

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)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №3

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

Выполнил: ст. группы ПВ-201
Морозов Данила Александрович

Проверил:
Осипов Олег Васильевич

Белгород 2022 г.

Лабораторная работа №3 «Аффинные преобразования на плоскости»

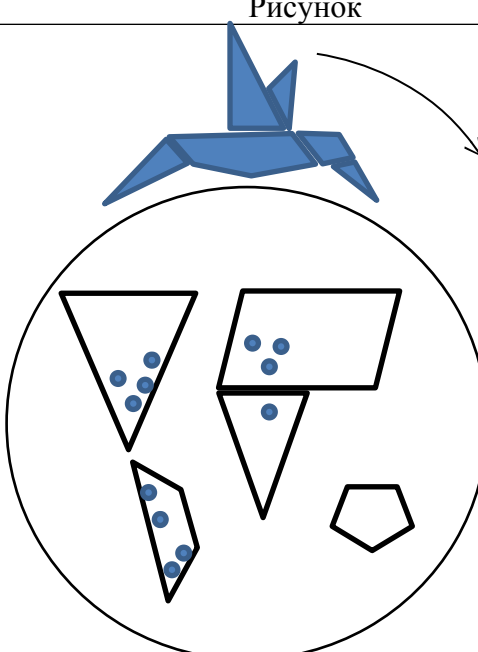
Цель работы:

Получение навыков выполнения аффинных преобразований на плоскости и создание графического приложения с использованием GDI в среде Qt Creator.

Задания 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. В программе должна быть предусмотрена возможность ввода пользователем исходных данных (из правой колонки таблицы №1).
3. Разбить окно на 2 равные части. В левой части должна выводиться основная анимация, в правой части её отражение относительно вертикальной линии, проходящей через центр окна.
4. Изображение должно масштабироваться по центру левой и правой части окна с отступом 10 пикселей от границ и вертикальной линии и реагировать на изменение размера окна (см. пример проекта lab_1_CSharp).
5. Раскрасить (залить) примитивы (круги, многоугольники и др.) по собственному усмотрению.

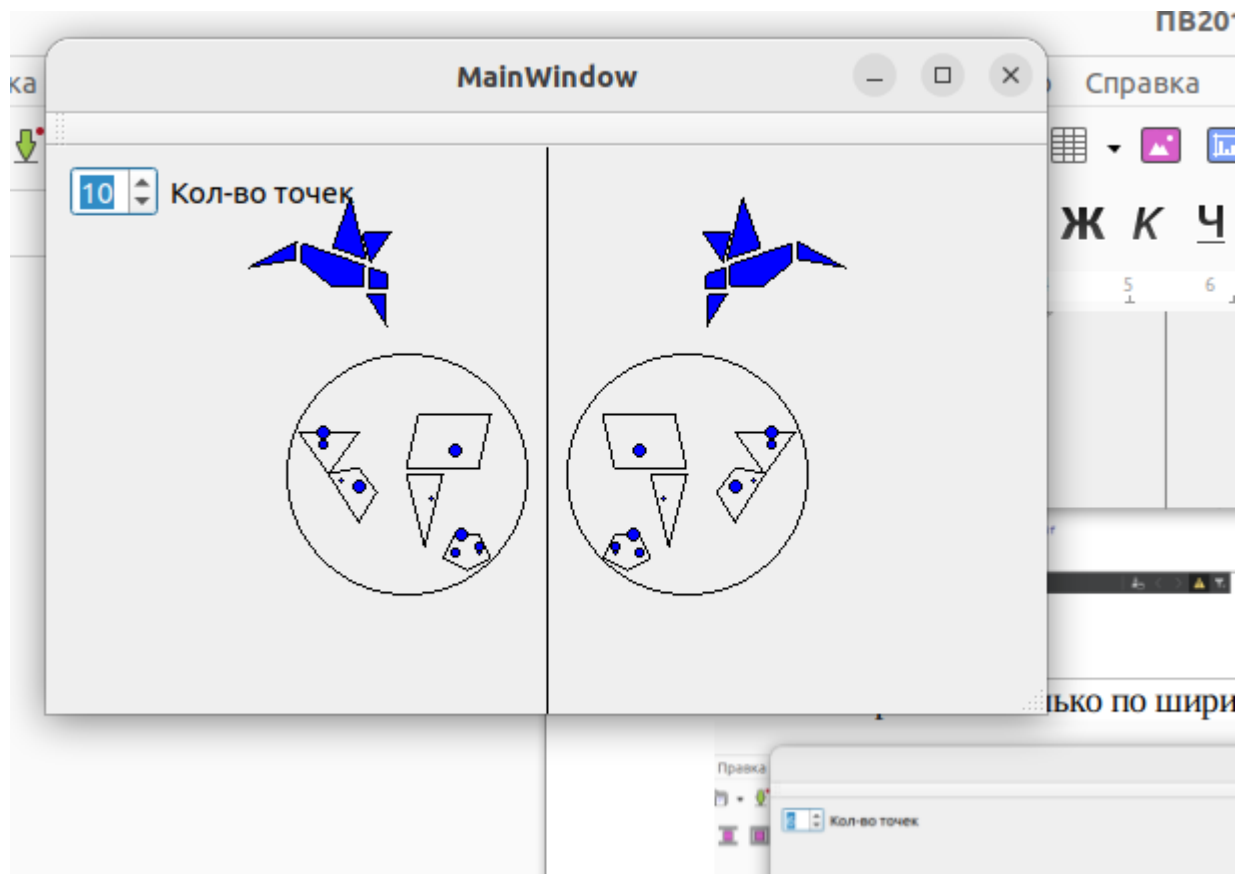
Вариант №1:

Вариант	Рисунок	Исходные данные
9		<p>Реализовать вращение птицы вокруг шара и следующую анимацию: на материках (геометрические фигуры внутри круга) появление случайным образом кругов случайного размера по случайным координатам. Раскраску фигур сделать произвольной.</p>

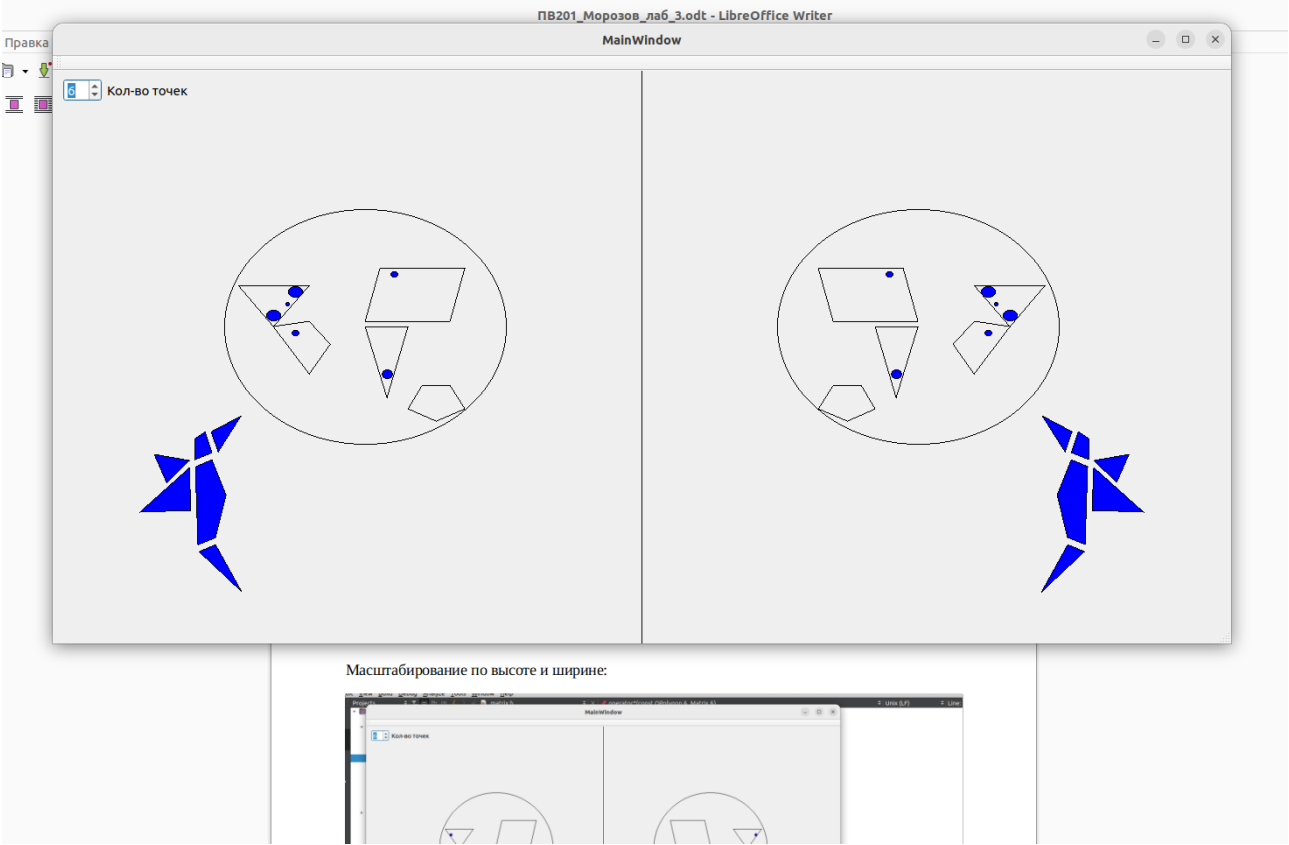
Выполнение:

Снимки экрана:

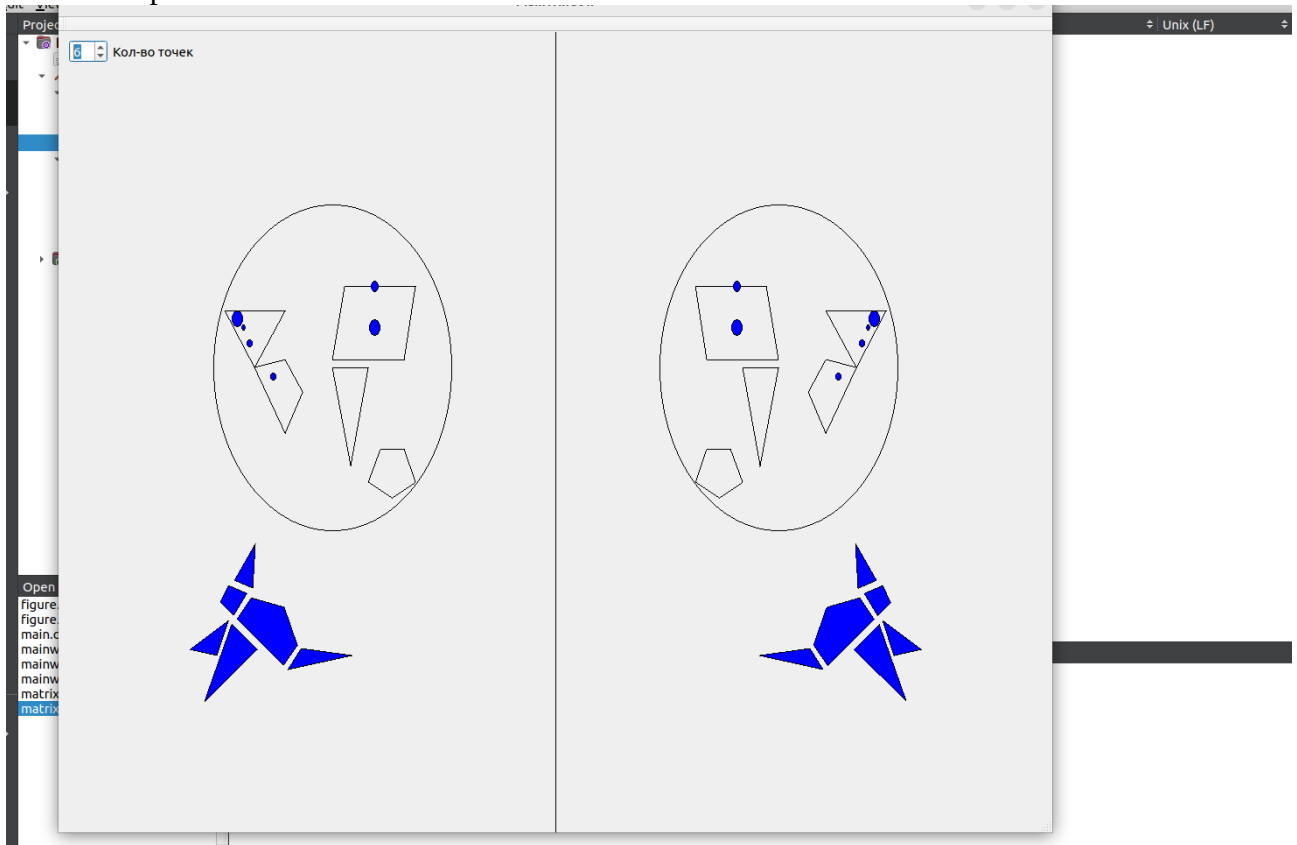
Стартовый размер:



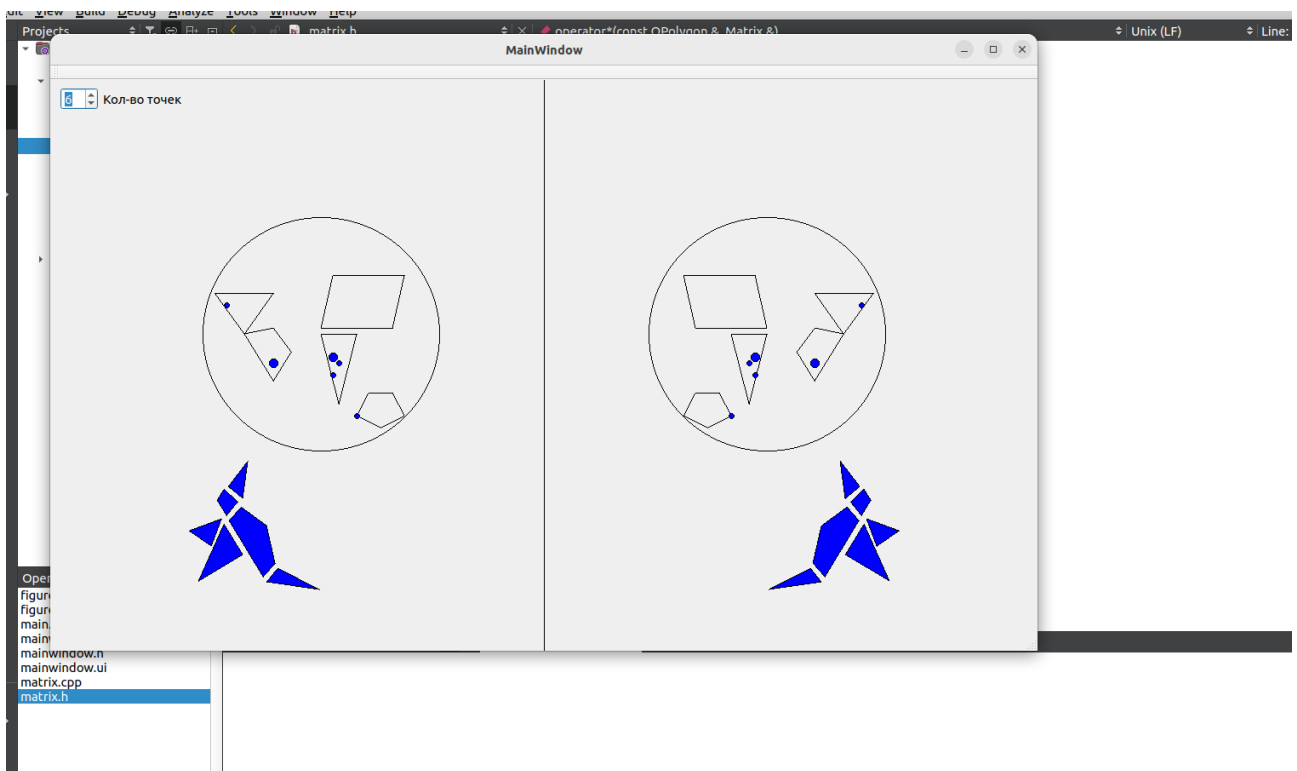
Масштабирование только по ширине:



Масштабирование только по высоте



Масштабирование по высоте и ширине:



Формулы:

Все необходимые фигуры строятся в начале координат, небольших размеров: например, хвост птицы состоит из точек с координатами (-2, 12), (5, 8), (4, 5). Это позволяет нам их в дальнейшем передвинуть на необходимые позиции и т. п. с помощью аффинных преобразований, реализованных с помощью наших матриц.

Где возможно — мы используем матрицы преобразований для трехмерных пространств (4x4), просто игнорируя последний столбец и последнюю строку — в нашем случае это возможно для всех используемых матриц, кроме матрицы переноса — она реализована в виде (3x3).

Матрица поворота:

$$\begin{bmatrix} \cos(\text{angle}) & \sin(\text{angle}) & 0 & 0 \\ -\sin(\text{angle}) & \cos(\text{angle}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Матрица переноса:

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix}$$

Матрица масштабирования ($kz = 1$):

$$\begin{bmatrix} kx & 0 & 0 & 0 \\ 0 & ky & 0 & 0 \\ 0 & 0 & kz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Матрица отражения:

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Сперва всегда применяется масштабирование. Для поворота вокруг определенной точки $P(x, y)$ мы сперва переносим все точки на $(-x, -y)$, затем производим поворот, и после — перенос на (x, y) . Отражение всегда идет последним.

Для расстановки точек мы — мы сперва получаем многоугольники (координаты без преобразований). Затем для каждого многоугольника: с помощью `boundingRect()` мы получаем прямоугольник, в который вписывается многоугольник, и ориентируясь по координатам его левого верхнего угла мы генерируем случайные координаты в пределах прямоугольника. Затем проверяем, лежит ли данная точка внутри многоугольника с помощью `containsPoint`, если нет — повторяем генерацию до тех пор, пока точка не окажется в

многоугольнике, иначе — добавляем точку в массив точек и переходим к следующей. Далее на них накладываются те же преобразования, что и на многоугольники: масштабирование и перенос, отражение.

Листинги кода:

Листинг №1: «main.cpp»:

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

Листинг №2: «mainwindow.h»:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QTimer>
#include <QPainter>
#include "matrix.h"
#include "figure.h"

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void rotate();
    void dots();
private:
    Ui::MainWindow *ui;
    QTimer *timer;
    QTimer *timer_dots;

    Matrix originTranslate;
    Matrix rotateZ;
    Matrix scale;
    double angle;

    void paintEvent(QPaintEvent *event);
    bird_st bird();
    std::vector<QPolygon> world();
    std::vector<QPolygon> inner_world;
    std::atomic<bool> canDrawPoints;
    int rotateX;
    int rotateY;
    QPoint rotatePoint;
```

```

        std::vector<QPoint> points;
        std::vector<int> sizes;
};
#endif // MAINWINDOW_H

```

Листинг №3: «mainwindow.cpp»:

```

#include "mainwindow.h"
#include "../ui_mainwindow.h"
#include <QRandomGenerator>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    timer = new QTimer;
    timer->setInterval(60);
    timer->start();
    timer_dots = new QTimer;
    timer_dots->setInterval(1000);
    timer_dots->start();
    connect(timer, &QTimer::timeout, this, &MainWindow::rotate);
    connect(timer_dots, &QTimer::timeout, this, &MainWindow::dots);

    originTranslate = TranslateMatrix2D(75, 75);
    rotateX = 35;
    rotateY = 35;
    rotatePoint.setX(rotateX);
    rotatePoint.setY(rotateY);
    scale = ScaleMatrix(6, 6, 1);
}

bird_st MainWindow::bird(void){
    bird_st birb;
    birb.tail().push_back(QPoint(-2, 12));
    birb.tail().push_back(QPoint(5, 8));
    birb.tail().push_back(QPoint(4, 5));

    birb.body().push_back(QPoint(5, 5));
    birb.body().push_back(QPoint(6, 8));
    birb.body().push_back(QPoint(12, 10));
    birb.body().push_back(QPoint(17, 8));
    birb.body().push_back(QPoint(16, 5));

    birb.closeWing().push_back(QPoint(10, 4));
    birb.closeWing().push_back(QPoint(16, 4));
    birb.closeWing().push_back(QPoint(10, -5));

    birb.farWing().push_back(QPoint(17, 4));
    birb.farWing().push_back(QPoint(14, 0));
    birb.farWing().push_back(QPoint(18, -2));

    birb.head().push_back(QPoint(18, 8));
    birb.head().push_back(QPoint(17, 5));
    birb.head().push_back(QPoint(20, 5));
    birb.head().push_back(QPoint(21, 7));

    birb.beak().push_back(QPoint(18, 9));
    birb.beak().push_back(QPoint(21, 8));
    birb.beak().push_back(QPoint(23, 13));
}

```



```

    int rtX = rotateX * scale[0][0] + originTranslate[0][2];
    int rtY = rotateY * scale[1][1] + originTranslate[1][2];
    auto translate{TranslateMatrix2D(rtX, rtY)};
    auto backTranslate{TranslateMatrix2D(-1 * rtX, -1 * rtY)};
    birb.apply({scale, originTranslate, backTranslate, rotateZ, translate});
    return birb;
}

```

```

std::vector<QPolygon> MainWindow::world(){
    std::vector<QPolygon> r(5);

    r[0].push_back({rotateX-18, rotateY-7});
    r[0].push_back({rotateX-8, rotateY-7});
    r[0].push_back({rotateX-13, rotateY});

    r[1].push_back({rotateX, rotateY});
    r[1].push_back({rotateX+6, rotateY});
    r[1].push_back({rotateX+3, rotateY+12});

    r[2].push_back({rotateX, rotateY-1});
    r[2].push_back({rotateX+12, rotateY-1});
    r[2].push_back({rotateX+14, rotateY-10});
    r[2].push_back({rotateX+2, rotateY-10});

    r[3].push_back({rotateX-16, rotateY+1});
    r[3].push_back({rotateX-6, rotateY+1});
    r[3].push_back({rotateX-8, rotateY-4});
    r[3].push_back({rotateX-14, rotateY-4});
    auto rt = RotateMatrixOZ(-15.0/16);
    auto btr = TranslateMatrix2D(-(rotateX-14), -(rotateY-4));
    auto tr = TranslateMatrix2D(rotateX-14, (rotateY-4));
    auto ttr = TranslateMatrix2D(6, 3);
    r[3] = r[3] * btr * rt * tr * ttr;

    r[4].push_back({rotateX+8, rotateY+10});
    r[4].push_back({rotateX+12, rotateY+10});
    r[4].push_back({rotateX+14, rotateY+14});
    r[4].push_back({rotateX+10, rotateY+16});
    r[4].push_back({rotateX+6, rotateY+14});

    return r;
}

```

```

void MainWindow::paintEvent(QPaintEvent *event){
    QPainter painter(this);
    scale = ScaleMatrix(6 * (MainWindow::width() / 1000.0), 6 *
(MainWindow::height() / 600.0), 1);
    auto refPoint = QPoint{MainWindow::width()/2, MainWindow::height()/2};
    painter.drawLine(QPoint{MainWindow::width()/2, MainWindow::height()},
QPoint{MainWindow::width()/2, 0});
    auto translateRefPoint = TranslateMatrix2D(refPoint.x(), refPoint.y());
    auto translateRefPointBack = TranslateMatrix2D(-refPoint.x(), -
refPoint.y());
    rotateZ = RotateMatrixOZ(angle);
    rotatePoint.setX(rotateX * scale[0][0] + originTranslate[0][2]);
    rotatePoint.setY(rotateY * scale[1][1] + originTranslate[1][2]);
    auto reflect = ReflectMatrixOZ();
    auto blueBrush = QBrush();
    blueBrush.setColor(Qt::blue);
    blueBrush.setStyle(Qt::SolidPattern);
    painter.setBrush(blueBrush);
    for(auto poly : bird().inner){

```

