МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра Вычислительной техники

ОТЧЁТ

по лабораторной работе №5

по дисциплине «Компьютерная графика»

Тема: Исследование алгоритмов выявления видимости сложных сцен

Вариант 5

Студенты гр. 9308 _____ Соболев М.С.
_____ Степовик В.С.
_____ Дубенков С.А.
Преподаватель _____ Матвеева И.В.

Санкт-Петербург,
2022

Оглавление

1. Введение

1.1. Цель работы

Исследование алгоритмов выявления видимости сложных сцен.

1.2. Задание

Вариант 5: Обеспечить реализацию видимости совокупности произвольных многогранников на основе использования Z буфера.

1.3. Используемые ресурсы

Для выполнения лабораторной работы использовался язык C++ и фреймворк Qt для визуализации.

2. Основные теоретические положения

Алгоритм, использующий z-буфер — это один из простейших алгоритмов удаления невидимых поверхностей. Впервые он был предложен Кетмулом. Работает этот алгоритм в пространстве изображения. Идея z-буфера является простым обобщением идеи о буфере кадра. Буфер кадра используется для запоминания атрибутов (интенсивности) каждого пикселя в пространстве изображения, z-буфер - это отдельный буфер глубины, используемый для запоминания координаты z или глубины каждого видимого пикселя в пространстве изображения. В процессе работы глубина или значение z каждого нового пикселя, который нужно занести в буфер кадра, сравнивается с глубиной того пикселя, который уже занесен в z-буфер. Если это сравнение показывает, что новый пиксел расположен впереди пикселя, находящегося в буфере кадра, то новый пиксел заносится в этот буфер и, кроме того, производится корректировка z-буфера новым значением z. Если же сравнение дает противоположный результат, то никаких действий не производится. По сути, алгоритм является поиском по x и y наибольшего значения функции z (x, y).

Главное преимущество алгоритма – его простота. Кроме того, этот алгоритм решает задачу об удалении невидимых поверхностей и делает тривиальной визуализацию пересечений сложных поверхностей. Сцены могут быть любой сложности. Поскольку габариты пространства изображения фиксированы, оценка вычислительной трудоемкости алгоритма не более чем линейна. Поскольку элементы сцены или картинки можно заносить в буфер кадра или в z-буфер в произвольном порядке, их не нужно предварительно сортировать по приоритету глубины. Поэтому экономится вычислительное время, затрачиваемое на сортировку по глубине.

Основной недостаток алгоритма - большой объём требуемой памяти. Если сцена подвергается видовому преобразованию и отсекается до фиксированного

диапазона значений координат z, то можно использовать z-буфер с фиксированной точностью. Информацию о глубине нужно обрабатывать с большей точностью, чем координатную информацию на плоскости (x, y); обычно бывает достаточно 20-ти бит. Буфер кадра размером 512´512´24 бит в комбинации с z-буфером размером 512´512´20 бит требует почти 1.5 мегабайт памяти. Однако снижение цен на память делает экономически оправданным создание специализированных запоминающих устройств для z-буфера и связанной с ним аппаратуры.

Альтернативой созданию специальной памяти для z-буфера является использование для этой цели оперативной памяти. Уменьшение требуемой памяти достигается разбиением пространства изображения на 4, 16 или больше квадратов или полос. В предельном варианте можно использовать z-буфер размером в одну строку развертки. Для последнего случая имеется интересный алгоритм построчного сканирования. Поскольку каждый элемент сцены обрабатывается много раз, то сегментирование z-буфера, вообще говоря, приводит к увеличению времени, необходимого для обработки сцены. Однако сортировка на плоскости, позволяющая не обрабатывать все многоугольники в каждом из квадратов или полос, может значительно сократить этот рост.

Другой недостаток алгоритма z-буфера состоит в трудоёмкости и высокой стоимости устранения лестничного эффекта, а также реализации эффектов прозрачности и просвечивания. Поскольку алгоритм заносит пиксели в буфер кадра в произвольном порядке, то нелегко получить информацию, необходимую для методов устранения лестничного эффекта, основывающихся на предварительной фильтрации. При реализации эффектов прозрачности и просвечивания пиксели могут заноситься в буфер кадра в некорректном порядке, что ведет к локальным ошибкам.

Формальное описание алгоритма z-буфера таково:

1. Заполнить буфер кадра фоновым значением интенсивности или цвета.

2. Заполнить z-буфер минимальным значением z.

3. Преобразовать каждый многоугольник в растровую форму в произвольном порядке.

4. Для каждого Пиксел(x,y) в многоугольнике вычислить его глубину z(x,y).

5. Сравнить глубину z(x,y) со значением Zбуфер(x,y), хранящимся в z-буфере в этой же позиции.

Если z(x,y) > Zбуфер (x,y), то записать атрибут этого многоугольника (интенсивность, цвет и т. п.) в буфер кадра и заменить Zбуфер(x,y) на z(x,y). В противном случае никаких действий не производить.

На псевдокоде алгоритм можно представить так:

```
for all objects;
  for all covered pixels;
    compare z;
```

В качестве предварительного шага там, где это целесообразно, применяется удаление нелицевых граней.

Если известно уравнение плоскости, несущей каждый многоугольник, то вычисление глубины каждого пикселя на сканирующей строке можно проделать пошаговым способом. Грань при этом рисуется последовательно (строка за строкой). Для нахождения необходимых значений используется линейная интерполяция.

