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 №4

по дисциплине: Компьютерная графика
тема: «Аффинные преобразования в пространстве»

Выполнил: ст. группы ПВ-201
Морозов Данила Александрович

Проверил:
Осипов Олег Васильевич

Белгород 2022 г.

Лабораторная работа №4
«Аффинные преобразования в пространстве»

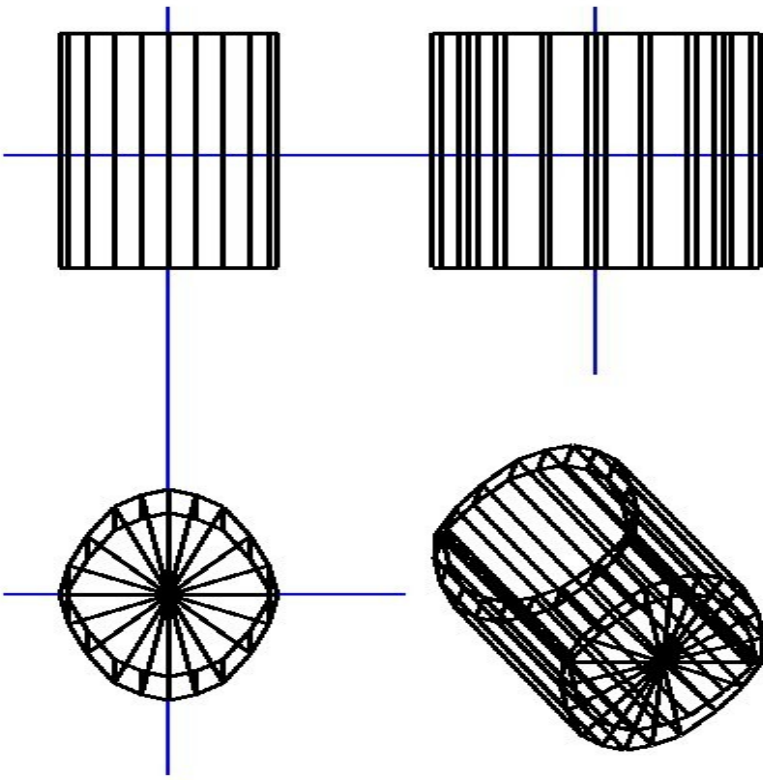
Цель работы:

Получение навыков использования аффинных преобразований в пространстве и создание графического приложения с использованием GDI в среде Qt Creator для визуализации простейших трехмерных объектов

Задания к работе:

1. Окно поделить на 4 части одинаковые части:
   a. На верхней левой части должна отображаться фронтальная проекция (вид спереди);
   b. Правая верхняя часть – профильная проекция (вид сбоку);
   c. Левая нижняя часть должна отображать вид сверху (горизонтальную проекцию);
   d. На правой нижней части должна отображаться проекция, вид которой выбирает пользователь: центральная, косоугольная кабинетная, косоугольная свободная, параллельная, ортографическая.
2. Предусмотреть возможность выбора вида проекции пользователем, например, с помощью элемента QComboBox.
3. Первые три проекции отобразить без перспективных искажений. Для четвёртой предусмотреть возможность отдаления/приближения и поворота фигуры клавишами или с помощью мыши.
4. Во всех проекциях нужно отобразить на экране только каркас фигуры, т.е. только рёбра объектов. Трёхмерные объекты хранить в памяти как массив многоугольников (не массив отрезков).

