

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)**



ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа №4
по дисциплине: Компьютерная графика
тема: «Аффинные преобразования в пространстве»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили:
ст. пр. Осипов Олег Васильевич

Белгород 2024 г.

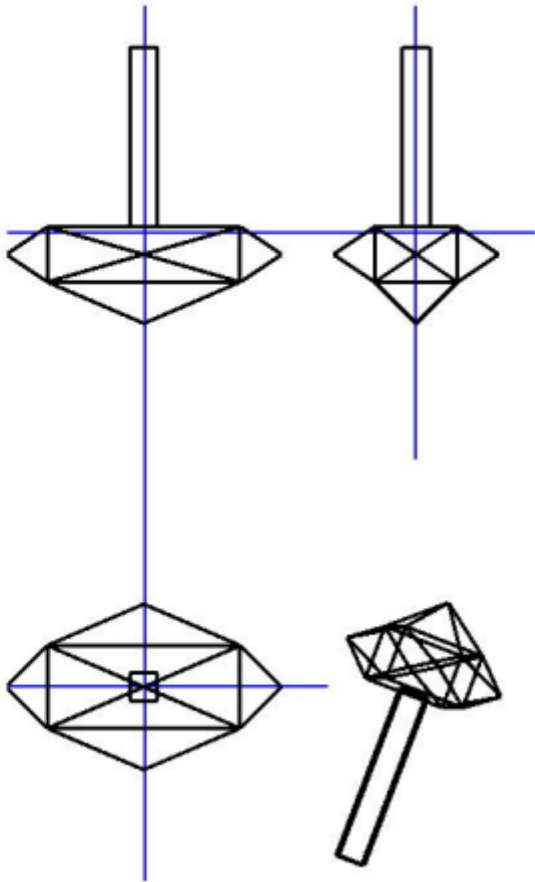
Лабораторная работа №2
Аффинные преобразования на плоскости
Вариант 8

Цель работы: получение навыков использования аффинных преобразований в пространстве и создание графического приложения с использованием GDI в среде Visual Studio для визуализации простейших трёхмерных объектов.

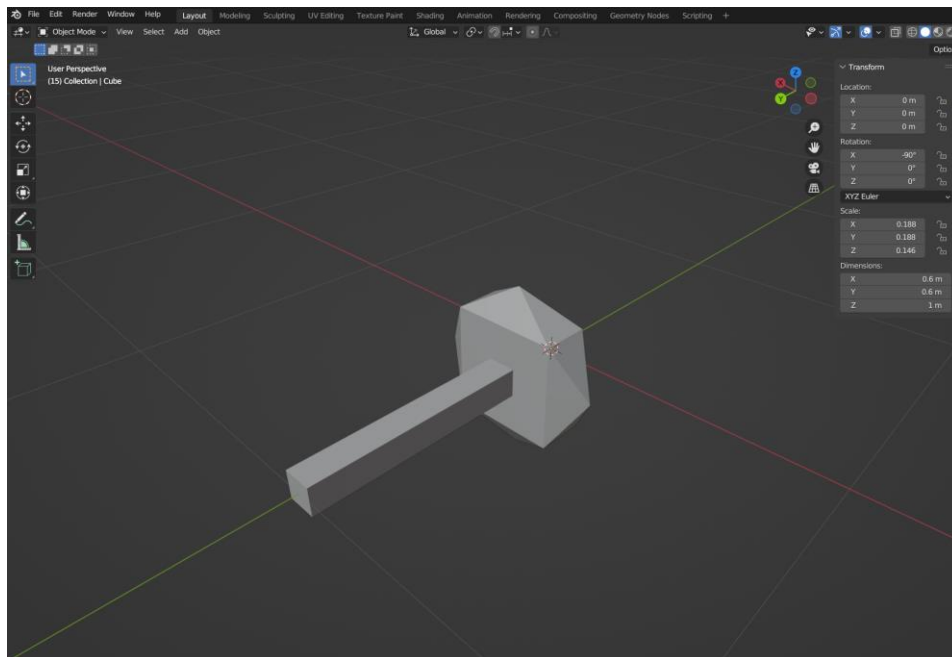
Задания для выполнения к работе:

1. Разработать алгоритм и составить программу для построения на экране трёхмерных изображений в соответствии с номером варианта. В качестве исходных данных взять указанные в таблице №1

Задание:

8		Изменять угол пирамид при движении колесика мыши
---	--	--

Для начала необходимо смоделировать молот. Сделать это можно при помощи Blender



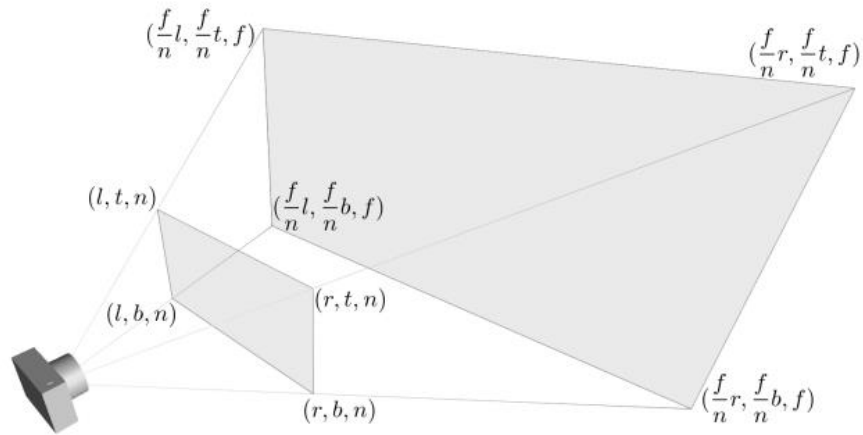
Перенесём точки в программу, предварительно триангулировав (разбив на треугольники) фигуру, однако шипы молота будет формировать иначе. Введём некий offset – отступ, острота шипа. Будем выдвигать в зависимости от этого offset шип посредством изменения позиции точки, таким образом сможем изменять угол пирамид.

Для формирования изображения будем каждый вектор-точку умножать на матрицы аффинных преобразований размера, вращения и трансформации. После чего будем умножать на матрицу проекции и приведения экранной системы координат к правосторонней системе координат.

Ортогографическая проекция – одна из самых простых проекций. Наши точки почти готовы для отображения, только нам необходимо сделать так, чтобы координата Z была от -1 до 1 для формирования буфера глубины. Обозначим границы, в которых будет видно экранное изображение плоскостями сечения x_1 – левая, x_2 – правая, y_1 – нижняя, y_2 – верхняя, $-z_1$ – ближняя плоскость сечения и $-z_2$ – дальняя. Эти плоскости сечения формируют параллелепипед, который будет виден пользователю, а от границы до границы точки будут принимать значения от -1 до 1. Матрица будет иметь вид

$$\begin{pmatrix} \frac{2}{x_2 - x_1} & 0 & 0 & -\frac{x_2 + x_1}{x_2 - x_1} \\ 0 & \frac{2}{y_2 - y_1} & 0 & -\frac{y_2 + y_1}{y_2 - y_1} \\ 0 & 0 & \frac{-2}{z_2 - z_1} & -\frac{z_2 + z_1}{z_2 - z_1} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Перспективная проекция, в отличие от ортогографической проекции, формирует усечённую пирамиду. Дальняя граница Z должна расширяться, а точки сходиться к наблюдателю. Аналогично введём плоскости отсечения



Составим для x, y уравнения прямых и нормализуем их:

$$X = \frac{z(r + l) + 2xn}{-z(r - l)}$$

$$Y = \frac{z(t + b) + 2yn}{-z(t - b)}$$

Координата Z формируется немного другим образом:

$$Z = \frac{z(f + n) + 2nz}{z(f - n)}$$

После чего можем составить матрицу

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Её неудобство заключается в том, что w не всегда равна 1. Поэтому необходимо дополнительно разделить x, y, z на эту координату.