Для рисунка y меняется от y_1 до y_2 и далее до y_3, при этом для каждой строки определяется x_a, z_a, x_b, z_b:

$$x_a = x_1 + (x_2 - x_1) \cdot \frac{y - y_1}{y_2 - y_1};$$

$$x_b = x_1 + (x_3 - x_1) \cdot \frac{y - y_1}{y_3 - y_1};$$

$$z_a = z_1 + (z_2 - z_1) \cdot \frac{y - y_1}{y_2 - y_1};$$

$$z_b = z_1 + (z_3 - z_1) \cdot \frac{y - y_1}{y_3 - y_1}.$$

На сканирующей строке x меняется от x_a до x_b и для каждой точки строки определяется глубина z:

$$z = z_a + (z_b - z_a) \cdot \frac{x - x_a}{x_b - x_a}.$$

Далее алгоритм сравнивает уже хранящийся элемент в z-буфере с тем, что получилось для данного многоугольника.

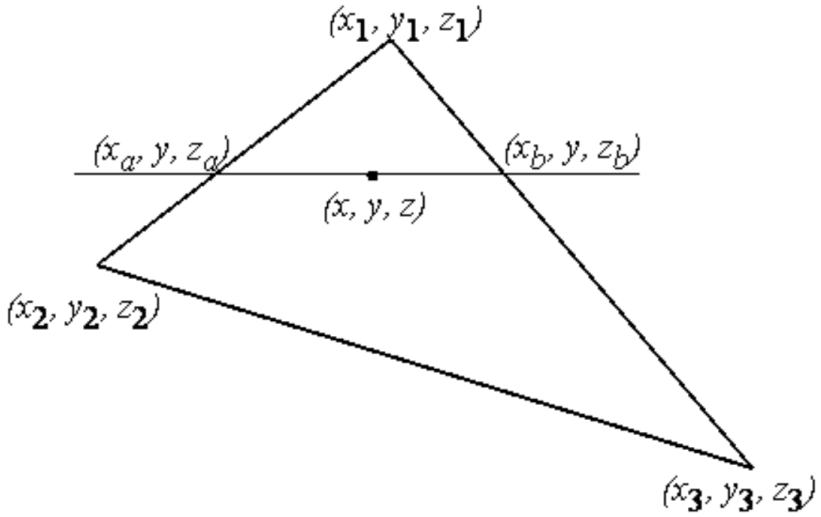

Рисунок 1. Пример сканирования изображения

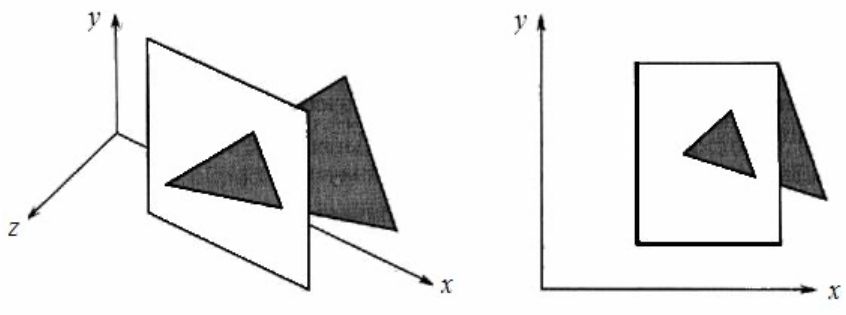

Рисунок 2. Пример работы алгоритма

3. Ход работы

```
#include <QWidget>
#include <QPainter>
#include <QKeyEvent>
#include <QColor>
#include <cmath>
#include <iostream>
#include <limits>

#include "./include/mainWindow.h"

#include "./include/Point.h"
#include "./include/Triangle.h"
#include "./include/matrix.h"



using namespace std;

double deg2rad(double a) { return (M_PI*a) / 180.0; }

unsigned calcColour(unsigned char r, unsigned char g, unsigned char b)
{
   unsigned res = 0, buff = 0;
   buff = b;
   res |= buff << 0;
   buff = g;
   res |= buff << 8;
   buff = r;
   res |= buff << 16;
   buff = 255;
   res |= buff << 24;
   return res;
}

MainWindow::MainWindow(QWidget *parent) : QWidget(parent), cam()
{
   resize(W, H);
   this->setStyleSheet("background-color: rgb(200,200,200); margin:0px; border:1px solid rgb(0, 0, 0); ");
```

```
    double newy = cam_begin_R*cos(deg2rad(cam_begin_angle));
    double newz = cam_begin_R*sin(deg2rad(cam_begin_angle));
    cam.move(0.0, -newy, newz);
    cranch(false);


    C_ = NULL;
    refresh_C_();

    display = new unsigned*[H];
    for(size_t li = 0; li < H; ++li)
        display[li] = new unsigned[W];
    refresh_display();

    z_buffer = new double*[W];
    for(size_t li = 0; li < W; ++li)
        z_buffer[li] = new double[H];
    refresh_z_buffer();
}

MainWindow::~MainWindow()
{
    if(C_ != NULL)
        delete C_;

    for(size_t li = 0; li < H; ++li)
        delete display[li];
    delete display;

    for(size_t li = 0; li < W; ++li)
        delete z_buffer[li];
    delete z_buffer;
}

void MainWindow::paintEvent(QPaintEvent *e)
{
    Q_UNUSED(e);
```

```
    refresh_display();
    refresh_z_buffer();

    putRectangle3D(Point(-5, 10, 10), Point(5, 10, 10), Point(-5, 10, -5), Point(5, 10, -5),
            calcColour(150, 150, 150));


    putRectangle3D(Point(-0.5, -0.5, 10), Point(0.5, 0.5, 10), Point(-0.5, -0.5, -5), Point(0.5, 0.5, -5),
            calcColour(155, 103, 60));
    putRectangle3D(Point(-0.5, 0.5, 10), Point(0.5, -0.5, 10), Point(-0.5, 0.5, -5), Point(0.5, -0.5, -5),
            calcColour(155, 103, 60));
    putTriangle3D(Triangle(Point(-5, 5, 7), Point(-5, -5, 7), Point(0, 0, 12)),
                    calcColour(255, 0, 0));
    putTriangle3D(Triangle(Point(5, -5, 7), Point(5, 5, 7), Point(0, 0, 12)),
                    calcColour(255, 0, 0));
    putTriangle3D(Triangle(Point(5, 5, 7), Point(-5, 5, 7), Point(0, 0, 12)),
                    calcColour(0, 0, 255));
    putTriangle3D(Triangle(Point(5, -5, 7), Point(-5, -5, 7), Point(0, 0, 12)),
                    calcColour(0, 0, 255));

    QPainter qp(this);
    for(size_t li = 0; li < H; ++li)
        for(size_t lj = 0; lj < W; ++lj)
            {
                qp.setPen(QColor(display[li][lj]));
                qp.drawPoint(lj, li);
            }
}

void MainWindow::cranch(bool where)
{
    const double drotate = 1;
    int times = (int)(cam_begin_angle/drotate + 0.5);
    for(int i = 0; i < times; ++i)
    {
        cam.rotateOX(where?drotate:-drotate);
    }
}
```

```
void MainWindow::keyPressEvent(QKeyEvent *event)
{
    int key = event->key();

    double dd = 5;

    if(key == Qt::Key_Left)
