 0 3 – координаты центра.

0 0 255 – цвет.

Для тетраэдера:

1 – тип фигуры (1 – тетраэдр).

-2 -2 -2 – координаты вершин грани.

2 -2 -2 – координаты вершин грани.

0 2 0 – координаты вершин грани.

0 -2 2 – координаты вершин грани.

0 0 0 – координаты вершин грани.

0 0 255 – цвет.

1. **Описание управления**

Выбор направленеи движения осуществляется с помощью мыши.

Управление клавишами.

W/w – вперёд.  
S/s – назад.  
A/a – поворот влево.  
D/d – поворот вправо.  
E/e – вкл/выкл режим исключения/включения фигур.

Q/q – включение/выключение выбора направления мышью.

Левая стрелка – выбрать фигуру (выделяется каркас).  
T/t - включение/отключение режима трассировки.  
R/r – включить/исключить фигуру.

Z/z – включить/выключить основание.

y – включить/выключить фоновый свет.

n – включить/выключить цвет тетраэдера.

h – включить/выключить цвет сферы.

Источник света точечный.  
1 – белый свет.  
2 – кравный свет.  
3 – фиолетоый свет.  
4 – синий свет.  
i – поднять источний света.

p – опустить источний света.

o – переместить вперёд источний света.

l – переместить назад источний света.

k – переместить влево источний света.

; – переместить вправо источний света.  
- – уменьшить размер пикселя в режиае трассировки.  
= – увеличить размер пикселя в режиае трассировки.

1. Текст программы

camera.h

#include <math.h>

#include <windows.h>

#include "glut.h"

#include <vector>

#include <gl\gl.h>

#include "normal.h"

class Camera

{

public:

Vector3f Position; //позиция камеры/положение точки наблюдателя/

Vector3f View; //направление наблюдения

Vector3f UpVector; //вектор поворота сцены

Camera(); //конструктор

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"

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

//Передвинуть камеру

Position.x += \_View.x \* speed; //изменить положение

Position.y += \_View.y \* speed;

Position.z += \_View.z \* speed; //камеры

View.x += \_View.x \* speed; //изменить направление

View.y += \_View.y \* speed;

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);

//пересчитаем координаты 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;

static float lastRotX = 0.0f;

lastRotX = currentRotX; //Сохраняем последний угол вращения

//и используем заново currentRotX

//если поворот больше одного градуса, умешьшим 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 = Cross(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 = Cross(View - Position, UpVector);

//нормализуем ось

vAxis = Normalize(vAxis);

//вращаем

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

}

}

//если в пределах 1.0f -1.0f - просто вращаем

else

{

Vector3f vAxis = Cross(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 вращатся

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

#include<math.h>

void sol\_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 sol\_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

#include "camera.h"

//луч

class Ray

{

public:

Vector3f start; //стартовая точка луча

Vector3f dir; //направление луча

int recurseLevel; //текущий уровень рекурсии

Ray(); //конструктор

void SetStart(Vector3f \_start); //установить стартовую точку

void SetDir(Vector3f \_dir); //установить направление

};

//информация 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);

};

//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"

//описание методов класса Ray

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);

}

//описание методов класса HitInfo

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);

}

//описание методов класса Intersection

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]);

}

normal.h

#define \_USE\_MATH\_DEFINES

#include<math.h>

//ВЫЧИСЛЕНИЕ НОРМАЛИ/ОТРИСОВКА ПОЛИГОНОВ/ ПРОТИВ/!!!!/ ЧАСОВОЙ СТРЕЛКИ

class Vector3f

{

public:

float x, y, z;

Vector3f(){}; //конструктор по умолчанию

Vector3f(float \_x, float \_y, float \_z) //конструктор

{

x = \_x;

y = \_y;

z = \_z;

}

//Переопределение оператора +

Vector3f operator + (Vector3f \_Vector)

{

return Vector3f(\_Vector.x + x, \_Vector.y + y, \_Vector.z + z);

}

//Переопределение оператора -

Vector3f operator - (Vector3f \_Vector)

{

return Vector3f(x - \_Vector.x, y - \_Vector.y, z - \_Vector.z);

}

//Переопределение оператора \*

Vector3f operator \* (float num)

{

return Vector3f(x \* num, y \* num, z \* num);

}

//Переопределение оператора /

Vector3f operator / (float num)

{

return Vector3f(x / num, y / num, z / num);

}

};

//Вычисление нормали двух векторов

Vector3f Cross(Vector3f \_Vec1, Vector3f \_Vec2);

//Вычисление нормы вектора

float Norm(Vector3f \_Vec);

//Нормализация вектора

Vector3f Normalize(Vector3f \_Vec);

//Вычисление скалярного произведения

float scalar(Vector3f \_Vec1, Vector3f \_Vec2);

//Вектор между двумя точками

Vector3f Vector(Vector3f \_Point1, Vector3f \_Point2);

//Вычисление нормали полигона

Vector3f Normal(Vector3f \_1, Vector3f \_2, Vector3f \_3);

normal.cpp

#include "Normal.h"

//Вычисление нормали двух векторов

Vector3f Cross(Vector3f \_Vec1, Vector3f \_Vec2)

{

Vector3f \_Normal;

//вычисление векторного произведения

\_Normal.x = ((\_Vec1.y\*\_Vec2.z) - (\_Vec1.z\*\_Vec2.y));

\_Normal.y = ((\_Vec1.z\*\_Vec2.x) - (\_Vec1.x\*\_Vec2.z));

\_Normal.z = ((\_Vec1.x\*\_Vec2.y) - (\_Vec1.y\*\_Vec2.x));

return \_Normal;

}

//Вычисление нормы вектора

float Norm(Vector3f \_Vec)

{

return (float)sqrt((\_Vec.x\*\_Vec.x) + (\_Vec.y\*\_Vec.y) + (\_Vec.z\*\_Vec.z));

}

//Нормализация вектора

Vector3f Normalize(Vector3f \_Vec)

{

//Вычислить норму вектора

float norm = Norm(\_Vec);

//нормализовать вектор

\_Vec = \_Vec / norm;

return \_Vec;

}

//Вычисление скалярного произведения

float scalar(Vector3f \_Vec1, Vector3f \_Vec2)

{

return \_Vec1.x\*\_Vec2.x + \_Vec1.y\*\_Vec2.y + \_Vec1.z\*\_Vec2.z;

}

//Вектор между двумя точками

Vector3f Vector(Vector3f \_Point1, Vector3f \_Point2)

{

Vector3f \_Vector;

\_Vector.x = \_Point1.x - \_Point2.x;

\_Vector.y = \_Point1.y - \_Point2.y;

\_Vector.z = \_Point1.z - \_Point2.z;

return \_Vector;

}

//Вычисление нормали полигона

Vector3f Normal(Vector3f \_1, Vector3f \_2, Vector3f \_3)

{

Vector3f \_Vector1 = Vector(\_3, \_2);

Vector3f \_Vector2 = Vector(\_2, \_1);

Vector3f \_Normal = Cross(\_Vector1, \_Vector2);

\_Normal = Normalize(\_Normal);

return \_Normal;

}

scene.h

//СЦЕНА

#include <math.h>

#include <windows.h>

#include "glut.h"

#include <vector>

#include <gl\gl.h>

#include <stdio.h>

#include "helpTracer.h"

#include "gauss.h"

using namespace std;

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

//И все остальные вспомогательные подпрограммы

//определить матрицу трансформирования

void makeTransformMatrix(double \*\*TMatrix, Vector3f shift, float scale);

//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);

//==================================================================

//int lighting\_num = 1;

class material\_properties

{

public:

GLfloat Ambient[4]; //фоновое отражение

GLfloat Diffuse[4]; //рассеянное отражение

GLfloat Specular[4]; //зеркальное отражение

GLfloat Shininess; //коэффициент блеска

};

//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 refl);

};

//источник света

class Light

{

public:

Vector3f position; //позиция источника света

Color color; //цвет источника света

Light(); //конструктор

void setPosition(Vector3f pos); //установить позицию источника света

void setColor(Color col); //установить цвет источника света

};

//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

material\_properties Properties;

Sphere(); //конструктор по умолчанию

Sphere(GLfloat \_radius, Vector3f \_center, Color \_color); //конструктор

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 пересечениях, используется для определения тени)

};

//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

material\_properties Properties; //свойства материала тетраэдра

Tetrahedron(); //конструктор по умолчанию

Tetrahedron(Vector3f \_coord[], Vector3f \_center, Color \_color); //конструктор

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 пересечениях, используется для определения тени)

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

};

//объект сцены: квадрат

class Square

{

public:

Vector3f center\_coord; //координаты центра плоскости

GLfloat scaleX; //длина по оси Х

GLfloat scaleZ; //длина по оси Z

Color color; //цвет квадрата

//свойства материала квадрата

material\_properties Properties;

Square(); //конструктор по умолчанию

Square(Vector3f \_center, GLfloat \_scaleX, GLfloat \_scaleZ, Color \_color); //конструктор

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)

};

//сцена

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

int lighting\_num;

Scene(); //конструктор

void set\_data(); //считать из файла координаты фигур и их параметры

Light set\_l(); // установить освещение

void set\_material();

void switch\_forward(); //переключение между фигурами вперед

void switch\_backward(); //переключение между фигурами назад

bool ambi = true; //параметр учёта фонового света

bool clsh = true; //параметр учёта цвета сферы

bool cltr = true; //параметр учёта цвета сферы

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"

//описание методов класса Color

Light \_Light;

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;

}

//описание методов класса Light

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);

}

//описание методов класса Sphere

Sphere::Sphere()

{

}

Sphere::Sphere(GLfloat \_radius, Vector3f \_center, Color \_color)

{

radius = \_radius;

center\_coord = \_center;

color = \_color;

FrameColor = Color(255, 0, 0);

display = true;

//свойства материала по умолчанию

Properties.Ambient[0] = 0.2f; Properties.Ambient[1] = 0.2f; Properties.Ambient[2] = 0.2f; Properties.Ambient[3] = 1.0f;

Properties.Diffuse[0] = 0.8f; Properties.Diffuse[1] = 0.8f; Properties.Diffuse[2] = 0.8f; Properties.Diffuse[3] = 1.0f;

Properties.Specular[0] = 0.0f; Properties.Specular[1] = 0.0f; Properties.Specular[2] = 0.0f; Properties.Specular[3] = 1.0f;

/\*Properties.Emission[0] = 0.0f; Properties.Emission[1] = 0.0f; Properties.Emission[2] = 0.0f; Properties.Emission[3] = 1.0f;\*/

Properties.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);

//коорднаты 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);

//коорднаты 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;

}

//описание методов класса Tetrahedron

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

Properties.Ambient[0] = 0.2f; Properties.Ambient[1] = 0.2f; Properties.Ambient[2] = 0.2f; Properties.Ambient[3] = 1.0f;

Properties.Diffuse[0] = 0.8f; Properties.Diffuse[1] = 0.8f; Properties.Diffuse[2] = 0.8f; Properties.Diffuse[3] = 1.0f;

Properties.Specular[0] = 0.0f; Properties.Specular[1] = 0.0f; Properties.Specular[2] = 0.0f; Properties.Specular[3] = 1.0f;

/\*Properties.Emission[0] = 0.0f; Properties.Emission[1] = 0.0f; Properties.Emission[2] = 0.0f; Properties.Emission[3] = 1.0f;\*/

Properties.Shininess = 0.0f;

}

void Tetrahedron::Draw()

{

Vector3f normal;

//сохраняем текущую матрицу

glPushMatrix();

//перемещаемся в центр фигуры

glTranslated(center\_coord.x, center\_coord.y, center\_coord.z);

//рисуем тетраэдр

normal = Normal(coord[0], coord[1], coord[2]);

glBegin(GL\_POLYGON);

glNormal3f(normal.x, normal.y, normal.z);

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();

normal = Normal(coord[1], coord[3], coord[2]);

glBegin(GL\_POLYGON);

glNormal3f(normal.x, normal.y, normal.z);

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();

normal = Normal(coord[3], coord[0], coord[2]);

glBegin(GL\_POLYGON);

glNormal3f(normal.x, normal.y, normal.z);

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();

normal = Normal(coord[1], coord[0], coord[3]);

glBegin(GL\_POLYGON);

glNormal3f(normal.x, normal.y, normal.z);

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 = Normal(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;

}

//выбираем плоскость для проекции XY

else if( fabs(N.z) >= fabs(N.y) )

{

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;

}

//выбираем плоскость для проекции XZ

else

{

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;

}

//описание методов класс Square

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

Properties.Ambient[0] = 0.2f; Properties.Ambient[1] = 0.2f; Properties.Ambient[2] = 0.2f; Properties.Ambient[3] = 1.0f;

Properties.Diffuse[0] = 0.8f; Properties.Diffuse[1] = 0.8f; Properties.Diffuse[2] = 0.8f; Properties.Diffuse[3] = 1.0f;

Properties.Specular[0] = 0.2f; Properties.Specular[1] = 0.2f; Properties.Specular[2] = 0.2f; Properties.Specular[3] = 1.0f;

/\*Properties.Emission[0] = 0.0f; Properties.Emission[1] = 0.0f; Properties.Emission[2] = 0.0f; Properties.Emission[3] = 1.0f;\*/

Properties.Shininess = 0.6f;

}

void Square::Draw()

{

Vector3f normal = Normal(Vector3f(-1, 0, -1), Vector3f(-1, 0, 1), Vector3f(1, 0, 1));

//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);

glNormal3f(normal.x, normal.y, normal.z);

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 вгзляда

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 х

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 вгзляда

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 х

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;

}

//описание методов класса Scene

Scene::Scene()

{

lighting\_num = 1;

set\_l();

set\_data();

add\_del = false;

sphere\_mod = false;

tetrahedron\_mod = false;

active\_Sphere = 0;

active\_Tetrahedron = 0;

}

Light Scene::set\_l()

{

//источники освещения

vector\_Light.clear();

if (lighting\_num == 1)

{

\_Light.position = Vector3f(-5, 2, -5);

\_Light.color = Color(1, 1, 1);

vector\_Light.push\_back(\_Light);

}

if (lighting\_num == 2)

{

\_Light.position = Vector3f(-13.0, 1.0, 8.0);

\_Light.color = Color(1, 0, 0);

vector\_Light.push\_back(\_Light);

}

if (lighting\_num == 3)

{

\_Light.position = Vector3f(0, 15, 0);

\_Light.color = Color(1, 0, 1);

vector\_Light.push\_back(\_Light);

}

if (lighting\_num == 4)

{

\_Light.position = Vector3f(0, 15, 0);

\_Light.color = Color(0, 0, 1);

vector\_Light.push\_back(\_Light);

}

return \_Light;

}

void Scene::set\_material()

{

char str[10] = "mat1S.txt";

for (int j= 0; j < vector\_Sphere.size(); j++)

{

FILE \*in\_mat = fopen(str, "r");

int is;

fscanf(in\_mat, "%d", &is);

//если указано фоновое отражение

if(is)

for(int i=0; i<4; i++)

fscanf(in\_mat, "%f", &vector\_Sphere[j].Properties.Ambient[i]);

fscanf(in\_mat, "%d", &is);

//если указано рассеянное отражение

if(is)

for(int i=0; i<4; i++)

fscanf(in\_mat, "%f", &vector\_Sphere[j].Properties.Diffuse[i]);

fscanf(in\_mat, "%d", &is);

//если указано зеркальное отражение

if(is)

for(int i=0; i<4; i++)

fscanf(in\_mat, "%f", &vector\_Sphere[j].Properties.Specular[i]);

fscanf(in\_mat, "%d", &is);

//если указан коэффициент блеска

if(is)

fscanf(in\_mat, "%f", &vector\_Sphere[j].Properties.Shininess);

str[3]++;

fclose(in\_mat);

}

char str\_[10] = "mat1T.txt";

for (int j = 0; j < vector\_Tetrahedron.size(); j++)

{

FILE \*in\_mat = fopen(str\_, "r");

int is;

fscanf(in\_mat, "%d", &is);

//если указано фоновое отражение

if (is)

for (int i = 0; i < 4; i++)

fscanf(in\_mat, "%f", &vector\_Tetrahedron[j].Properties.Ambient[i]);

fscanf(in\_mat, "%d", &is);

//если указано рассеянное отражение

if (is)

for (int i = 0; i < 4; i++)

fscanf(in\_mat, "%f", &vector\_Tetrahedron[j].Properties.Diffuse[i]);

fscanf(in\_mat, "%d", &is);

//если указано зеркальное отражение

if (is)

for (int i = 0; i < 4; i++)

fscanf(in\_mat, "%f", &vector\_Tetrahedron[j].Properties.Specular[i]);

fscanf(in\_mat, "%d", &is);

//если указан коэффициент блеска

if (is)

fscanf(in\_mat, "%f", &vector\_Tetrahedron[j].Properties.Shininess);

str\_[3]++;

fclose(in\_mat);

}

}

void Scene::set\_data()

{

FILE \*in, \*in\_mat;

Sphere \_Sphere;

Tetrahedron \_Tetrahedron;

int type;

in = fopen("scene\_objects.txt", "r");

while(!feof(in))

{

fscanf(in, "%d", &type);

switch (type)

{

//сфера

case 0:

{

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);

vector\_Sphere.push\_back(\_Sphere);

}

break;

//тетраэдр

case 1:

{

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);

vector\_Tetrahedron.push\_back(\_Tetrahedron);

}

break;

}

}

fclose(in);

set\_material();

//плоскость

square = Square(Vector3f(0,-2,0), 10, 10, Color(0,0,0));

}

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==============

//И все остальные вспомогательные подпрограммы

//определить матрицу 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

sol\_gauss(TMatrix, point, sol\_point, 4);

sol\_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**

#include"scene.h"

const int MAX = 4; //максимальный уровень рекурсии

struct sq

{

int x1;

int y1;

int x2;

int y2;

sq(int \_1, int \_2, int \_3, int \_4)

{

x1 = \_1;

y1 = \_2;

x2 = \_3;

y2 = \_4;

}

sq(){};

};

struct pix

{

Color color;

sq p;

pix(Color c, sq p\_)

{

color = c;

p = p\_;

}

pix() {};

};

extern vector<pix> PIX;

//трассировщик лучей

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(); //конструктор по умолчанию

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

void draw();

void Raytrace(); //трассировка

Color Shade(Ray ray); //ядро трассировки

Intersection getFirstHit(Ray ray); //находит объект, с которым луч соударяется первым

};

**tracer.cpp**

#include"tracer.h"

//описание методов класса Raytracer

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::draw()

{

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0, nCols, 0, nRows);

glDisable(GL\_LIGHTING);

for (int i = 0; i < PIX.size(); i++)

{

pix temp = PIX[i];

//закрашиваем блок пикселей полученным цветом

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

glRecti(temp.p.x1, temp.p.y1, temp.p.x2, temp.p.y2);

}

}

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

//параметры из этой ф-и gluPerspective(60, ratio, 0.1f, 100.0f);

//расчитываем новую систему координат, связанную с наблюдателем

Vector3f n = camera.Position-camera.View;

Vector3f u = Cross(camera.UpVector, n);

Vector3f v = Cross(n, u);

//нормализуем эту систему

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;

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; //коэффицент отклонения луча

//начальный цвет черный

color = Color(0, 0, 0);

//посылаем луч из двух точек

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));

//вычисляем усредненный цвет

color = Color(color.red/2., color.green/2., color.blue/2.);

sq temp(col, row, col + blockSize, row + blockSize);

pix t(color, temp);

PIX.push\_back(t);

}

}

}

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;

}

double getDistance(Light light, HitInfo col)//расчёт расстояния от источника света до точки фигуры

{

double x = abs(pow((light.position.x - col.hitPoint.x), 2));

double y = abs(pow((light.position.y - col.hitPoint.y), 2));

double z = abs(pow((light.position.z - col.hitPoint.z), 2));

return sqrt(x + y + z);

}

Color Raytracer::Shade(Ray ray)

{

Color color; //возвращаемый цвет данного луча

Intersection best;

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)

{

Color buf;

buf.red = scene.vector\_Light[0].color.red - color.red;

buf.green = scene.vector\_Light[0].color.green - color.green;

buf.blue = scene.vector\_Light[0].color.blue - color.blue;

return scene.vector\_Light[0].color;

}

//соударение со сферой

if(typeObj == 0)

{

//установить цвет объекта

if(scene.clsh == true)

color = scene.vector\_Sphere[numObj].color;

//нормаль в точке соударения

Vector3f normal;

normal = h.hitNormal;

normal = Normalize(normal);

float eps = 0.0001;

Ray feeler;

feeler.start = h.hitPoint - ray.dir\*eps;

feeler.recurseLevel = 1;

//цикл по всем источникам освещения

for(int i = 0; i < scene.vector\_Light.size(); i++)

{

//добавить фоновое освещение

double d = getDistance(scene.vector\_Light[i], h);

d = 1 / (0.01 + 0.04 \* d);

if (scene.ambi == true)

{

Color ambientCol = Color(scene.vector\_Sphere[numObj].Properties.Ambient[0] \* scene.vector\_Light[i].color.red,

scene.vector\_Sphere[numObj].Properties.Ambient[1] \* scene.vector\_Light[i].color.green,

scene.vector\_Sphere[numObj].Properties.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;

//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);

float mDotS = scal(s, normal);

//если точка соударения повернута к свету

if(mDotS > 0.0)

{

Color c = Color(mDotS\*scene.vector\_Sphere[numObj].Properties.Diffuse[0]\*scene.vector\_Light[i].color.red / d,

mDotS\*scene.vector\_Sphere[numObj].Properties.Diffuse[1]\*scene.vector\_Light[i].color.green / d,

mDotS\*scene.vector\_Sphere[numObj].Properties.Diffuse[2]\*scene.vector\_Light[i].color.blue / d);

//добавить диффузную составляющую

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

}

Vector3f \_h = v + s;

\_h = Normalize(\_h);

float mDotH = scal(\_h, normal);

if(mDotH > 0)

{

float phong = pow(mDotH, scene.vector\_Sphere[numObj].Properties.Shininess);

Color specColor = Color(phong\*scene.vector\_Sphere[numObj].Properties.Specular[0]\*scene.vector\_Light[i].color.red / d,

phong\*scene.vector\_Sphere[numObj].Properties.Specular[1]\*scene.vector\_Light[i].color.green / d,

phong\*scene.vector\_Sphere[numObj].Properties.Specular[2]\*scene.vector\_Light[i].color.blue / d);

//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].Properties.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].Properties.Specular[0], scene.vector\_Sphere[numObj].Properties.Specular[1], scene.vector\_Sphere[numObj].Properties.Specular[2]);

//добавить отраженный свет

color.add(Shade(reflectedRay), c);

}

}

}

//соударение с тетраэдром

if(typeObj == 1)

{

//установить цвет объекта

if (scene.cltr == true)

color = scene.vector\_Tetrahedron[numObj].color;

//нормаль в точке соударения

Vector3f normal;

normal = h.hitNormal;

normal = Normalize(normal);

float eps = 0.0001;

Ray feeler;

feeler.start = h.hitPoint - ray.dir\*eps;

feeler.recurseLevel = 1;

//цикл по всем источникам освещения

for(int i = 0; i < scene.vector\_Light.size(); i++)

{

double d = getDistance(scene.vector\_Light[i], h);

d = 1 / (0.01 + 0.04 \* d);

//добавить фоновое освещение

if (scene.ambi == true)

{

Color ambientCol = Color(scene.vector\_Sphere[numObj].Properties.Ambient[0] \* scene.vector\_Light[i].color.red,

scene.vector\_Sphere[numObj].Properties.Ambient[1] \* scene.vector\_Light[i].color.green,

scene.vector\_Sphere[numObj].Properties.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);

float mDotS = scal(s, normal);

//если точка соударения повернута к свету

if(mDotS > 0.0)

{

Color c = Color(mDotS\*scene.vector\_Tetrahedron[numObj].Properties.Diffuse[0]\*scene.vector\_Light[i].color.red / d,

mDotS\*scene.vector\_Tetrahedron[numObj].Properties.Diffuse[1]\*scene.vector\_Light[i].color.green / d,

mDotS\*scene.vector\_Tetrahedron[numObj].Properties.Diffuse[2]\*scene.vector\_Light[i].color.blue / d);

//добавить диффузную составляющую

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

}

Vector3f \_h = v + s;

\_h = Normalize(\_h);

float mDotH = scal(\_h, normal);

if(mDotH > 0)

{

float phong = pow(mDotH, scene.vector\_Tetrahedron[numObj].Properties.Shininess);

Color specColor = Color(phong\*scene.vector\_Tetrahedron[numObj].Properties.Specular[0]\*scene.vector\_Light[i].color.red / d,

phong\*scene.vector\_Tetrahedron[numObj].Properties.Specular[1]\*scene.vector\_Light[i].color.green / d,

phong\*scene.vector\_Tetrahedron[numObj].Properties.Specular[2]\*scene.vector\_Light[i].color.blue / d);

//добавить зеркальную составляющую

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

}

//если достигнут максимальный уровень рекурсии

if(ray.recurseLevel == MAX)

return color;