Аксометрическая проекция заключается в том, что необходимо повернуть систему так, чтобы соотношение сторон имело определённые ограничения. Рассмотрим тетрадр с точками $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$. В изометрической проекции все три отрезка от центра к точкам должны иметь одинаковый размер. В диметрической – хотя бы две стороны. В триметрической ограничений нет. Триметрическая проекция самая простая – в ней нет ограничений, и мы можем повернуть модель на любой угол по y и x . В диметрической проекции любые две стороны должны иметь одинаковое соотношение. Пусть у нас есть матрица вращения по x и y . Избавимся заранее от координаты z , потому что она не будет использоваться в дальнейшем.

$$\begin{bmatrix} \cos \phi & \sin \phi \sin \theta & 0 & 0 \\ 0 & \cos \theta & 0 & 0 \\ \sin \phi & -\cos \phi \sin \theta & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Теперь возьмём векторы нашего тетраэдра и умножим их на матрицу трансформации.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} \cos \phi & \sin \phi \sin \theta & 0 & 0 \\ 0 & \cos \theta & 0 & 0 \\ \sin \phi & -\cos \phi \sin \theta & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

В итоге получим матрицу:

$$\begin{bmatrix} \cos \phi & \sin \phi \sin \theta & 0 & 1 \\ 0 & \cos \theta & 0 & 1 \\ \sin \phi & -\cos \phi \sin \theta & 0 & 1 \end{bmatrix}$$

Обозначим элементы полученной матрицы как:

$$\begin{bmatrix} x_x^* & y_x^* & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & 1 \\ x_y^* & y_y^* & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & 1 \\ x_z^* & y_z^* & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & 1 \end{bmatrix}$$

x_x^*, y_x^* – новые координаты точки (1, 0, 0).

x_y^*, y_y^* – новые координаты точки (0, 1, 0).

x_z^*, y_z^* – новые координаты точки (0, 0, 1).

Значит, можем вычислить длины новых координат:

$$f_x = \sqrt{x_x^{*2} + y_x^{*2}},$$

$$f_y = \sqrt{x_y^{*2} + y_y^{*2}},$$

$$f_z = \sqrt{x_z^{*2} + y_z^{*2}}.$$

f – коэффициенты искажения точек. В диаметрической проекции хотя бы два таких коэффициента должны быть равны. Допустим, $f_x = f_y$.

$$\begin{cases} \cos^2 \phi + \sin^2 \phi \sin^2 \theta = \cos^2 \theta \\ f_z^2 = \sin^2 \phi + \cos^2 \phi \cdot \sin^2 \theta \end{cases}$$

$$\begin{cases} \sin^2 \phi = \frac{\sin^2 \theta}{(1 - \sin^2 \theta)} \\ f_z^2 = \frac{\sin^2 \theta}{(1 - \sin^2 \theta)} + \left(1 - \frac{\sin^2 \theta}{(1 - \sin^2 \theta)}\right) \cdot \sin^2 \theta \end{cases}$$

