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 №3

по дисциплине: Компьютерная графика
тема: «Аффинные преобразования на плоскости»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили:
ст. пр. Осипов Олег Васильевич

Белгород 2024 г.

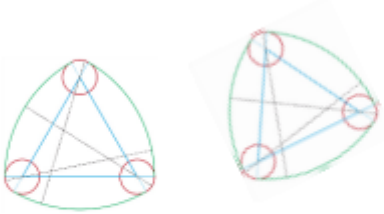
Лабораторная работа №2
Аффинные преобразования на плоскости
Вариант 8

Цель работы: получение навыков выполнения аффинных преобразований на плоскости и создание графического приложения на языке C++ для создания простейшей анимации

Задания для выполнения к работе:

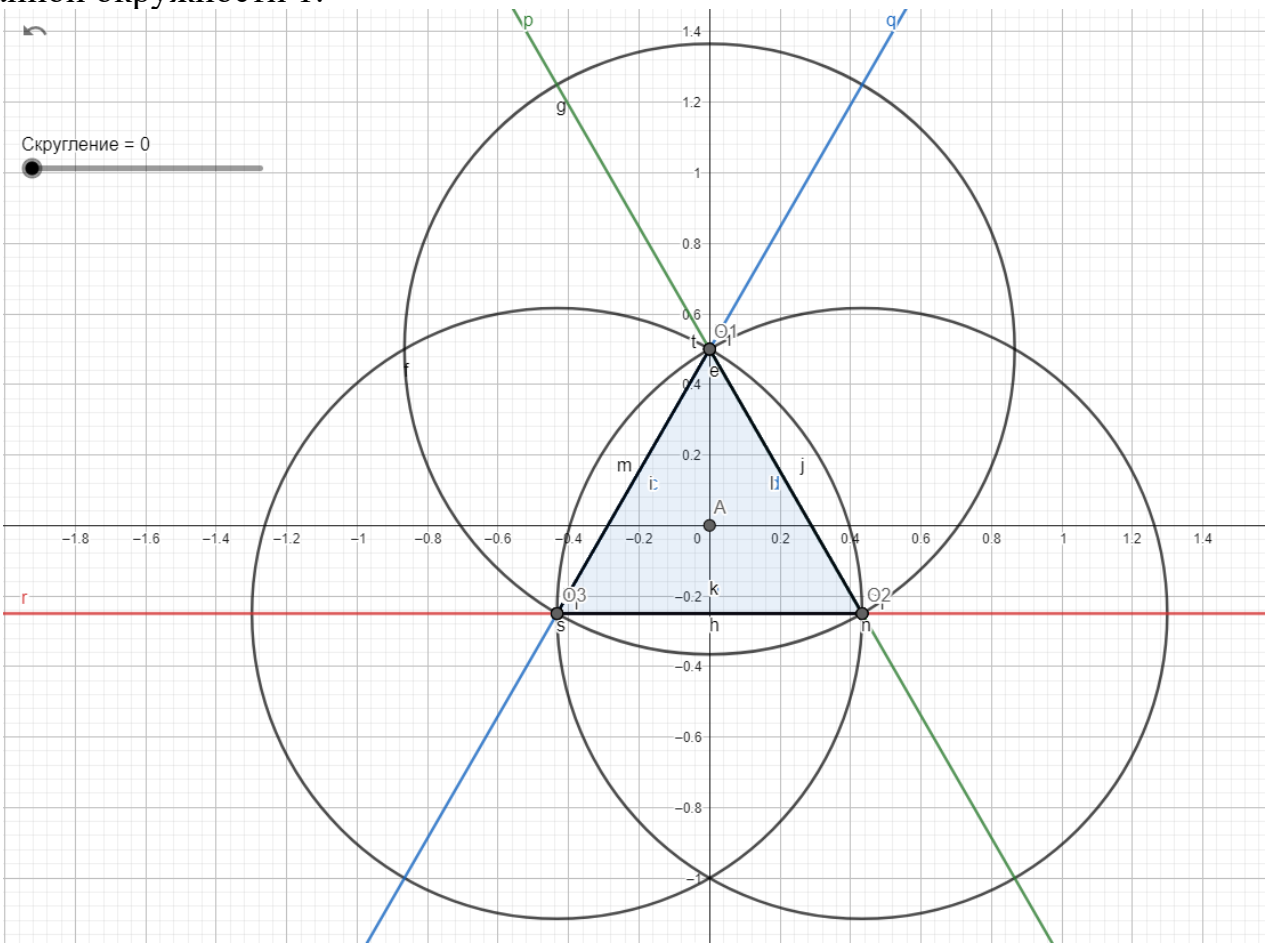
1. Разработать модуль для выполнения аффинных преобразований на плоскости с помощью матриц. В модуле должны быть реализованы перегруженные операции действия с матрицами (умножение), с векторами и матрицами (умножение вектора-строки на матрицу), конструкторы различных матриц (переноса, масштабирования, переноса, отражения).
2. Разработать алгоритм и составить программу для построения на экране изображения в соответствии с номером варианта (по журналу старосты). В качестве исходных данных взять указанные в таблице №1 лаб. работы №1.