//\_Light.color;

//если объект достаточно блестящий

if(scene.vector\_Tetrahedron[numObj].Properties.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].Properties.Specular[0], scene.vector\_Tetrahedron[numObj].Properties.Specular[1], scene.vector\_Tetrahedron[numObj].Properties.Specular[2]);

//добавить отраженную составляющую

color.add(Shade(reflectedRay), c);

}

}

}

//соударение с плоскостью

if(typeObj == 10)

{

//установить цвет объекта

color.red = scene.square.color.red;

color.green = scene.square.color.green;

color.blue = scene.square.color.blue;

//нормаль в точке соударения

Vector3f normal;

normal = h.hitNormal;

normal = Normalize(normal);

float eps = 0.0001;

Ray feeler;

feeler.start = h.hitPoint - ray.dir\*eps;

feeler.recurseLevel = 1;

//цикл по всем источникам освещения

for(int i = 0; i < scene.vector\_Light.size(); i++)

{

double d = getDistance(scene.vector\_Light[i], h);

//добавить фоновое освещение

Color ambientCol = Color(scene.square.Properties.Ambient[0]\*scene.vector\_Light[i].color.red,

scene.square.Properties.Ambient[1]\*scene.vector\_Light[i].color.green,

scene.square.Properties.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;

//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 с = Color(mDotS\*scene.square.Properties.Diffuse[0]\*scene.vector\_Light[i].color.red / d,

mDotS\*scene.square.Properties.Diffuse[1]\*scene.vector\_Light[i].color.green / d,

mDotS\*scene.square.Properties.Diffuse[2]\*scene.vector\_Light[i].color.blue / d);

//добавить диффузную составляющую

color.add(с.red, с.green, с.blue);

}

Vector3f \_h = v + s;

\_h = Normalize(\_h);

float mDotH = scal(\_h, normal);

if (mDotH <= 0) continue;

float phong = pow(mDotH, scene.square.Properties.Shininess);

Color specColor = Color(phong\*scene.square.Properties.Specular[0]\*scene.vector\_Light[i].color.red / d,

phong\*scene.square.Properties.Specular[1]\*scene.vector\_Light[i].color.green / d,

phong\*scene.square.Properties.Specular[2]\*scene.vector\_Light[i].color.blue / d);

//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.Properties.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.Properties.Specular[0], scene.square.Properties.Specular[1], scene.square.Properties.Specular[2]);

//добавить отраженную составляющую

color.add(Shade(reflectedRay), c);

}

}

}

return color;

}

**header.h**

#define \_USE\_MATH\_DEFINES

#include <math.h>

#include <windows.h>

#include "glut.h"

#include <vector>

#include <gl\gl.h>

#include <stdio.h>

#include "tracer.h"

#include <iostream>

#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 camera1; //камера

Scene scene1; //сцена

Raytracer raytracer1; //трассировщик

Light \_LT;

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

bool raytracer\_mode = false; //режим трассировки

bool en\_material = false;

bool g\_bLight = true;

int block\_size = 1;

bool changes = false;

**main.cpp**

#include "header.h"

vector<pix> PIX;

void Reshape(GLint w, GLint h)

{

if(!raytracer\_mode)

{

//изменить размеры окна

width = w;

height = h;

//вычислить соотношение между шириной и высотой

//предотвратить деление на 0

if(height == 0)

height = 1;

ratio = 1. \* width / height;

//установить матрицу проекции/определяет объем сцены

glMatrixMode(GL\_PROJECTION);

//загрузить единичную матрицу

glLoadIdentity();

//определить окно просмотра

glViewport(0, 0, width, height);

//использовать перспективную проекциию

gluPerspective(60, ratio, 0.1f, 100.0f);

//возврат к матрице модели

glMatrixMode(GL\_MODELVIEW);

}

else

{

//изменить 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);

}

changes = true;

}

void tune\_lighting()

{

if(en\_material)

glEnable(GL\_COLOR\_MATERIAL);

else

glDisable(GL\_COLOR\_MATERIAL);

GLfloat Diffuse[] = { 3.0, 3.0, 3.0, 1.0 };

glMaterialfv(GL\_FRONT\_AND\_BACK, GL\_AMBIENT\_AND\_DIFFUSE, Diffuse);

// отрисовать источник света

for (int i = 0; i < scene1.vector\_Light.size(); i++)

{

Light temp = scene1.vector\_Light[i];

Sphere \*l = new Sphere(0.1, temp.position, temp.color);

l->Draw();

GLfloat light1\_Diffuse[] = { temp.color.red, temp.color.green, temp.color.blue, 1.0 }; // цвет света

GLfloat light1\_position[] = { temp.position.x, temp.position.y, temp.position.z, 1.0 }; // местоположение света

glEnable(GL\_LIGHT2);

glLightfv(GL\_LIGHT2, GL\_DIFFUSE, light1\_Diffuse);

glLightfv(GL\_LIGHT2, GL\_POSITION, light1\_position);

glLightf(GL\_LIGHT2, GL\_CONSTANT\_ATTENUATION, 0);

glLightf(GL\_LIGHT2, GL\_LINEAR\_ATTENUATION, 0.04);

glLightf(GL\_LIGHT2, GL\_QUADRATIC\_ATTENUATION, 0.04);

}

}

void Display(void)

{

//очистка буфера цвета и глубины

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

//включить буфер глубины

glEnable(GL\_DEPTH\_TEST);

if (g\_bLight)

glEnable(GL\_LIGHTING); //включить свет

else

glDisable(GL\_LIGHTING); //выключить свет

//обнуление трансформации

glLoadIdentity();

//установить вид камеры

gluLookAt(camera1.Position.x, camera1.Position.y, camera1.Position.z,

camera1.View.x, camera1.View.y, camera1.View.z,

camera1.UpVector.x, camera1.UpVector.y, camera1.UpVector.z);

// настроить освещение

tune\_lighting();

//отрисовать сцену

scene1.Draw();

//есди включен режим трассировки

if(raytracer\_mode)

{

//установить параметры трассировщика

raytracer1 = Raytracer(camera1.Position, scene1, camera1, ratio, width, height, block\_size);

if (changes) {

cout << "\nВычисляю лучи";

PIX.clear();

raytracer1.Raytrace();//запустить трассировку

cout << "\nВычислил!\n";

changes = false;

}

raytracer1.draw();

}

glDisable(GL\_LIGHT0);

glDisable(GL\_LIGHT1);

glDisable(GL\_LIGHT2);

glDisable(GL\_LIGHT3);

glDisable(GL\_LIGHT4);

glutSwapBuffers();

glutPostRedisplay();

}

void Process\_Mouse\_Move(int x, int y)

{

if (rot)

{

changes = true;

camera1.SetViewByMouse(width, height);

}

}

void Process\_Normal\_Keys(unsigned char key, int x, int y)

{

changes = true;

//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);

}

if (key == '=')

block\_size++;

if (key == '-')

if(block\_size > 1)

block\_size--;

//включение/выключение режима включения/исключения фигур

if(key == 'e' || key == 'E')

{

//переключить режим

scene1.add\_del = !scene1.add\_del;

//если режим включен

if(scene1.add\_del)

{

//задать начальные установки

scene1.sphere\_mod = true;

scene1.tetrahedron\_mod = false;

scene1.active\_Sphere = 0;

scene1.active\_Tetrahedron = 0;

}

//иначе

else

{

scene1. sphere\_mod = false;

scene1.tetrahedron\_mod = false;

}

}

//включить исключить фигуру

if(key == 'r' || key == 'R')

{

//если включен режив включения/исключения фигур

if(scene1.add\_del)

{

if(scene1.sphere\_mod)

scene1.vector\_Sphere[scene1.active\_Sphere].display = !scene1.vector\_Sphere[scene1.active\_Sphere].display;

if(scene1.tetrahedron\_mod)

scene1.vector\_Tetrahedron[scene1.active\_Tetrahedron].display = !scene1.vector\_Tetrahedron[scene1.active\_Tetrahedron].display;

}

}

if (key == 'z' || key == 'Z')

{

en\_material = !en\_material;

}

//движение камеры

if(key == 'w' || key == 'W')

{

camera1.MoveCamera(kSpeed);

}

if (key == 's' || key == 'S')

{

camera1.MoveCamera(-kSpeed);

}

if (key == 'a' || key == 'A')

{

camera1.RotateAroundPoint(camera1.View, -kSpeed\*2.0f, 0.0f, 1.0f, 0.0f);

}

if (key == 'd' || key == 'D')

{

camera1.RotateAroundPoint(camera1.View, kSpeed\*2.0f, 0.0f, 1.0f, 0.0f);

}

// выбор источника света

if (key == '1')

{

scene1.lighting\_num = 1;

scene1.set\_l();

}

if (key == '2')

{

scene1.lighting\_num = 2;

scene1.set\_l();

}

if (key == '3')

{

scene1.lighting\_num = 3;

scene1.set\_l();

}

if (key == '4')

{

scene1.lighting\_num = 4;

scene1.set\_l();

}

//перемещение источника света

if(key == 'i')//поднять

{

scene1.vector\_Light[0].position.y++;

}

if(key == 'p')//опустить

{

scene1.vector\_Light[0].position.y--;

}

if(key == 'o')//вперёд

{

scene1.vector\_Light[0].position.x++;

}

if(key == 'l')//назад

{

scene1.vector\_Light[0].position.x--;

}

if(key == ';')//вправо

{

scene1.vector\_Light[0].position.z++;

}

if(key == 'k')//влево

{

scene1.vector\_Light[0].position.z--;

}

if (key == 'y')//включить/выключить фоновый свет

{

if(scene1.ambi == true)

scene1.ambi = false;

else

scene1.ambi = true;

Reshape(width, height);

}

if (key == 'h')//включить/выключить цвет сферы

{

if (scene1.clsh == true)

scene1.clsh = false;

else

scene1.clsh = true;

Reshape(width, height);

}

if (key == 'n')//включить/выключить цвет тетраэдера

{

if (scene1.cltr == true)

scene1.cltr = false;

else

scene1.cltr = true;

Reshape(width, height);

}

glutPostRedisplay();

}

void Process\_Special\_Keys(int key, int x, int y)

{

//переключать фигуры вперед

if(key == GLUT\_KEY\_UP)

{

if(scene1.add\_del)

scene1.switch\_forward();

}

//переключать фигуры назад

if(key == GLUT\_KEY\_DOWN)

{

if(scene1.add\_del)

scene1.switch\_backward();

}

//сменить активный контейнер фигур

if(key == GLUT\_KEY\_LEFT)

{

if(scene1.add\_del)

if(scene1.sphere\_mod)

{

if(scene1.vector\_Tetrahedron.size()>0)

{

scene1.sphere\_mod = false;

scene1.tetrahedron\_mod = true;

}

}

else

{

if(scene1.vector\_Sphere.size()>0)

{

scene1.sphere\_mod = true;

scene1.tetrahedron\_mod = false;

}

}

}

glutPostRedisplay();

}

void Initialize()

{

ratio = 1. \* width / height;

camera1.PositionCamera(-12.0f, 4.0f, -3.0f, -5.5f, 1.0f, -1.5f, 0.0f, 1.0f, 0.0f); //установить начальную позицию камеры

}

int main(int argc, char \*argv[])

{

width = 1200;

height = 800;

glutInit(&argc, argv);

setlocale(LC\_ALL, "Russian");

glutInitDisplayMode(GLUT\_DEPTH | GLUT\_DOUBLE | GLUT\_RGBA);

glutInitWindowSize(width, height);

glutCreateWindow("Трассировка");

Initialize();

glutKeyboardFunc(Process\_Normal\_Keys);

glutSpecialFunc(Process\_Special\_Keys);

glutDisplayFunc(Display);

glutReshapeFunc(Reshape);

glutPassiveMotionFunc(Process\_Mouse\_Move);

glutMainLoop();

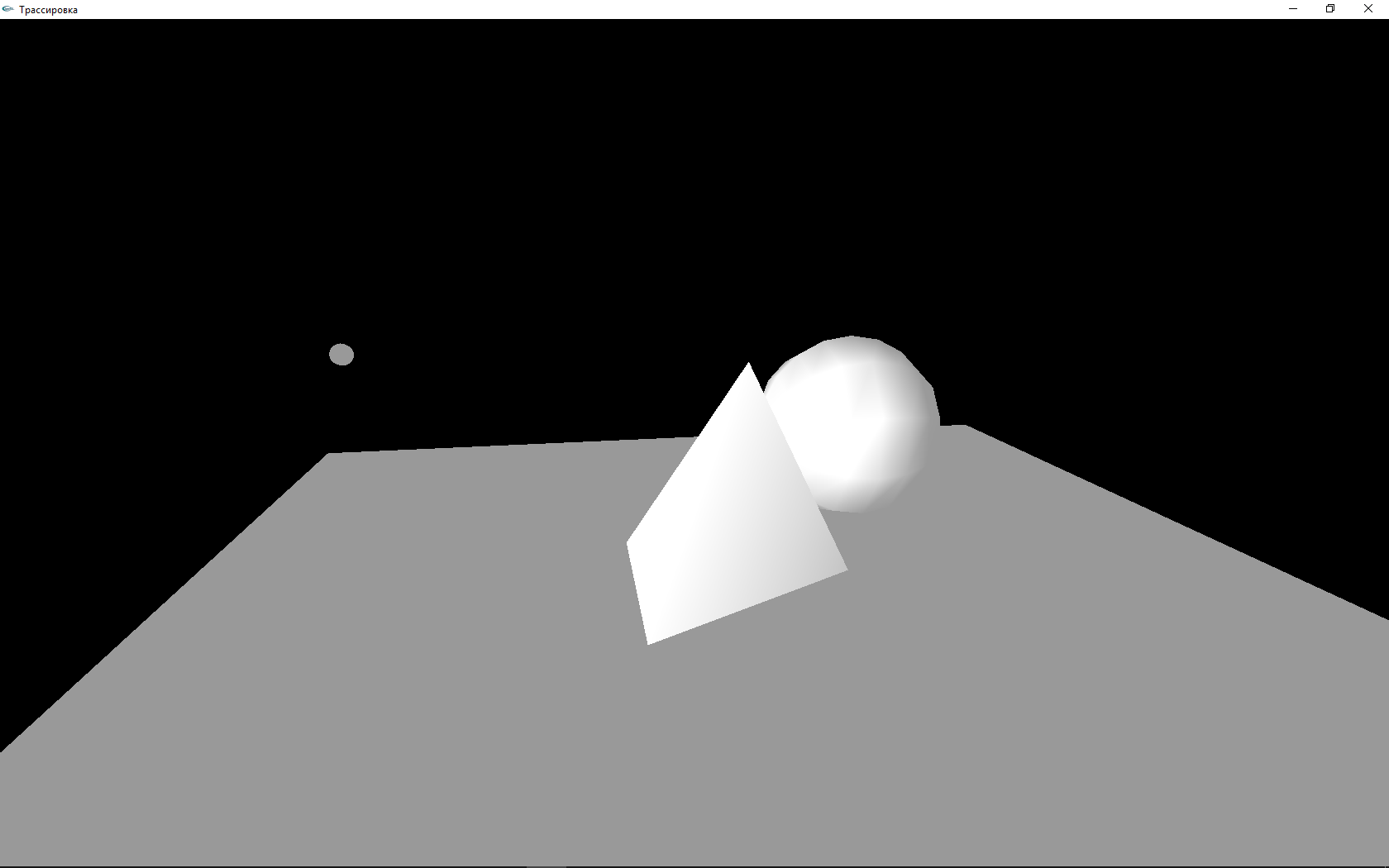
}

1. Тесты

Демострация работы программы вне режима трассировки

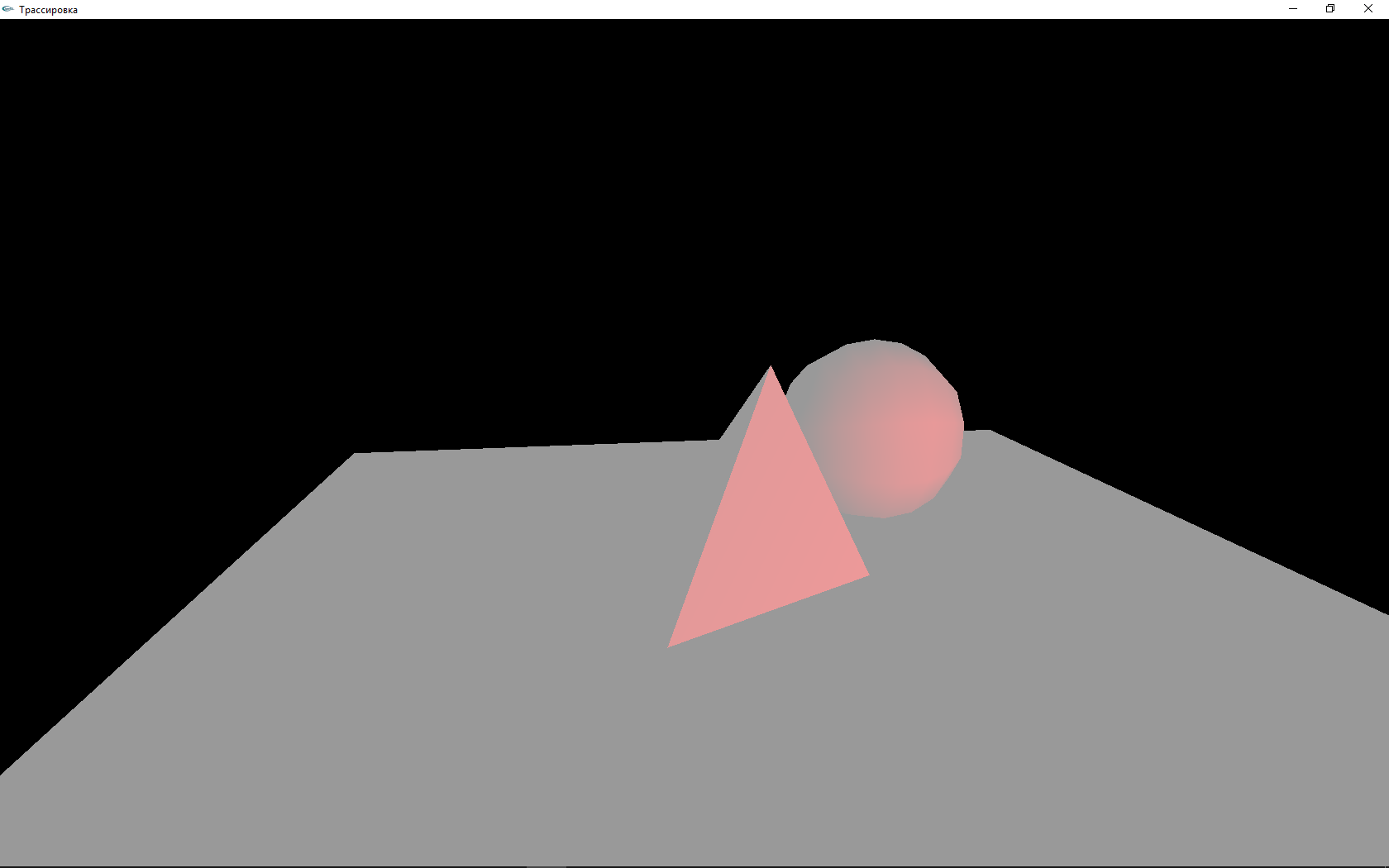
Тест 1.

Малая сфера слева – визуализация источника света.

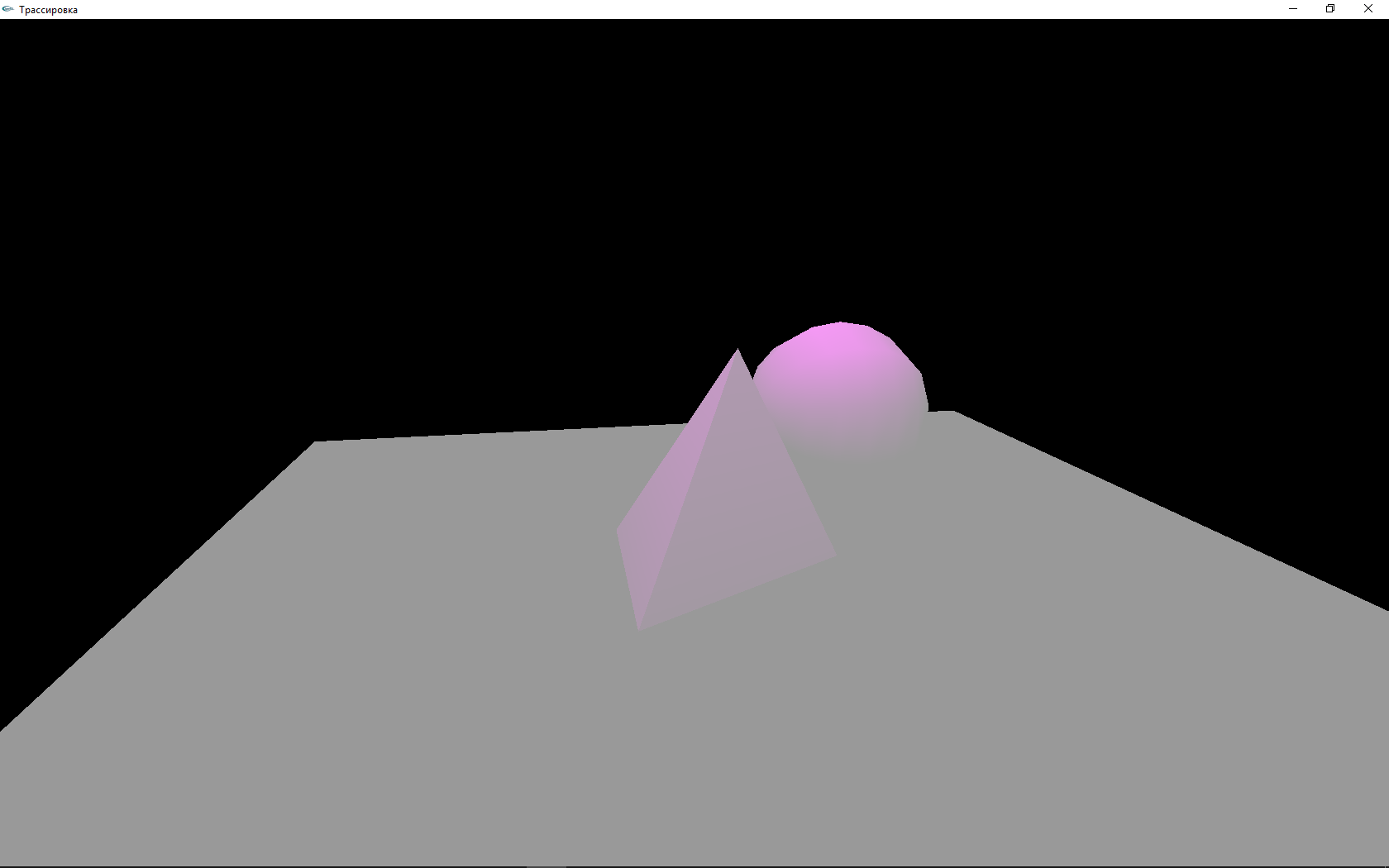


Тест 2. Демострация работы источников света.