$$2 \sin^4 \theta - \sin^2 \theta (f_z^2 + 2) + f_z^2 = 0$$

Пусть $V = \sin^2 \theta$

$$2V^2 - (f_z^2 + 2)V + f_z^2 = 0$$

$$V_1 = \frac{(f_z^2 + 2) + f_z^2 - 2}{4} = \frac{f_z^2}{2}$$

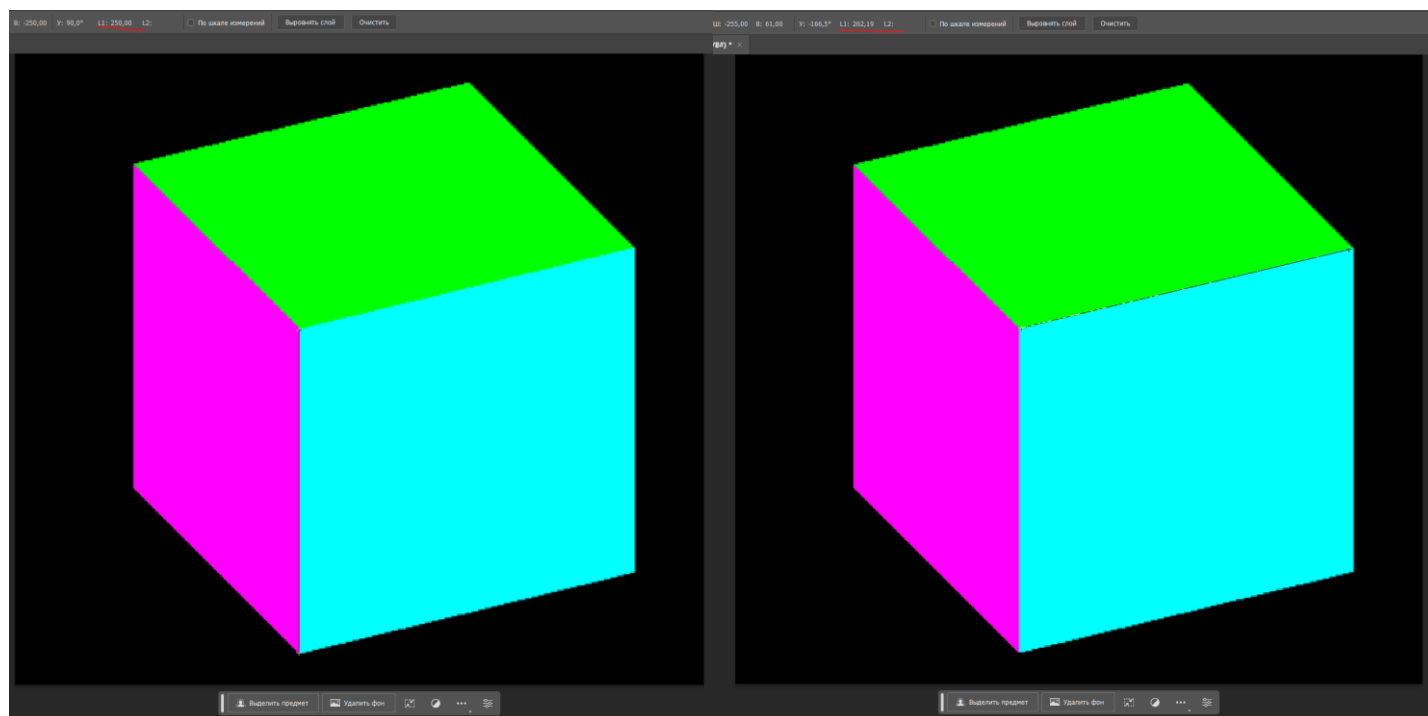
$$V_2 = 1, \text{ не подходит.}$$

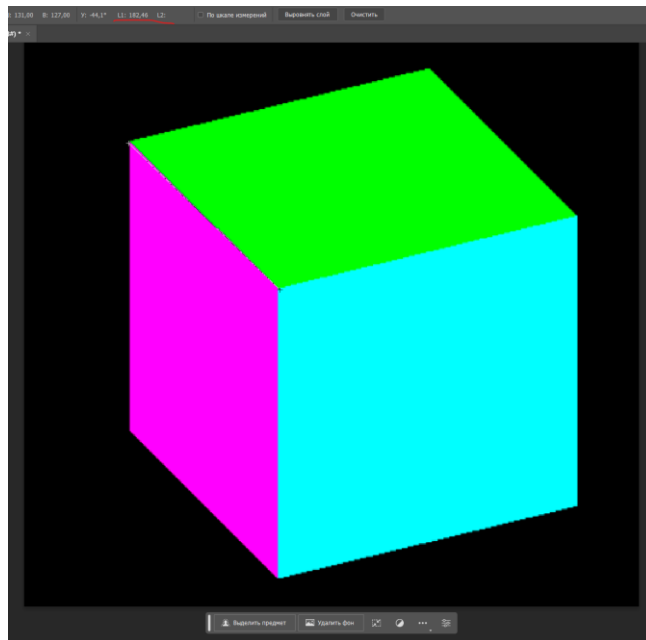
$$\sin^2 \theta = \frac{f_z^2}{2} \Rightarrow \theta = \arcsin\left(\sqrt{\frac{f_z^2}{2}}\right); \phi = \arcsin\left(\sqrt{\frac{1-f_z^2}{2}}\right)$$

Ограничения: $f_z \leq 1$.

Можем использовать любой f_z в любом промежутке. Пусть, например, $f_z = 3/8$. Тогда вращение $\phi = 0,5152212$; $\theta = 0,45779986$ в радианах.

Проверим вычисления в программе:





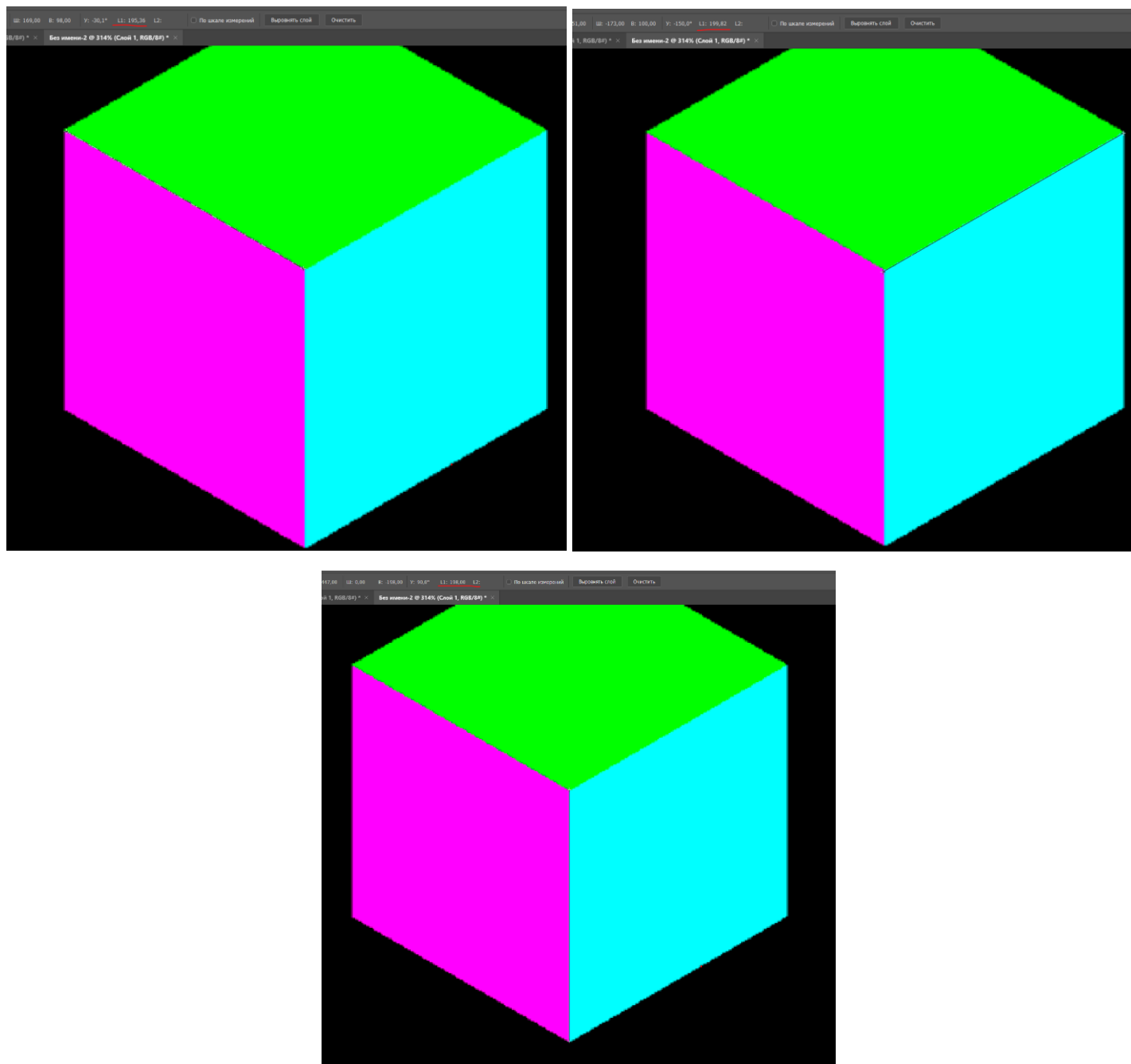
Размеры в пределах погрешности совпали, получили диметрическую проекцию. Чтобы получить изометрическую проекцию, необходимо, чтобы $f_z = f_y = f_x$. Приравняем f_x к f_y , а f_y к f_z и получим

и второе и третье, получим:

$$\sin^2 \phi = \frac{\sin^2 \theta}{1 - \sin^2 \theta} \quad \sin^2 \theta = \frac{1 - \sin^2 \phi}{1 + \sin^2 \phi}$$

$$\begin{aligned} \sin^2 \theta &= \frac{1 - \sin^2 \phi}{1 + \sin^2 \phi} \\ \sin^2 \theta &= \frac{1 - \sin^2 \phi - \sin^2 \phi}{1 - \sin^2 \phi + \sin^2 \phi} \\ \sin^2 \theta &= \frac{1 - 2\sin^2 \phi}{1} \\ \sin^2 \theta &= 1 - 2\sin^2 \phi \\ \sin^2 \theta &= \frac{1}{3}; \quad \theta = \arcsin\left(\pm\sqrt{\frac{1}{3}}\right) \approx 0,6154 \\ \sin^2 \phi &= \frac{\frac{1}{3}}{1 - \frac{1}{3}} \\ \frac{2}{3} \sin^2 \phi &= \frac{1}{3}; \quad \sin^2 \phi = \frac{1}{2}; \quad \phi = \pm 45^\circ = \pm \frac{\pi}{4} \end{aligned}$$

Проверим полученные значения в программе:



Длины совпали в пределах погрешности. Вычисления верны.