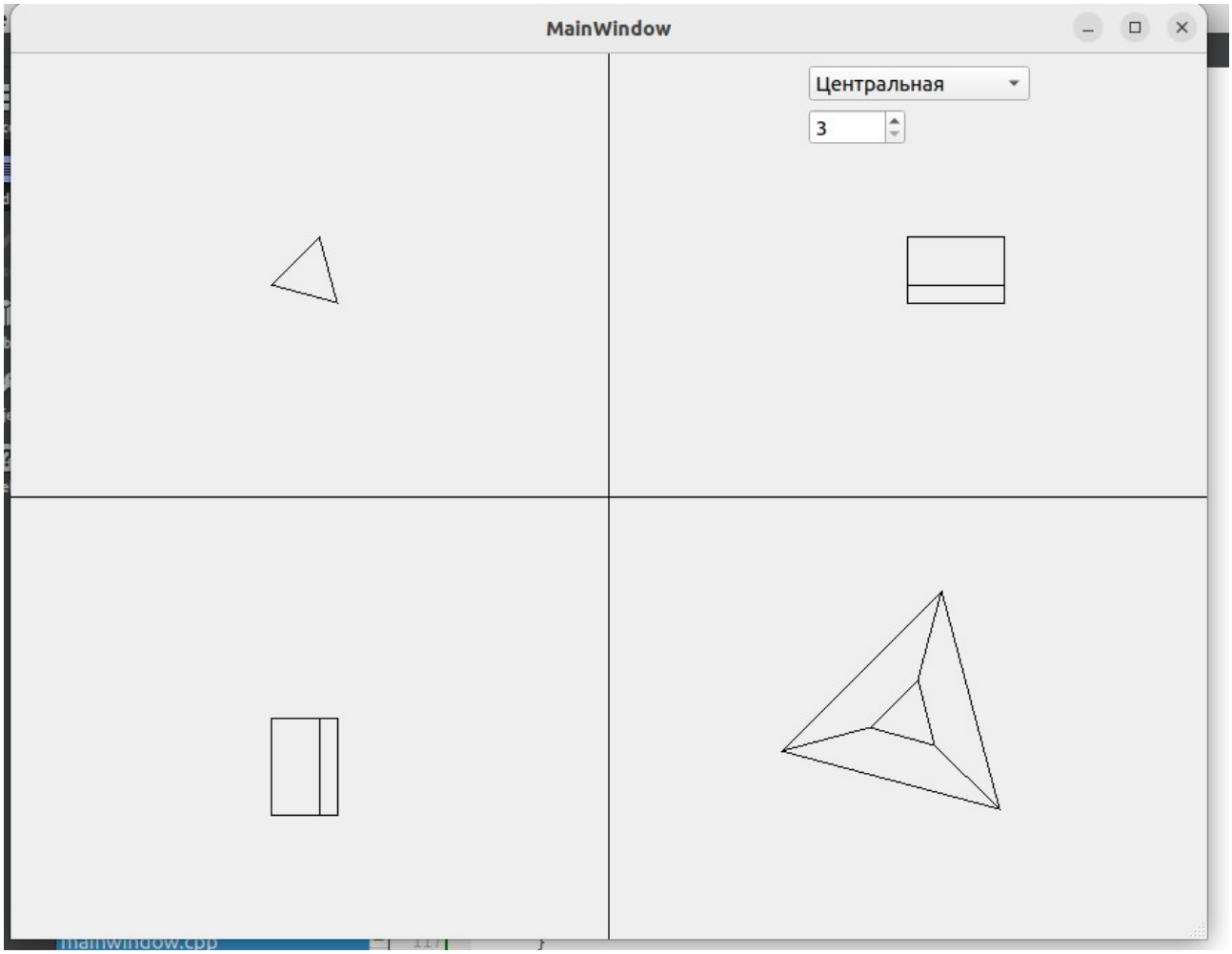
Вариант №9:

9	 Изменять количество граней при движении колесика мыши (делать стакан более круглым)