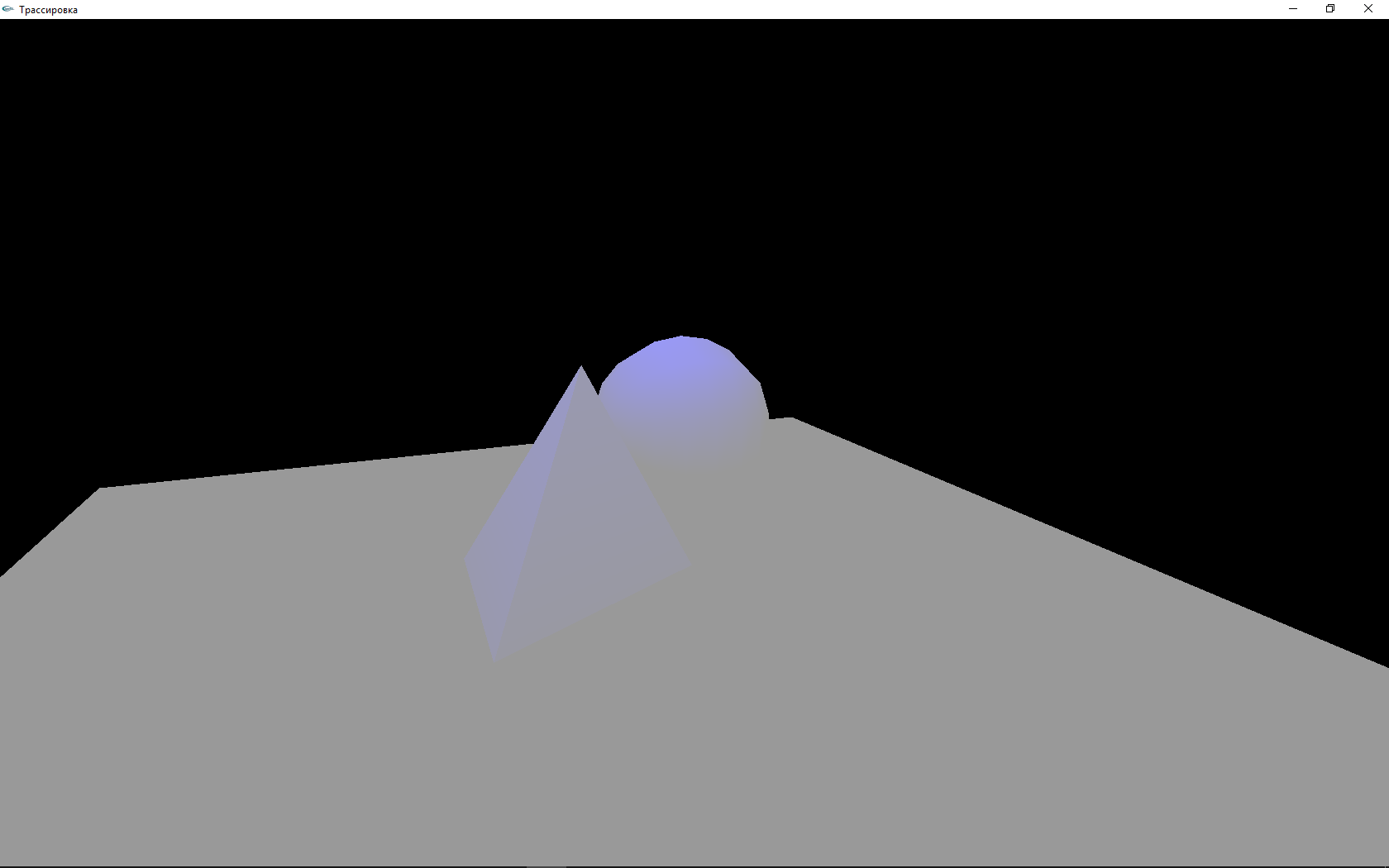
Красный свет.



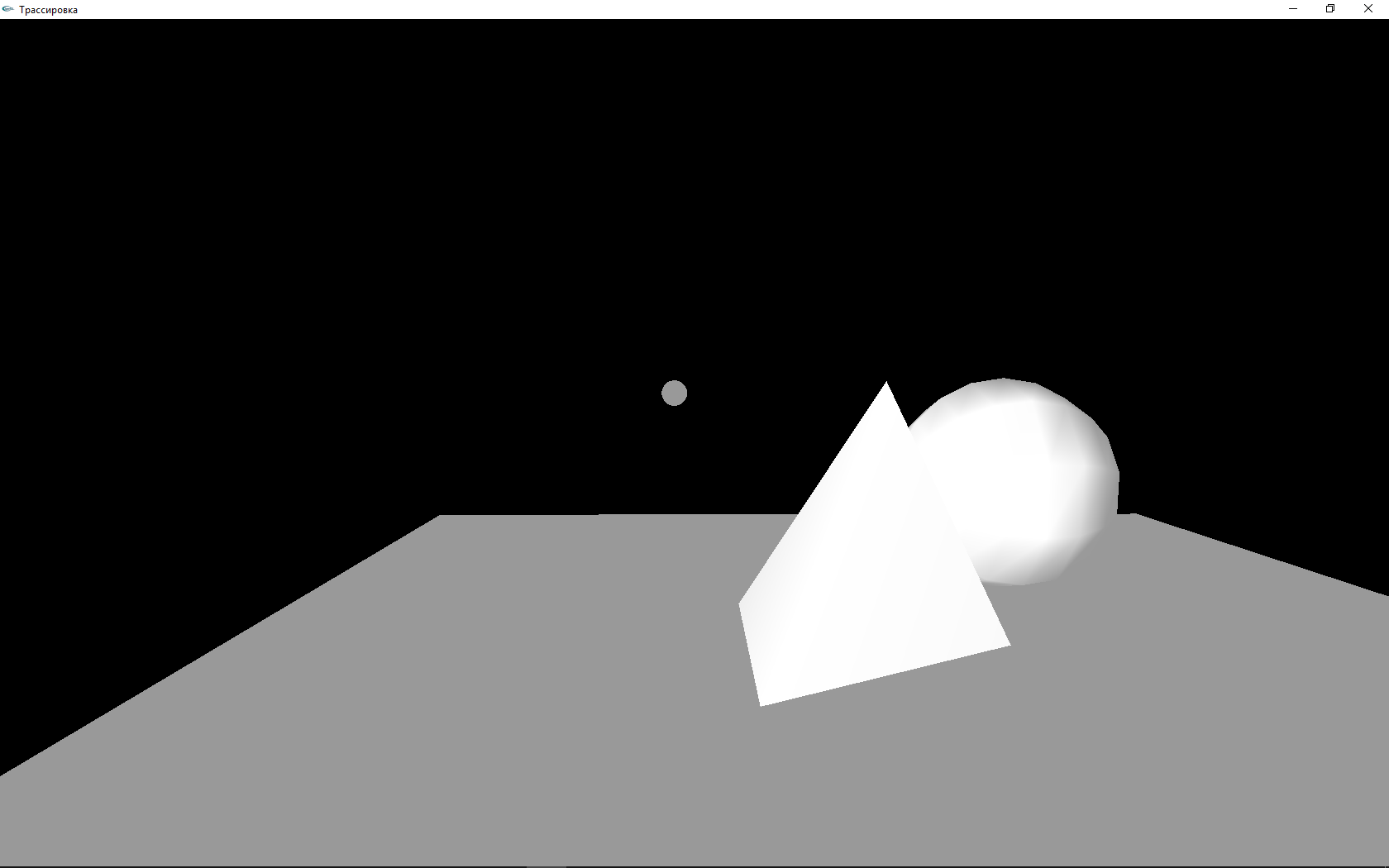
Филетовый свет.

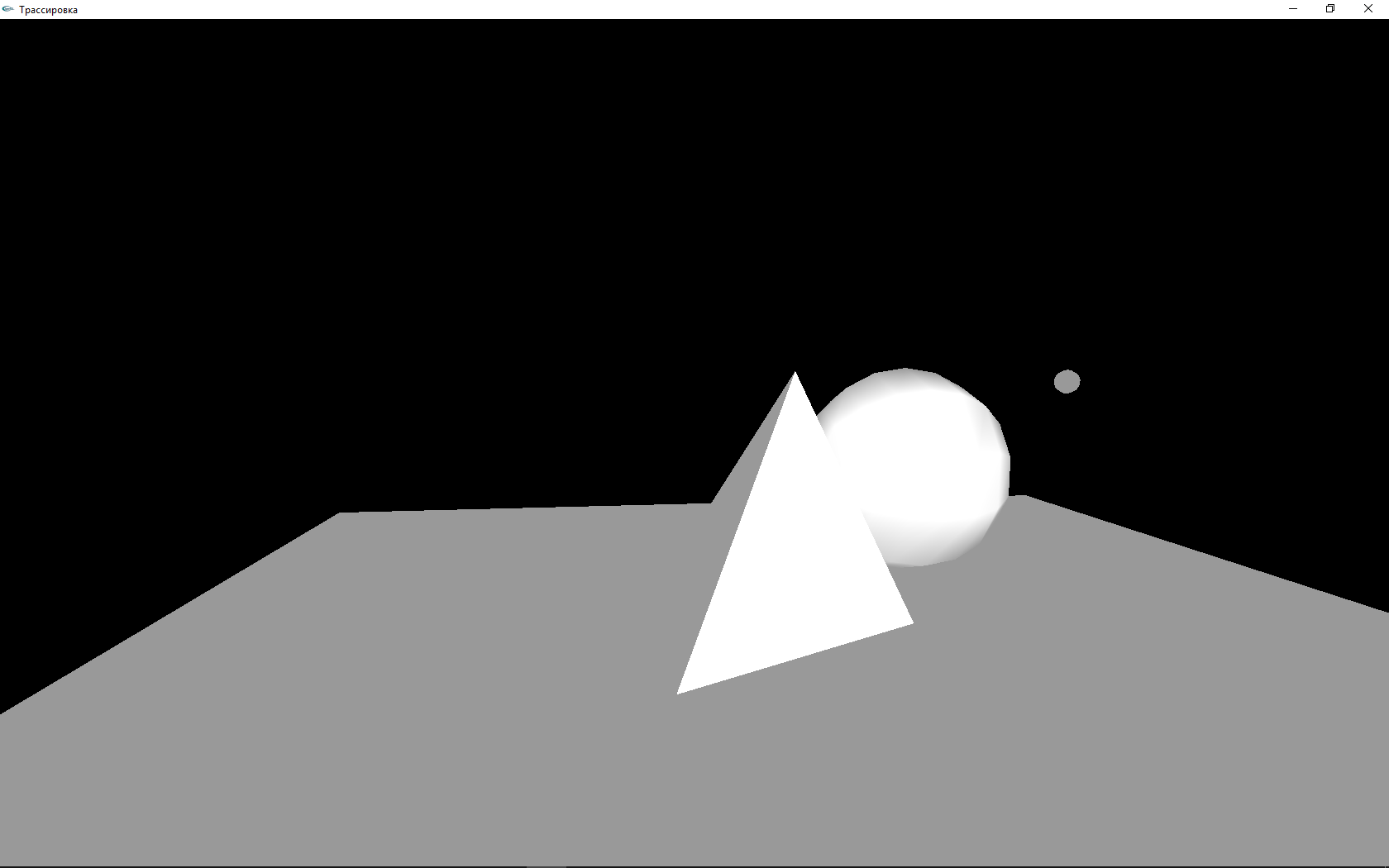


Синий свет.

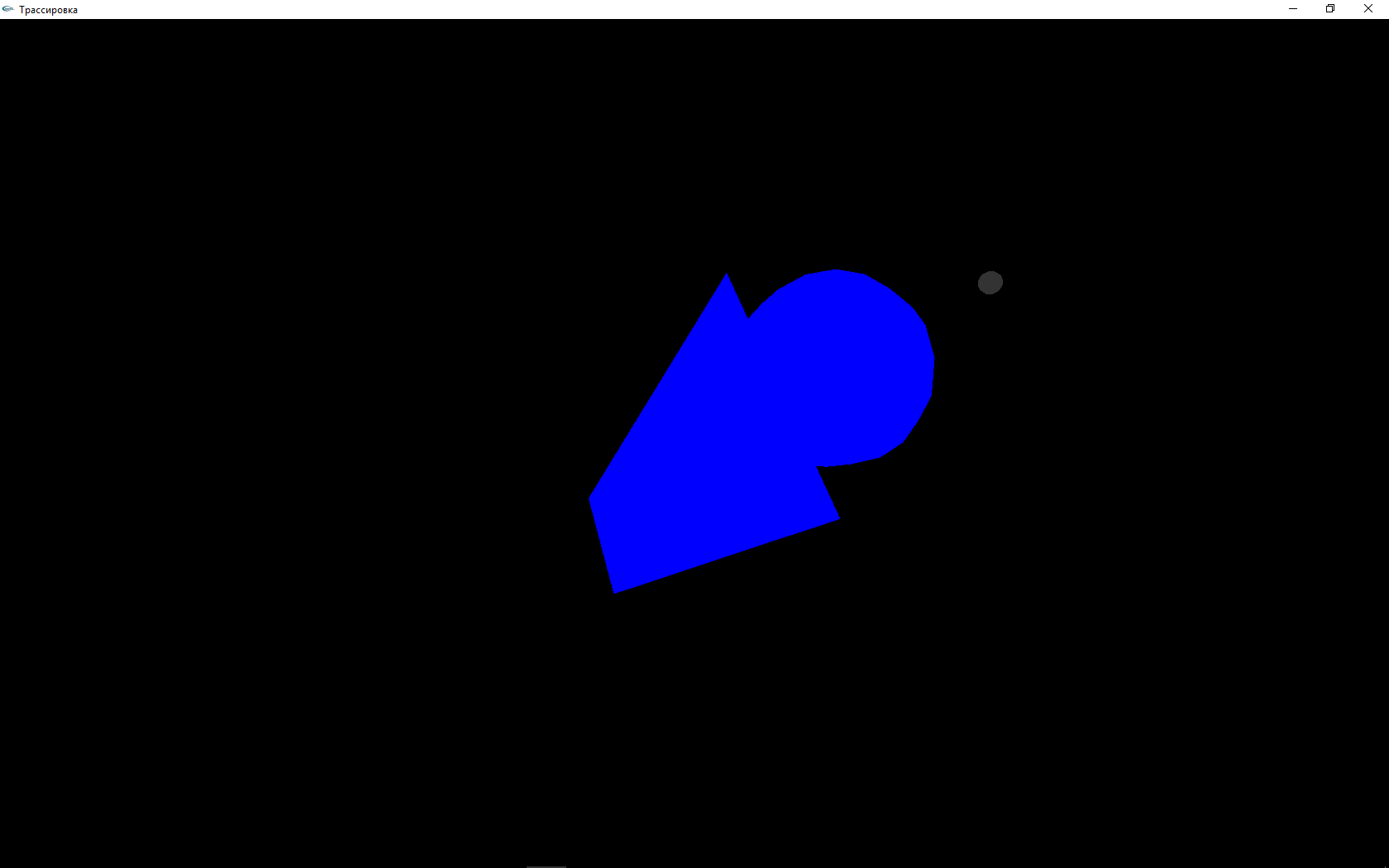


Тест 3. Движение источника света.



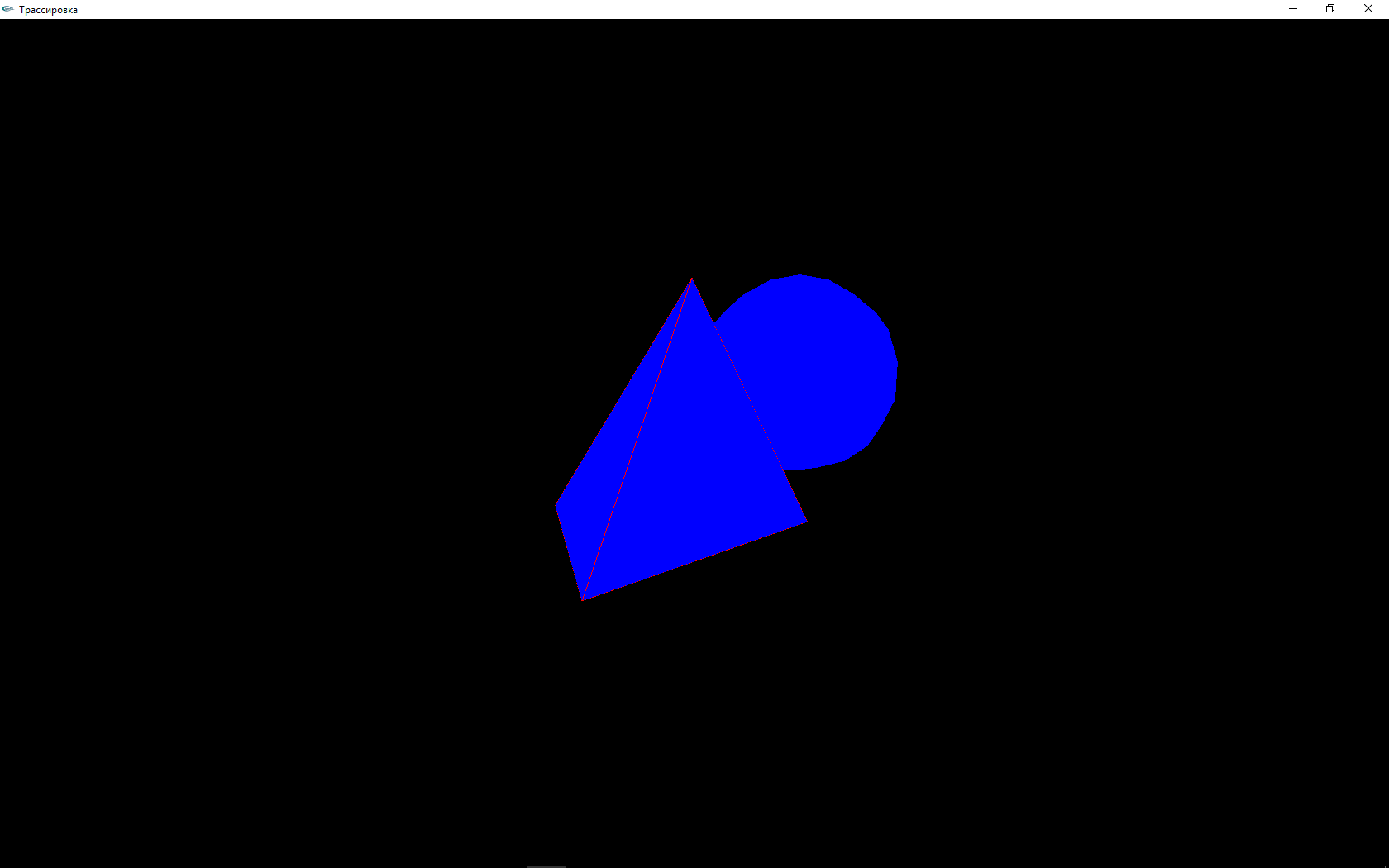


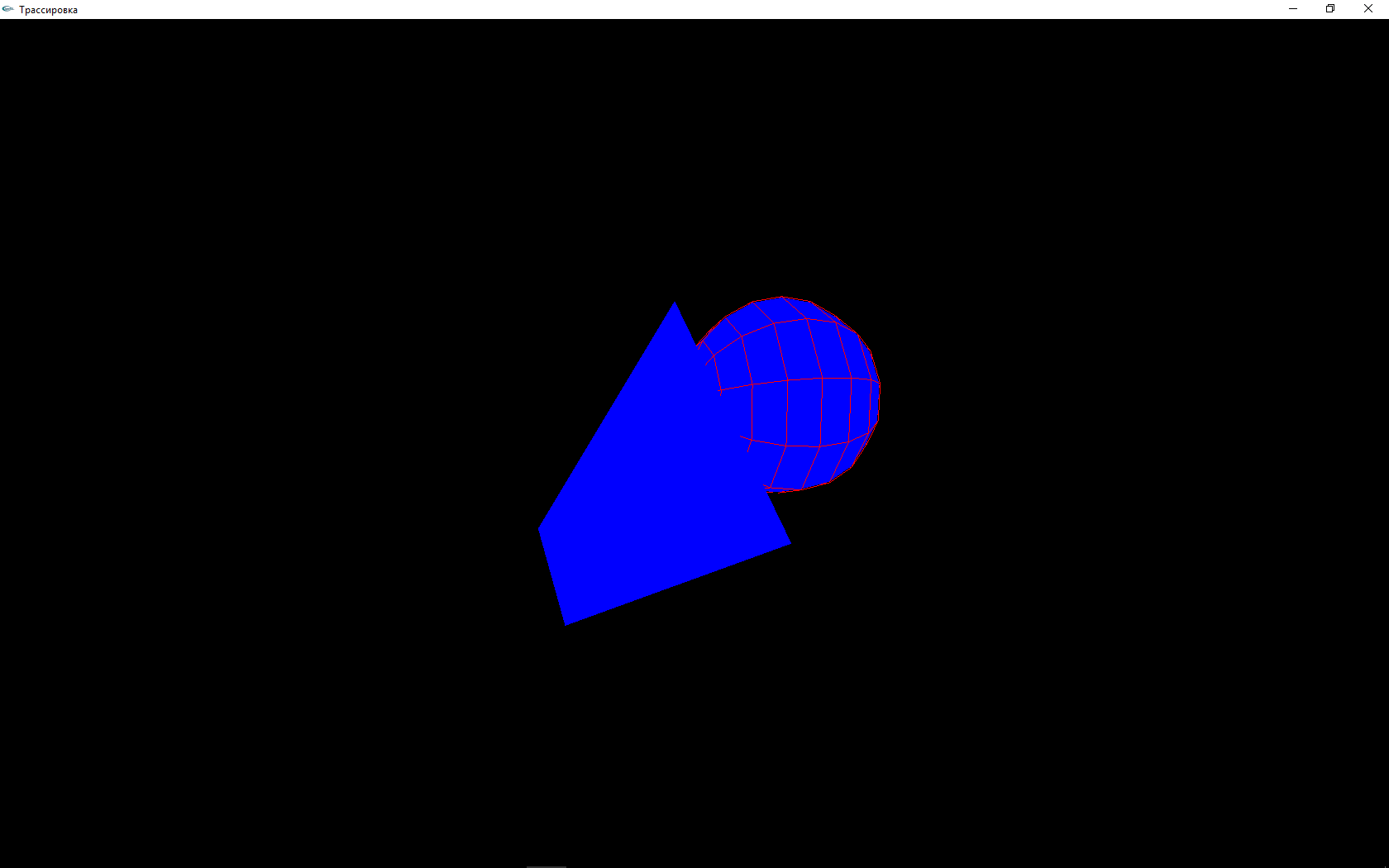
Тест 4. Отключение поверхности.



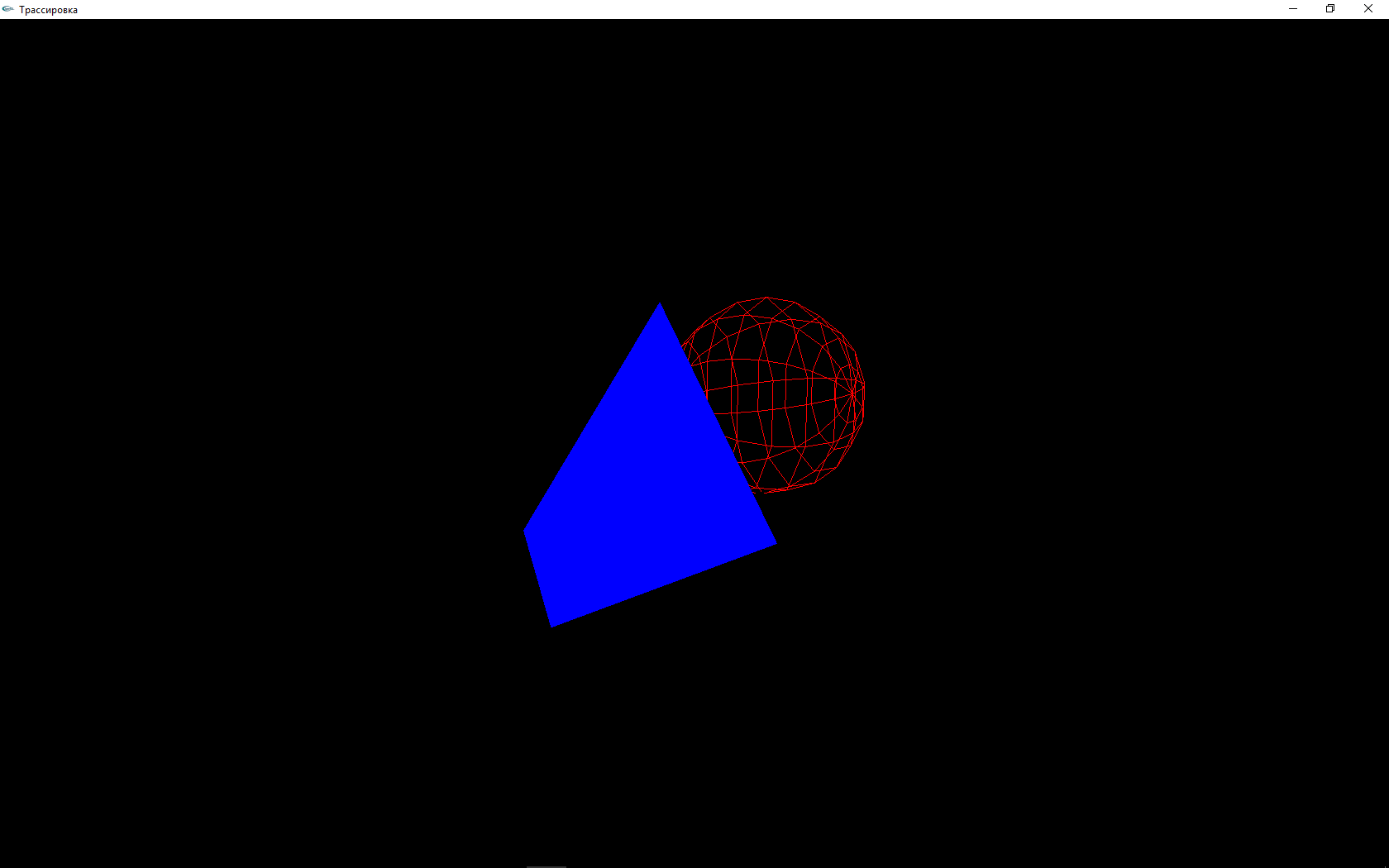
Тест 5. Отключение фигур.