Color.h

```
#pragma once

#include <math.h>

// Структура для задания цвета
struct COLOR
{
    unsigned char RED;           // Компонента красного цвета
    unsigned char GREEN;        // Компонента зелёного цвета
    unsigned char BLUE;         // Компонента синего цвета
    unsigned char ALPHA;        // Прозрачность (альфа канал)

    COLOR(int red, int green, int blue, int alpha = 255)
        : RED(red), GREEN(green), BLUE(blue), ALPHA(alpha) { }
};

// Пиксель в буфере кадра
struct PIXEL
{
    unsigned char RED;           // Компонента красного цвета
    unsigned char GREEN;        // Компонента зелёного цвета
    unsigned char BLUE;         // Компонента синего цвета
    float Z;                     // Глубина пикселя
    PIXEL() : RED(0), GREEN(0), BLUE(0), Z(INFINITY) { }
};
```

Frame.h

```
#ifndef FRAME_H
#define FRAME_H

#include <math.h>
#include "Matrix.h"
#include "Color.h"
#include "Perspective.h"

#define INSIDE 0 // 0000
#define LEFT 1 // 000
#define RIGHT 2 // 0010
#define BOTTOM 4 // 0100
#define TOP 8 // 1000
#define SHOW_POLYGON 0b01
#define SHOW_GRID 0b10

template<typename TYPE> void swap(TYPE& a, TYPE& b)
{
    TYPE t = a;
    a = b;
    b = t;
}

// Буфер кадра
class Frame
{
    // Указатель на массив пикселей
    // Буфер кадра будет представлять собой матрицу, которая располагается в памяти в виде
    // непрерывного блока
    PIXEL* pixels;

public:
    // Размеры буфера кадра
```

```

int width, height;
Matrix transform;
Perspective perspective;

Frame(int _width, int _height, Perspective _perspective = Perspective::FRUSTUM, Matrix
_transform = Matrix()) :
    width(_width), height(_height), perspective(_perspective), transform(_transform)
{
    int size = width * height;

    // Создание буфера кадра в виде непрерывной матрицы пикселей
    pixels = new PIXEL[size];
}

// Задаёт цвет color пикселю с координатами (x, y)
void SetPixel(int x, int y, COLOR color)
{
    PIXEL* pixel = pixels + (size_t)y * width + x; // Находим нужный пиксель в
матрице пикселей
    pixel->RED = color.RED;
    pixel->GREEN = color.GREEN;
    pixel->BLUE = color.BLUE;
}

// Возвращает цвет пикселя с координатами (x, y)
COLOR GetPixel(int x, int y)
{
    PIXEL* pixel = pixels + (size_t)y * width + x; // Находим нужный пиксель в
матрице пикселей
    return COLOR(pixel->RED, pixel->GREEN, pixel->BLUE);
}

int getCohenSutherland(int x, int y) {
    int y_max = height - 1;
    int y_min = 0;
    int x_max = width - 1;
    int x_min = 0;
    // initialized as being inside
    int code = INSIDE;

    if (x < x_min) // to the left of rectangle
        code |= LEFT;
    else if (x > x_max) // to the right of rectangle
        code |= RIGHT;
    if (y < y_min) // below the rectangle
        code |= BOTTOM;
    else if (y > y_max) // above the rectangle
        code |= TOP;

    return code;
}

void cohenSutherlandClip(int x1, int y1,
    int x2, int y2, bool *accepted,
    int *resX1, int *resY1, int* resX2, int* resY2)
{
    int y_max = height - 1;
    int y_min = 0;
    int x_max = width - 1;
    int x_min = 0;

    int code1 = getCohenSutherland(x1, y1);
    int code2 = getCohenSutherland(x2, y2);

    bool accept = false;

```

```

while (true) {
    if ((code1 == 0) && (code2 == 0)) {
        accept = true;
        break;
    }
    else if (code1 & code2) {
        break;
    }
    else {
        int code_out;
        int x, y;

        if (code1 != 0)
            code_out = code1;
        else
            code_out = code2;

        if (code_out & TOP) {
            x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1);
            y = y_max;
        }
        else if (code_out & BOTTOM) {
            x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1);
            y = y_min;
        }
        else if (code_out & RIGHT) {
            y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1);
            x = x_max;
        }
        else if (code_out & LEFT) {
            y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1);
            x = x_min;
        }

        if (code_out == code1) {
            x1 = x;
            y1 = y;
            code1 = getCohenSutherland(x1, y1);
        }
        else {
            x2 = x;
            y2 = y;
            code2 = getCohenSutherland(x2, y2);
        }
    }
}

if (accept) {
    *accepted = true;
    *resX1 = x1;
    *resY1 = y1;
    *resX2 = x2;
    *resY2 = y2;
}
else {
    *accepted = false;
}
}

// Рисование отрезка
void DrawLine(int x1, int y1, int x2, int y2, COLOR color)
{
    bool shouldDraw = false;
    int resX1, resY1, resX2, resY2;
    cohenSutherlandClip(x1, y1, x2, y2, &shouldDraw, &resX1, &resY1, &resX2, &resY2);
    if (!shouldDraw) return;
    x1 = resX1;
    y1 = resY1;
}

```

```

x2 = resX2;
y2 = resY2;

PIXEL* pixel;
int dy = y2 - y1, dx = x2 - x1;
if (dx == 0 && dy == 0)
{
    pixel = pixels + (size_t)y1 * width + x1;
    pixel->RED = color.RED;
    pixel->GREEN = color.GREEN;
    pixel->BLUE = color.BLUE;
    pixel->Z = -1;
    return;
}

if (abs(dx) > abs(dy))
{
    if (x2 < x1)
    {
        // Обмен местами точек (x1, y1) и (x2, y2)
        swap(x1, x2);
        swap(y1, y2);
        dx = -dx; dy = -dy;
    }

    int y = y1;
    int sign_factor = dy < 0 ? 1 : -1;
    int sumd = -2 * (y - y1) * dx + sign_factor * dx;
    for (int x = x1; x <= x2; x++)
    {
        if (sign_factor * sumd < 0) {
            y -= sign_factor;
            sumd += sign_factor * dx;
        }

        sumd += dy;

        pixel = pixels + (size_t)y * width + x;
        pixel->RED = color.RED;
        pixel->GREEN = color.GREEN;
        pixel->BLUE = color.BLUE;
        pixel->Z = -1;
    }
}
else
{
    if (y2 < y1)
    {
        // Обмен местами точек (x1, y1) и (x2, y2)
        swap(x1, x2);
        swap(y1, y2);
        dx = -dx; dy = -dy;
    }

    int x = x1;
    int sign_factor = dx > 0 ? 1 : -1;
    int sumd = 2 * (x - x1) * dy + sign_factor * dy;
    for (int y = y1; y <= y2; y++)
    {
        if (sign_factor * sumd < 0) {
            x += sign_factor;
            sumd += sign_factor * dy;
        }

        sumd -= dx;

        pixel = pixels + (size_t)y * width + x;
    }
}