Задание:

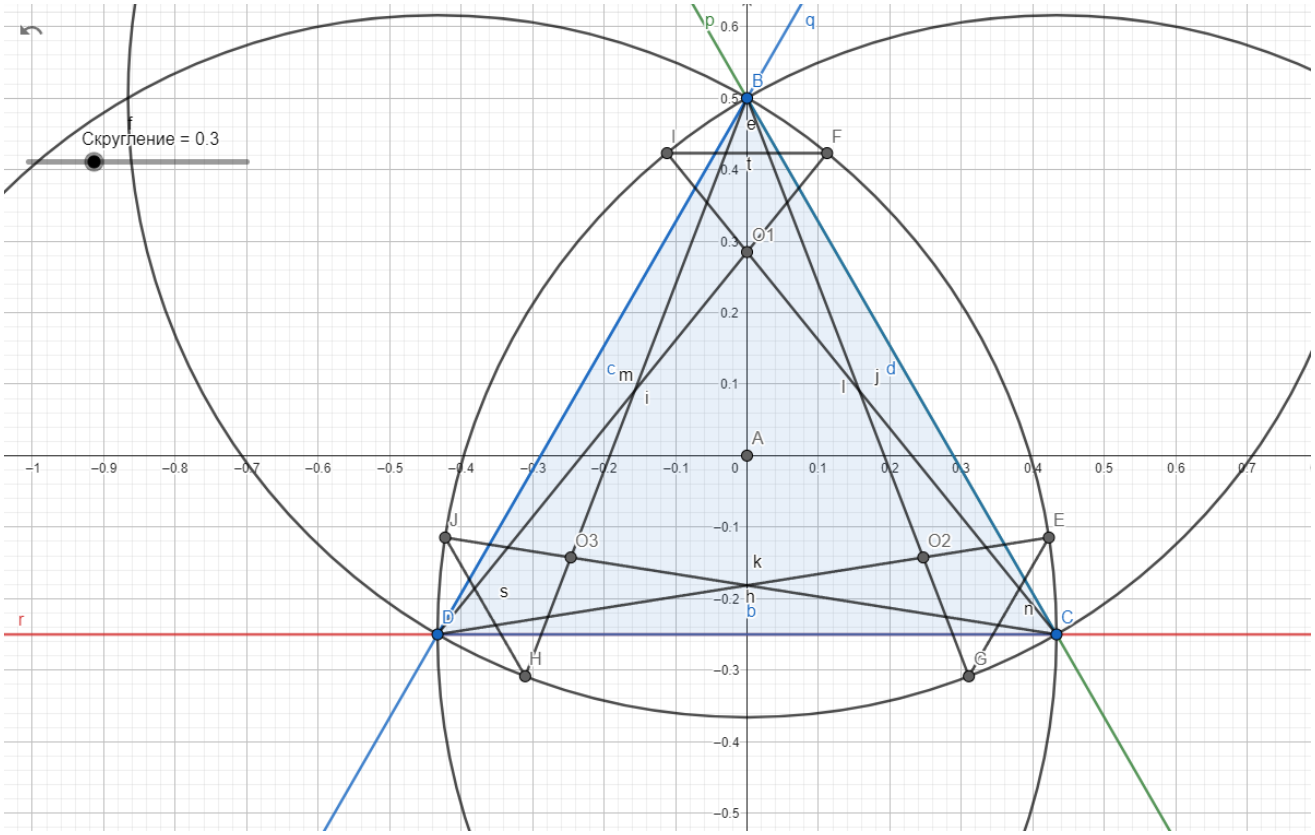


Нарисовать несколько треугольников Рёло с закруглёнными краями, хаотически движущихся в пределах экрана. Треугольники должны вращаться, плавно масштабироваться. Рассчитать цвет таким образом, чтобы он менялся плавно от края треугольника к центру.