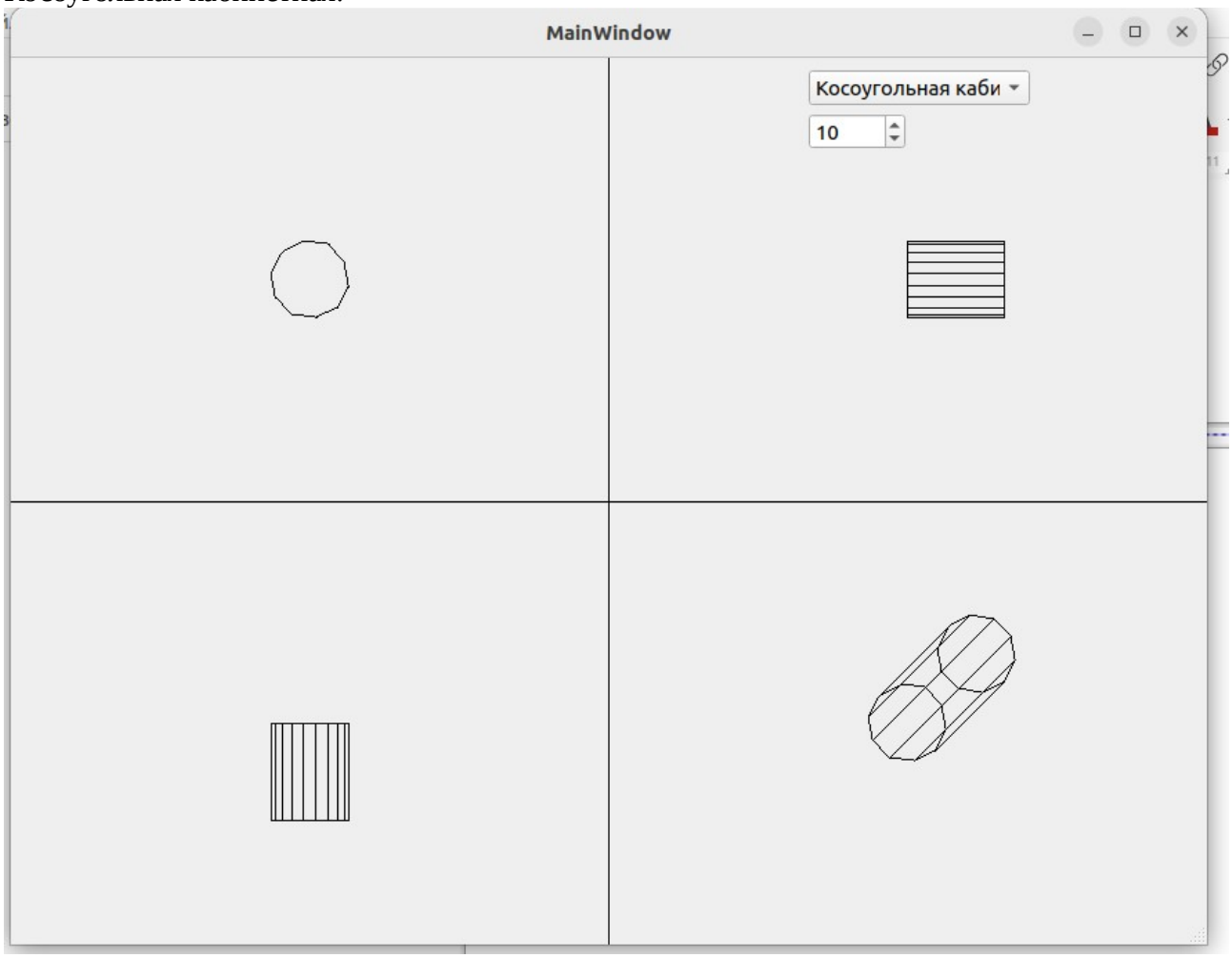
Выполнение:

Снимки экрана:

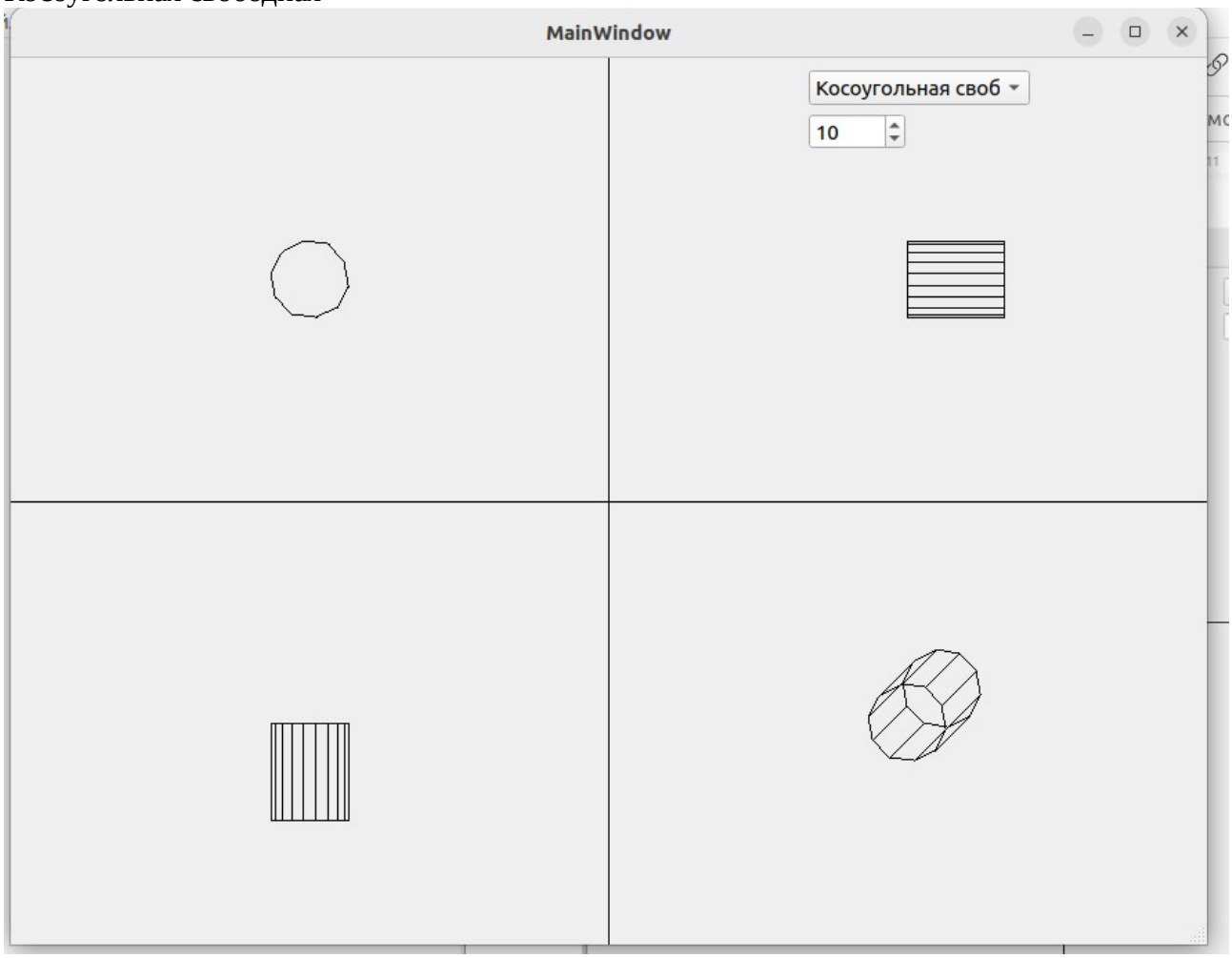
Центральная



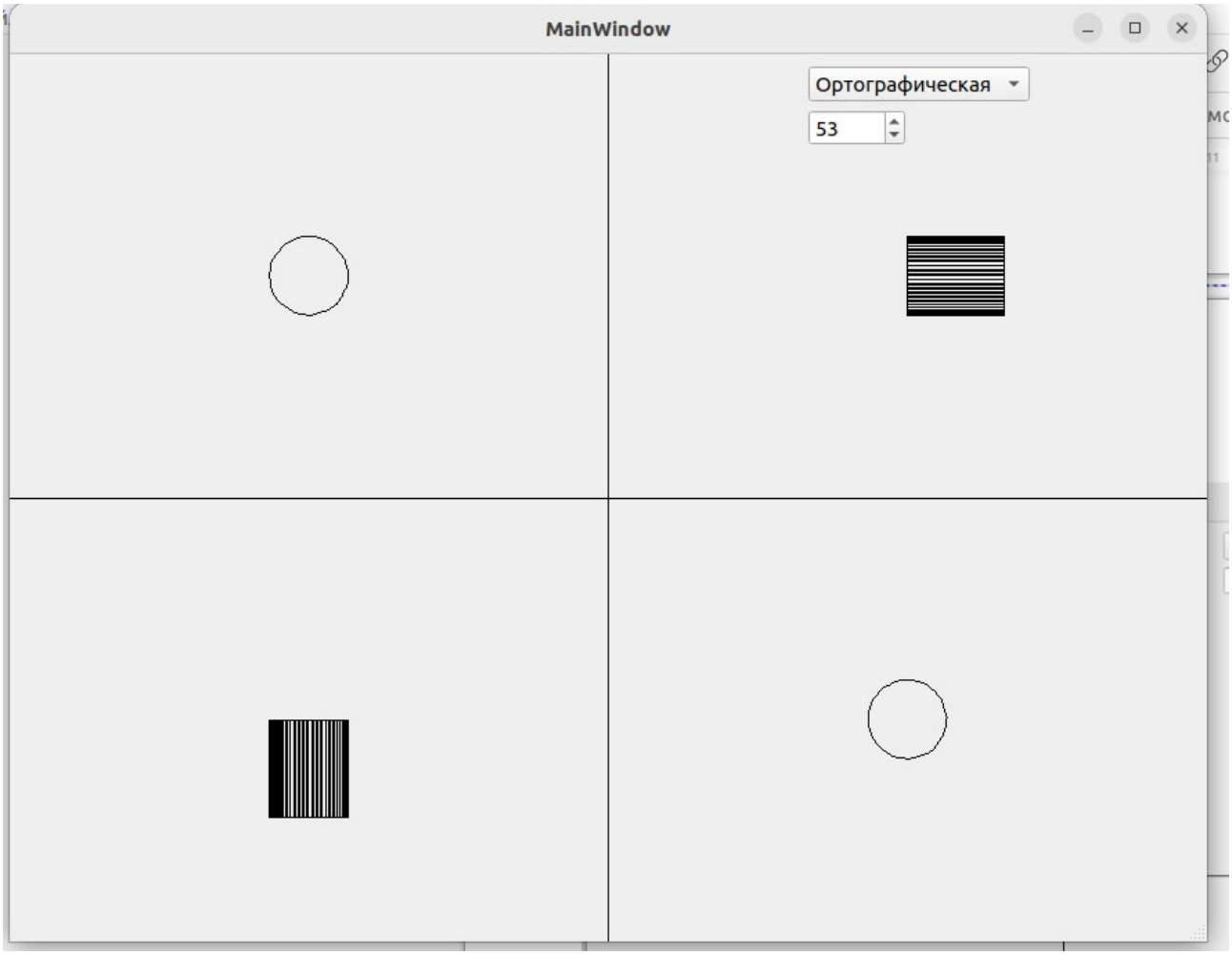
Косоугольная кабинетная:



Косоугольная свободная



Ортографическая:



Формулы:

Указанное пользователем количество точек располагается равномерно по окружности — тем самым мы получаем переднюю плоскость. Затем мы её копируем и переносим назад — получаем заднюю плоскость фигуры. Затем между каждой четверкой точек передней и задней плоскости формируются боковые плоскости.
Плоскости содержат в себе точки, фигуры содержат в себе плоскости. Каждая плоскость способна вернуть соответствующий себе 2D многоугольник для правильного отображения на экран, каждую из названных сущностей можно сразу умножить на матрицу преобразования. Например, вся фигура сразу может быть повернута умножением на матрицу поворота, нет необходимости вручную умножать каждую плоскость и т. п.
Далее показаны только переносы и начальные повороты.