```

```

        pixel->RED = color.RED;
        pixel->GREEN = color.GREEN;
        pixel->BLUE = color.BLUE;
        pixel->Z = -1.1;
    }
}

void Triangle(float x0, float y0, float z0, float x1, float y1, float z1, float x2, float
y2, float z2, COLOR color, char drawMode = SHOW_GRID | SHOW_POLYGON, COLOR gridColor = {255, 255,
255, 0})
{
    if (!drawMode) return;

    if (drawMode & SHOW_GRID) {
        DrawLine(x0, y0, x1, y1, gridColor);
        DrawLine(x1, y1, x2, y2, gridColor);
        DrawLine(x0, y0, x2, y2, gridColor);
    }

    if (!(drawMode & SHOW_POLYGON))
        return;

    // Отсортируем точки таким образом, чтобы выполнилось условие: y0 < y1 < y2
    if (y1 < y0)
    {
        swap(x1, x0);
        swap(y1, y0);
        swap(z1, z0);
    }
    if (y2 < y1)
    {
        swap(x2, x1);
        swap(y2, y1);
        swap(z2, z1);
    }
    if (y1 < y0)
    {
        swap(x1, x0);
        swap(y1, y0);
        swap(z1, z0);
    }

    // Определяем номера строк пикселей, в которых располагаются точки треугольника
    int Y0 = (int) (y0 + 0.5f);
    int Y1 = (int) (y1 + 0.5f);
    int Y2 = (int) (y2 + 0.5f);

    // Отсечение невидимой части треугольника
    if (Y0 < 0) Y0 = 0;
    else if (Y0 > height) Y0 = height;

    if (Y1 < 0) Y1 = 0;
    else if (Y1 > height) Y1 = height;

    if (Y2 < 0) Y2 = 0;
    else if (Y2 > height) Y2 = height;

    float S = (y1 - y2) * (x0 - x2) + (x2 - x1) * (y0 - y2); // Площадь треугольника

    // Рисование верхней части треугольника
    for (int Y = Y0; Y < Y1; Y++)
    {
        float y = Y + 0.5; // Координата y середины пикселя

        // Вычисление координат граничных пикселей
        int X0 = (int) ((y - y0) / (y1 - y0) * (x1 - x0) + x0 + 0.5f);

```

```

int X1 = (int) ((y - y0) / (y2 - y0) * (x2 - x0) + x0 + 0.5f);

if (X0 > X1) swap(X0, X1); // Сортировка пикселей
if (X0 < 0) X0 = 0; // Отсечение невидимых пикселей в строке y
if (X1 > width) X1 = width;

for (int X = X0; X < X1; X++)
{
    double x = X + 0.5; // Координата x середины пикселя

    // Середина пикселя имеет координаты (x, y)
    // Вычислим барицентрические координаты этого пикселя
    float h0 = ((y1 - y2) * (x - x2) + (x2 - x1) * (y - y2)) / S;
    float h1 = ((y2 - y0) * (x - x2) + (x0 - x2) * (y - y2)) / S;
    float h2 = ((y0 - y1) * (x - x1) + (x1 - x0) * (y - y1)) / S;

    float Z = h0 * z0 + h1 * z1 + h2 * z2; // Глубина пикселя

    // Определение глубины точки в экранной системе координат

    PIXEL* pixel = pixels + (size_t)Y * width + X; // Вычислим адрес
    пикселя (Y, X) в матрице пикселей pixels
    // pixel->Z - глубина пикселя, которая уже записана в буфер
    кадра
    // Если текущий пиксель находится ближе того пикселя, который
    уже записан в буфере кадра
    if (Z > -1 && Z < 1 && Z < pixel->Z)
    { // то обновляем пиксель в буфере кадра
        //float zmax = 0.8, zmin = 0.1;
        //color = COLOR(255 - (Z - zmin) / (zmax - zmin) * 255,
100, 100);

        //color = COLOR((2.5 - Z) / 4 * 255, 0, 0);
        pixel->RED = color.RED;
        pixel->GREEN = color.GREEN;
        pixel->BLUE = color.BLUE;
        pixel->Z = Z;
    }
}

// Рисование нижней части треугольника
for (int Y = Y1; Y < Y2; Y++)
{
    float y = Y + 0.5; // Координата y середины пикселя

    // Вычисление координат граничных пикселей
    int X0 = (int)((y - y1) / (y2 - y1) * (x2 - x1) + x1 + 0.5f);
    int X1 = (int)((y - y0) / (y2 - y0) * (x2 - x0) + x0 + 0.5f);

    if (X0 > X1) swap(X0, X1); // Сортировка пикселей
    if (X0 < 0) X0 = 0; // Отсечение невидимых пикселей в строке y
    if (X1 > width) X1 = width;

    for (int X = X0; X < X1; X++)
    {
        double x = X + 0.5; // Координата x середины пикселя

        // Середина пикселя имеет координаты (x, y)
        // Вычислим барицентрические координаты этого пикселя
        float h0 = ((y1 - y2) * (x - x2) + (x2 - x1) * (y - y2)) / S;
        float h1 = ((y2 - y0) * (x - x2) + (x0 - x2) * (y - y2)) / S;
        float h2 = ((y0 - y1) * (x - x1) + (x1 - x0) * (y - y1)) / S;

        float Z = h0 * z0 + h1 * z1 + h2 * z2; // Глубина пикселя

        // Определение глубины точки в экранной системе координат

```

```

// Если текущий пиксель находится ближе того пикселя, который
уже записан в буфере кадра
PIXEL* pixel = pixels + (size_t)Y * width + X;
if (Z > -1 && Z < 1 && Z < pixel->Z)
{
    //float zmax = 0.8, zmin = 0.1;
    //color = COLOR(255 - (Z - zmin) / (zmax - zmin) * 255,
100, 100);

    //color = COLOR((2.5 - Z) / 4 * 255, 0, 0);
    pixel->RED = color.RED;
    pixel->GREEN = color.GREEN;
    pixel->BLUE = color.BLUE;
    pixel->Z = Z;
}
}
}

void Quad(float x0, float y0, float z0, float x1, float y1, float z1, float x2, float y2,
float z2, float x3, float y3, float z3, COLOR color)
{
    Triangle(x0, y0, z0, x1, y1, z1, x2, y2, z2, color);
    Triangle(x2, y2, z2, x3, y3, z3, x0, y0, z0, color);
}

~Frame(void)
{
    delete []pixels;
}

};

#endif // FRAME_H

```

Matrices.h

```

#ifndef MATRIX_H
#define MATRIX_H

class Matrix
{
    float M[4][4];
public:

    // По умолчанию матрица инициализируется как единичная
    Matrix() : M {
        { 1, 0, 0, 0 },
        { 0, 1, 0, 0 },
        { 0, 0, 1, 0 },
        { 0, 0, 0, 1 } }
    {}

    // Конструктор, который инициализирует матрицу M поэлементно значениями аргументов
    Matrix( float A00, float A01, float A02, float A03,
            float A10, float A11, float A12, float A13,
            float A20, float A21, float A22, float A23,
            float A30, float A31, float A32, float A33) : M {
        { A00, A01, A02, A03 },
        { A10, A11, A12, A13 },
        { A20, A21, A22, A23 },
        { A30, A31, A32, A33 } }
    {}
}