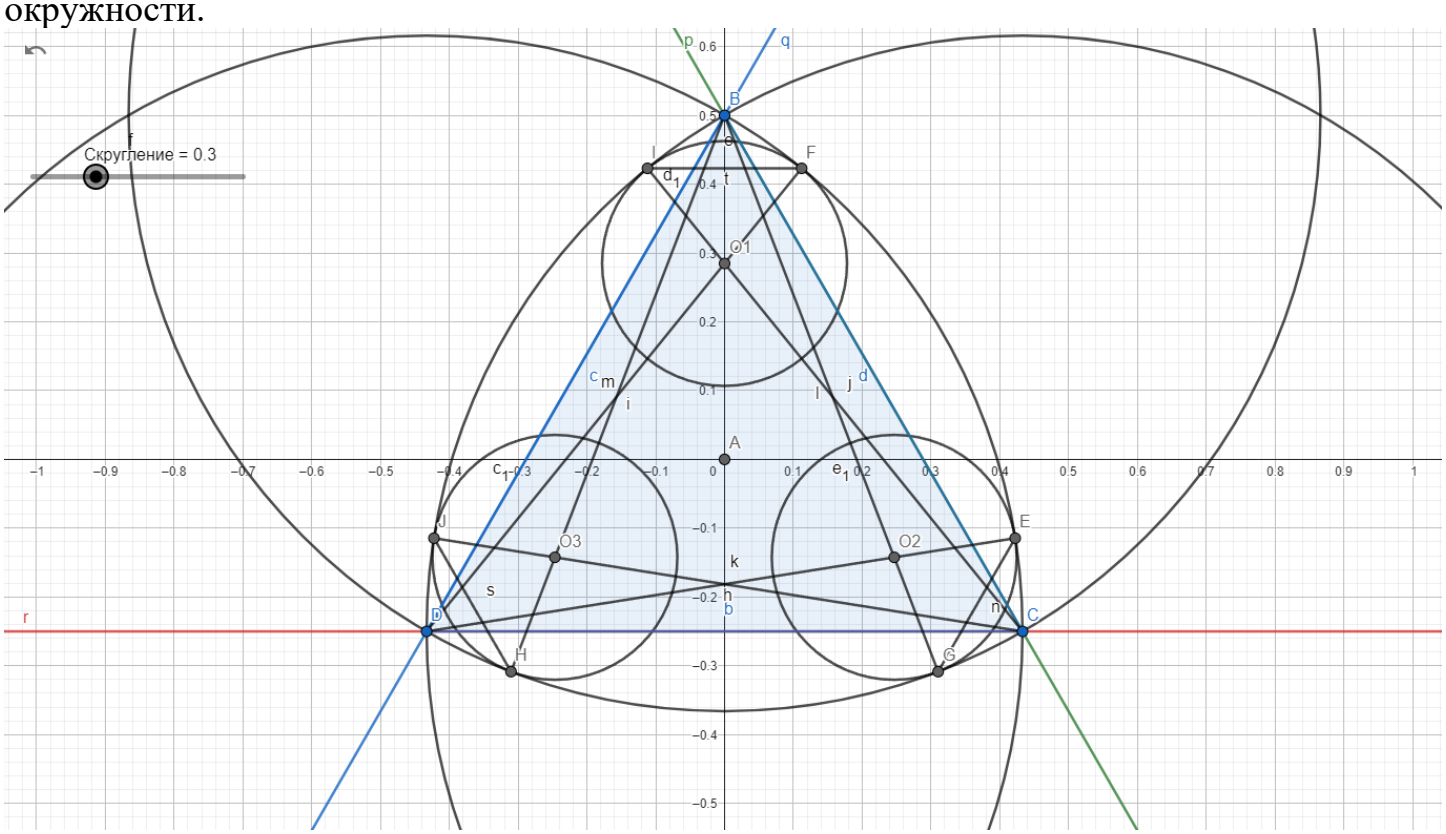
Треугольник Рёло – это треугольник, в котором добавлены три дуги, центр которых находится в одной из трёх точек, а две остальные – точки на окружности. Так и выполним. Предположим, что у нас есть равносторонний треугольник с центром A диаметром описанной окружности 1.



Для того чтобы скруглить края такого треугольника необходимо добавить в углы окружности, которая соприкасается с каждой из двух дуг и в этой точке имеет одинаковую касательную. Чтобы построить касательную, необходимо из центра к окружности провести радиус и к радиусу провести перпендикуляр. Следовательно, центр маленькой окружности должен лежать на радиусе и радиус этой маленькой окружности должен совпадать с большим радиусом. Проведём такие радиусы, угол между радиусом и стороной треугольника должен быть одинаковым (от 0 до 30 градусов):

