Література

1. Пєтух А.М. Інтерполяція в задачах контурного формоутворення : [монографія] / Пєтух А.М., Обідник Д.Т., Романюк О.Н. – Вінниця : УНІВЕРСУМ-Вінниця, 2007. – 103 с.
2. Романюк О. Н. Особливості гексагональної моделі піксела / О. В. Мельник, О. Н. Романюк // Вимірювальна та обчислювальна техніка в технологічних процесах : міжнародний науково-технічний журнал. – Хмельницький : ХНУ, 2014. – № 1 (46). – С. 91–95.
3. Романюк О.Н. Використання методу оцінювальної функції для задач антиаліайзингу / О. Н. Романюк, М. С. Курінний // Сборник научных трудов НГУ. – Дніпропетровськ : НГУ, 2004. – № 19. – Том 2. – С. 200–208.
4. Wuthrich C. A. AnAlgorithmic Comparison Between Squareand Hexagonal-based Grid / C. A. Wuthrich, P. Stucki // CVGIP: Graphical Modelsand Image Processing. – 1999. – Vol. 53. – P. 324–339.

Отримана/Received : 3.5.2017 р. Надрукована/Printed :9.6.2017 р.
Рецензент: д. т. н., проф. Павлов С. В.

УДК: 004.932

Н.С. СВИРНЕВСКИЙ
Хмельницкий национальный университет

РАЗРАБОТКА АРХИТЕКТУРЫ ПРОГРАММЫ ДЛЯ РЕШЕНИЯ ЗАДАЧ, СВЯЗАННЫХ С АФФИННЫМИ ПРЕОБРАЗОВАНИЯМИ ПРОСТРАНСТВА

В статье описывается процесс создания архитектуры программы для решения геометрических задач на основе математического аппарата аффинных преобразований, отталкиваясь от элементарных графических возможностей, предоставляемых C++WinAPI приложением. Программа тестирована на примере решения задачи сканирования рупорной антенны. Продемонстрировано многообразие возможностей геометрических преобразований и эффективность управления ими.

Ключевые слова: программа, архитектура, аффинные преобразования, матрица, антенна.

N.S. SVIRNEVSKYI
Khmelnytskyi National University

DEVELOPMENT OF POGRAM ARCHITECTURE FOR SOLVING THE PROBLEMS ASSOCIATED WITH AFFINE TRANSFORMATION OF THE SPACE

This article describes creation of program architecture for solving of geometric problems based on the mathematical apparatus of affine transformation, starting from the basic graphics capabilities provided by C ++ WinAPI application. The program was tested on an example of solving the problem of scanning horn antenna. It demonstrated the variety of possible geometric transformations and their management efficiency.

Keywords: program, architecture, affine transformation, matrix, antenna.

Введение

Архитектура программы – это ее организация, воплощенная в компонентах (модулях), объединенных для выполнения определенной функции. Различают стандартные модули (библиотеки), входящие в язык программирования и пользовательские модули (программы), предназначенные для упрощения работы программистов.

Эта статья посвящена разработке программы, предназначенной для решения геометрических задач на основе аффинных преобразований пространства. К таким задачам, прежде всего, относятся – создание геометрических объектов, визуализация и обработка изображений, включая создание диалогов для поступления исходной информации и управления результатами.

Анализ исследований и публикаций

Известно большое количество стандартных модулей, обеспечивающих создание и обработку изображений [1, 2], решение многих задач в которых реализуется через аффинные преобразования [3]. При этом в них ограничены возможности для решения ряда нестандартных задач [4, 5], так как аффинные преобразования инкапсулированы в функциях и командах и предназначены лишь для конкретных действий по визуализации и обработке изображений.

На сегодняшний день, несмотря на многообразие стандартных графических библиотек и программных систем, по-прежнему, остается актуальной проблема создания эффективных пользовательских программных модулей для решения задач универсального характера, поскольку многие научные разработки выполняются именно с помощью их.

Формулирование цели

Разработать архитектуру программы для решения задач, связанных с аффинными преобразованиями пространства. Описать процесс создания программы на базе математического аппарата аффинных преобразований 2D и 3D пространства, отталкиваясь от элементарных графических

возможностей, предоставляемых в оконной (физической) системе координат.