```

```

// Конструктор, который инициализирует матрицу M двумерным массивом A
Matrix(const float A[4][4]) : M {
    { A[0][0], A[0][1], A[0][2], A[0][3] },
    { A[1][0], A[1][1], A[1][2], A[1][3] },
    { A[2][0], A[2][1], A[2][2], A[2][3] },
    { A[3][0], A[3][1], A[3][2], A[3][3] } }
{}

Matrix operator * (const Matrix& A) const
{
    Matrix R;
    char i, j;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            R.M[i][j] = M[i][0] * A.M[0][j] + M[i][1] * A.M[1][j] + M[i][2] * A.M[2][j] +
M[i][3] * A.M[3][j];
    return R;
}

static Matrix RotationX(float Angle)
{
    Matrix R;
    float cosA = cos(Angle);
    float sinA = sin(Angle);
    R.M[0][0] = 1; R.M[0][1] = 0;    R.M[0][2] = 0;    R.M[0][3] = 0;
    R.M[1][0] = 0; R.M[1][1] = cosA; R.M[1][2] = sinA; R.M[1][3] = 0;
    R.M[2][0] = 0; R.M[2][1] = -sinA; R.M[2][2] = cosA; R.M[2][3] = 0;
    R.M[3][0] = 0; R.M[3][1] = 0;    R.M[3][2] = 0;    R.M[3][3] = 1;
    return R;
}

static Matrix RotationY(float Angle)
{
    Matrix R;
    float cosA = cos(Angle);
    float sinA = sin(Angle);
    R.M[0][0] = cosA; R.M[0][1] = 0; R.M[0][2] = -sinA; R.M[0][3] = 0;
    R.M[1][0] = 0;    R.M[1][1] = 1; R.M[1][2] = 0;    R.M[1][3] = 0;
    R.M[2][0] = sinA; R.M[2][1] = 0; R.M[2][2] = cosA; R.M[2][3] = 0;
    R.M[3][0] = 0;    R.M[3][1] = 0; R.M[3][2] = 0;    R.M[3][3] = 1;
    return R;
}

static Matrix RotationZ(float Angle)
{
    Matrix R;
    float cosA = cos(Angle);
    float sinA = sin(Angle);
    R.M[0][0] = cosA; R.M[0][1] = sinA; R.M[0][2] = 0; R.M[0][3] = 0;
    R.M[1][0] = -sinA; R.M[1][1] = cosA; R.M[1][2] = 0; R.M[1][3] = 0;
    R.M[2][0] = 0;    R.M[2][1] = 0;    R.M[2][2] = 1; R.M[2][3] = 0;
    R.M[3][0] = 0;    R.M[3][1] = 0;    R.M[3][2] = 0; R.M[3][3] = 1;
    return R;
}

static Matrix Translation(float dx, float dy, float dz)
{
    Matrix R;
    R.M[0][0] = 1; R.M[0][1] = 0; R.M[0][2] = 0; R.M[0][3] = 0;
    R.M[1][0] = 0; R.M[1][1] = 1; R.M[1][2] = 0; R.M[1][3] = 0;
    R.M[2][0] = 0; R.M[2][1] = 0; R.M[2][2] = 1; R.M[2][3] = 0;
    R.M[3][0] = dx; R.M[3][1] = dy; R.M[3][2] = dz; R.M[3][3] = 1;
    return R;
}

static Matrix Scale(float kx, float ky, float kz)
{

```



```

Matrix R;
R.M[0][0] = kx; R.M[0][1] = 0; R.M[0][2] = 0; R.M[0][3] = 0;
R.M[1][0] = 0; R.M[1][1] = ky; R.M[1][2] = 0; R.M[1][3] = 0;
R.M[2][0] = 0; R.M[2][1] = 0; R.M[2][2] = kz; R.M[2][3] = 0;
R.M[3][0] = 0; R.M[3][1] = 0; R.M[3][2] = 0; R.M[3][3] = 1;
return R;
}

// Матрица для преобразования мировых координат в экранные координаты области (порта) вывода
// (X0, Y0) - экранные координаты левого нижнего угла области (порта) вывода
// width - ширина порта вывода (в пикселях)
// height - высота порта вывода (в пикселях)
// nearPlane - координата z ближней плоскости
// farPlane - координата z дальней плоскости
static Matrix Viewport(float X0, float Y0, float width, float height, float nearPlane = -1.0f,
float farPlane = 1.0f)
{
    return Matrix(
        width / 2,          0,          0,          0,
        0,          height / 2,          0,          0,
        0,          0,          (farPlane - nearPlane) / 2,          0,
        X0 + width / 2,    Y0 + height / 2,    (farPlane + nearPlane) / 2,          1);
}

// Матрица перспективного проектирования
static Matrix Frustum(float left, float right, float bottom, float top, float nearPlane, float
farPlane)
{
    float width = right - left;
    float invheight = top - bottom;
    float clip = farPlane - nearPlane;
    Matrix R;
    R.M[0][0] = 2.0f * nearPlane / width;
    R.M[1][0] = 0.0f;
    R.M[2][0] = (left + right) / width;
    R.M[3][0] = 0.0f;
    R.M[0][1] = 0.0f;
    R.M[1][1] = 2.0f * nearPlane / invheight;
    R.M[2][1] = (top + bottom) / invheight;
    R.M[3][1] = 0.0f;
    R.M[0][2] = 0.0f;
    R.M[1][2] = 0.0f;
    R.M[2][2] = -(nearPlane + farPlane) / clip;
    R.M[3][2] = -2.0f * nearPlane * farPlane / clip;
    R.M[0][3] = 0.0f;
    R.M[1][3] = 0.0f;
    R.M[2][3] = -1.0f;
    R.M[3][3] = 0.0f;
    return R;
}

// Матрица ортогографического проектирования
static Matrix Ortho(float left, float right, float bottom, float top, float nearPlane, float
farPlane)
{
    float width = right - left;
    float invheight = top - bottom;
    float clip = farPlane - nearPlane;
    Matrix R;
    R.M[0][0] = 2.0f / width;
    R.M[1][0] = 0.0f;
    R.M[2][0] = 0.0f;
    R.M[3][0] = -(left + right) / width;
    R.M[0][1] = 0.0f;

```

```

    R.M[1][1] = 2.0f / invheight;
    R.M[2][1] = 0.0f;
    R.M[3][1] = -(top + bottom) / invheight;
    R.M[0][2] = 0.0f;
    R.M[1][2] = 0.0f;
    R.M[2][2] = -2.0f / clip;
    R.M[3][2] = -(nearPlane + farPlane) / clip;
    R.M[0][3] = 0.0f;
    R.M[1][3] = 0.0f;
    R.M[2][3] = 0.0f;
    R.M[3][3] = 1.0f;
    return R;
}

static Matrix Axonometric(float xAngle, float yAngle, float left, float right, float bottom,
float top, float nearPlane, float farPlane) {
    return
        Matrix::RotationY(-3.14 / 2 + yAngle) *
        Matrix::RotationX(xAngle) *
        Matrix::Ortho(left, right, bottom, top, nearPlane, farPlane);
}

friend class Vector;
};

#endif MATRIX_H

```

Model.h

```

#pragma once

#include <vector>
#include <tuple>
#include "Vector.h"

// Возвращает точки объекта
const std::vector<Vector> get_points(float scale_factor = 1.) {
    std::vector<Vector> result = {
        // Вершина рукоятки
        {-0.05, 1, 0.05}, // 0
        {0.05, 1, 0.05}, // 1
        {-0.05, 1, -0.05}, // 2
        {0.05, 1, -0.05}, // 3

        // Нижняя часть рукоятки
        {-0.05, 0.25, 0.05}, // 4
        {0.05, 0.25, 0.05}, // 5
        {-0.05, 0.25, -0.05}, // 6
        {0.05, 0.25, -0.05}, // 7

        // Нижняя часть молота
        {-0.4, 0.25, 0.2}, // 8
        {0.4, 0.25, 0.2}, // 9
        {-0.4, 0.25, -0.2}, // 10
        {0.4, 0.25, -0.2}, // 11

        // Боковые шипы молота
        {-(0.4f + scale_factor), 0.17, 0}, // 12
        {(0.4f + scale_factor), 0.17, 0}, // 13
        {0, 0.17, -(0.2f + scale_factor)}, // 14
        {0, 0.17, (0.2f + scale_factor)}, // 15

        // Соединители боковых шипов
    };
}