    {
        //cam.rotateOX(60);
        cranch(true);

        double x_new, y_new;
        rotateVector(cam.o().x(), cam.o().y(), -dd, &x_new, &y_new);
        double dx = x_new - cam.o().x();
        double dy = y_new - cam.o().y();
        cam.move(dx, dy, 0.0);
        cam.rotateOZ(-dd);

        //cam.rotateOX(-60);
        cranch(false);
    }
    else if(key == Qt::Key_Right)
    {
        //cam.rotateOX(60);
        cranch(true);

        double x_new, y_new;
        rotateVector(cam.o().x(), cam.o().y(), dd, &x_new, &y_new);
        double dx = x_new - cam.o().x();
        double dy = y_new - cam.o().y();
        cam.move(dx, dy, 0.0);
        cam.rotateOZ(dd);

        //cam.rotateOX(-60);
        cranch(false);
    }

    refresh_C_();
```

```
    update();
}

void MainWindow::rotateVector(double x_old, double y_old, double angle_degrees, double *x_new, double *y_new)
{
    double a = deg2rad(angle_degrees);

    double x = x_old, y = y_old;

    double si = sin(a);
    double co = cos(a);

    *x_new = x*co - y*si;
    *y_new = x*si + y*co;
}

Point MainWindow::projectionOnCamera(const Point &p)
{
    return projectionOnCamera(p.x(), p.y(), p.z());
}

Point MainWindow::projectionOnCamera(double x, double y, double z)
{
    Point cam_o = cam.o();
    double obj_x = x - cam_o.x();
    double obj_y = y - cam_o.y();
    double obj_z = z - cam_o.z();

    Matrix<double> old_v(3, 1);
    old_v.set(obj_x, 0, 0); old_v.set(obj_y, 1, 0); old_v.set(obj_z, 2, 0);

    Matrix<double> new_v = C_->multiply(old_v);

    double x_ = new_v.get(0, 0);
    double y_ = new_v.get(1, 0);
    double z_ = new_v.get(2, 0);

    return Point(x_, y_, z_);
}
```

```
Point MainWindow::projection3DOn2D(const Point &p)
{
   return projection3DOn2D(p.x(), p.y(), p.z());
}

Point MainWindow::projection3DOn2D(double x, double y, double z)
{
   Point cam_o = cam.o();
   double obj_x = x - cam_o.x();
   double obj_y = y - cam_o.y();
   double obj_z = z - cam_o.z();

   Matrix<double> old_v(3, 1);
   old_v.set(obj_x, 0, 0); old_v.set(obj_y, 1, 0); old_v.set(obj_z, 2, 0);

   Matrix<double> new_v = C_->multiply(old_v);
   double x_ = new_v.get(0, 0);
   double y_ = new_v.get(1, 0);
   double z_ = new_v.get(2, 0);


   double xp, zp;

   xp = ((n*x_)/y_);
   zp = ((n*z_)/y_);


   double x_res, y_res;
   x_res = ((xp+r)*(W-1))/(r+r);
   y_res = ((zp+t)*(H-1))/(t+t);

   return Point(x_res, y_res, 0.0);
}

bool MainWindow::isOnDisplay(int x, int y)
{
   if(x < 0 || x >= (int)W)
      return false;
   if(y < 0 || y >= (int)H)
```

```
        return false;
    return true;
}

void MainWindow::refresh_C_()
{
    Point vx(cam.vr());
    Point vy(cam.vf());
    Point vz(cam.vu());

    Matrix<double> C(3, 3);
    C.set(vx.x(), 0, 0); C.set(vy.x(), 0, 1); C.set(vz.x(), 0, 2);
    C.set(vx.y(), 1, 0); C.set(vy.y(), 1, 1); C.set(vz.y(), 1, 2);
    C.set(vx.z(), 2, 0); C.set(vy.z(), 2, 1); C.set(vz.z(), 2, 2);
    //std::cout << C.toString() << std::endl;
    Matrix<double> C_buff = C.inverse();


    if(C_ != NULL)
        delete C_;
    C_ = new Matrix<double>(C_buff);
}

void MainWindow::refresh_display()
{
    for(size_t li = 0; li < H; ++li)
        for(size_t lj = 0; lj < W; ++lj)
            display[li][lj] = GlobalBackgroundColor;
}

void MainWindow::refresh_z_buffer()
{
    double max_double = numeric_limits<double>::infinity();
    for(size_t li = 0; li < W; ++li)
        for(size_t lj = 0; lj < H; ++lj)
            z_buffer[li][lj] = max_double;
}
```

```
void MainWindow::putPointOnScreen(int x, int y, unsigned **display, unsigned colo, double **z_buffer, const Triangle &tri)
{
   Point O(0.0, 0.0, 0.0);
   double _x = ((double)x*r*2.0)/((double)W-1) - r;
   double _y = ((double)y*t*2.0)/((double)H-1) - t;
   Point P(_x, n, _y);


   Point crossP = tri.crossLine(O, P);
   double z_crossed = crossP.y();

   if(z_crossed < z_buffer[x][y])
   {
      if(n <= z_crossed && z_crossed <= f)