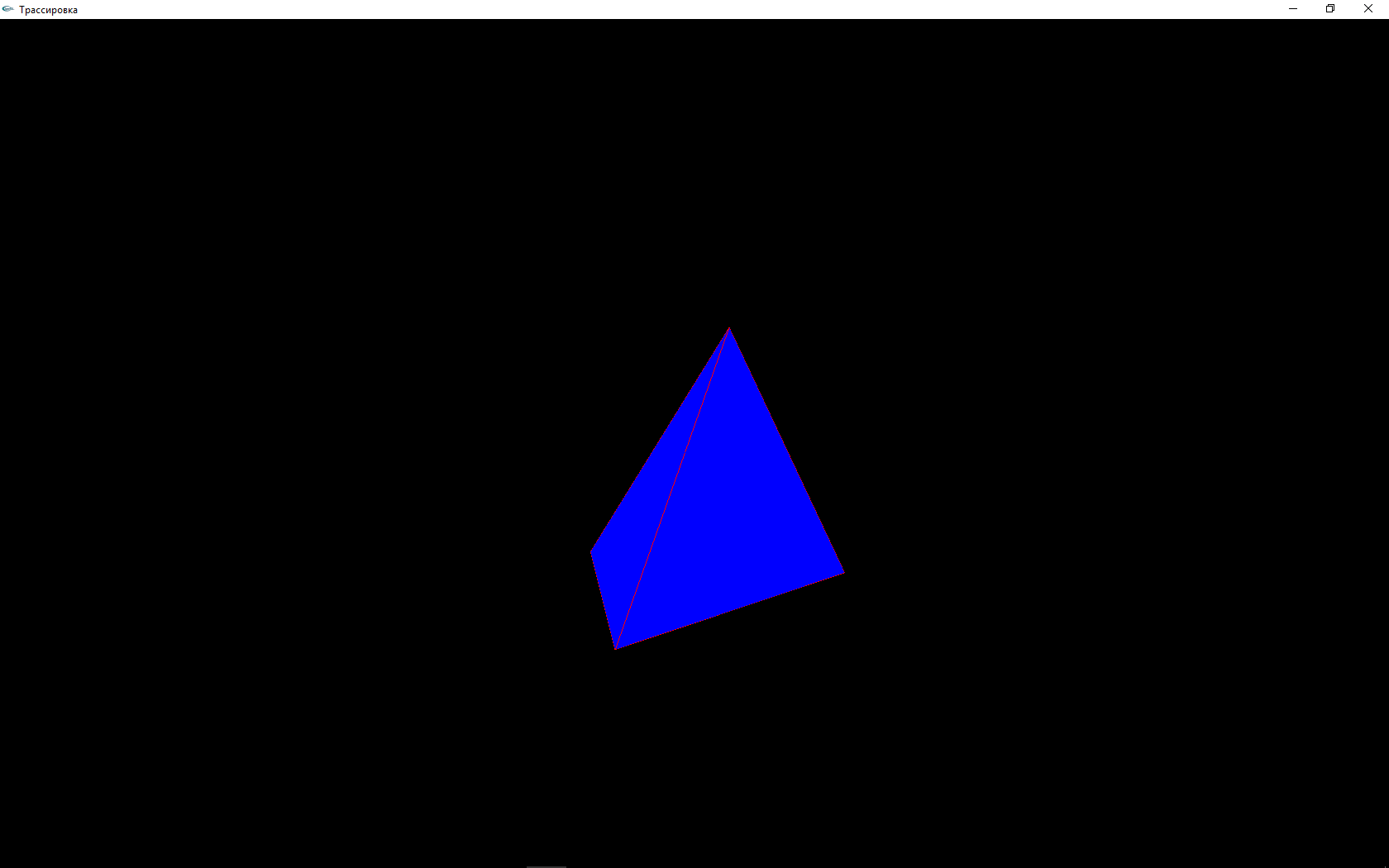
Выбор фигур.

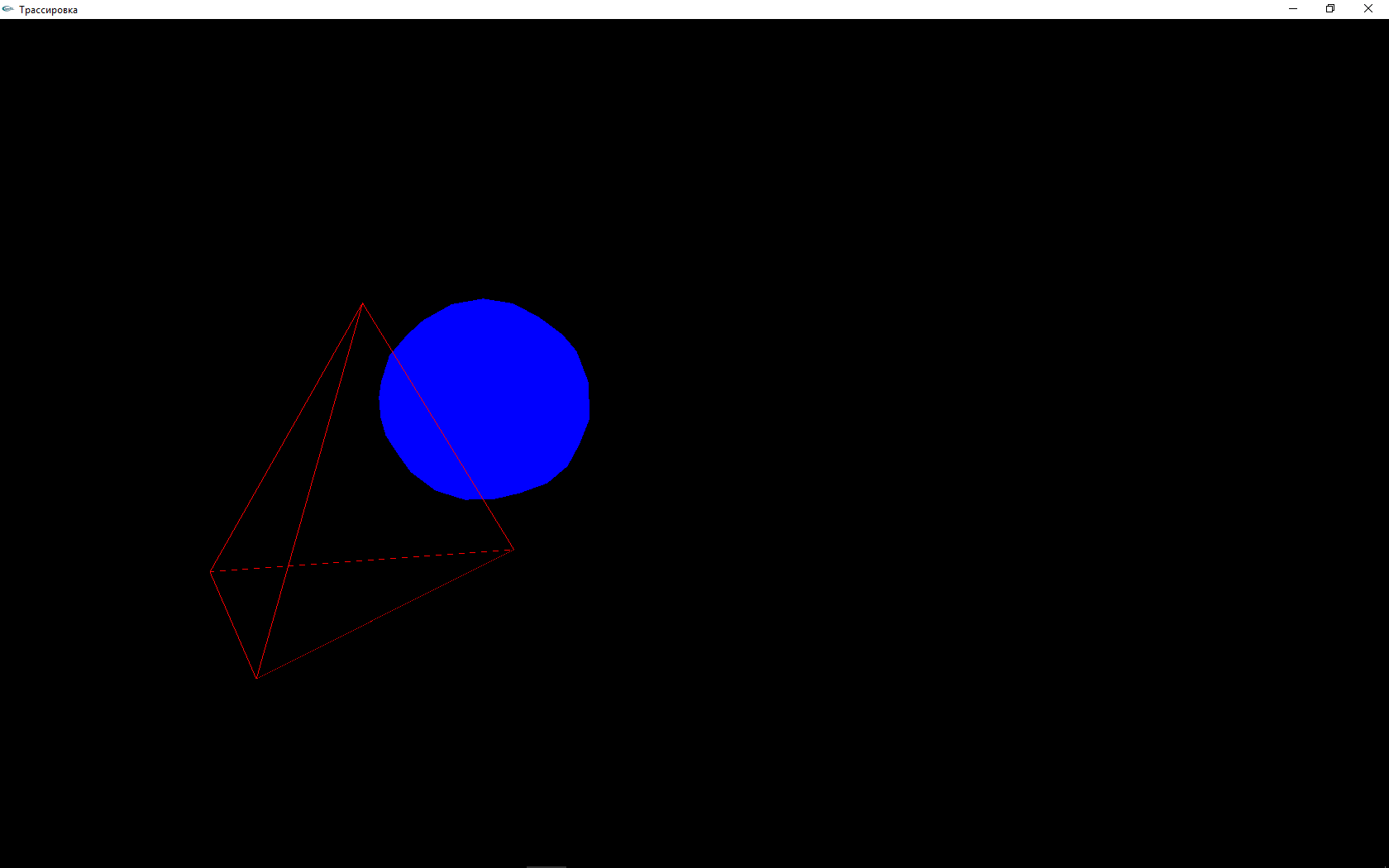


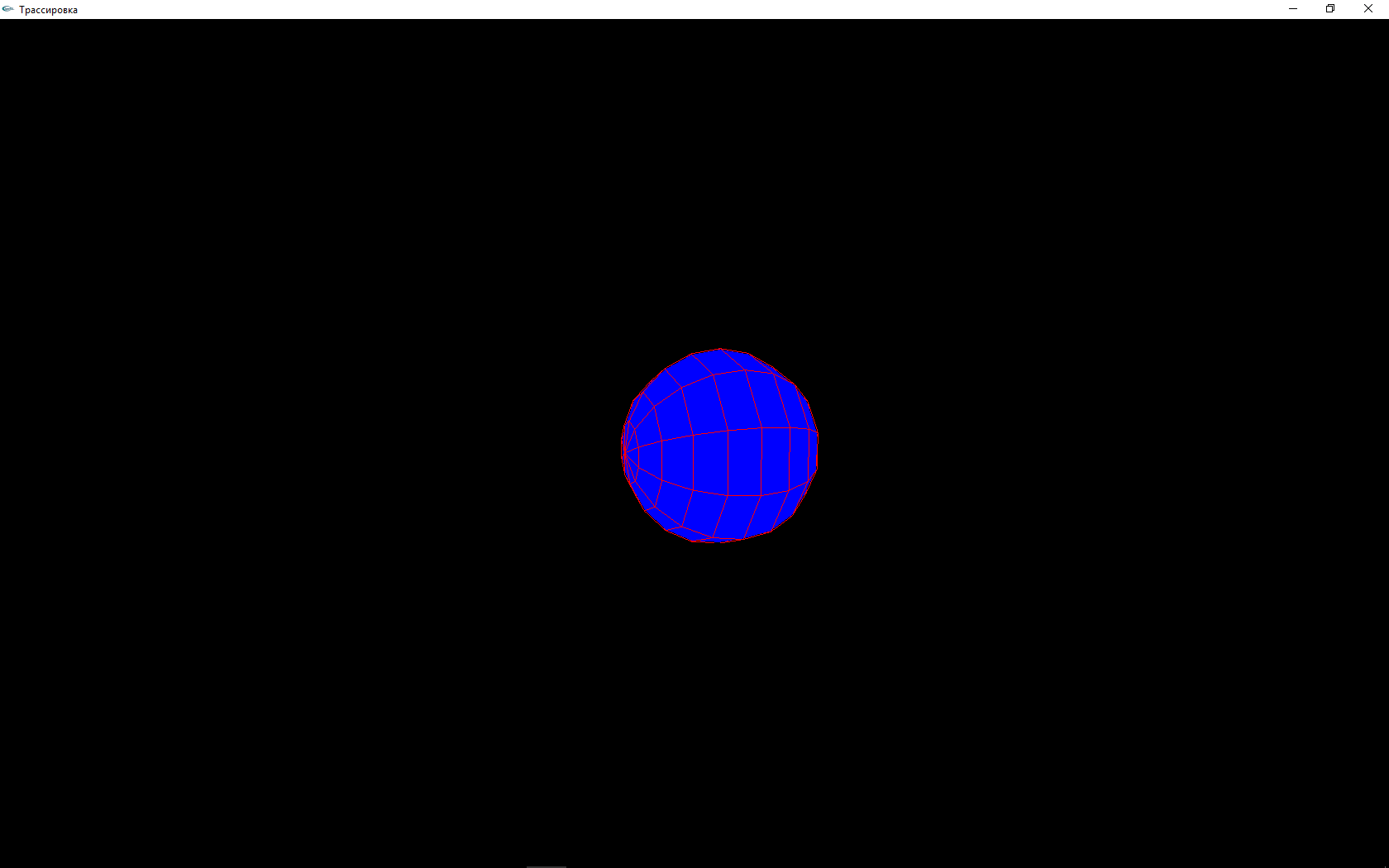


Исключение фигуры.



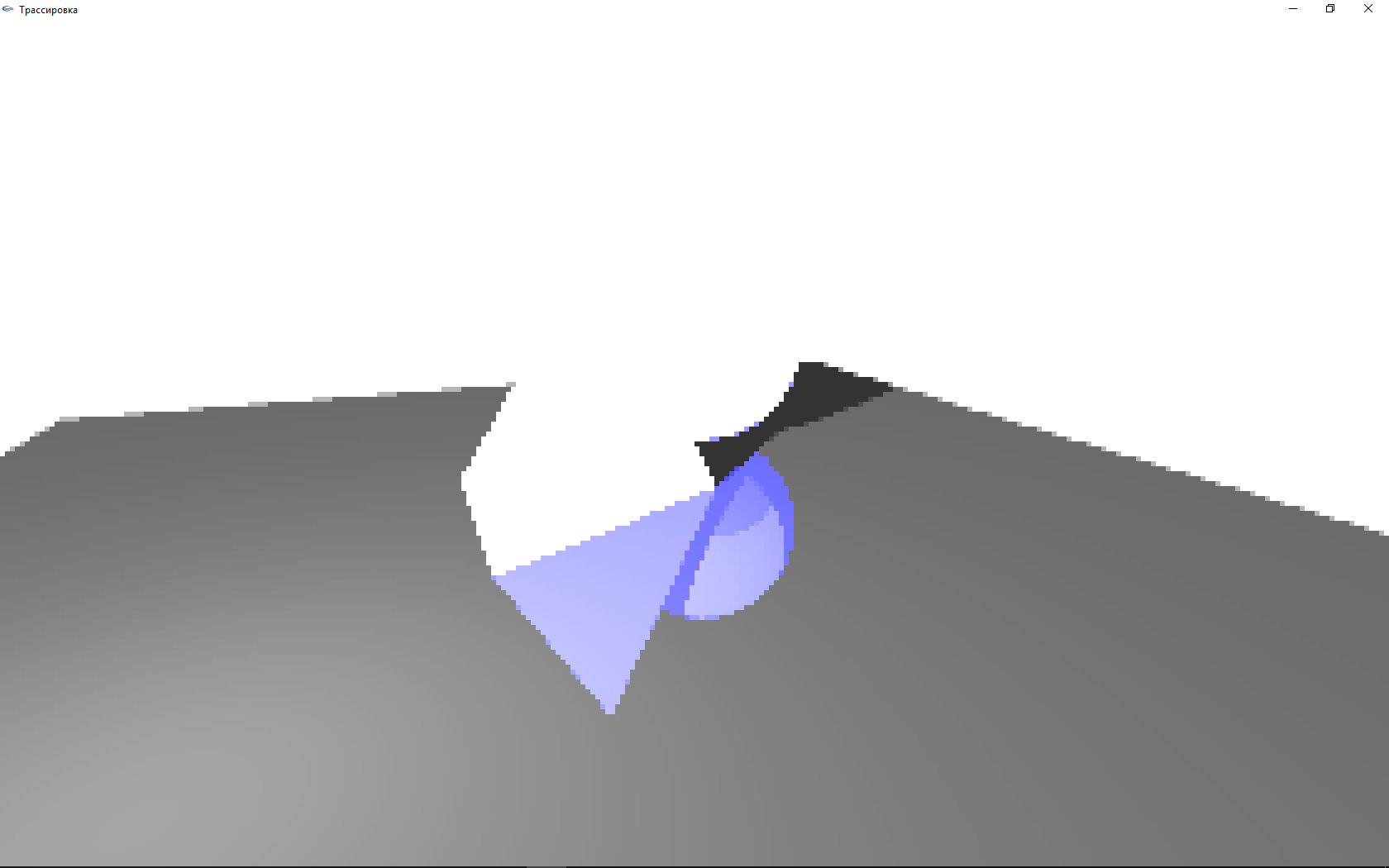




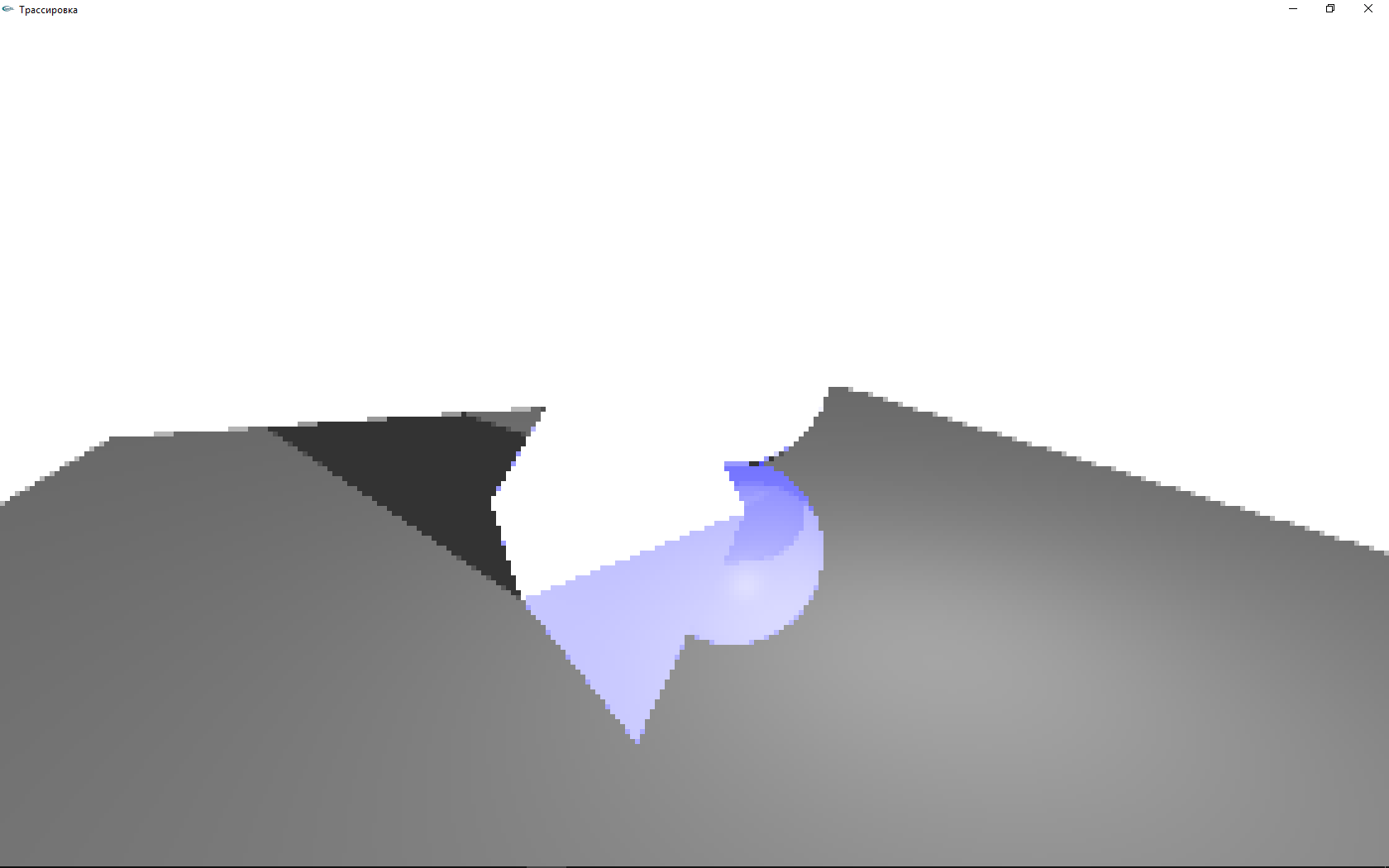


Демонстрация работы программы в режиме трассировки лучей.

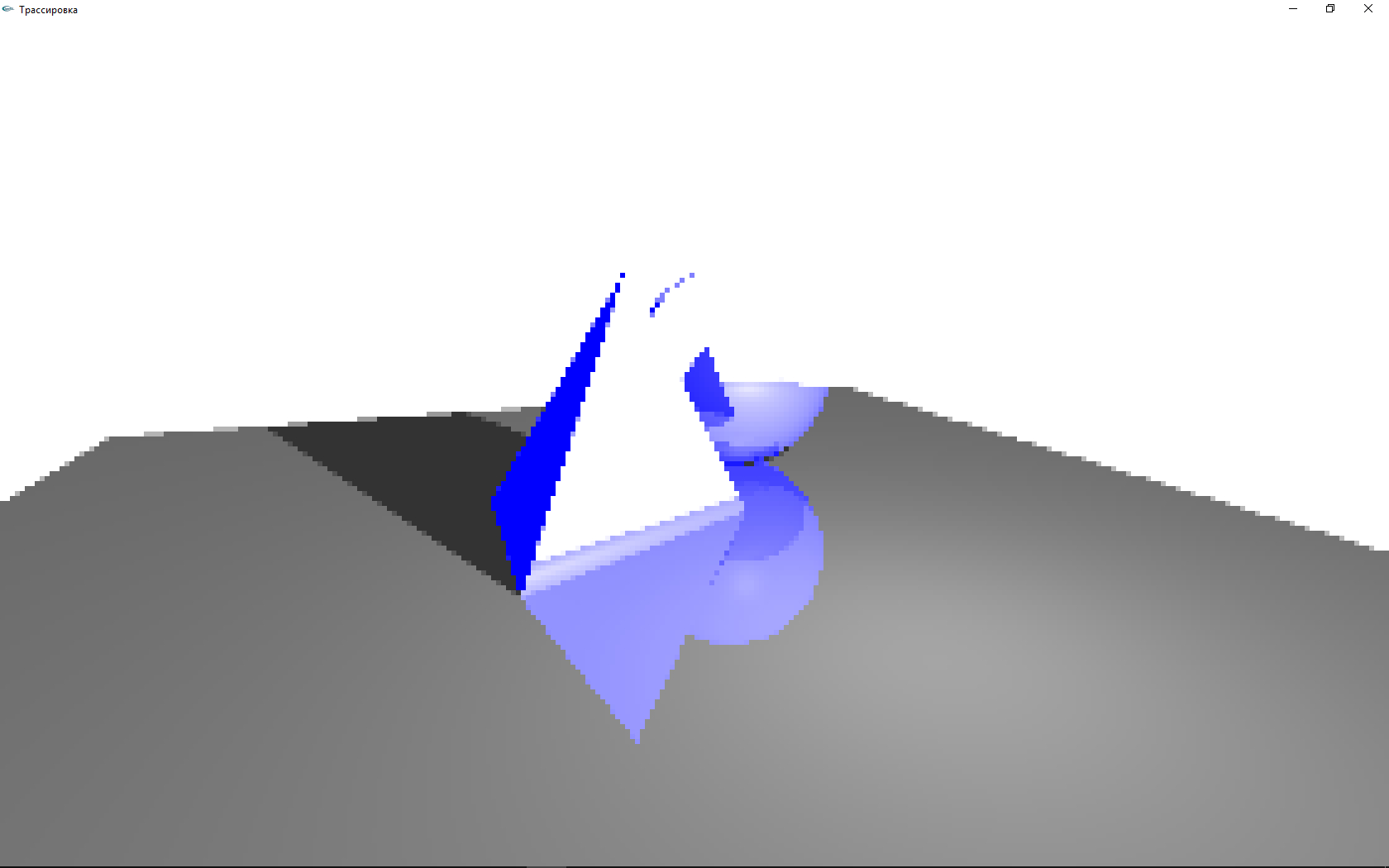
Тест 1. Трассировка лучей с учётом цвета фигур и фонового освещения.



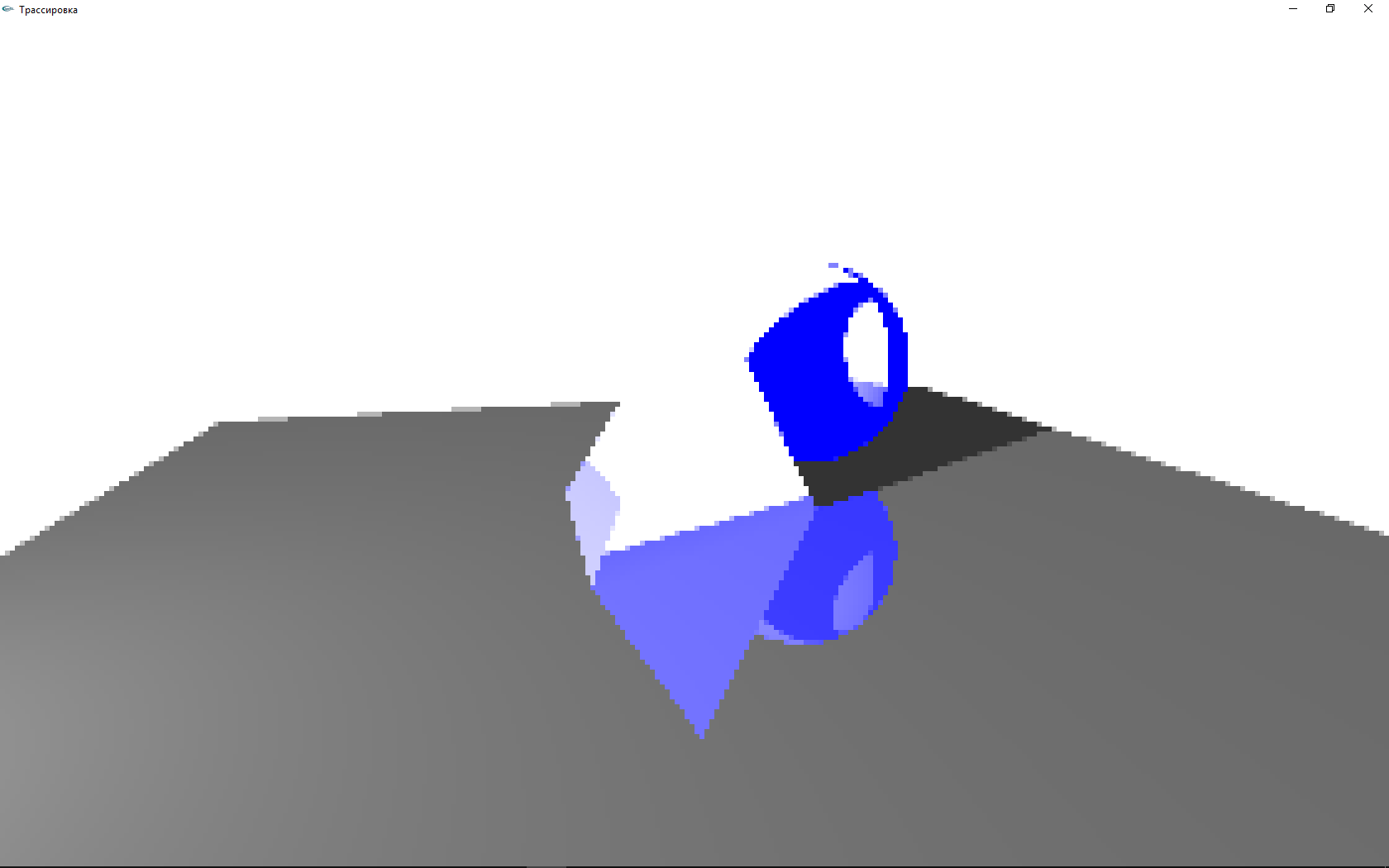
Перемещение источника освещения (тень изменяется, но фоновый источник света не позволяет изменить освещённость фигуры).