```

```

        {-0.4, 0.09, 0.2}, // 16
        {0.4, 0.09, 0.2}, // 17
        {-0.4, 0.09, -0.2}, // 18
        {0.4, 0.09, -0.2}, // 19

        // Нижний шип
        {0, 0.09f - scale_factor, 0} // 20
    };

    return result;
}

// Задаёт индексы точек для формирования полигонов
std::vector<std::tuple<int, int, int>> polygons = {
    // Вершина рукоятки
    {0, 1, 2},
    {3, 1, 2},

    // Стенки рукоятки
    {0, 1, 5},
    {1, 5, 7},
    {1, 3, 7},
    {3, 7, 6},
    {3, 2, 6},
    {2, 6, 4},
    {2, 0, 4},
    {0, 4, 5},

    // Нижняя часть рукоятки
    {5, 4, 9},
    {4, 9, 8},
    {6, 4, 8},
    {6, 8, 10},
    {6, 7, 10},
    {7, 10, 11},
    {7, 11, 5},
    {11, 5, 9},

    // Шипы молота
    {8, 10, 12},
    {8, 9, 15},
    {9, 11, 13},
    {11, 10, 14},
    {16, 18, 12},
    {16, 17, 15},
    {17, 19, 13},
    {19, 18, 14},

    {11, 19, 13},
    {11, 19, 14},

    {10, 18, 14},
    {10, 18, 12},

    {8, 16, 12},
    {8, 16, 15},

    {9, 17, 15},
    {9, 17, 13},

    // Нижний шип
    {19, 18, 20},
    {18, 16, 20},
    {16, 17, 20},
    {17, 19, 20}
};

```

```

std::vector<COLOR> materials = {
    {132, 39, 75},
    {126, 142, 215},
    {98, 82, 154},
    {142, 132, 211},
    {97, 103, 222},
    {70, 93, 36},
    {62, 122, 190},
    {83, 57, 98},
    {148, 167, 154},
    {111, 115, 224},
    {175, 218, 20},
    {168, 178, 84},
    {40, 168, 153},
    {132, 46, 214},
    {105, 215, 86},
    {109, 183, 95},
    {64, 121, 201},
    {141, 185, 103},
    {88, 116, 207},
    {42, 205, 195},
    {143, 190, 44},
    {218, 156, 52},
    {205, 68, 192},
    {61, 68, 142},
    {96, 129, 219},
    {196, 61, 81},
    {139, 69, 110},
    {205, 210, 134},
    {184, 158, 46},
    {225, 131, 143},
    {168, 156, 159},
    {143, 67, 152},
    {160, 210, 114},
    {125, 75, 154},
    {32, 191, 21},
    {73, 216, 93},
    {112, 113, 173},
    {146, 211, 170}
};

/*const std::vector<Vector> get_points(float scale_factor = 1.) {
    std::vector<Vector> result = {
        {0, 0, 0},
        {1, 0, 0},
        {1, 0, 1},
        {0, 0, 1},
        {0, 1, 0},
        {1, 1, 0},
        {1, 1, 1},
        {0, 1, 1},
    };

    return result;
};

std::vector<std::tuple<int, int, int>> polygons = {
    {0, 1, 2},
    {0, 3, 2},
    {4, 5, 6},
    {4, 7, 6},
    {0, 4, 3},
    {4, 7, 3},
    {0, 4, 1},
    {5, 4, 1},
    {5, 2, 1},
    {5, 2, 6},

```

```

        {2, 6, 3},
        {7, 6, 3},
    };

    std::vector<COLOR> materials = {
        {255, 0, 0},
        {255, 0, 0},
        {0, 255, 0},
        {0, 255, 0},
        {0, 0, 255},
        {0, 0, 255},
        {255, 255, 0},
        {255, 255, 0},
        {0, 255, 255},
        {0, 255, 255},
        {255, 0, 255},
        {255, 0, 255}
    };*/

```

Painter.h

```

#ifndef PAINTER_H
#define PAINTER_H

#include "Frame.h"
#include "Vector.h"
#include "Model.h"

// Время от начала запуска программы
float time = 0;
float scale = 0;
float x_offset = 0;
float y_offset = 0;
float z_offset = -4;
float x_rot = 0;
float y_rot = 0;
float z_rot = 0;
float fig_scale = 2.5;
int draw_mode = 1;
Perspective currentPerspective = static_cast<Perspective>(0);

// Тип проекции (перспективная или ортографическая)

class Painter
{
public:

    void Draw(Frame& frame)
    {

        float angle = time; // Угол поворота объекта

        auto A = get_points(scale);

        Matrix projection_matrix; // Матрица проектирования

        // Выбор матрицы проектирования
        if (frame.perspective == Perspective::ORTHO) //Ортографическое проектирование
        {
            projection_matrix = Matrix::Ortho(-2.0 * frame.width / frame.height, 2.0 *
frame.width / frame.height, -2.0, 2.0, 1, 140.0f);
        }
        else if (frame.perspective == Perspective::FRUSTUM) // Перспективное проектирование
        {
            projection_matrix = Matrix::Frustum(-0.5 * frame.width / frame.height, 0.5 *
frame.width / frame.height, -0.5, 0.5, 1, 140);
        }
    }
}