      {
         z_buffer[x][y] = z_crossed;
         size_t display_y = (size_t)y;
         size_t display_x = (size_t)x;
         display[H-1-display_y][display_x] = colo;
      }
   }
}

void MainWindow::printLineBeziers(int x1, int y1, int x2, int y2, unsigned colo, unsigned **display, double **z_buffer, const Triangle &tri)
{
   int deltaX = abs(x2 - x1);
   int deltaY = abs(y2 - y1);
   int signX = x1 < x2 ? 1 : -1;
   int signY = y1 < y2 ? 1 : -1;
   //
   int error = deltaX - deltaY;
   //
   //qp.drawPoint(x2, y2);
   if(isOnDisplay(x2, y2))
      putPointOnScreen(x2, y2, display, colo, z_buffer, tri);
   while(x1 != x2 || y1 != y2)
   {
```

```
        //qp.drawPoint(x1, y1);
        if(isOnDisplay(x1, y1))
            putPointOnScreen(x1, y1, display, colo, z_buffer, tri);
        int error2 = error * 2;
        //
        if(error2 > -deltaY)
        {
            error -= deltaY;
            x1 += signX;
        }
        if(error2 < deltaX)
        {
            error += deltaX;
            y1 += signY;
        }
    }
}

void MainWindow::putTriangle3D(const Triangle &tri, unsigned colo)
{
    Triangle tri_camera(projectionOnCamera(tri.p1()),
                        projectionOnCamera(tri.p2()),
                        projectionOnCamera(tri.p3()) );
    if(tri_camera.p1().y() < n && tri_camera.p2().y() < n && tri_camera.p3().y() < n)
        return;
    Point A;
    Point B;
    Point C;
    if(tri_camera.p1().y() < 0)
    {
        double x_ = -tri_camera.p1().x();
        double z_ = -tri_camera.p1().z();
        double y_ =  tri_camera.p1().y();


        double xp = ((n*x_)/y_);
        double zp = ((n*z_)/y_);
        double x_res, y_res;
        x_res = ((xp+r)*(W-1))/(r+r);
```

```
    y_res = ((zp+t)*(H-1))/(t+t);
    A = Point(x_res, y_res, 0.0);
}
else
    A = projection3DOn2D(tri.p1());

if(tri_camera.p2().y() < 0)
{
    double x_ = -tri_camera.p2().x();
    double z_ = -tri_camera.p2().z();
    double y_ =  tri_camera.p2().y();

    double xp = ((n*x_)/y_);
    double zp = ((n*z_)/y_);
    double x_res, y_res;
    x_res = ((xp+r)*(W-1))/(r+r);
    y_res = ((zp+t)*(H-1))/(t+t);
    B = Point(x_res, y_res, 0.0);
}
else
    B = projection3DOn2D(tri.p2());

if(tri_camera.p3().y() < 0)
{
    double x_ = -tri_camera.p3().x();
    double z_ = -tri_camera.p3().z();
    double y_ =  tri_camera.p3().y();

    double xp = ((n*x_)/y_);
    double zp = ((n*z_)/y_);
    double x_res, y_res;
    x_res = ((xp+r)*(W-1))/(r+r);
    y_res = ((zp+t)*(H-1))/(t+t);
    C = Point(x_res, y_res, 0.0);
}
else
    C = projection3DOn2D(tri.p3());

int x0 = (int)(A.x()), y0 = (int)(A.y());
```

```
int x1 = (int)(B.x()), y1 = (int)(B.y());
int x2 = (int)(C.x()), y2 = (int)(C.y());

int tmp = 0;
if(y0 > y1)
{
   tmp = y0;
   y0 = y1;
   y1 = tmp;
   tmp = x0;
   x0 = x1;
   x1 = tmp;
}
if(y0 > y2)
{
   tmp = y0;
   y0 = y2;
   y2 = tmp;
   tmp = x0;
   x0 = x2;
   x2 = tmp;
}
if(y1 > y2)
{
   tmp = y1;
   y1 = y2;
   y2 = tmp;
   tmp = x1;
   x1 = x2;
   x2 = tmp;
}

int cross_x1 = 0, cross_x2 = 0;


int dx1 = x1 - x0;
int dy1 = y1 - y0;
int dx2 = x2 - x0;
int dy2 = y2 - y0;
```

```
    int top_y = y0;

    while(top_y < y1)
    {
        cross_x1 = x0 + dx1 * (top_y - y0) / dy1;
        cross_x2 = x0 + dx2 * (top_y - y0) / dy2;
        printLineBeziers(cross_x1, top_y, cross_x2, top_y, colo, display, z_buffer, tri_camera);
        ++top_y;
    }

    dx1 = x2 - x1;
    dy1 = y2 - y1;

    while(top_y < y2)
    {
        cross_x1 = x1 + dx1 * (top_y - y1) / dy1;
        cross_x2 = x0 + dx2 * (top_y - y0) / dy2;
        printLineBeziers(cross_x1, top_y, cross_x2, top_y, colo, display, z_buffer, tri_camera);
        ++top_y;
    }
}

void MainWindow::putRectangle3D(const Point &lu, const Point &ru, const Point &ld, const Point &rd, unsigned colo)
{
    Triangle one1(lu, ru, ld);
    Triangle two2(ld, rd, ru);

    putTriangle3D(one1, colo);
    putTriangle3D(two2, colo);
}
```