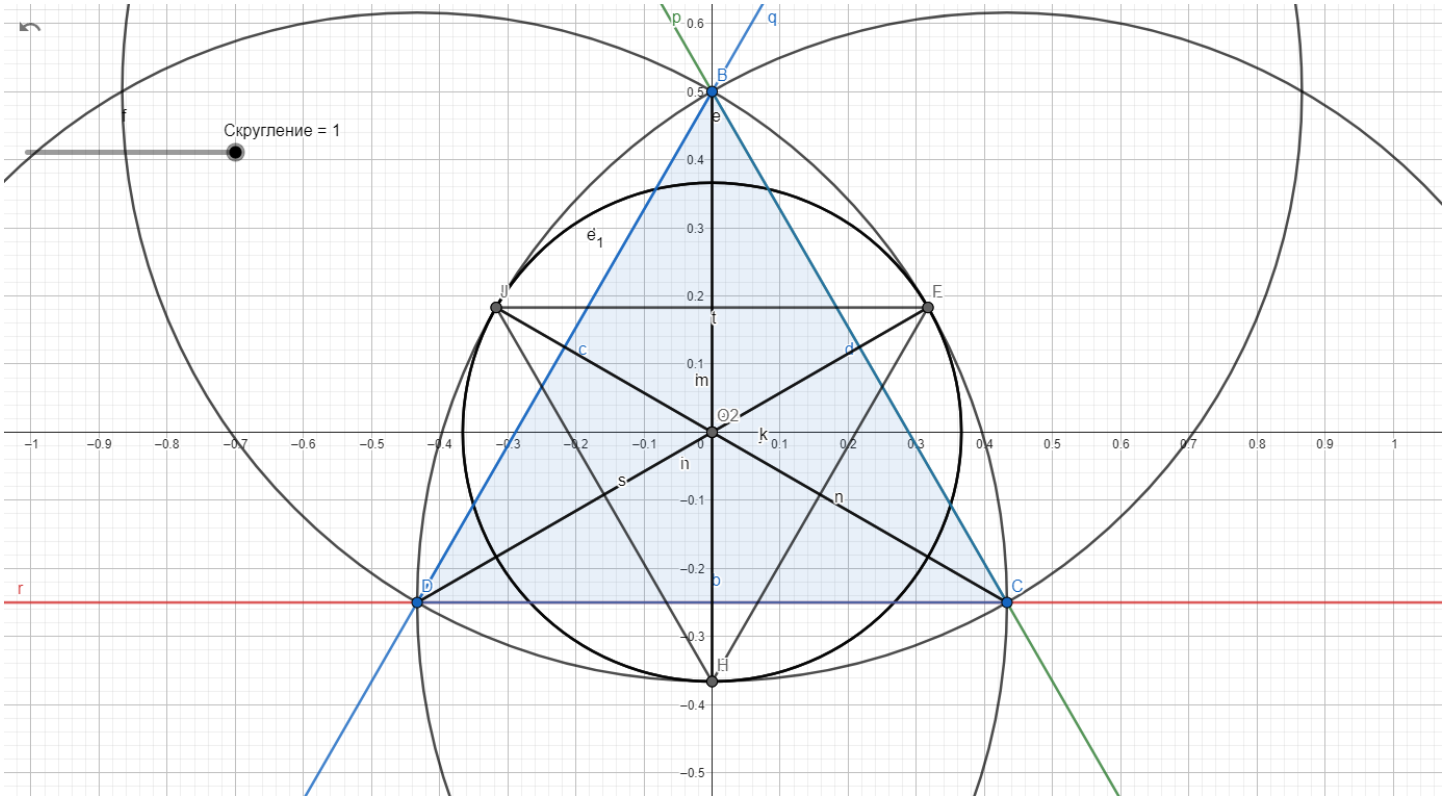


Получили радиусы BH, BG, CI, CJ, DE, DF. Можем отметить, что радиусы пересекаются в точках O1, O2, O3 и O1F = O1I = O2E = O2G = O3J = O3H. Можем построить окружности.



Таким образом, получили окружности с одинаковыми касательными, что даст качественное приятное скругление.

Ограничим угол 30 градусами, так как дальнейшее скругление не имеет смысла. При максимальном скруглении получим круг:



Определили основные точки, которые необходимы для отрисовки треугольника Рёло. В дальнейшем можем выполнить его преобразование, выполнив умножение векторов точек B, C, D, E, F, G, H, I, J на SRT-матрицу. В названии матрицы кроется порядок получения матрицы преобразования. Мы получаем S – матрицу масштабирования -, R – матрицу вращения -, и T – матрицу перемещения. Перемножаем матрицы масштабирования, вращения и перемещения и получаем искомую матрицу. После чего можно выполнять умножение полученной матрицы на вектор.

Отрисовать точку в треугольнике Рёло легко – будем ставить точку, если она находится в трёх больших окружностях сразу. Однако у нас есть скругление. Ограничим первый алгоритм отрисовки прямыми IF, EG, JH – точка должна быть также ниже этих прямых. После этого проверяем, находится ли точка в одной из маленьких окружностей.