Фронтальная проекция (вид спереди):

Matrix **m** = TranslateMatrix(width() * 0.25, height() * 0.25, 0);

Профильная проекция (вид сбоку):

Matrix **m** = TranslateMatrix(width() * 0.75, height * 0.25, 0) * RotateMatrixOY(qDegreesToRadians(90.0));

Горизонтальная проекция (вид сверху):

Matrix **m** = TranslateMatrix(width() * 0.25, height * 0.75, 0) * RotateMatrixOX(qDegreesToRadians(90.0));

Ортографическая / Центральная / Косоугольная кабинетная / Косоугольная свободная (объемный вид модели):
Matrix **m** = current * TranslateMatrix(width() * 0.75, height() * 0.75, 0) * RotateMatrixOX(qDegreesToRadians(45.0)) * RotateMatrixOY(qDegreesToRadians(45.0));

- OrthographicProc() – для ортографической проекции
- CentralProc(300); *// 300 - отдаленность точки C (см. лекцию) – для центральной проекции*
- ObliqueProc(-cos(qDegreesToRadians(45.0)), cos(qDegreesToRadians(45.0))); - для косоугольной кабинетной проекции
- ObliqueProc(-cos(qDegreesToRadians(45.0)) / 2, cos(qDegreesToRadians(45.0)) / 2); - для косоугольной свободной проекции

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 "matrix.h"

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    void paintEvent(QPaintEvent* event);
    void wheelEvent(QWheelEvent *wheelevent);
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    ~MainWindow();

private slots:
    void on_comboBox_currentIndexChanged(int index);

    void on_spinBox_valueChanged(int arg1);

private:
    Ui::MainWindow *ui;
    Matrix rotate;
```

```
    Matrix scale;
    int projection = 0;
    int N;
    double scaleV = 1;
    double angleOX;
    double angleOY;
    double angleOZ;
    double mx;
    double my;
};
#endif // MAINWINDOW_H
```

Листинг №3: «mainwindow.cpp»:

```
#include "mainwindow.h"
#include "./ui_mainwindow.h"

#include <QPainter>
#include <QWheelEvent>

#include "plane.h"
#include "figure.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    ui->comboBox->addItem("Центральная");
    ui->comboBox->addItem("Косоугольная кабинетная");
    ui->comboBox->addItem("Косоугольная свободная");
    ui->comboBox->addItem("Ортографическая");
    N = ui->spinBox->value();
}

void MainWindow::paintEvent(QPaintEvent* event) {
    QPainter painter(this);
    scale = ScaleMatrix(scaleV, scaleV, scaleV);
    rotate = RotateMatrixOX(angleOX) * RotateMatrixOY(angleOY) *
RotateMatrixOZ(angleOZ);
    int cX = width() / 2;
    int cY = height() / 2;
    painter.drawLine(cX, 0, cX, height());
    painter.drawLine(0, cY, width(), cY);

    std::vector<double> radius = {20, 20, 0, 1};
    std::vector<Point4D> points;

    for(int i = 0; i < N; i++){
        points.push_back(radius * RotateMatrixOZ(qDegreesToRadians(360.0/N *
i)));
    }
    Plane FrontPlane;
    Plane BackPlane;
    std::vector<Plane> sidePlanes;
    for(auto p : points){
        FrontPlane.pushPoint(p);
        auto p2 = p * TranslateMatrix(0, 0, -70);
        BackPlane.pushPoint(p2);
    }
    for(int i = 0; i < N; i++){
        Plane pt;
```