Изложение основного материала

Хорошая архитектура программы это, прежде всего, модульная архитектура. Деление на модули лучше всего производить исходя из тех задач, которые решает программа. Основная задача разбивается на составляющие ее подзадачи, которые могут выполняться независимо друг от друга. Каждый модуль должен отвечать за решение какой-то подзадачи и выполнять соответствующую ей функцию.

Изложение теории будет идти параллельно с практической реализацией. Поэтому, для начала нам потребуется шаблон приложения. Остановим свой выбор на C++WinAPI приложении, как наиболее оптимальном для разработки графических приложений, поскольку оно не требовательно к машинным ресурсам и предоставляет достаточно возможностей для взаимодействия с пользователем.

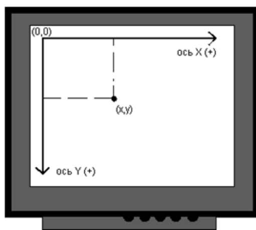
WinAPI приложение [6] является в своей основе процедурным приложением. Оно изначально состоит из одного файла (main.cpp) и содержит два основных элемента – функции winMain() и WndProc.

Функция winMain() составляет основу любого приложения. Она служит как бы точкой входа в приложение и отвечает за следующее:

- начальную инициализацию приложения;
- создание и регистрацию объекта класса окна приложения;
- создание и инициализацию цикла обработки событий.

Обработанное в бесконечном цикле событие переправляется (опосредовано через Windows) оконной функции WndProc. События в ней идентифицируются именем константы (WM_PAINT, WM_DESTROY и др.). Рисование осуществляется при помощи объектов типа HDC (дескриптор контекста устройства).

Нарисуем треугольник, задав координаты его вершин в оконной (физической) системе координат (рис.1). Начало этой системы координат располагается в левом верхнем углу экрана. Ось X направлена слева направо, ось Y – сверху вниз. В качестве единицы длины в этой системе координат используется пиксел.



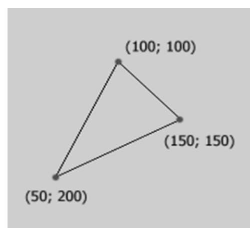
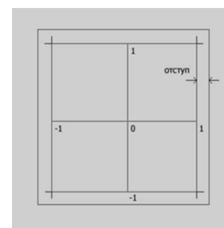


Рис. 1. Определение рисунка в оконной СК



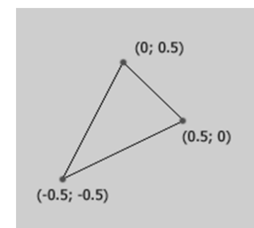


Рис. 2. Определение рисунка в логической СК

Для этого создадим новый файл draw.cpp:

```
#include <windows.h>
void Draw(HDC hdc){MoveToEx(hdc, 100, 100, NULL);
      LineTo(hdc, 150, 150); LineTo(hdc, 50, 200); LineTo(hdc, 100, 100);}
```

И файл draw.h:

```
void Draw(HDC hdc);
```

В файле main.cpp добавим в начало:

```
#include "draw.h"
```

А в событии WM_PAINT между заполнением области фоновым цветом и выводом изображения на основной контекст, появляется вызов процедуры рисования:

```
Draw(hCmpDC);
```

Оконная СК неудобна для пользователя из-за непривычного расположения осей (ось y направлена вниз), задания координат в пикселях и др. Устраним эти недостатки, используя логическую СК (рис. 2), в которой:

- центр координат перенесен из левого верхнего угла в центр экрана;
- направление оси Y меняется на противоположное;
- координаты (от -1 до +1) соотнесены с шириной и высотой окна;
- введен отступ от края окна (margin).