Painter.h

```
#ifndef PAINTER_H
#define PAINTER_H

#include "Frame.h"
#include "Matrices.h"
#include "RadialInterpolator.h"
#include "BarycentricInterpolator.h"
#include "SectorInterpolator.h"
#include "ReuleauxTriangleInterpolator.h"

// Установите 1 для отрисовки основного варианта, 0 - для отрисовки задания с защиты (сектор-круг)
#define MAIN_TASK 1

// Угол поворота фигуры
float global_angle = 0;

// Координаты последнего пикселя, который выбрал пользователь
struct
{
    int X, Y;
} global_clicked_pixel = { -1, -1 };

typedef struct
{
    float x;
    float y;
} coordinate;

class Painter
{
public:

    double sawtooth(double val, double height) {
        return abs(fmod(val, height) - height / 2);
    }

    void Draw(Frame& frame)
    {
        // Шахматная текстура
        for (int y = 0; y < frame.height; y++)
            for (int x = 0; x < frame.width; x++)
            {
                if ((x + y) % 2 == 0)
                    frame.SetPixel(x, y, { 23, 25, 23 });   // Золотистый цвет
                //frame.SetPixel(x, y, { 217, 168, 14 });
                else
                    frame.SetPixel(x, y, { 20, 20, 20 }); // Чёрный цвет
                //frame.SetPixel(x, y, { 255, 255, 255 }); // Белый цвет
            }

        // Код для отрисовки основного задания.
        if (MAIN_TASK) {
            int W = frame.width - frame.width * 0.02, H = frame.height - frame.height * 0.02;
            double t1XOffset = frame.width * 0.01 + sawtooth(312 + global_angle * 65, W * 2), t1YOffset = frame.height * 0.01 + sawtooth(41 + global_angle * 131, H * 2);
            double t2XOffset = frame.width * 0.01 + sawtooth(19 + global_angle * 102, W * 2), t2YOffset = frame.height * 0.01 + sawtooth(901 + global_angle * 12, H * 2);
            double t3XOffset = frame.width * 0.01 + sawtooth(71 + global_angle * 25, W * 2), t3YOffset = frame.height * 0.01 + sawtooth(47 + global_angle * 32, H * 2);
            SectorInterpolator t1s(t1XOffset, t1YOffset);
            RadialInterpolator t2s(t2XOffset, t2YOffset, t2XOffset, t2YOffset, { {255, 180, 0}, {180, 0, 99}, {188, 188, 188} }, global_angle / 5);
```

```
            BarycentricInterpolator t3s(frame.width * 0.01, frame.height * 0.01, frame.width *
0.01, frame.height * 0.01 + H, frame.width * 0.01 + W, frame.height * 0.01,
                { 91, 9, 39, 12 }, { 98, 192, 67 }, { 76, 100, 158 });
            ReuleauxTriangleInterpolator<SectorInterpolator> t1(t1XOffset, t1YOffset, 50 +
sawtooth(global_angle * 10, 30), 12 + global_angle, 0.3, t1s);
            ReuleauxTriangleInterpolator<RadialInterpolator> t2(t2XOffset, t2YOffset, 70 +
sawtooth(global_angle * 22, 11), 9 + 2 * global_angle, 0, t2s);
            ReuleauxTriangleInterpolator<BarycentricInterpolator> t3(t3XOffset, t3YOffset, 20 +
sawtooth(global_angle * 10, 100), 12 + global_angle, 0.7, t3s);

            frame.Triangle(frame.width * 0.01, frame.height * 0.01, frame.width * 0.01,
frame.height * 0.01 + H, frame.width * 0.01 + W, frame.height * 0.01, t1);
            frame.Triangle(frame.width * 0.01, frame.height * 0.01 + H, frame.width * 0.01 + W,
frame.height * 0.01 + H, frame.width * 0.01 + W, frame.height * 0.01, t1);
            frame.Triangle(frame.width * 0.01, frame.height * 0.01, frame.width * 0.01,
frame.height * 0.01 + H, frame.width * 0.01 + W, frame.height * 0.01, t2);
            frame.Triangle(frame.width * 0.01, frame.height * 0.01 + H, frame.width * 0.01 + W,
frame.height * 0.01 + H, frame.width * 0.01 + W, frame.height * 0.01, t2);
            frame.Triangle(frame.width * 0.01, frame.height * 0.01, frame.width * 0.01,
frame.height * 0.01 + H, frame.width * 0.01 + W, frame.height * 0.01, t3);
            frame.Triangle(frame.width * 0.01, frame.height * 0.01 + H, frame.width * 0.01 + W,
frame.height * 0.01 + H, frame.width * 0.01 + W, frame.height * 0.01, t3);
        }
        else {
        }
    }
};

#endif // PAINTER_H
```