```

        auto poly2 = poly * translateRefPointBack * reflect * translateRefPoint;
        painter.drawPolygon(poly);
        painter.drawPolygon(poly2);
    }
    auto defBrush = QBrush();
    painter.setBrush(defBrush);
    int rx = 20 * scale[0][0];
    int ry = 20 * scale[1][1];
    painter.drawEllipse(rotatePoint, rx, ry);
    painter.drawEllipse(rotatePoint * translateRefPointBack * reflect *
translateRefPoint, rx, ry);
    inner_world = world();
    for(auto poly : inner_world){
        poly = poly * scale * originTranslate;
        auto poly2 = poly * translateRefPointBack * reflect * translateRefPoint;
        painter.drawPolygon(poly);
        painter.drawPolygon(poly2);
    }
    painter.setBrush(blueBrush);
    if(canDrawPoints){
        int index = 0;
        for(auto &p : points){
            int r = sizes[index++];
            painter.drawEllipse(p * scale * originTranslate, int(r * scale[0][0]
/ 6.0), int(r * scale[1][1] / 6.0));
            painter.drawEllipse((p * scale * originTranslate *
translateRefPointBack * reflect * translateRefPoint), int(r * scale[0][0] /
6.0), int(r * scale[1][1] / 6.0));
        }
    }
}

void MainWindow::rotate(){
    angle -= 0.15f;
    this->update();
}

void MainWindow::dots(){
    if(inner_world.size() != 0){
        int amount = this->ui->spinBox->value();
        points.clear();
        sizes.clear();
        canDrawPoints = false;
        static QRandomGenerator gen;
        for(int i = 0; i < amount; i++){
            QPoint point;
            auto poly = inner_world[gen.generate()%5];
            QRect qr = poly.boundingRect();
            point.setX(gen.generate());
            point.setY(gen.generate());
            while(poly.containsPoint(point, Qt::OddEvenFill) == false){
                point.setX(qr.bottomLeft().x() + (gen.generate()) %
(qr.bottomRight().x() - qr.bottomLeft().x()) - (gen.generate()) %
(qr.bottomRight().y() - qr.topRight().y())/2);
                point.setY(qr.topLeft().y() + (gen.generate()) %
(qr.bottomRight().y() - qr.topRight().y()) - (gen.generate()) %
(qr.bottomRight().y() - qr.topRight().y())/2);
            }
            sizes.push_back(gen.generate()%5 + 2);
            points.push_back(point);
        }
    }
    canDrawPoints = true;
}

```

```

        this->update();
    }

MainWindow::~MainWindow()
{
    delete timer;
    delete timer_dots;
    delete ui;
}

```

Листинг №5: «matrix.cpp»:

```

#include "matrix.h"
#include <cassert>

std::vector<double> &Matrix::operator[](size_t i) {
    return this->matr[i];
}

Matrix::Matrix(std::vector<std::vector<double>> A) {
    for(int i = 0; i < A.size(); i++)
        matr[i] = A[i];
}

Matrix::Matrix(Matrix const & A) {
    matr = A.matr;
}

Matrix &Matrix::operator=(Matrix const & C) {
    matr = C.matr;
    return *this;
}

Matrix Matrix::operator*(Matrix& a) {
    Matrix R;

    if(a.matr.size() == 3){
        R.matr.resize(3);
        for(auto &v : R.matr)
            v.resize(3);
        R[0][0] = matr[0][0] * a[0][0] + matr[0][1] * a[1][0] + matr[0][2] * a[2][0];
        R[0][1] = matr[0][0] * a[0][1] + matr[0][1] * a[1][1] + matr[0][2] * a[2][1];
        R[0][2] = matr[0][0] * a[0][2] + matr[0][1] * a[1][2] + matr[0][2] * a[2][2];

        R[1][0] = matr[1][0] * a[0][0] + matr[1][1] * a[1][0] + matr[1][2] * a[2][0];
        R[1][1] = matr[1][0] * a[0][1] + matr[1][1] * a[1][1] + matr[1][2] * a[2][1];
        R[1][2] = matr[1][0] * a[0][2] + matr[1][1] * a[1][2] + matr[1][2] * a[2][2];

        R[2][0] = matr[2][0] * a[0][0] + matr[2][1] * a[1][0] + matr[2][2] * a[2][0];
        R[2][1] = matr[2][0] * a[0][1] + matr[2][1] * a[1][1] + matr[2][2] * a[2][1];
        R[2][2] = matr[2][0] * a[0][2] + matr[2][1] * a[1][2] + matr[2][2] * a[2][2];
    }
}

```

```

    } else if(a.matr.size() == 4){
        R[0][0] = matr[0][0] * a[0][0] + matr[0][1] * a[1][0] + matr[0][2] * a[2]
[0] + matr[0][3] * a[3][0];
        R[0][1] = matr[0][0] * a[0][1] + matr[0][1] * a[1][1] + matr[0][2] * a[2]
[1] + matr[0][3] * a[3][1];
        R[0][2] = matr[0][0] * a[0][2] + matr[0][1] * a[1][2] + matr[0][2] * a[2]
[2] + matr[0][3] * a[3][2];
        R[0][3] = matr[0][0] * a[0][3] + matr[0][1] * a[1][3] + matr[0][2] * a[2]
[3] + matr[0][3] * a[3][3];