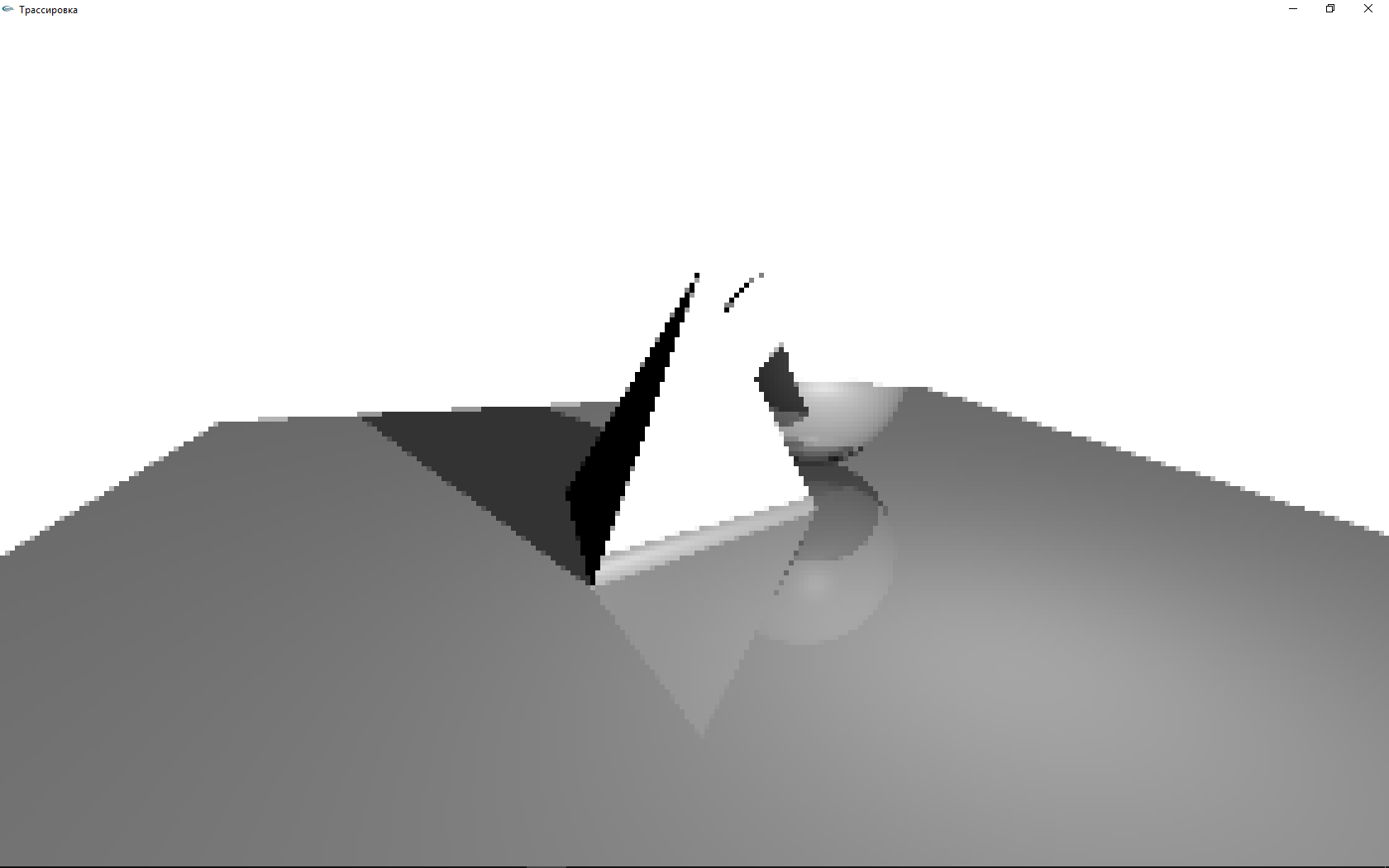
Тест 2. Трассировка лучей без фонового освещения.



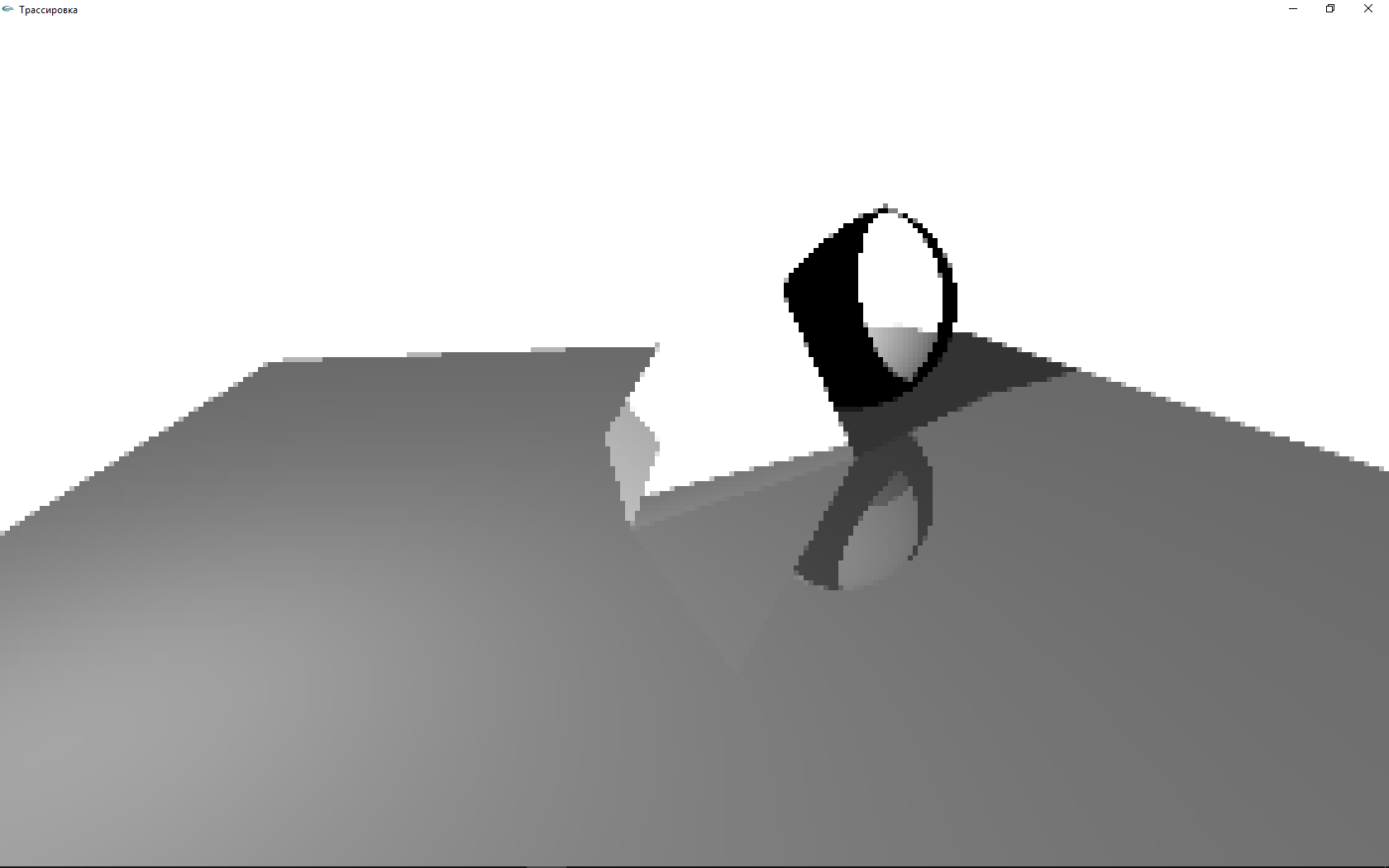
Перемещение источника освещения.



Тест 3. Трассировка лучей без учёта цвета фигур.

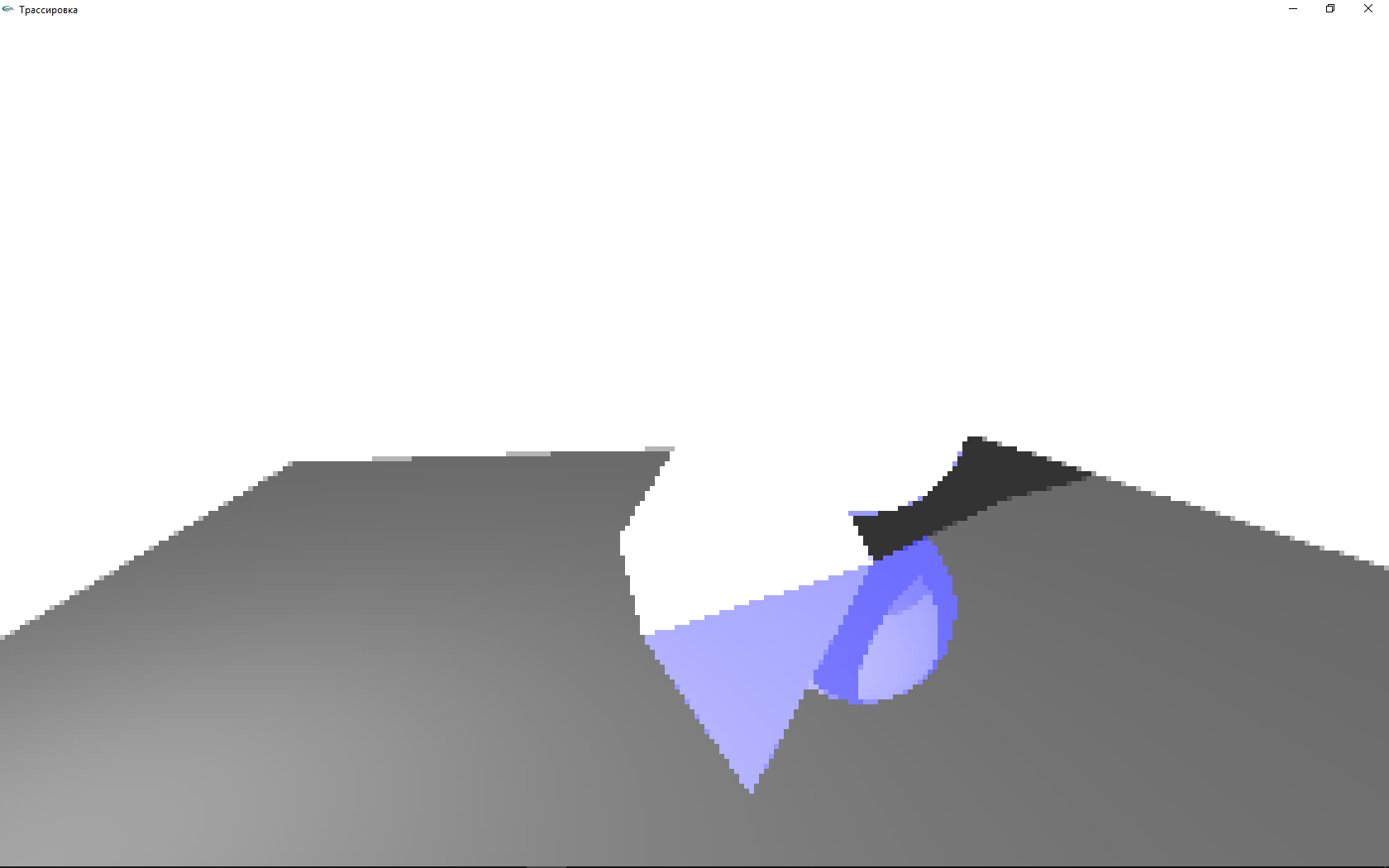


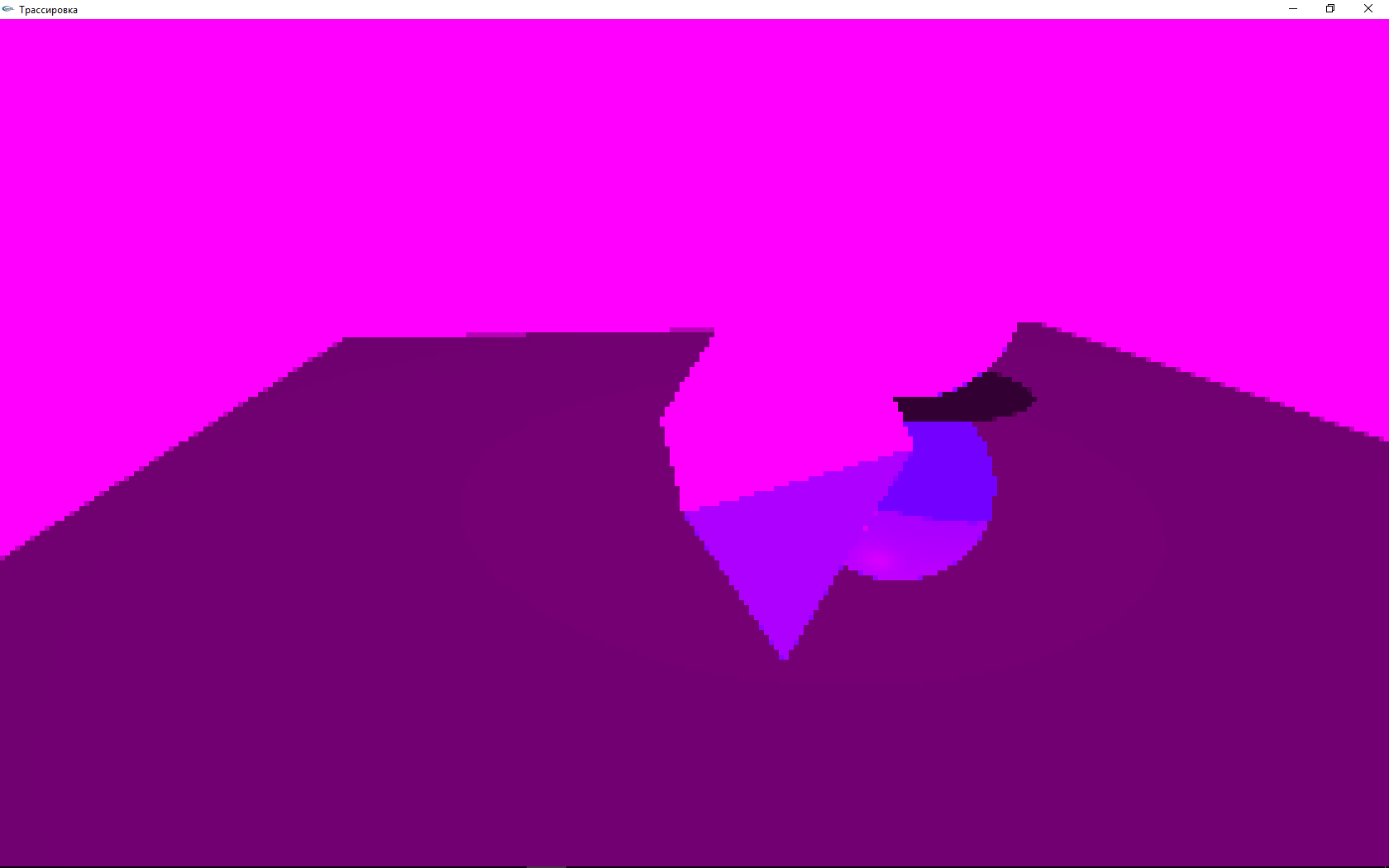
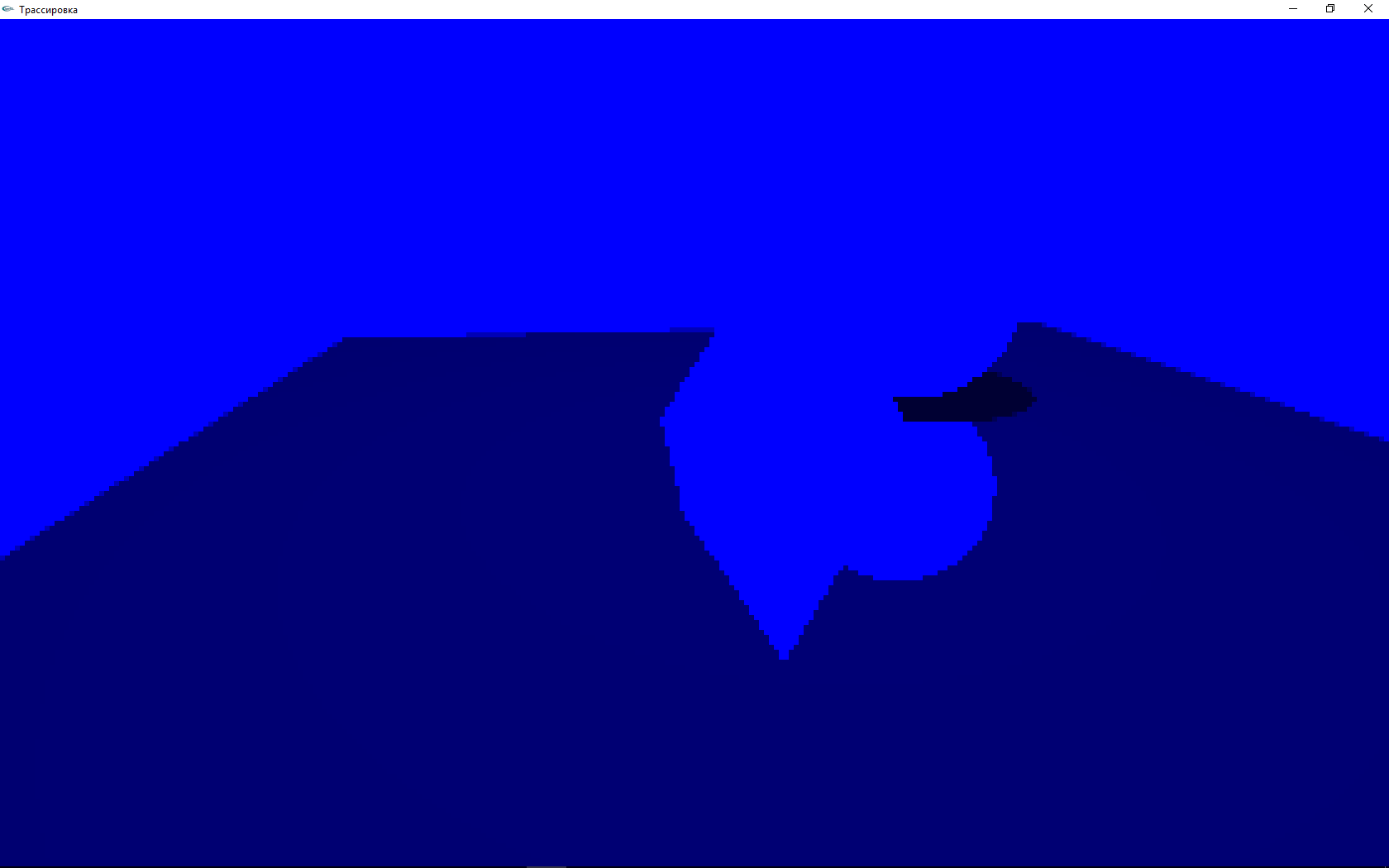
Перемещение источника освещения.



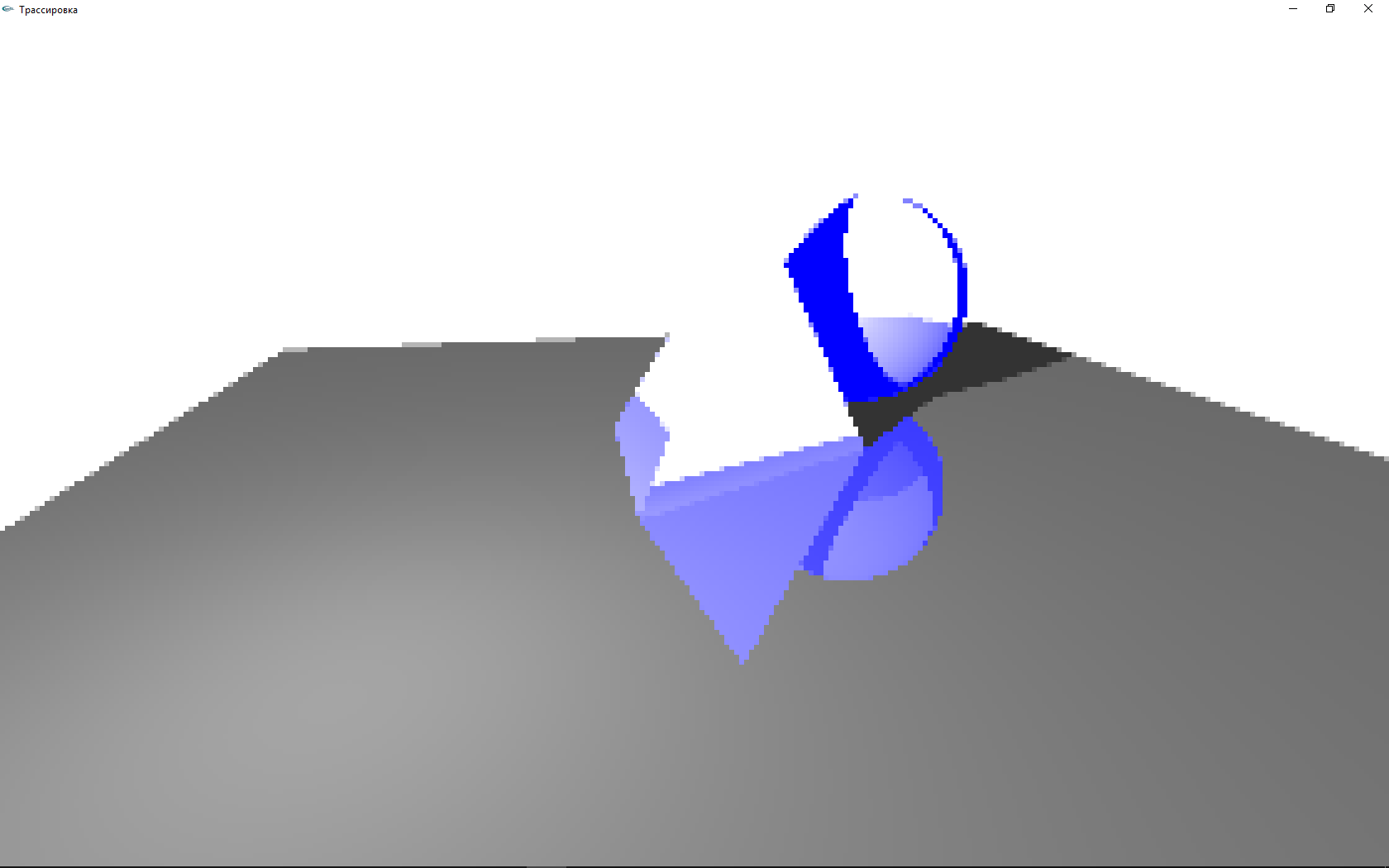
Тест 4. Различный цвета источника освещения.