ReuleauxTriangleInterpolator.h

```
#pragma once
#include "Frame.h"
#include "Matrices.h"

# define PI 3.14159265358979323846
# define EPS 0.0000000001

// Класс для расчёта треугольника Рёло с закруглением краёв
template <typename ColorInterpolator>
class ReuleauxTriangleInterpolator
{
    Vector B;
    Vector C;
    Vector D;
    Vector E;
    Vector F;
    Vector G;
    Vector H;
    Vector I;
    Vector J;
    Vector O1;
    Vector O2;
    Vector O3;
    double smallCircleRadius;
    double sideSize;
    ColorInterpolator colorInterpolator;

    std::pair<bool, Vector> findPointsIntersection(double ax0, double ay0, double ax1, double
ay1,
        double bx0, double by0, double bx1, double by1) {
        double adx = ax1 - ax0;
        double ady = ay1 - ay0;
        double bdx = bx1 - bx0;
        double bdy = by1 - by0;
```

```
        double denom = bdy * adx - bdx * ady;
        if (denom == 0) {
            return { false, Vector()};
        }
        double t = (bdx * (ay0 - by0) + bdy * (bx0 - ax0)) / denom;
        return { true, Vector(ax0 + adx * t, ay0 + ady * t) };
    }

public:
    ReuleauxTriangleInterpolator(double x, double y, double size, double angle, double rounding,
ColorInterpolator colorInterpolator) : colorInterpolator(colorInterpolator) {
        angle = angle < 0 ? (2 * PI / 3. + fmod(angle, 2 * PI / 3.)) : (fmod(angle, 2 * PI /
3.));
        rounding = max(min(rounding, 1), EPS);
        double roundang = rounding * (PI / 6);
        // Определим основные точки
        double R = .5;
        double a = R * sqrt(3);
        double h = 3 * R / 2;

        // Основные точки для последующей трансформации, см. схему GeoGebra
        this->B = Vector( 0, 0.5 );
        this->C = Vector(sqrt(3) / 4, -0.25 );
        this->D = Vector(-C.x(), C.y());
        this->E = Vector(D.x() + a * cos(roundang), D.y() + a * sin(roundang));
        this->F = Vector(D.x() + a * cos(PI / 3 - roundang), D.y() + a * sin(PI / 3 - roundang));
        this->G = Vector(B.x() + a * cos(-PI / 3 - roundang), B.y() + a * sin(-PI / 3 -
roundang));
        this->H = Vector(B.x() + a * cos(-2 * PI / 3 + roundang), B.y() + a * sin(-2 * PI / 3 +
roundang));
        this->I = Vector(C.x() + a * cos(2 * PI / 3 + roundang), C.y() + a * sin(2 * PI / 3 +
roundang));
        this->J = Vector(C.x() + a * cos(PI - roundang), C.y() + a * sin(PI - roundang));

        this->O1 = findPointsIntersection(D.x(), D.y(), F.x(), F.y(), C.x(), C.y(), I.x(),
I.y()).second;
        this->O2 = findPointsIntersection(D.x(), D.y(), E.x(), E.y(), B.x(), B.y(), G.x(),
G.y()).second;
        this->O3 = findPointsIntersection(B.x(), B.y(), H.x(), H.y(), C.x(), C.y(), J.x(),
J.y()).second;

        Matrix S = Matrix::scale(size);
        Matrix Ro = Matrix::rotation(angle);
        Matrix T = Matrix::transfrom(x, y);
        Matrix SRT = (T * Ro) * S;

        B = SRT * B;
        C = SRT * C;
        D = SRT * D;
        E = SRT * E;
        F = SRT * F;
        G = SRT * G;
        H = SRT * H;
        I = SRT * I;
        J = SRT * J;
        O1 = SRT * O1;
        O2 = SRT * O2;
        O3 = SRT * O3;

        this->sideSize = a * size;
        this->smallCircleRadius = sqrt(pow(F.x() - O1.x(), 2) + pow(F.y() - O1.y(), 2));
    }

    COLOR color(float x, float y) {
        double O1dy = F.y() - I.y();
        double O1dx = F.x() - I.x();
        double O2dy = G.y() - E.y();
```

```
        double O2dx = G.x() - E.x();
        double O3dy = J.y() - H.y();
        double O3dx = J.x() - H.x();

        if (pow(x - C.x(), 2) + pow(y - C.y(), 2) <= sideSize * sideSize &&
            pow(x - B.x(), 2) + pow(y - B.y(), 2) <= sideSize * sideSize &&
            pow(x - D.x(), 2) + pow(y - D.y(), 2) <= sideSize * sideSize &&
            (x - I.x()) * O1dy - (y - I.y()) * O1dx >= EPS &&
            (x - E.x()) * O2dy - (y - E.y()) * O2dx >= EPS &&
            (x - H.x()) * O3dy - (y - H.y()) * O3dx >= EPS) {
            return colorInterpolator.color(x, y);
        }
        else if (
            pow(x - O1.x(), 2) + pow(y - O1.y(), 2) <= smallCircleRadius * smallCircleRadius ||
            pow(x - O2.x(), 2) + pow(y - O2.y(), 2) <= smallCircleRadius * smallCircleRadius ||
            pow(x - O3.x(), 2) + pow(y - O3.y(), 2) <= smallCircleRadius * smallCircleRadius) {
                return colorInterpolator.color(x, y);
        }

        return COLOR(0, 0, 0, 0);
    }
};
```