        R[1][0] = matr[1][0] * a[0][0] + matr[1][1] * a[1][0] + matr[1][2] * a[2]
[0] + matr[1][3] * a[3][0];
        R[1][1] = matr[1][0] * a[0][1] + matr[1][1] * a[1][1] + matr[1][2] * a[2]
[1] + matr[1][3] * a[3][1];
        R[1][2] = matr[1][0] * a[0][2] + matr[1][1] * a[1][2] + matr[1][2] * a[2]
[2] + matr[1][3] * a[3][2];
        R[1][3] = matr[1][0] * a[0][3] + matr[1][1] * a[1][3] + matr[1][2] * a[2]
[3] + matr[1][3] * a[3][3];

        R[2][0] = matr[2][0] * a[0][0] + matr[2][1] * a[1][0] + matr[2][2] * a[2]
[0] + matr[2][3] * a[3][0];
        R[2][1] = matr[2][0] * a[0][1] + matr[2][1] * a[1][1] + matr[2][2] * a[2]
[1] + matr[2][3] * a[3][1];
        R[2][2] = matr[2][0] * a[0][2] + matr[2][1] * a[1][2] + matr[2][2] * a[2]
[2] + matr[2][3] * a[3][2];
        R[2][3] = matr[2][0] * a[0][3] + matr[2][1] * a[1][3] + matr[2][2] * a[2]
[3] + matr[2][3] * a[3][3];

        R[3][0] = matr[3][0] * a[0][0] + matr[3][1] * a[1][0] + matr[3][2] * a[2]
[0] + matr[3][3] * a[3][0];
        R[3][1] = matr[3][0] * a[0][1] + matr[3][1] * a[1][1] + matr[3][2] * a[2]
[1] + matr[3][3] * a[3][1];
        R[3][2] = matr[3][0] * a[0][2] + matr[3][1] * a[1][2] + matr[3][2] * a[2]
[2] + matr[3][3] * a[3][2];
        R[3][3] = matr[3][0] * a[0][3] + matr[3][1] * a[1][3] + matr[3][2] * a[2]
[3] + matr[3][3] * a[3][3];
    } else{
        assert(false);
    }
    return R;
}

```

```

std::vector<double> Matrix::operator*(std::vector<double> point) {
    if(point.size() == 3){
        double tmp[3] = {point[0], point[1], point[2]};
        point[0] = matr[0][0] * tmp[0] + matr[0][1] * tmp[1] + matr[0][2] *
tmp[2];
        point[1] = matr[1][0] * tmp[0] + matr[1][1] * tmp[1] + matr[1][2] *
tmp[2];
        point[2] = matr[2][0] * tmp[0] + matr[2][1] * tmp[1] + matr[2][2] *
tmp[2];
    } else if (point.size() == 4){
        double tmp[4] = {point[0], point[1], point[2], point[3]};
        point[0] = matr[0][0] * tmp[0] + matr[0][1] * tmp[1] + matr[0][2] * tmp[2]
+ matr[0][3] * tmp[3];
        point[1] = matr[1][0] * tmp[0] + matr[1][1] * tmp[1] + matr[1][2] * tmp[2]
+ matr[1][3] * tmp[3];
        point[2] = matr[2][0] * tmp[0] + matr[2][1] * tmp[1] + matr[2][2] * tmp[2]
+ matr[2][3] * tmp[3];
        point[3] = matr[3][0] * tmp[0] + matr[3][1] * tmp[1] + matr[3][2] * tmp[2]
+ matr[3][3] * tmp[3];
    } else{
        assert(false);
    }
}

```

```

    }

    return point;
}

QPoint Matrix::operator*(QPoint point) {
    std::vector<double> inner = {double(point.x()), double(point.y()), 1.0};
    inner = (*this)*inner;
    return QPoint(inner[0], inner[1]);
}

Matrix RotateMatrixOX(double angle) {
    Matrix A;
    A[0][0] = 1;      A[0][1] = 0;      A[0][2] = 0;      A[0][3] = 0;
    A[1][0] = 0;      A[1][1] = cos(angle); A[1][2] = -sin(angle); A[1][3] = 0;
    A[2][0] = 0;      A[2][1] = sin(angle); A[2][2] = cos(angle); A[2][3] = 0;
    A[3][0] = 0;      A[3][1] = 0;      A[3][2] = 0;      A[3][3] = 1;

    return A;
}

Matrix RotateMatrixOY(double angle) {
    Matrix A;
    A[0][0] = cos(angle); A[0][1] = 0;      A[0][2] = -sin(angle); A[0][3]
= 0;
    A[1][0] = 0;          A[1][1] = 1;      A[1][2] = 0;          A[1][3]
= 0;
    A[2][0] = sin(angle); A[2][1] = 0;      A[2][2] = cos(angle); A[2][3]
= 0;
    A[3][0] = 0;          A[3][1] = 0;      A[3][2] = 0;          A[3][3]
= 1;

    return A;
}

Matrix RotateMatrixOZ(double angle){
    Matrix A;