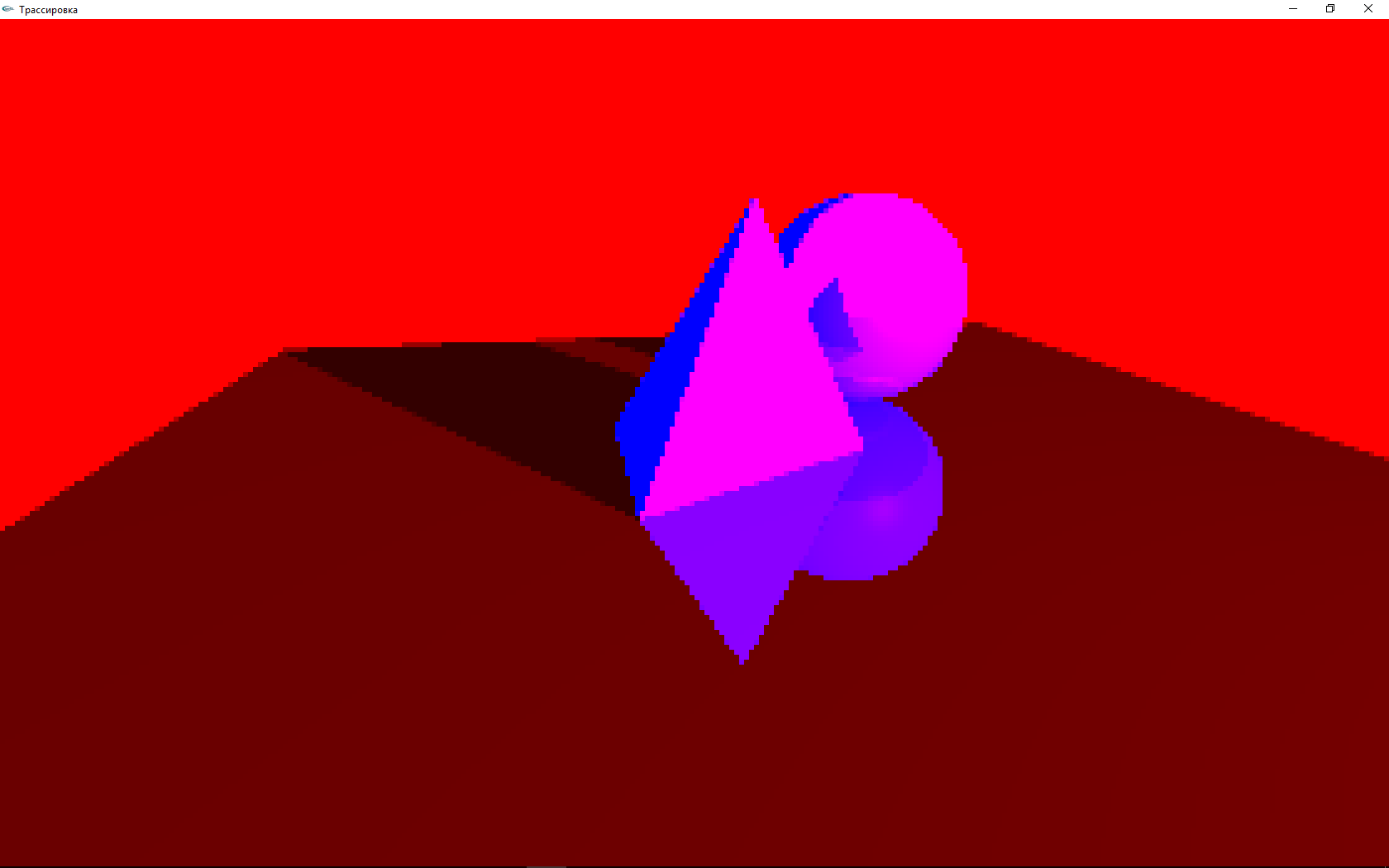
Для фонового освещения и цветов фигур.

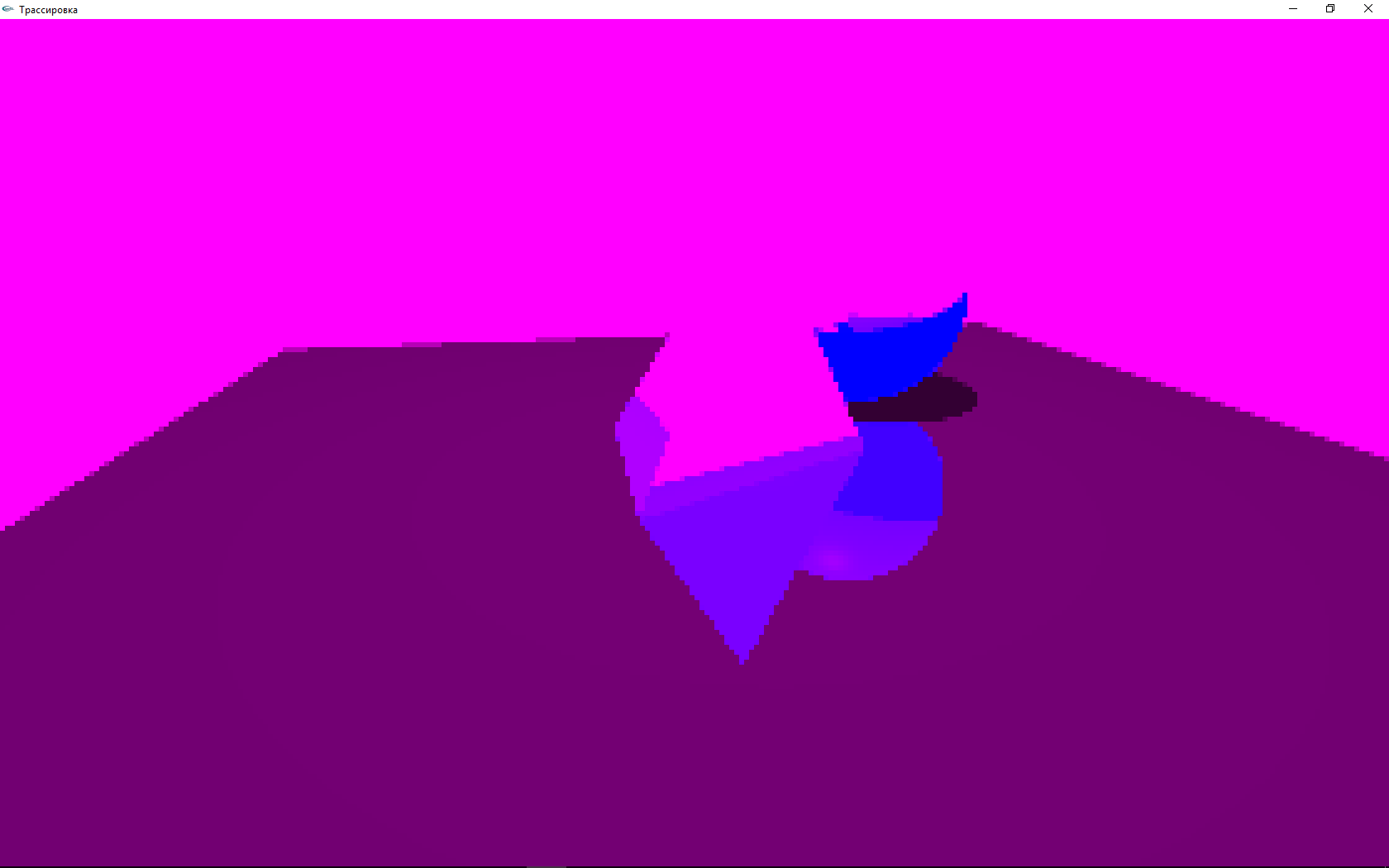


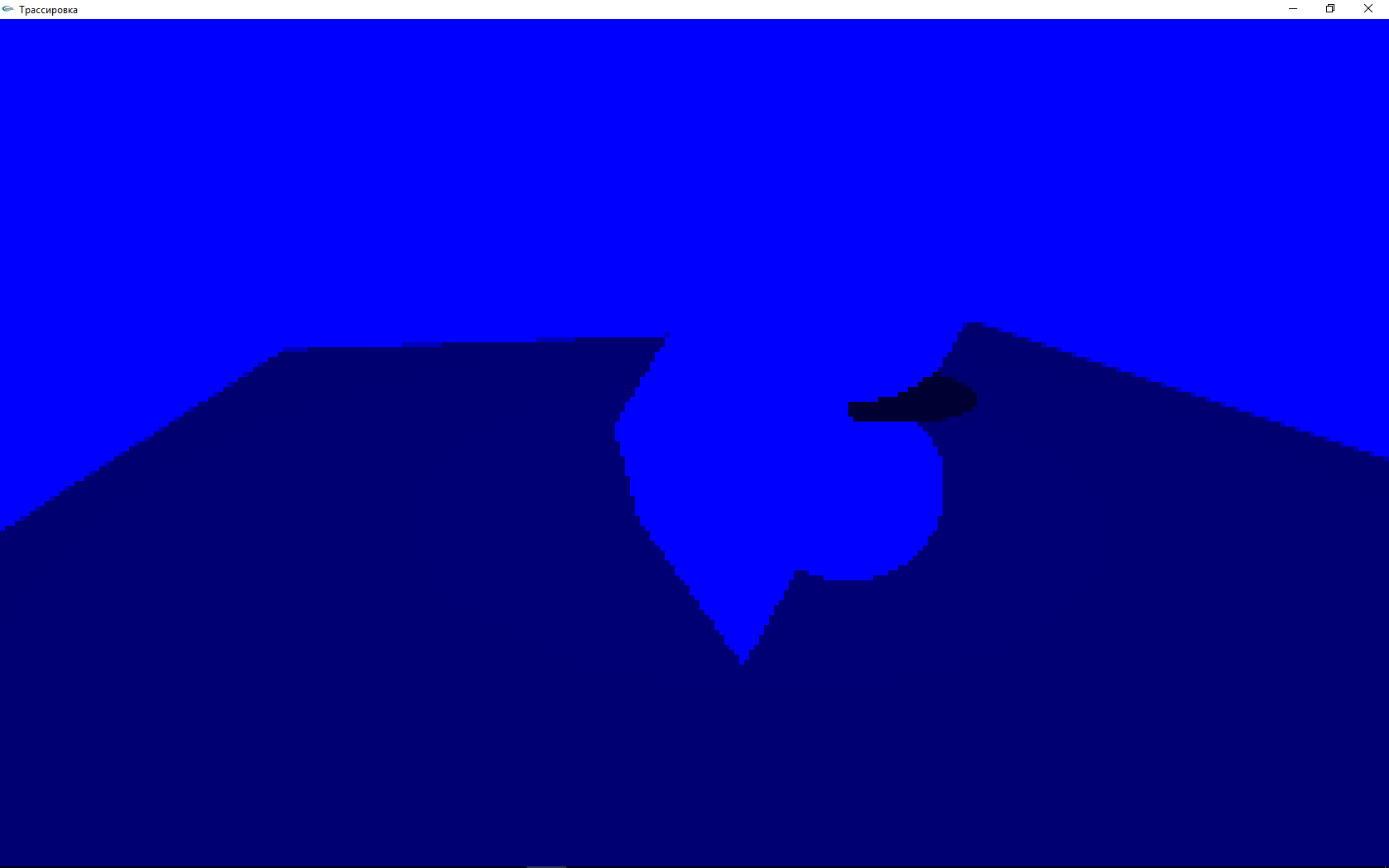
  

Без фонового освещения.

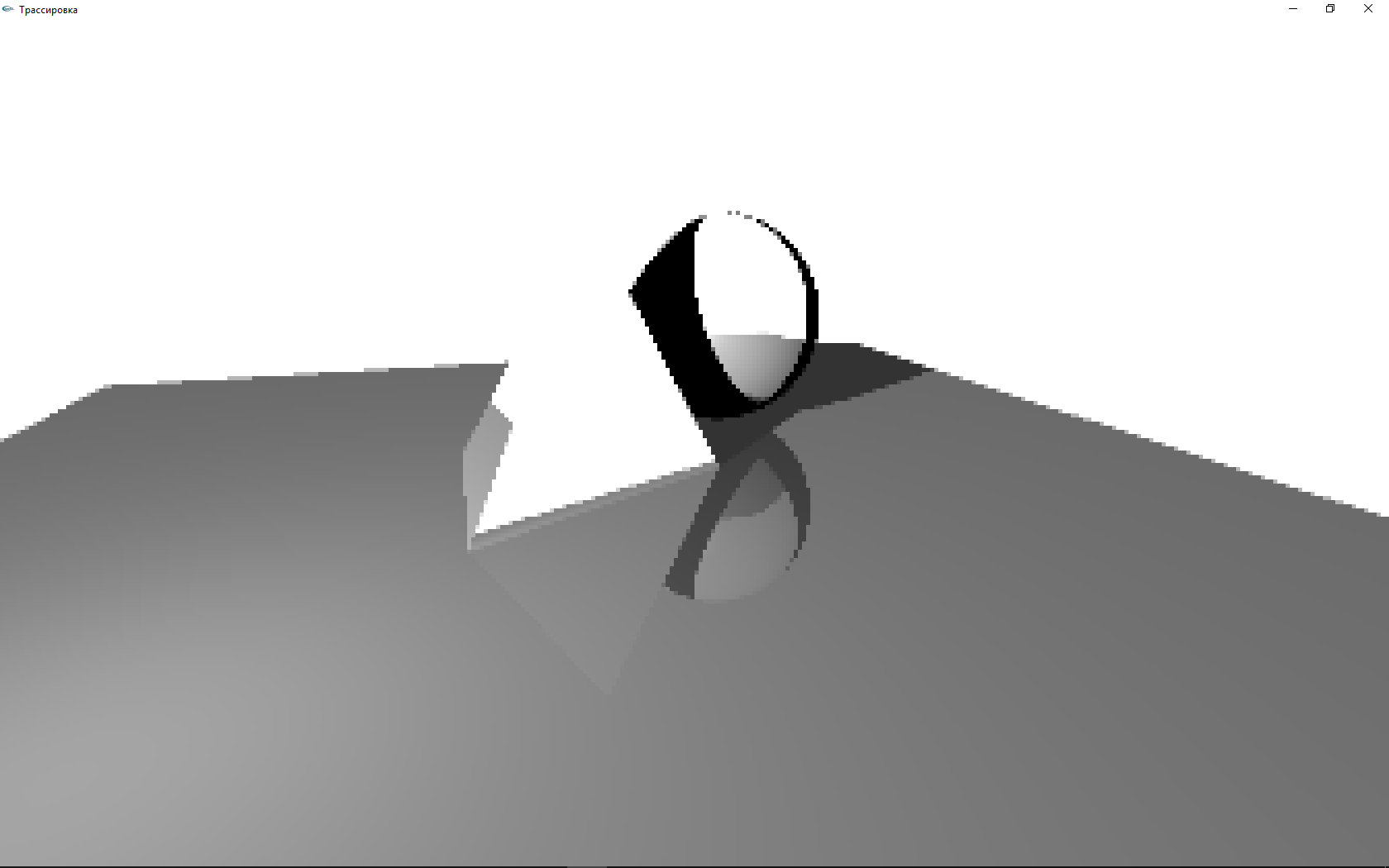


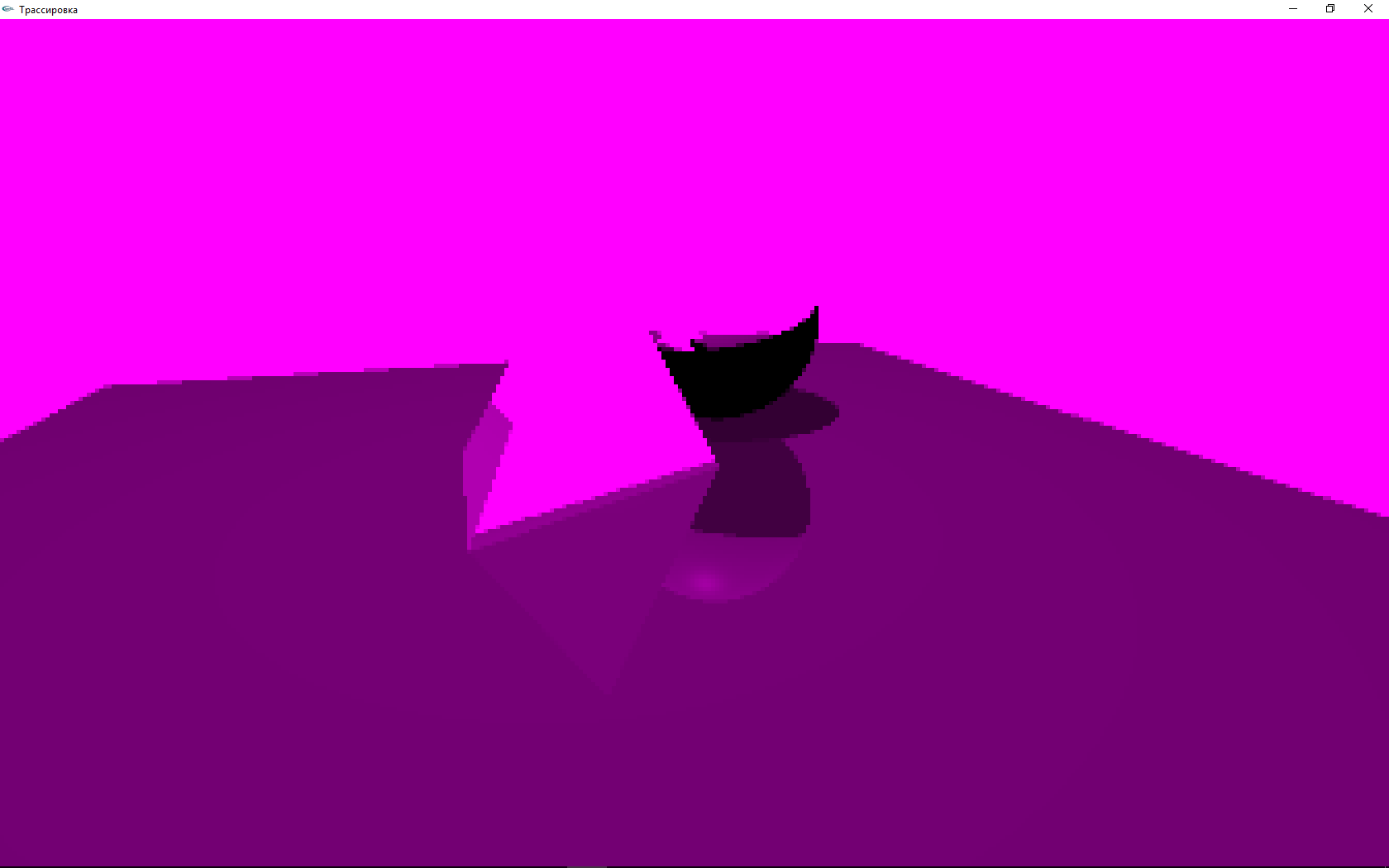
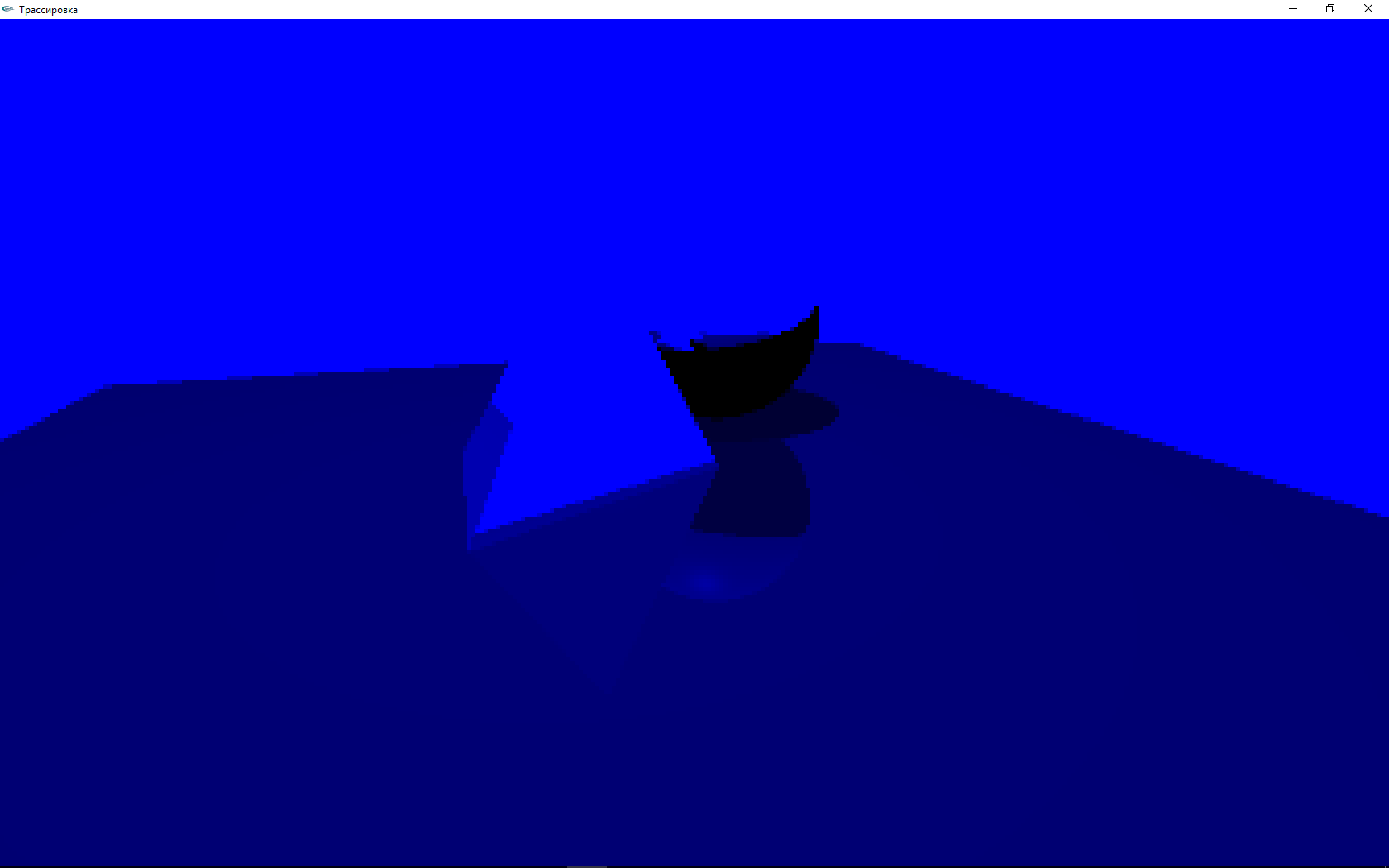