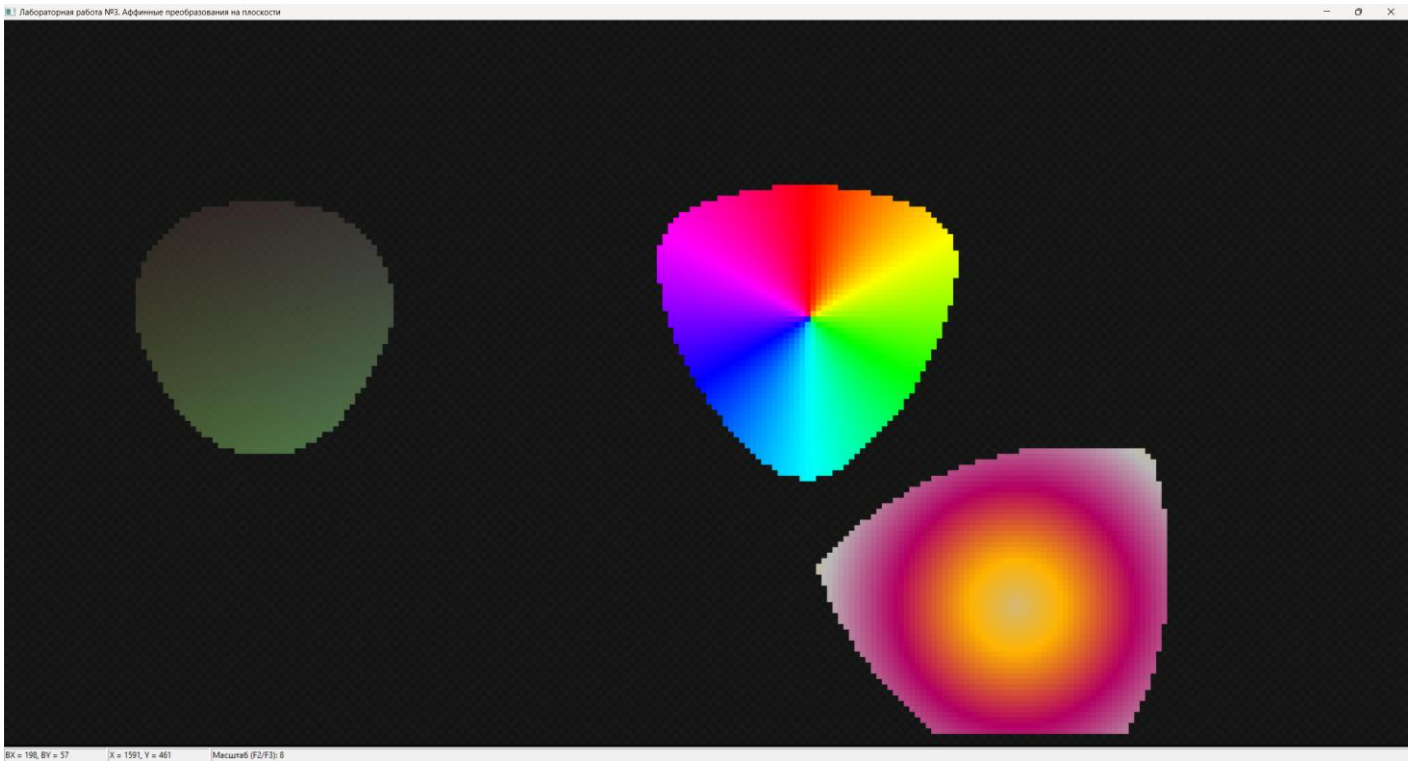
Matrices.h

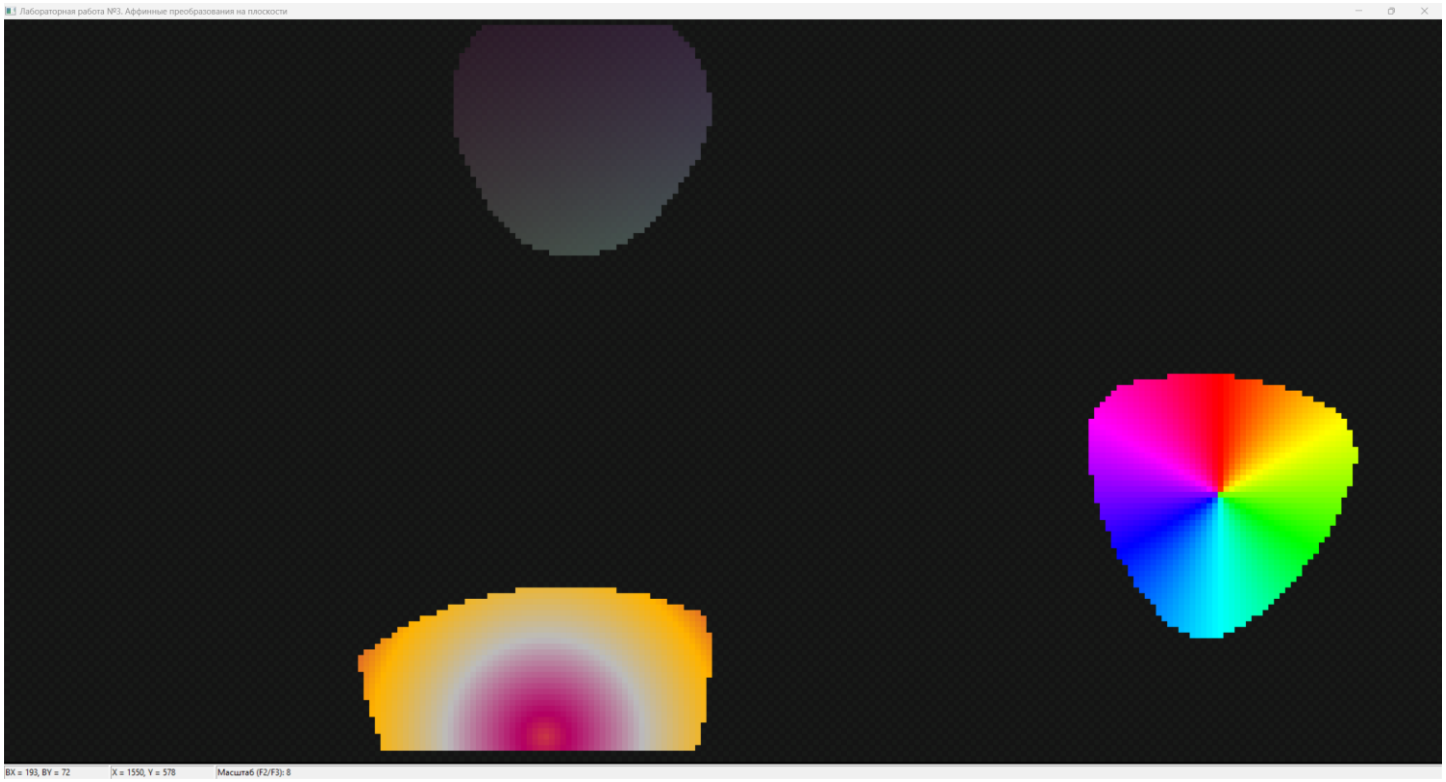
```
#pragma once
#pragma once

#include <string>
#include <vector>

class Vector {
public:
    double vector[3];
    Vector(std::initializer_list<double> v) {
        memcpy(vector, v.begin(), sizeof(double) * 3);
    }
    Vector(std::vector<double> v) {
        memcpy(vector, &v[0], sizeof(double) * 3);
    }
    Vector(double x, double y) {
        this->vector[0] = x;
        this->vector[1] = y;
        this->vector[2] = 1;
    }
    Vector() {
        this->vector[0] = 0;
        this->vector[1] = 0;
        this->vector[2] = 1;
    }

public:
    double x() {
        return this->vector[0];
    }

    double y() {
        return this->vector[1];
    }
};

class Matrix {
public:
    double data[9];
    double* matrix[3];
```

```
    Matrix(std::initializer_list<double> v) {
        memcpy(data, v.begin(), sizeof(double) * 9);
        matrix[0] = data;
        matrix[1] = data + 3;
        matrix[2] = data + 6;
    }
    Matrix(std::vector<double> v) {
        memcpy(data, &v[0], sizeof(double) * 9);
        matrix[0] = data;
        matrix[1] = data + 3;
        matrix[2] = data + 6;
    }

    static Matrix rotation(double angle) {
        return { cos(angle), -sin(angle),  0,
                 sin(angle),  cos(angle),  0,
                 0,           0,  1 };
    }

    static Matrix scale(double scale) {
        return { scale, 0, 0,
                 0, scale, 0,
                 0, 0, 1 };
    }

    static Matrix transfrom(double x, double y) {
        return { 1, 0, x,