Преобразования из логической системы координат в оконную осуществляются при помощи зависимостей:

$$X_Window = MARGIN + (1.0/2)*(X_Log + 1)*(Width - 2 * MARGIN);$$

$$Y_Window = MARGIN + (-1.0/2)*(Y_Log - 1)*(Height - 2 * MARGIN);$$

Ниже приводится новый код модуля draw.cpp:

```
#include <windows.h>
int Width, Height;
const int MARGIN = 10;
void SetWindowSize(int _Width, int _Height){Width = _Width; Height = _Height;}
int Tx(double X_Log){int X_Window;
     X_Window = MARGIN + (1.0 / 2) * (X_Log + 1) * (Width - 2 * MARGIN); return X_Window;}
int Ty(double Y_Log){int Y_Window;
Y_Window = MARGIN +(-1.0 / 2)*(Y_Log - 1)*(Height - 2* MARGIN); return Y_Window;}
void Draw(HDC hdc){
```

```
	MoveToEx(hdc, Tx(0.0), Ty(0.5), NULL);LineTo(hdc, Tx(0.5), Ty(0.0));
LineTo(hdc, Tx(-0.5), Ty(-0.5));LineTo(hdc, Tx(0.0), Ty(0.5)); }
```

В draw.h добавляется объявление новой функции:

```
void Draw(HDC hdc);
void SetWindowSize(int _Width, int _Height);
```

В модуль main.cpp добавляем обработчик события WM_SIZE и WM_ERASEBKGND:

```
case WM_SIZE:
GetClientRect(hWnd, &Rect);
 SetWindowSize(Rect.right - Rect.left, Rect.bottom - Rect.top);
     break;
case WM_ERASEBKGND:return 1; break;
```

Теперь, при изменении размеров окна треугольник подстраивается под новые размеры. При изменении размеров окна происходит перерисовка изображения, поскольку событие WM_SIZE автоматически вызывает событие WM_PAINT.

Мы нарисовали треугольник в логической системе координат. Теперь обеспечим возможность поворота треугольника при каждом нажатии клавиш ← или →. Для пересчета положения каждой вершины треугольника используем преобразование координат при повороте. Рассмотрим произвольный вектор r, задающий некоторую точку в системе координат xy (рис. 3).

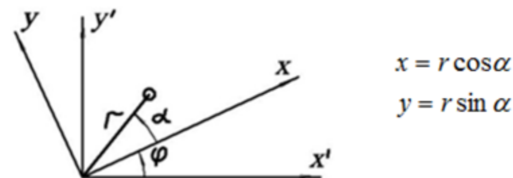


Рис. 3. Преобразование координат при повороте

При повороте на угол φ координаты точки запишутся в виде:

$$x' = r\cos(\alpha+\varphi) = r(\cos\alpha\cos\varphi - \sin\alpha\sin\varphi) = x\cos\varphi - y\sin\varphi$$

$$y' = r\sin(\alpha+\varphi) = r(\sin\alpha\cos\varphi + \cos\alpha\sin\varphi) = x\sin\varphi + y\cos\varphi$$

Выделим из этих уравнений 2 пары зависимостей – для пересчета координат точек в новой СК (1) и для обновления значений косинуса и синуса угла (2):

$$\begin{aligned} x' &= x\cos\varphi - y\sin\varphi \\ y' &= x\sin\varphi + y\cos\varphi \end{aligned} \qquad (1)$$

$$\begin{aligned} \cos(\alpha+\varphi) &= \cos\alpha\cos\varphi - \sin\alpha\sin\varphi \\ \sin(\alpha+\varphi) &= \sin\alpha\cos\varphi + \cos\alpha\sin\varphi \end{aligned}. \qquad (2)$$

Для решения задачи пошагового вращения треугольника необходимо сохранять параметры предыдущего положения треугольника. В качестве таких параметров могут использоваться предыдущие координаты, либо суммарный угол поворота. Будем сохранять угловые характеристики – синусы и косинусы, поскольку именно они используются в выражениях (1).

В программе модифицируем все модули (main.cpp, draw.h, draw.cpp) и добавляем новые (geometry.h, matrix.h, matrix.cpp).