```
        pt.pushPoint({FrontPlane[i], FrontPlane[(i+1)%N], BackPlane[(i+1)%N],
BackPlane[i]});
        sidePlanes.push_back(pt);
    }

    Figure mainFigure;
    mainFigure.pushBack(FrontPlane);
    mainFigure.pushBack(BackPlane);
    for(auto p : sidePlanes)
        mainFigure.pushBack(p);
    auto leftUpper = mainFigure * TranslateMatrix(width() * 0.25, height() *
0.25, 0);
    auto rightUpper = mainFigure * RotateMatrixOY(qDegreesToRadians(90.0)) *
TranslateMatrix(width() * 0.75, height() * 0.25, 0);
    auto leftBottom = mainFigure * RotateMatrixOX(qDegreesToRadians(90.0)) *
TranslateMatrix(width() * 0.25, height() * 0.75, 0);
    auto rightBottom = mainFigure;
    switch(projection){
        case 0:
            rightBottom = rightBottom * rotate * CentralProc(100.0) * scale *
TranslateMatrix(width() * 0.75, height() * 0.75, 0);
        break;
        case 1:
            rightBottom = rightBottom * rotate * ObliqueProc(-
cos(qDegreesToRadians(45.0)), cos(qDegreesToRadians(45.0))) * scale *
TranslateMatrix(width() * 0.75, height() * 0.75, 0);
        break;
        case 2:
            rightBottom = rightBottom * rotate * ObliqueProc(-
cos(qDegreesToRadians(45.0)) / 2.0, cos(qDegreesToRadians(45.0)) / 2.0) * scale
* TranslateMatrix(width() * 0.75, height() * 0.75, 0);
        break;
        case 3:
            rightBottom = rightBottom * rotate * OrthographicProc() * scale *
TranslateMatrix(width() * 0.75, height() * 0.75, 0);
        break;
    }

    leftUpper.draw(painter);
    rightUpper.draw(painter);
    leftBottom.draw(painter);
    rightBottom.draw(painter);
}

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

void MainWindow::on_comboBox_currentIndexChanged(int index)
{
    projection = index;
    angleOY = 0;
    angleOX = 0;
    scaleV = 1;
    this->update();
}

void MainWindow::on_spinBox_valueChanged(int arg1)
{
    N = arg1;
    this->update();
```

```
}

void MainWindow::wheelEvent(QWheelEvent *wheelevent) {
  int p = wheelevent->pixelDelta().y();
  if (p > 0){
      if(QApplication::keyboardModifiers() == Qt::ControlModifier){
          scaleV += 0.02;
      } else{
          ui->spinBox->setValue(ui->spinBox->value() + 1);
      }
  } else{
      if(QApplication::keyboardModifiers() == Qt::ControlModifier){
          scaleV -= 0.02;
      } else{
          ui->spinBox->setValue(ui->spinBox->value() - 1);
      }
  }
  this->update();
}

void MainWindow::mousePressEvent(QMouseEvent *event) {
  if (event->buttons() & Qt::LeftButton && rect().contains(event->pos())) {
    mx = event->x();
    my = event->y();
  }
}

void MainWindow::mouseMoveEvent(QMouseEvent *event) {
  double offsetX = event->x() - mx;
  double offsetY = event->y() - my;
  angleOY += offsetX / 100;
  angleOX += offsetY / 100;
  mx = event->x();
  my = event->y();
  this->update();
}
```

Листинг №4: «matrix.hpp»:

```
#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 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;
}