4. Пример работы программы

Пример работы программы представлен на рисунках ниже. Программа запускается через системный терминал, для поворота используются стрелки «вправо» и «влево», чтобы вращать камеру вправо и влево относительно центра соответственно.

На экране появляется крыша в виде зонта и стена. Все элементы раскрашены в разные цвета, и при повороте камеры меняется видимость различных объектов. Благодаря тому, что объекты либо исчезают из вида, либо накладываются друг на друга при «проекции» на экран, но при этом корректно отображаются для пользователя, можно убедиться в том, что алгоритм с использованием Z-буфера работает.

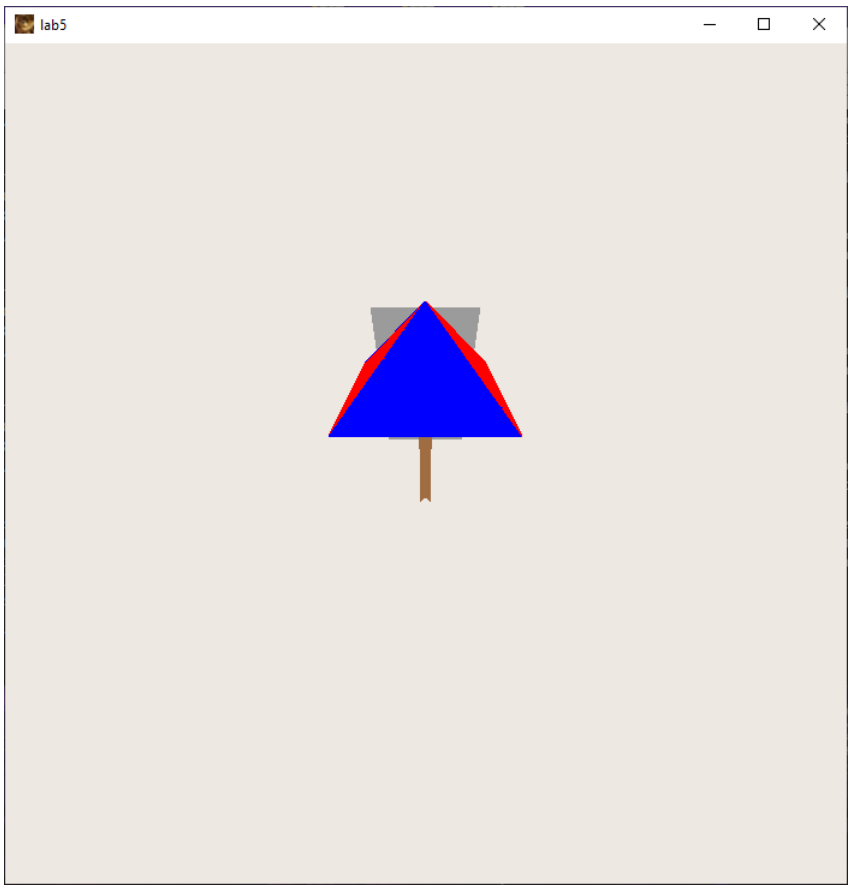

Рисунок 3. Пример работы программы

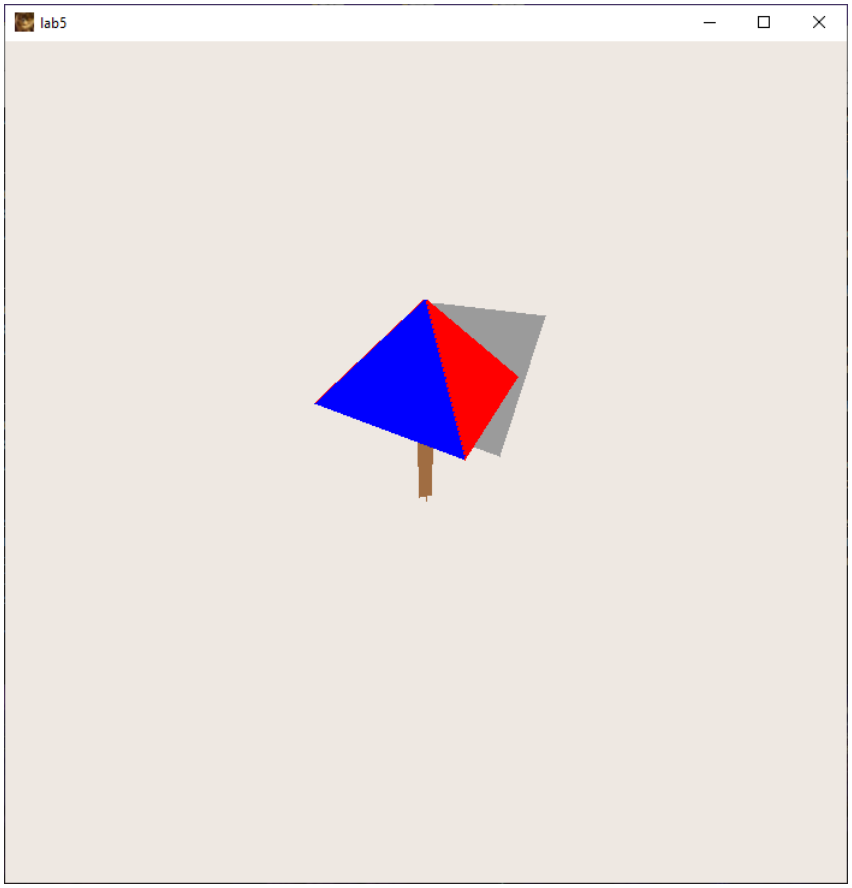

Рисунок 4. Пример работы программы

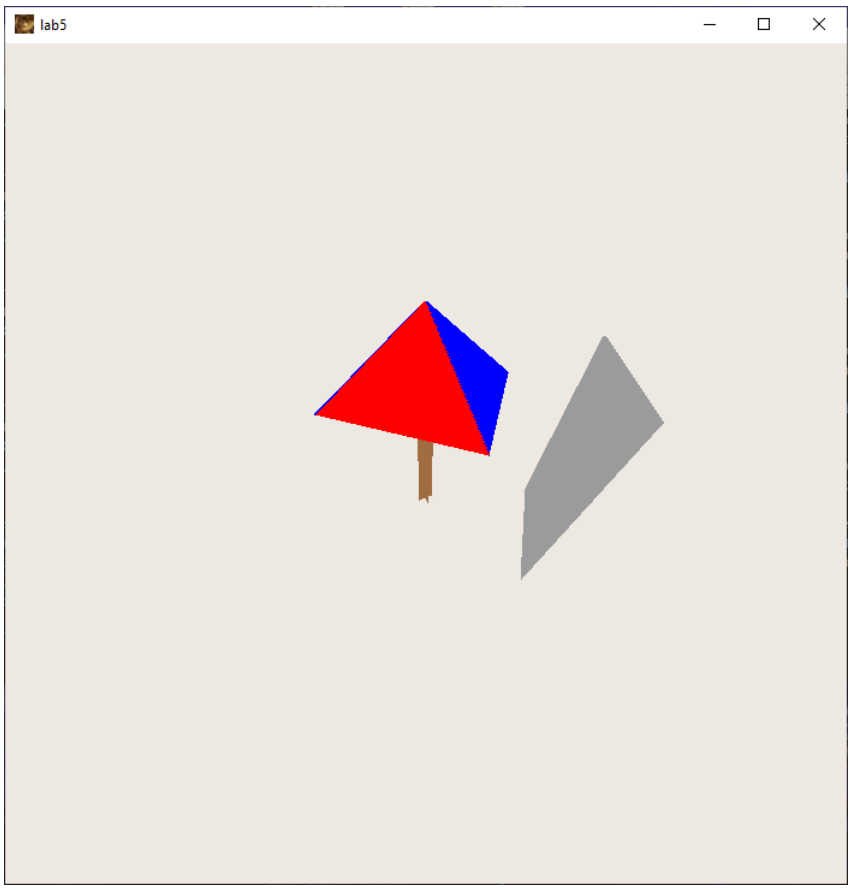

Рисунок 5. Пример работы программы

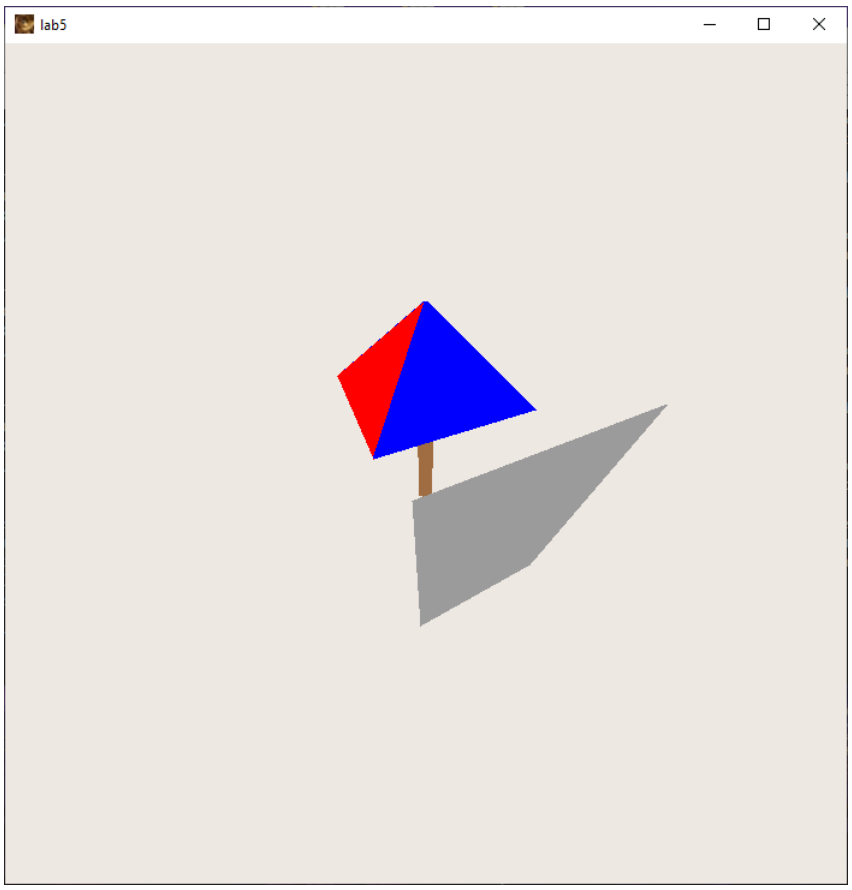

Рисунок 6. Пример работы программы