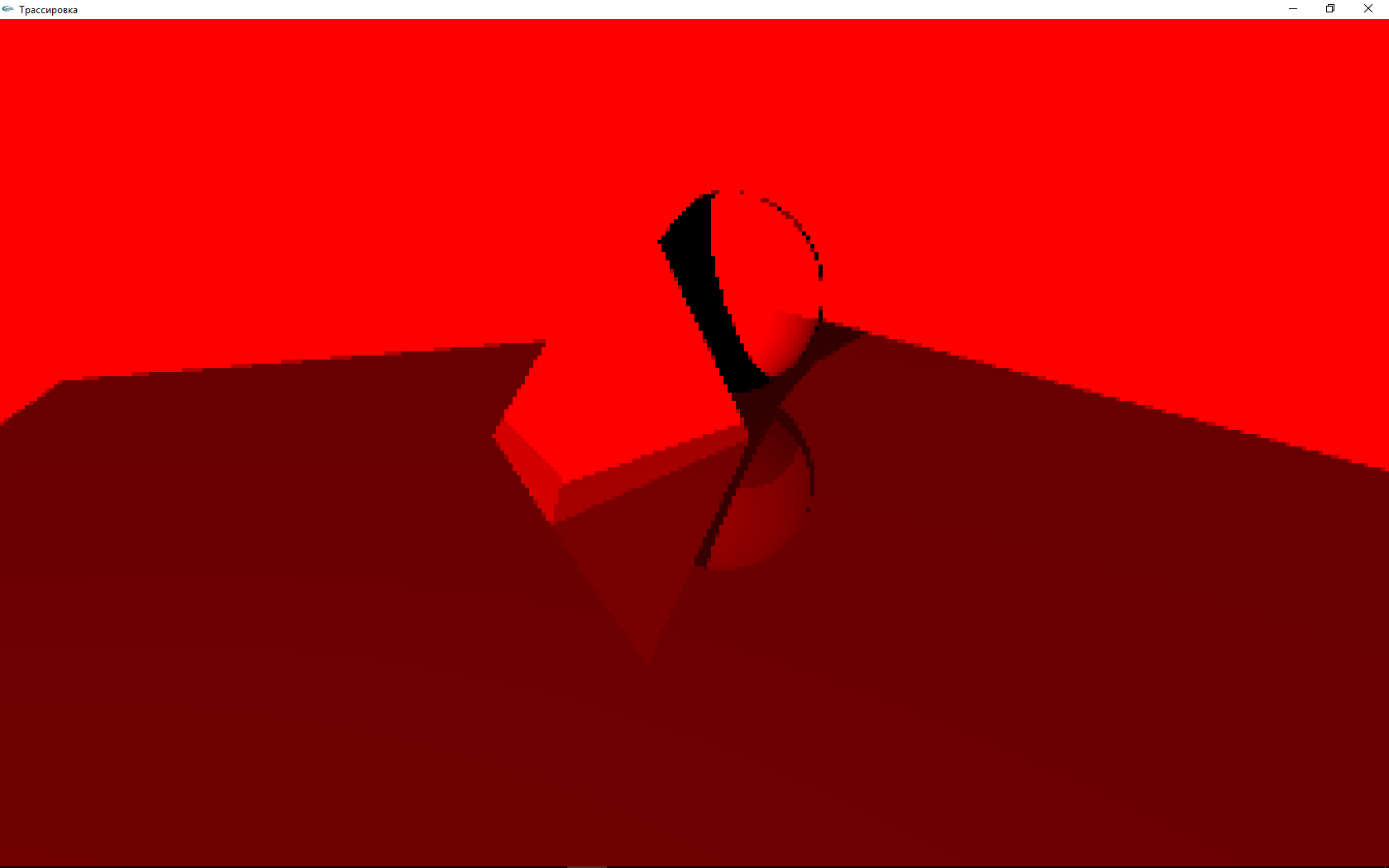
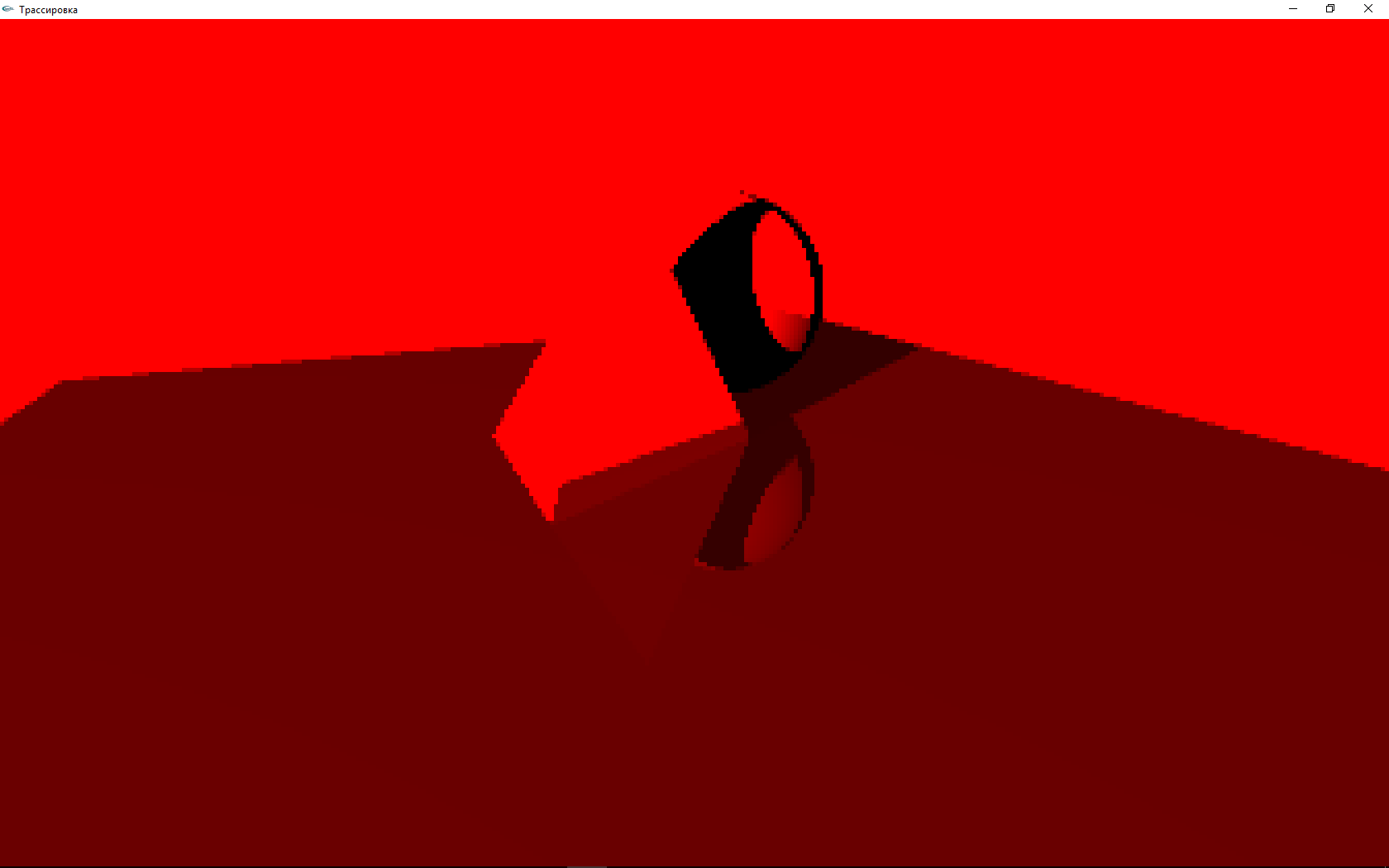
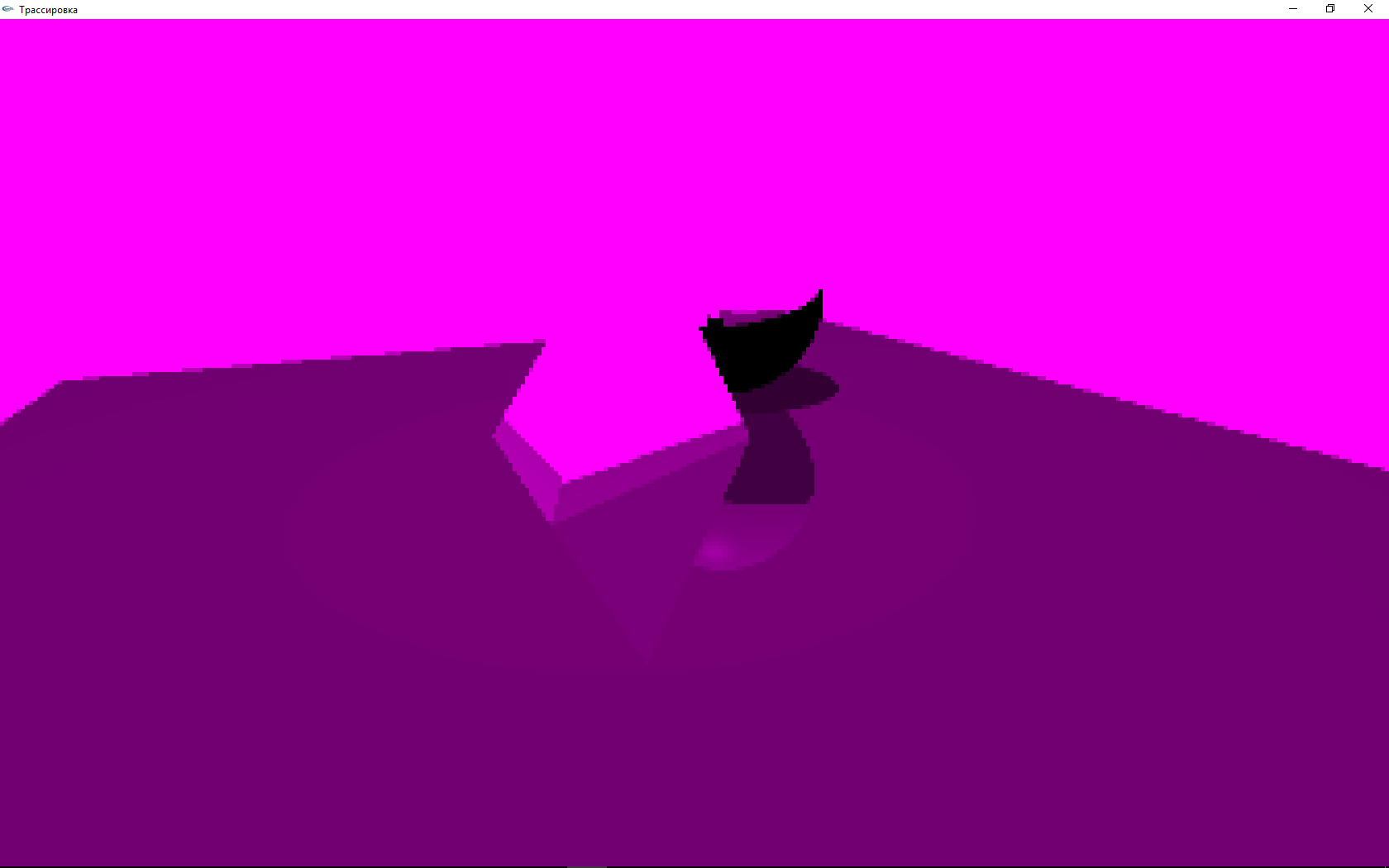
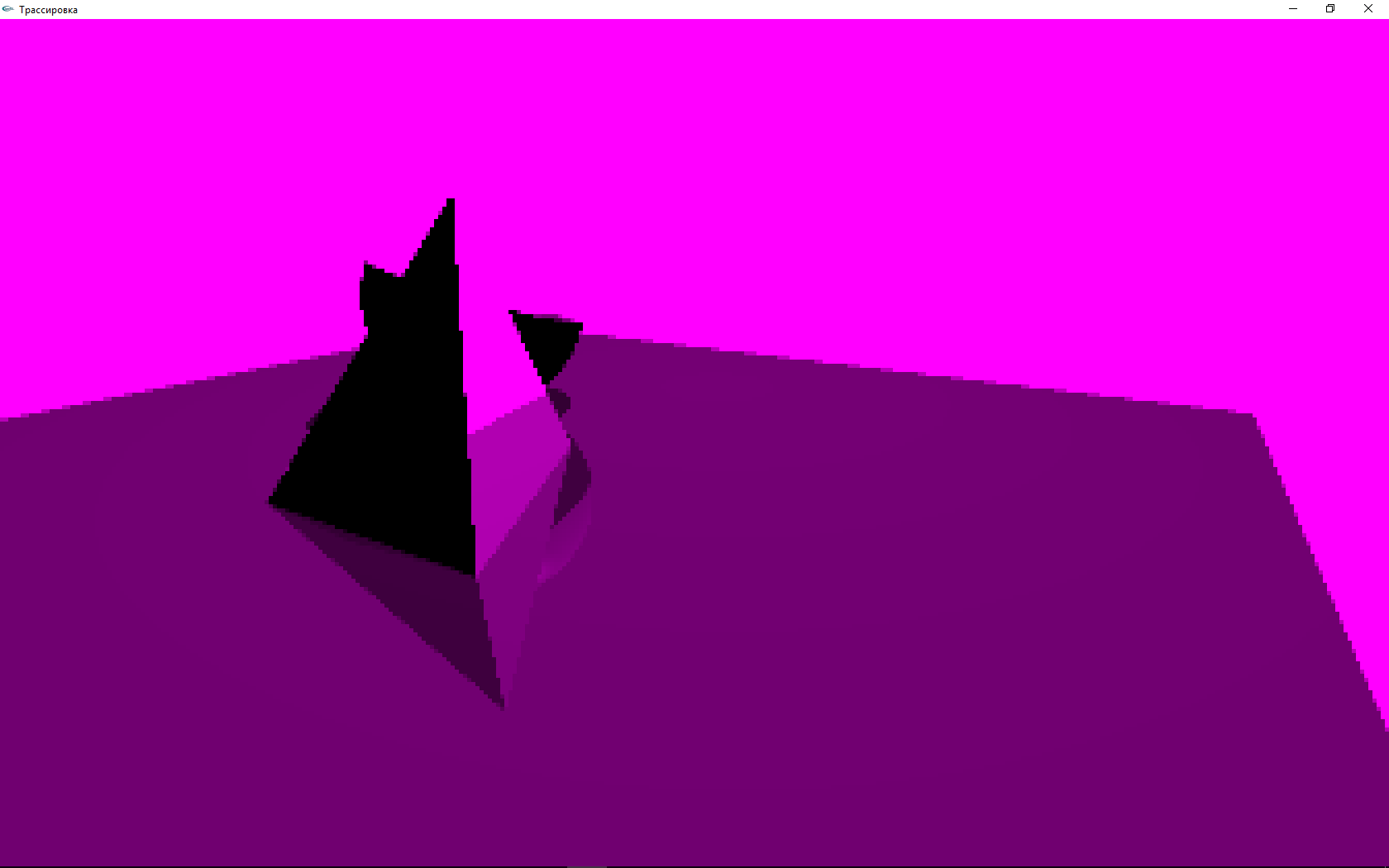
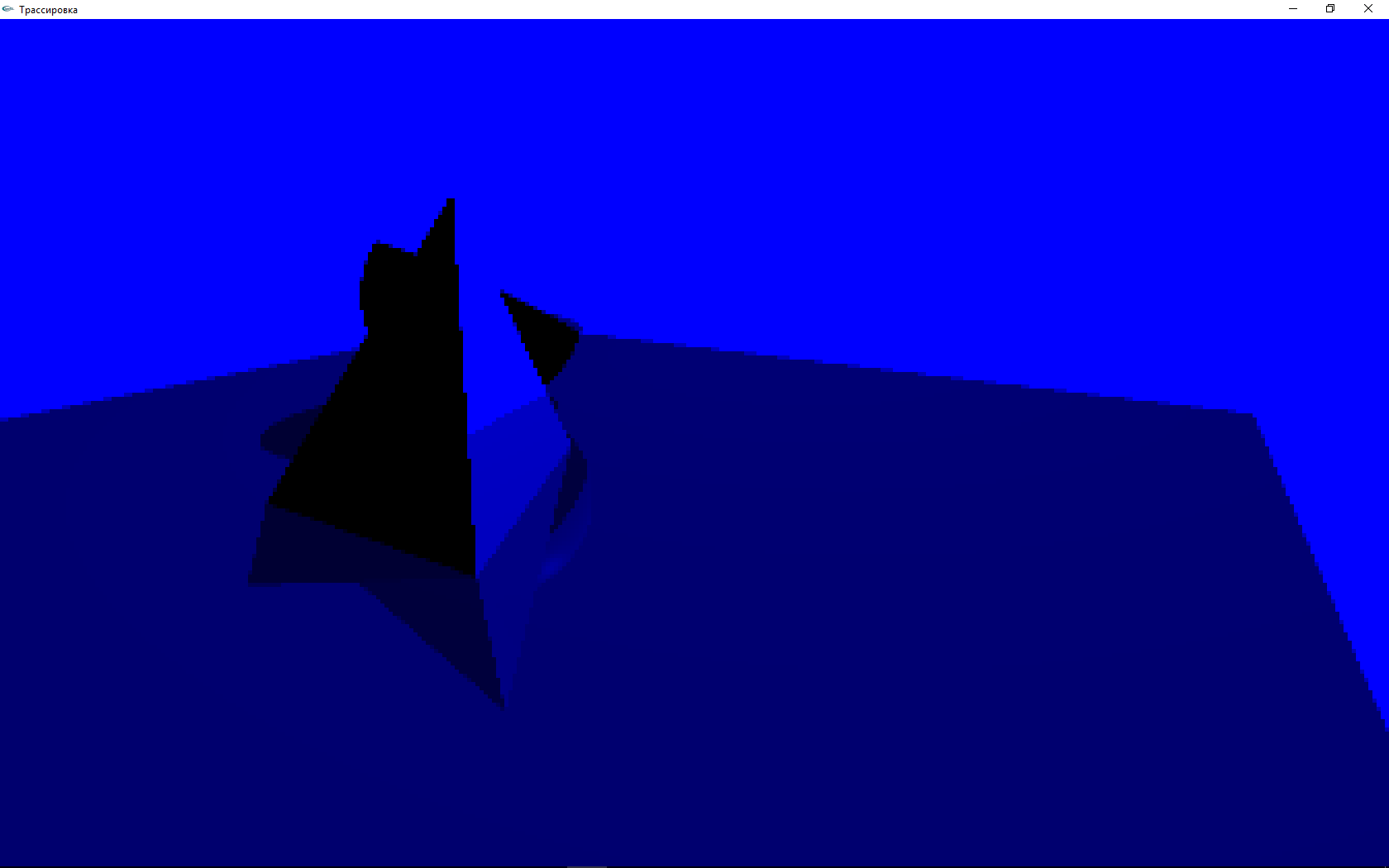
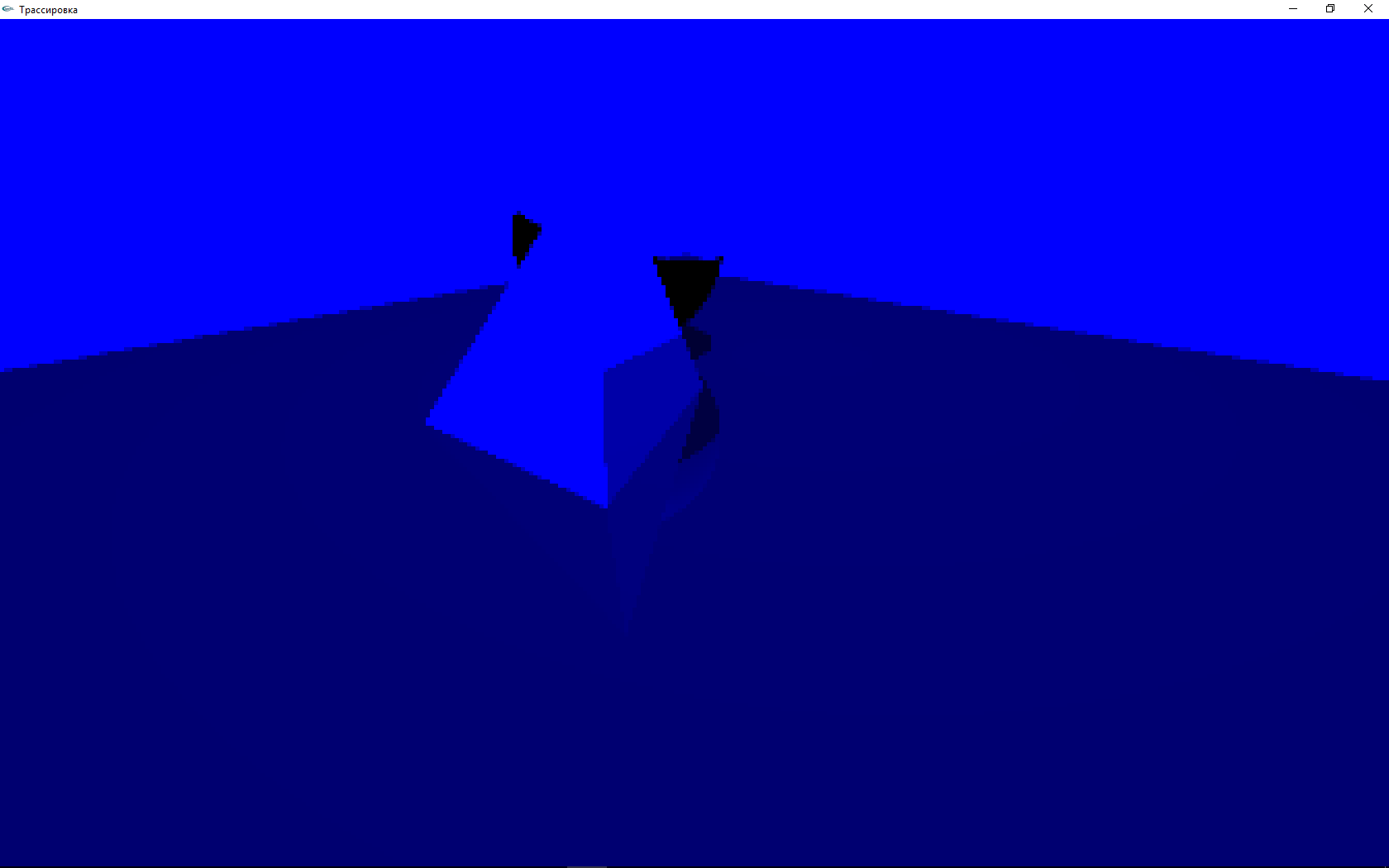


Без цвета фигур.