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

  return A;
}

Matrix CentralProc(double c) {
  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] = 0;    A[2][3] = 0;
  A[3][0] = 0;    A[3][1] = 0;    A[3][2] = 1 / c;    A[3][3] = 1;

  return A;
}

Matrix ObliqueProc(double a, double b) {
  Matrix A;
  A[0][0] = 1;    A[0][1] = 0;    A[0][2] = a;    A[0][3] = 0;
  A[1][0] = 0;    A[1][1] = 1;    A[1][2] = b;    A[1][3] = 0;
  A[2][0] = 0;    A[2][1] = 0;    A[2][2] = 0;    A[2][3] = 0;
  A[3][0] = 0;    A[3][1] = 0;    A[3][2] = 0;    A[3][3] = 1;

  return A;
}
```

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 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;
}

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

  return A;
}

Matrix CentralProc(double c) {
  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] = 0;    A[2][3] = 0;
  A[3][0] = 0;    A[3][1] = 0;    A[3][2] = 1 / c;    A[3][3] = 1;

  return A;
}

Matrix ObliqueProc(double a, double b) {
  Matrix A;
  A[0][0] = 1;    A[0][1] = 0;    A[0][2] = a;    A[0][3] = 0;
  A[1][0] = 0;    A[1][1] = 1;    A[1][2] = b;    A[1][3] = 0;
  A[2][0] = 0;    A[2][1] = 0;    A[2][2] = 0;    A[2][3] = 0;
  A[3][0] = 0;    A[3][1] = 0;    A[3][2] = 0;    A[3][3] = 1;

  return A;
}
```

Листинг №6: «plane.hpp»:

```
#ifndef PLANE_H
#define PLANE_H

#include <vector>
#include <QPainter>
#include <initializer_list>

#include "matrix.h"

using Point4D = std::vector<double>;
using Vector4D = std::vector<double>;
```

```
class Plane
{
public:
    Plane();
    void pushPoint(std::initializer_list<Point4D> points);
    void pushPoint(Point4D point);
    QPolygon getPoly(void);

    std::vector<double> &operator[](size_t i);

    friend Plane operator*(Plane plane, Matrix a){
        Plane r = plane;
        for(auto &p : r.inner){
            p = a * p;
        }
        return r;
    }
    friend Plane operator*(Matrix a, Plane plane){
        return plane * a;
    }
private:
    std::vector<Point4D> inner;
};

#endif // FIGURE_H
```

Листинг №7: «plane.cpp»:

```
#include "plane.h"
Plane::Plane()
{

}

void Plane::pushPoint(std::initializer_list<Point4D> points){
    for(auto p : points){
        inner.push_back(p);
    }
}

std::vector<double> &Plane::operator[](size_t i){
    return inner[i];
}

void Plane::pushPoint(Point4D point){
    inner.push_back(point);
}

QPolygon Plane::getPoly(void){
    QPolygon r;
    for(auto p : inner){
        r << QPoint{int(p[0]/p[3]), int(p[1]/p[3])};
    }
    return r;
}
```

Листинг №8: «figure.h»:

```
#ifndef FIGURE_H
```

```
#define FIGURE_H

#include <vector>

#include <QPainter>

#include "plane.h"
#include "matrix.h"

class Figure
{
public:
    Figure();
    friend Figure operator*(Figure fig, Matrix a){
        Figure f = fig;
        for(int i = 0; i < f.planes.size(); i++){
            f.planes[i] = f.planes[i] * a;
        }
        return f;
    }
    friend Figure operator*(Matrix a, Figure fig){
        return fig * a;
    }
    void pushBack(Plane p);
    void draw(QPainter &painter);
    std::vector<Plane> planes;
};

#endif // FIGURE_H
```

Листинг №9: «figure.cpp»:

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

}

void Figure::pushBack(Plane p){
    planes.push_back(p);
}

void Figure::draw(QPainter &painter){
    for(auto p : planes){
        painter.drawPolygon(p.getPoly());
    }
}
```

Вывод:

В ходе выполнения лабораторной работы мы получили навыки использования аффинных преобразований в пространстве и создания графического приложения с использованием GDI в среде Qt Creator для визуализации простейших трехмерных объектов.