В файле main.cpp добавляются обработчики WM_CREATE и WM_KEYPRESSED, клавиши ← и → идентифицируются в программе константами VK_LEFT и VK_RIGHT, начальный поворот плоскости инициализируется вызовом функции InitRotation, текущий поворот – функцией AddRotation. Перерисовка изображения (вызов события WM_PAINT) обеспечивается функцией InvalidateRect.

```
case WM_CREATE:InitRotation();break;
case WM_KEYDOWN:
		int KeyPressed;
		KeyPressed = int(wParam);
		if (KeyPressed == VK_RIGHT){ AddRotation(-PI / 10);}
		if (KeyPressed == VK_LEFT){ AddRotation(PI / 10);}
   InvalidateRect(hWnd, NULL, FALSE); //вызов события WM_PAINT
break;
```

В файлы draw.h, draw.cpp и main.cpp вносим изменения, которые обеспечивают рисование треугольника в зависимости от текущего значения угла. В определение функций InitRotation и AddRotation входит вызов функции SetRotationMatrix, которая осуществляет инициализацию массива из 4-х тригонометрических характеристик угла. Переменная массива объявляется типом Matrix. Обновление массива реализует функция MultiplyMatrices. В функции Draw каждая вершина треугольника объявляется типом _Point. При каждом запуске функции Draw вершины инициализируются начальными координатами точек, затем функция ApplyMatrixtoPoint обеспечивает пересчет координат в соответствии с текущими значениями массива тригонометрических характеристик угла. Поворот точек треугольника реализуется в логической системе координат, затем осуществляется преобразование в оконные координаты и рисование сторон треугольника.

draw.h

```
void Draw(HDC hdc);
void SetWindowSize(int _Width, int _Height);
void InitRotation();
void AddRotation(double alpha);
```

draw.cpp

```
#include <windows.h>
```

```
#include "geometry.h"
#include "matrix.h"
int Width, Height;
Matrix current_rot;
const int MARGIN = 10;
void InitRotation(){SetRotationMatrix(0.0, current_rot);}
void AddRotation(double alpha){
 Matrix additional_rot;
    SetRotationMatrix(alpha, additional_rot);
    MultiplyMatrices(current_rot, current_rot, additional_rot);}
void SetWindowSize(int _Width, int _Height) {Width = _Width;Height = _Height;}
_Point T(_Point point){_Point TPoint;
TPoint.x = MARGIN + (1.0 / 2)*(point.x + 1)*(Width - 2 * MARGIN);
TPoint.y = MARGIN +(-1.0 / 2)*(point.y - 1)*(Height - 2 * MARGIN);
      return TPoint;}
void Draw(HDC hdc)
{_Point triangle[3];
      triangle[0].x = 0.0; triangle[0].y = 0.5; triangle[1].x = 0.5;
      triangle[1].y = 0.0; triangle[2].x = -0.5; triangle[2].y = -0.5;
      for (int i = 0; i < 3; i++){
            ApplyMatrixtoPoint(current_rot, triangle[i]); triangle[i] = T(triangle[i]);}
      for (int i = 0; i <= 3; i++){
          int j = i % 3;
          if (i == 0){MoveToEx(hdc, triangle[j].x, triangle[j].y, NULL);}
            else{ LineTo(hdc, triangle[j].x, triangle[j].y);}}
}
```

Как было выше отмечено, по мере усложнения программы необходимо осуществлять ее структуризацию. Очевидно, что по функциональному признаку все действия с массивами целесообразно описать в отдельном модуле. Обратите внимание, что входными параметрами ряда описанных ниже функций есть не переменные, а ссылки на переменные. Это дает возможность при вызове этих функций обновлять входные параметры.

geometry.h

```
#ifndef _POINT
      struct _Point{double x, y;};
      #define _POINT
#endif
```

matrix.h

```
#include "geometry.h"
typedef double Matrix[4];
void SetRotationMatrix(double alpha, Matrix &matrix);
void MultiplyMatrices(Matrix &dest, Matrix &left, Matrix &right);
void ApplyMatrixtoPoint(Matrix rot, _Point &point);
```

matrix.cpp

```
#include <math.h>
#include <memory.h>
#include "matrix.h"
#include "geometry.h"
void SetRotationMatrix(double alpha, Matrix &matrix){
      matrix[0] = cos(alpha);
      matrix[1] = -sin(alpha);
      matrix[2] = sin(alpha);
      matrix[3] = cos(alpha);}
void MultiplyMatrices(Matrix &dest, Matrix &left, Matrix &right){
       Matrix _dest;
      _dest[0] = left[0] * right[0] + left[1] * right[2];
      _dest[1] = left[0] * right[1] + left[1] * right[3];
      _dest[2] = left[2] * right[0] + left[3] * right[2];
      _dest[3] = left[2] * right[1] + left[3] * right[3];
      memcpy(dest, _dest, sizeof(Matrix));}
void ApplyMatrixtoPoint(Matrix rot, _Point &point){
      double _x, _y;
      _x = point.x;
      _y = point.y;
      point.x = _x * rot[0] + _y * rot[1];
      point.y = _x * rot[2] + _y * rot[3];}
```