                 0, 1, y,
                 0, 0, 1 };
    }

    static Matrix mirrorHorizontal() {
        return { -1, 0,0,
                 0, 1, 0,
                 0, 0, 1 };
    }

    static Matrix mirrorVertical() {
        return { 1, 0,0,
                 0, -1, 0,
                 0, 0, 1 };
    }

    Matrix operator * (Matrix& another) {
        double dataNew[9] = {};
        double* matrixNew[3];
        matrixNew[0] = dataNew;
        matrixNew[1] = dataNew + 3;
        matrixNew[2] = dataNew + 6;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                matrixNew[i][j] = 0;

                for (int k = 0; k < 3; k++) {
                    matrixNew[i][j] += this->matrix[i][k] * another.matrix[k][j];
                }
            }
        }

        return Matrix(std::vector<double>(dataNew, dataNew + 9));
    }

    Vector operator * (Vector& vec) {
        return Vector({
            vec.vector[0] * this->matrix[0][0] + vec.vector[1] * this->matrix[0][1] +
vec.vector[2] * this->matrix[0][2],
```

```
            vec.vector[0] * this->matrix[1][0] + vec.vector[1] * this->matrix[1][1] +
vec.vector[2] * this->matrix[1][2],
            vec.vector[0] * this->matrix[2][0] + vec.vector[1] * this->matrix[2][1] +
vec.vector[2] * this->matrix[2][2] });
    }
};
```

Ссылка на репозиторий:
https://github.com/IAmProgrammist/comp_graphics/tree/lab_3_affine_transformations





Вывод: в ходе лабораторной работы получены навыки выполнения аффинных преобразований на плоскости и создание графического приложения на языке C++ для создания простейшей анимации.