```

```

    else if (frame.perspective == Perspective::TRIMETRIC) {
        projection_matrix = Matrix::Axonometric(3.14 / 4., 3.14 / 6., -2.0 * frame.width /
frame.height, 2.0 * frame.width / frame.height, -2.0, 2.0, 1, 140.0f);
    }
    else if (frame.perspective == Perspective::DIMETRIC) {
        projection_matrix = Matrix::Axonometric(0.5152212, 0.45779986, -2.0 * frame.width /
frame.height, 2.0 * frame.width / frame.height, -2.0, 2.0, 1, 140.0f);
    }
    else if (frame.perspective == Perspective::ISOMETRIC) {
        projection_matrix = Matrix::Axonometric(0.615479708, 3.14 / 4., -2.0 * frame.width /
frame.height, 2.0 * frame.width / frame.height, -2.0, 2.0, 1, 140.0f);
    }

    Matrix proj_viewport = projection_matrix * // Проектирование
        Matrix::Viewport(0, 0, frame.width, frame.height);

    Matrix general_matrix =
        frame.transform * // Перенос куба против оси z
        proj_viewport; // Преобразование нормализованных координат в оконные

    std::vector<Vector> B(A.size());

    for (int i = 0; i < A.size(); i++)
    {
        B[i] = A[i] * general_matrix;

        // Преобразование однородных координат в обычные
        B[i].x /= B[i].w;
        B[i].y /= B[i].w;
        B[i].z /= B[i].w;
        B[i].w = 1;
    }

    for (int i = 0; i < polygons.size(); i++) {
        auto pointA = B[std::get<0>(polygons[i])];
        auto pointB = B[std::get<1>(polygons[i])];
        auto pointC = B[std::get<2>(polygons[i])];
        auto polygonColor = materials[i];

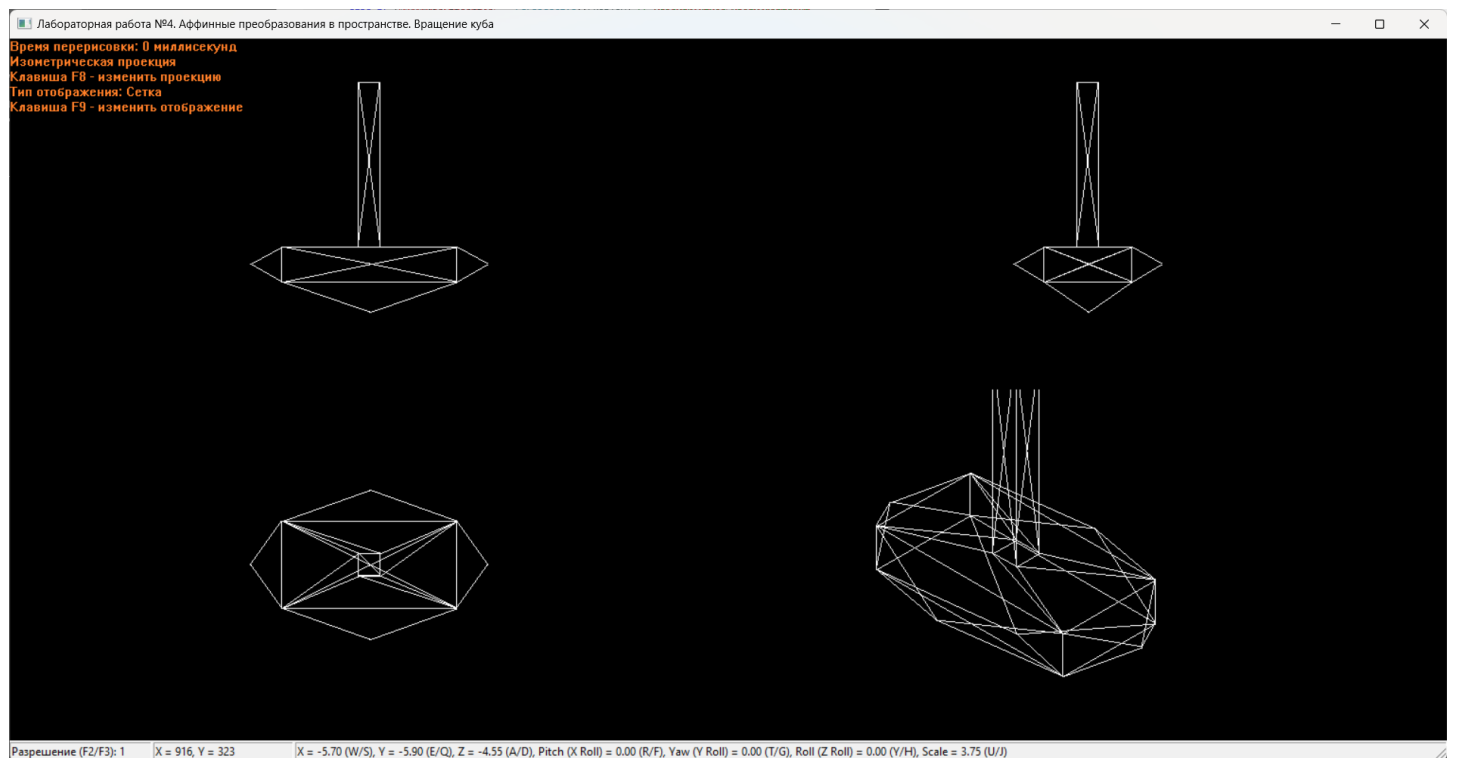
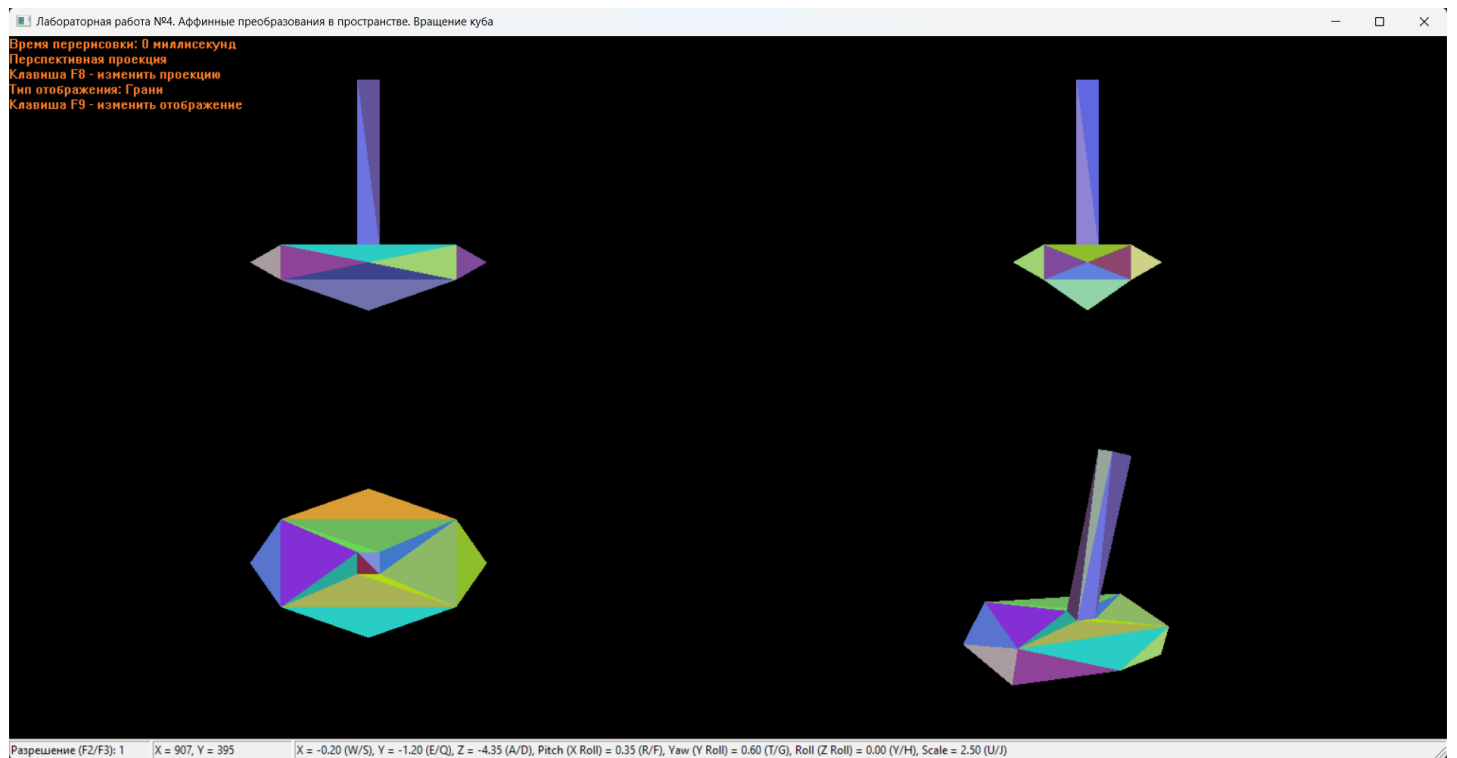
        frame.Triangle(pointA.x, pointA.y, pointA.z, pointB.x, pointB.y, pointB.z, pointC.x,
pointC.y, pointC.z, polygonColor, draw_mode);
    }
}
};

#endif // PAINTER_H

```

Ссылка на репозиторий:

https://github.com/IAmProgrammist/comp_graphics/tree/lab_4_affine_3d_simple_cube



Вывод: в ходе лабораторной работы получены навыки использования аффинных преобразований в пространстве и создание графического приложения с использованием GDI в среде Visual Studio для визуализации простейших трёхмерных объектов.