    A[0][0] = cos(angle); A[0][1] = sin(angle); A[0][2] = 0;      A[0][3] = 0;
    A[1][0] = -sin(angle); A[1][1] = cos(angle); A[1][2] = 0;      A[1][3] = 0;
    A[2][0] = 0;          A[2][1] = 0;          A[2][2] = 1;      A[2][3] = 0;
    A[3][0] = 0;          A[3][1] = 0;          A[3][2] = 0;      A[3][3] = 1;

    return A;
}

Matrix ScaleMatrix(double kx, double ky, double kz) {
    Matrix A;

    A[0][0] = kx;      A[0][1] = 0;      A[0][2] = 0;      A[0][3] = 0;
    A[1][0] = 0;      A[1][1] = ky;      A[1][2] = 0;      A[1][3] = 0;
    A[2][0] = 0;      A[2][1] = 0;      A[2][2] = kz;      A[2][3] = 0;
    A[3][0] = 0;      A[3][1] = 0;      A[3][2] = 0;      A[3][3] = 1;

    return A;
}

Matrix ReflecMatrixOX() {
    Matrix A;
    A[0][0] = 1;      A[0][1] = 0;      A[0][2] = 0;      A[0][3] = 0;

```

```

    A[1][0] = 0;    A[1][1] = 1;    A[1][2] = 0;    A[1][3] = 0;
    A[2][0] = 0;    A[2][1] = 0;    A[2][2] = -1;   A[2][3] = 0;
    A[3][0] = 0;    A[3][1] = 0;    A[3][2] = 0;    A[3][3] = 1;

    return A;
}

Matrix ReflecMatrixOY() {
    Matrix A;
    A[0][0] = 1;    A[0][1] = 0;    A[0][2] = 0;    A[0][3] = 0;
    A[1][0] = 0;    A[1][1] = -1;   A[1][2] = 0;    A[1][3] = 0;
    A[2][0] = 0;    A[2][1] = 0;    A[2][2] = 1;    A[2][3] = 0;
    A[3][0] = 0;    A[3][1] = 0;    A[3][2] = 0;    A[3][3] = 1;

    return A;
}

Matrix ReflecMatrixOZ() {
    Matrix A;
    A[0][0] = -1;   A[0][1] = 0;    A[0][2] = 0;    A[0][3] = 0;
    A[1][0] = 0;    A[1][1] = 1;    A[1][2] = 0;    A[1][3] = 0;
    A[2][0] = 0;    A[2][1] = 0;    A[2][2] = 1;    A[2][3] = 0;
    A[3][0] = 0;    A[3][1] = 0;    A[3][2] = 0;    A[3][3] = 1;

    return A;
}

Matrix TranslateMatrix2D(double dx, double dy) {
    Matrix A;
    A[0][0] = 1;    A[0][1] = 0;    A[0][2] = dx;
    A[1][0] = 0;    A[1][1] = 1;    A[1][2] = dy;
    A[2][0] = 0;    A[2][1] = 0;    A[2][2] = 1;

    return A;
}

Matrix TranslateMatrix(double dx, double dy, double dz) {
    Matrix A;
    A[0][0] = 1;    A[0][1] = 0;    A[0][2] = 0;    A[0][3] = dx;
    A[1][0] = 0;    A[1][1] = 1;    A[1][2] = 0;    A[1][3] = dy;
    A[2][0] = 0;    A[2][1] = 0;    A[2][2] = 1;    A[2][3] = dz;
    A[3][0] = 0;    A[3][1] = 0;    A[3][2] = 0;    A[3][3] = 1;

    return A;
}

```

Листинг №6: «figure.h»:

```

#ifndef FIGURE_H
#define FIGURE_H

#include <QPainter>
#include <vector>
#include "matrix.h"

struct figure{
    std::vector<QPolygon> inner;
    figure();

    friend figure operator*(Matrix &a, figure const &f){

```

```

        figure R;
        R.inner = f.inner;
        for(auto &QP : R.inner)
            QP = a * QP;
        return R;
    }
    void apply(std::vector<Matrix> const &M);

    friend figure operator*(figure const &f, Matrix &a){
        return a*f;
    }
};

struct bird_st : public figure{
    bird_st(){inner.resize(6);}
    QPolygon &tail(){return inner[0];}
    QPolygon &body(){return inner[1];}
    QPolygon &closeWing(){return inner[2];}
    QPolygon &farWing(){return inner[3];}
    QPolygon &head(){return inner[4];}
    QPolygon &beak(){return inner[5];}
};

#endif // FIGURE_H

```

Листинг №7: «figure.cpp»:

```

#include "figure.h"
figure::figure()
{

}

void figure::apply(std::vector<Matrix> const &M){
    for(auto &poly : inner){
        for(auto matr : M){
            poly = poly * matr;
        }
    }
}

```

Вывод:

В ходе выполнения лабораторной работы мы изучили аффинные преобразования на плоскости и создали графическое приложение с помощью GDI в среде Qt Creator.