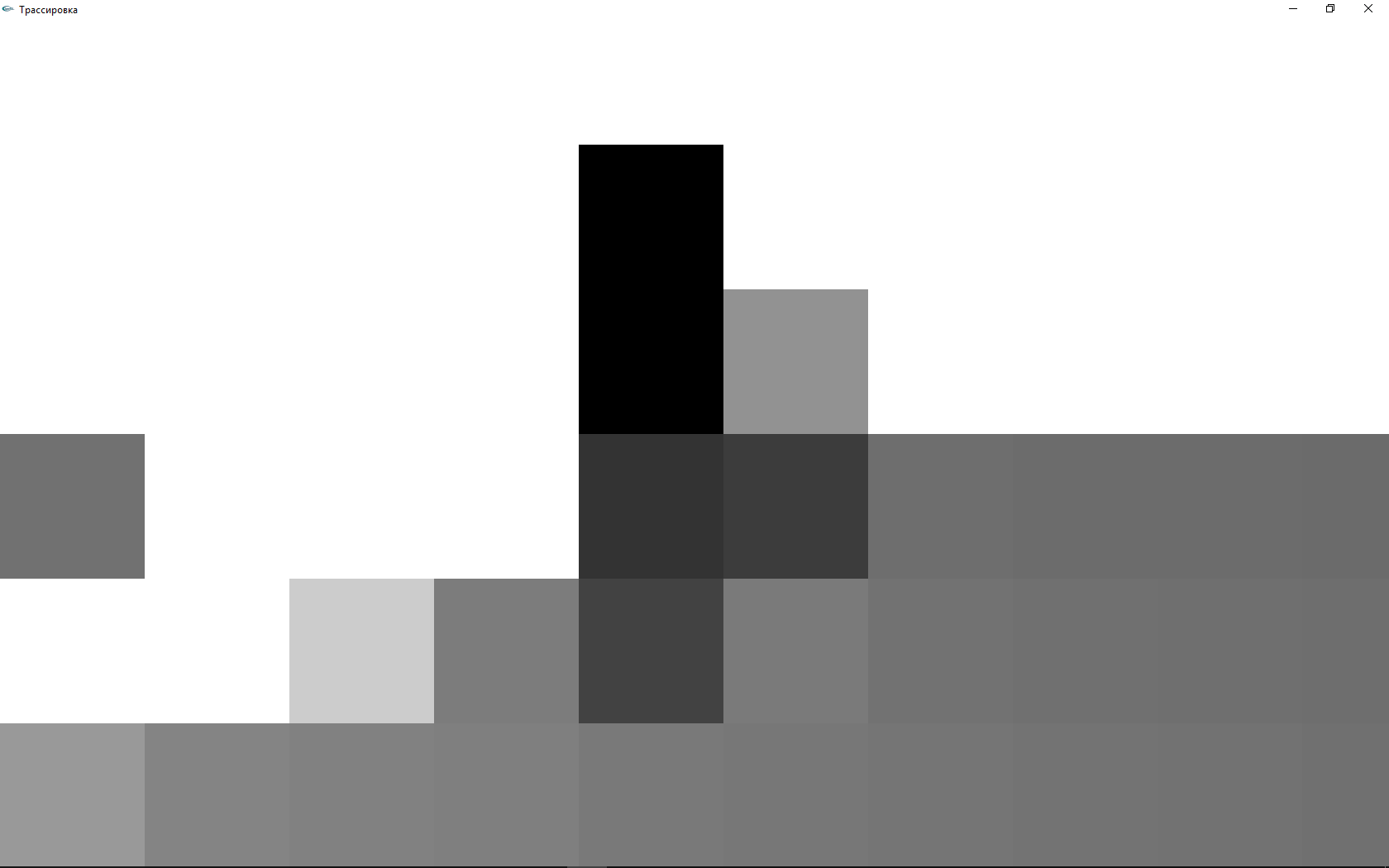


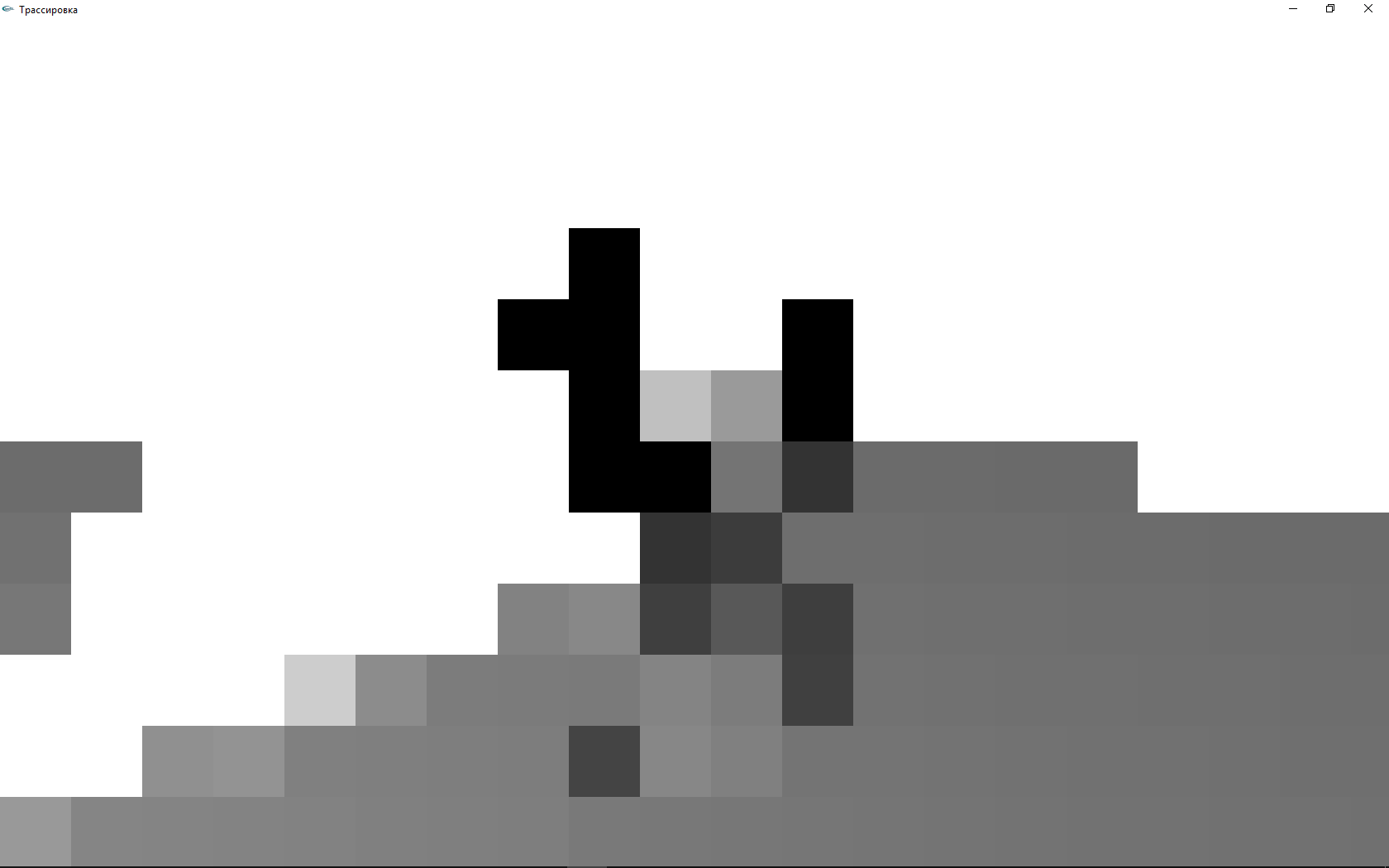
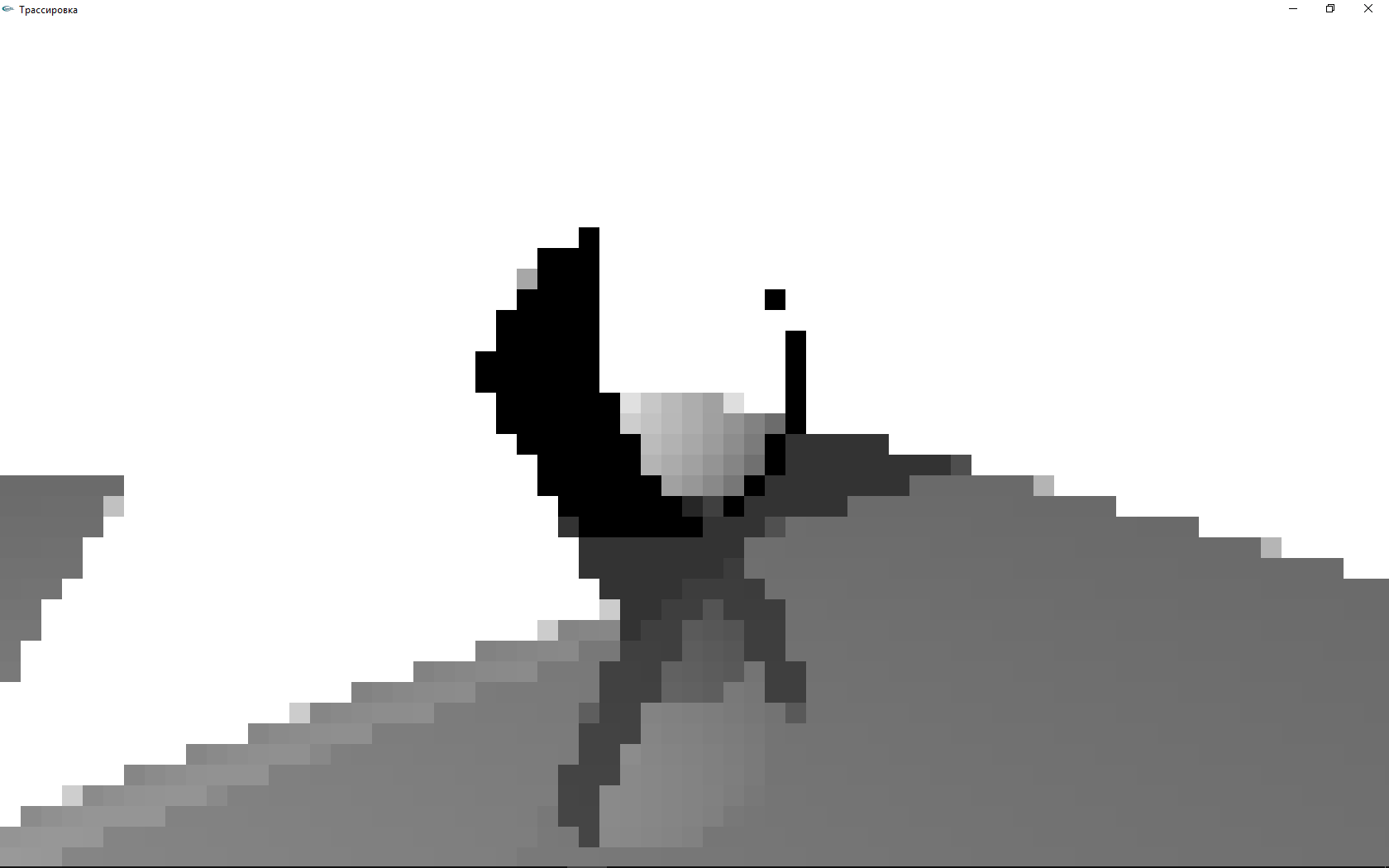
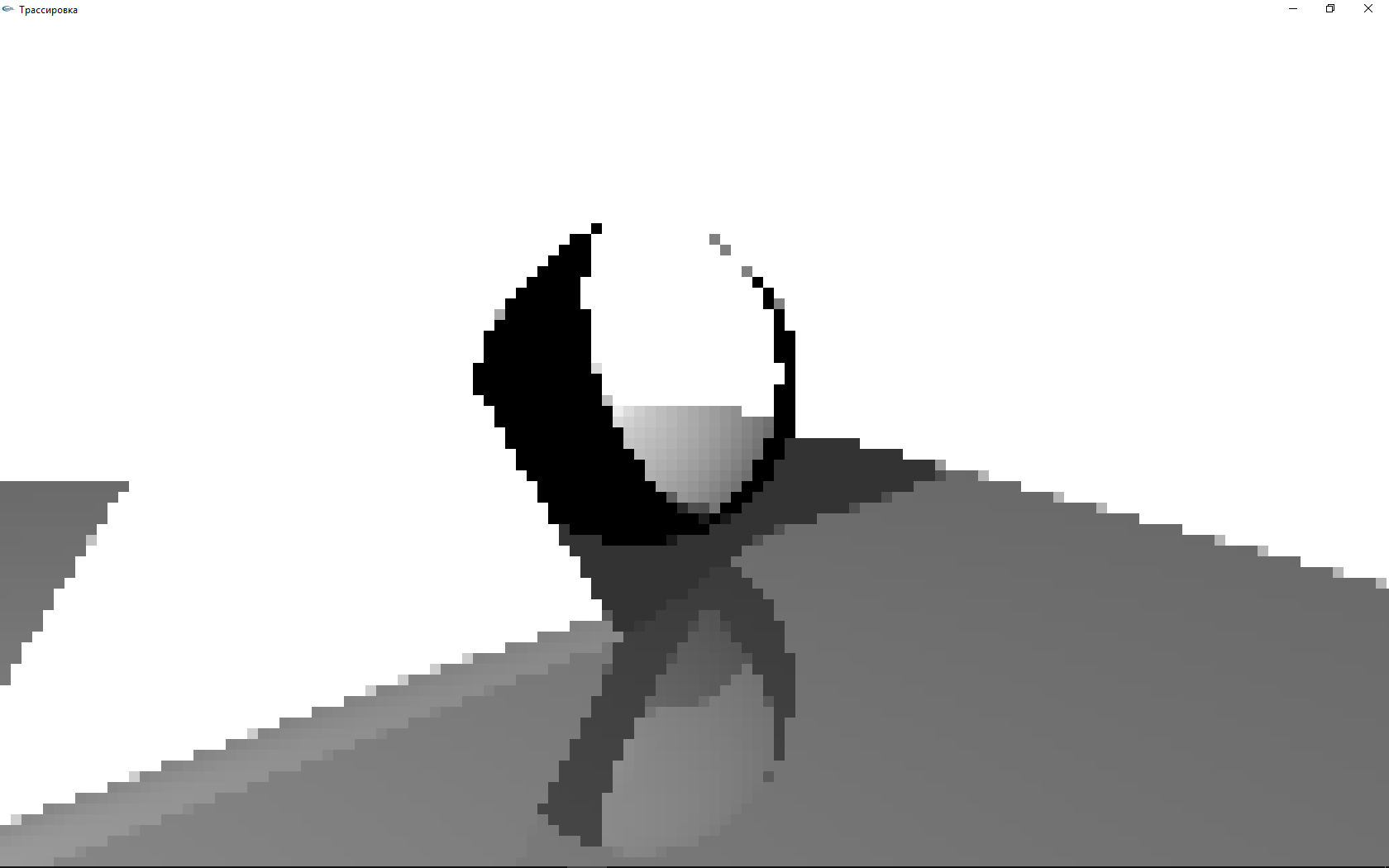
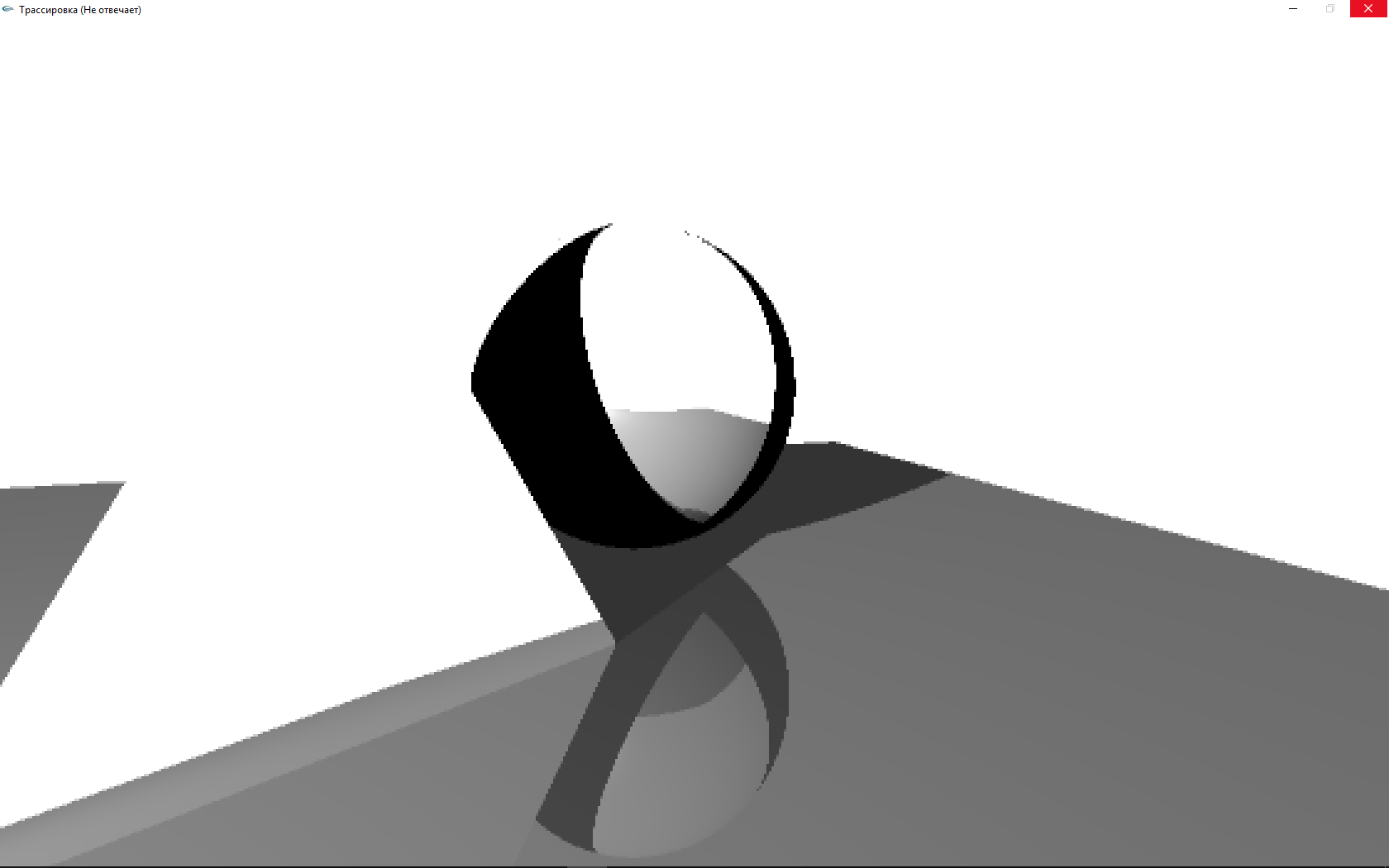
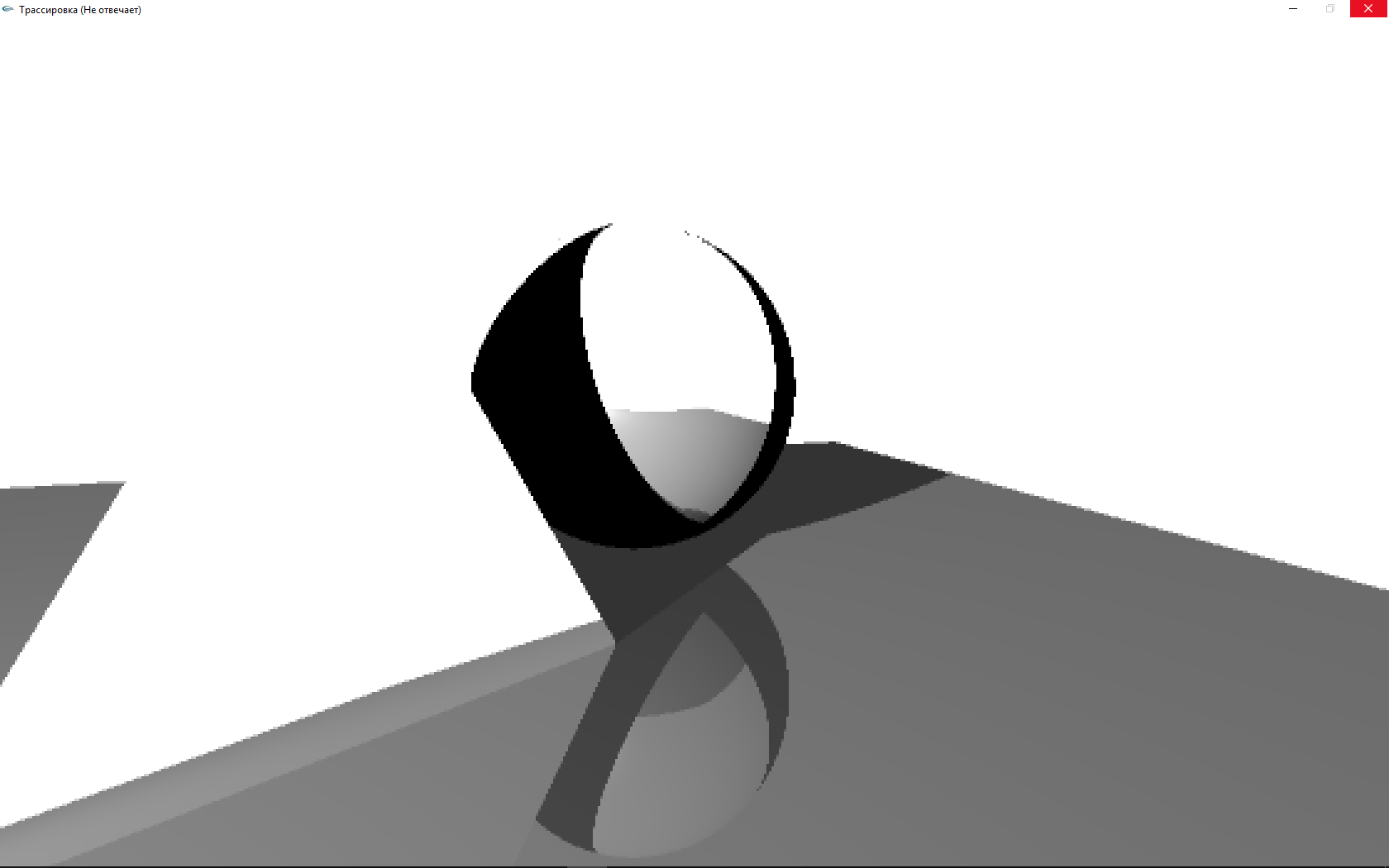
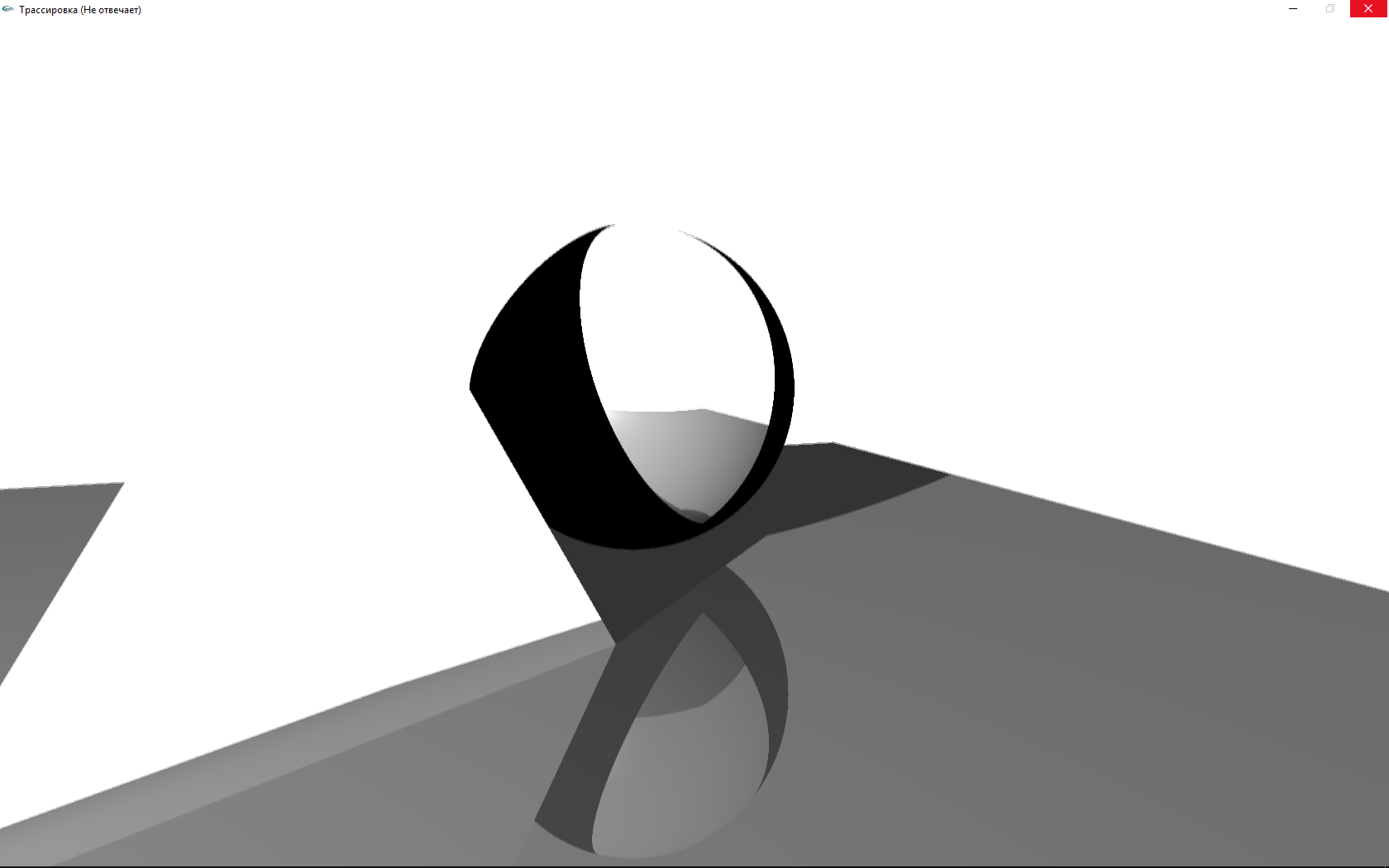
  

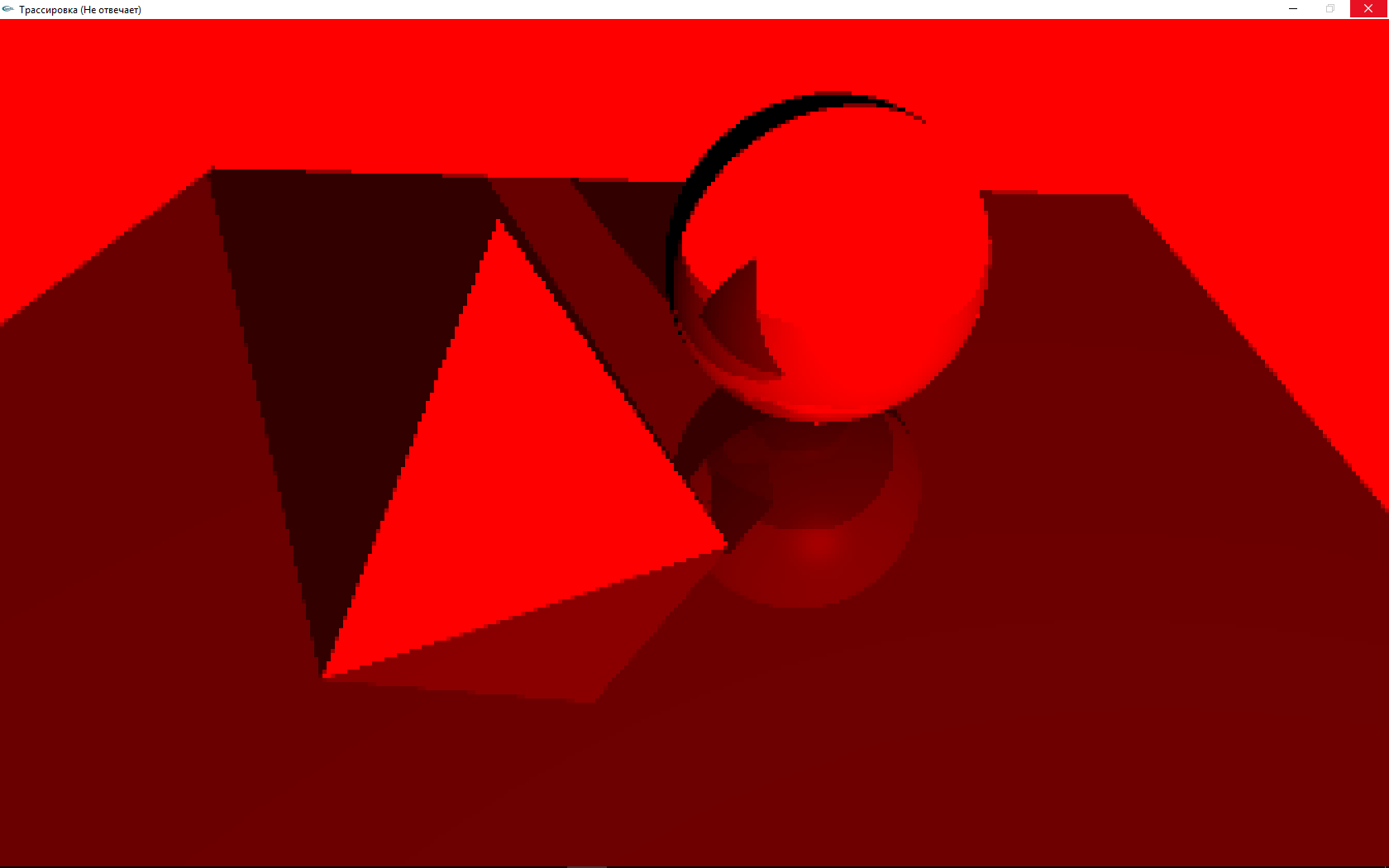
Пермещение источника освещения.

Тест 5. Изменение размеров пикселей (сглаживание).





Тест 6. Трассировка для для синей сферы, красным светом.