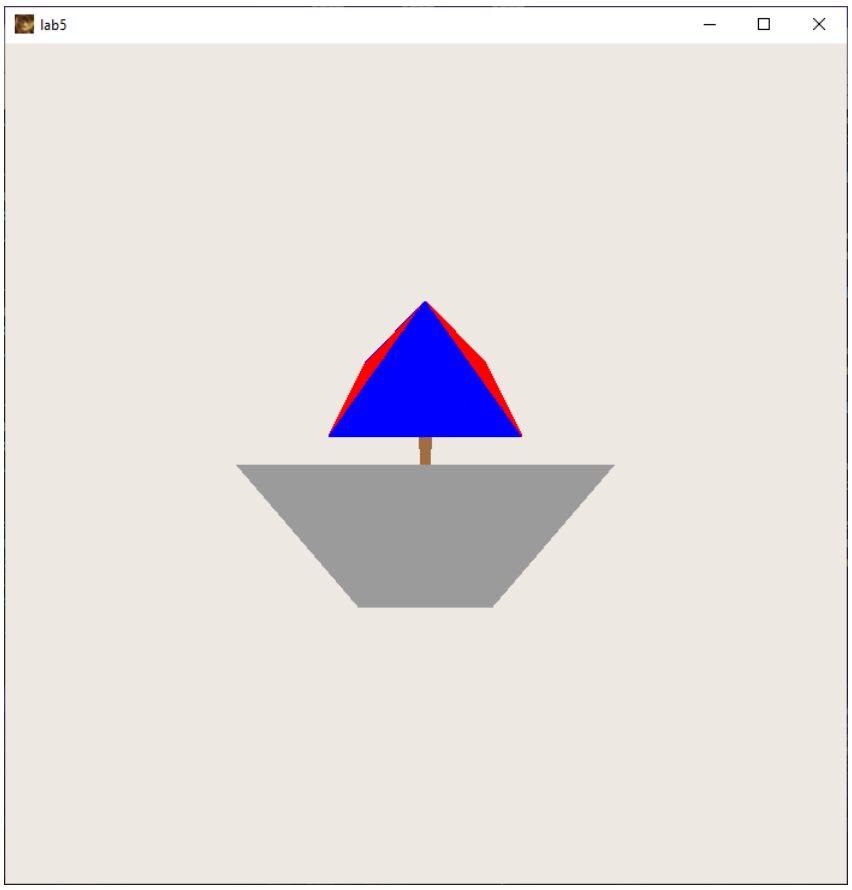

Рисунок 7. Пример работы программы

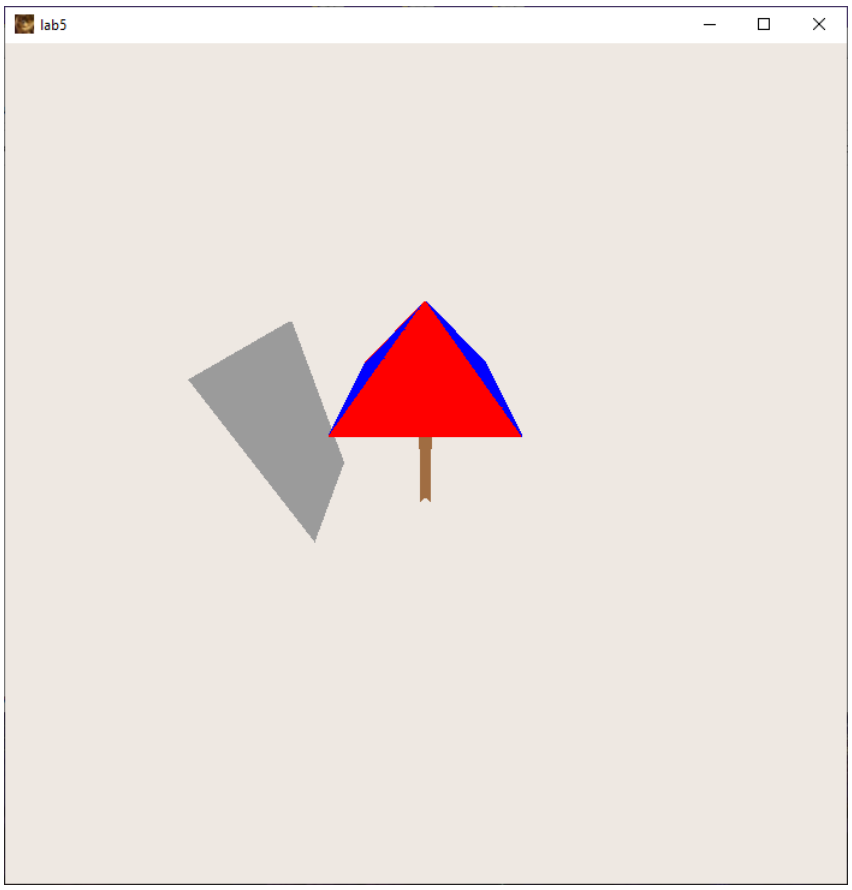

Рисунок 8. Пример работы программы

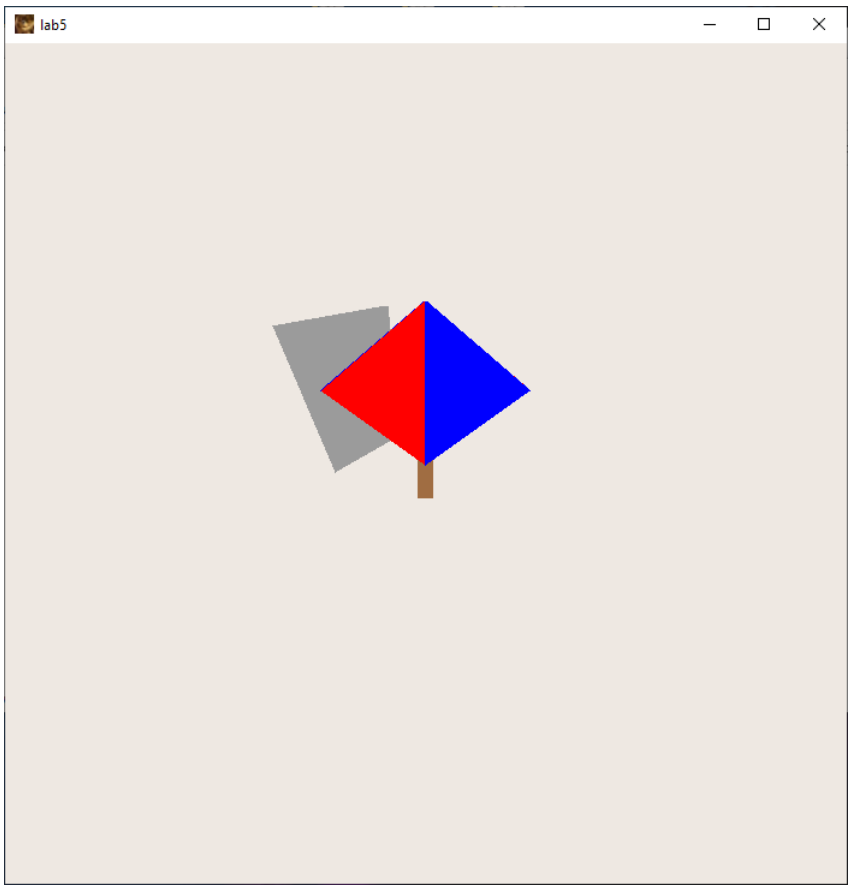

Рисунок 9. Пример работы программы

5. Вывод

В ходе выполнения лабораторной работы №5 «Исследование алгоритмов выявления видимости сложных сцен» были исследованы алгоритмы выявления видимости сложных сцен, построена специальная 3D-модель, позволяющая наглядно показать работоспособность алгоритма на основе Z-буфера, реализована сцена с возможностью поворачивать камеру и отслеживать изменения в раскраске разных объектов при изменении камеры. Таким образом и были исследованы алгоритмы выявления видимости сложных сцен.

6. Список использованных источников

1. Онлайн-курс «Компьютерная графика» в LMS Moodle [сайт]. URL: https://vec.etu.ru/moodle/course/view.php?id=8194.

2. Растровая и векторная графика [сайт]. URL: http://compgraph.tpu.ru/zbuffer.htm.