На этом этапе мы уже обеспечили возможность поворота треугольника при каждом нажатии клавиш ← или →. Теперь поставим задачу обеспечения непрерывного вращения треугольника. Для реализации задачи необходимо обеспечить изменение угла поворота треугольника с последующим обновлением изображения. Причем, это должно выполняться в цикле.

В WinAPI для работы со временем обычно используют сообщение WM_TIMER:

```
case WM_TIMER:     // обработка сообщения WM_TIMER
AddRotation(PI / 10);
InvalidateRect(hWnd, NULL, FALSE);  //вызов события WM_PAINT
break;
```

Это сообщение будет посылаться вашей программе через интервал времени, который вы зададите при создании таймера:

```
SetTimer(hWnd,TIMER_2SEC,2000,NULL);// интервал две секунды
KillTimer (hWnd,TIMER_2SEC);  // уничтожение таймера
```

Создание таймера необходимо предусматривать до его использования, например в событии WM_CREATE. Кроме этого необходимо в заголовке файла определить идентификатор таймера:

```
const unsigned int TIMER_2SEC = 1; // идентификатор таймера
```

Можно обеспечить запуск вращения треугольника при нажатии на клавишу P (пуск) и остановку – при нажатии на клавишу S (stop). Пример оформления события нажатия на клавишу приводится ниже:

```
case WM_KEYDOWN:
   int KeyPressed;
   KeyPressed = int(wParam);
     if (KeyPressed == int('P')) { ... }
break;
```

Ранее мы обеспечили возможность поворота треугольника при каждом нажатии клавиш ← или →. Теперь разберемся, как поворачивать изображение (плоскость) с помощью мыши. Начнем с теории.

Пусть имеется пара точек *M0(x0, y0), M(x, y)* – начальное и конечное положения мыши (рис. 4). По этим точкам нужно определить угол поворота плоскости.

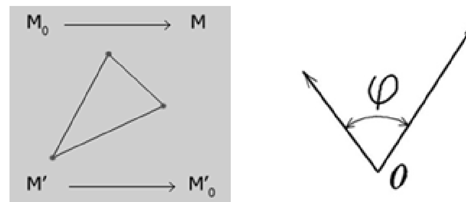


Рис. 4. Угол между векторами, определяемый точками

Воспользуемся формулами для расчета скалярного и векторного произведения векторов:

$$\cos\varphi = \frac{x_1x_2 + y_1y_2}{|l_1||l_2|} = \frac{x_1}{|l_1|}\frac{x_2}{|l_2|} + \frac{y_1}{|l_1|}\frac{y_2}{|l_2|} = \overline{x_1}\,\overline{x_2} + \overline{y_1}\,\overline{y_2}$$

$$\sin\varphi = \frac{x_1y_2 - y_1x_2}{|l_1||l_2|} = \frac{x_1}{|l_1|}\frac{y_2}{|l_2|} - \frac{y_1}{|l_1|}\frac{x_2}{|l_2|} = \overline{x_1}\,\overline{y_2} - \overline{y_1}\,\overline{x_2}$$

$$l = \sqrt{x^2 + y^2}\ , \qquad \overline{x} = \frac{x}{l}\ , \qquad \overline{y} = \frac{y}{l}\ .$$

Класс Rotation объединяет действия и данные, связанные с вращением. Функция Rotate устанавливает преобразования вращения, опираясь на текущее и предыдущее положения мыши. Функция InitRotation запоминает координаты курсора в объекте класса vec. Эти функции вызываются при событиях нажатия кнопки мышки и перемещения курсора. Класс Vec инкапсулирует координаты конца вектора (начало вектора в середине окна) и действия над векторами. Переопределены операции * и ^ под операции скалярного и векторного произведения векторов. Функции set, length и unit устанавливают координаты и определяют длину вектора и нормализуют его. Внесены следующие изменения в файле main.cpp (см. выделенное жирным шрифтом):

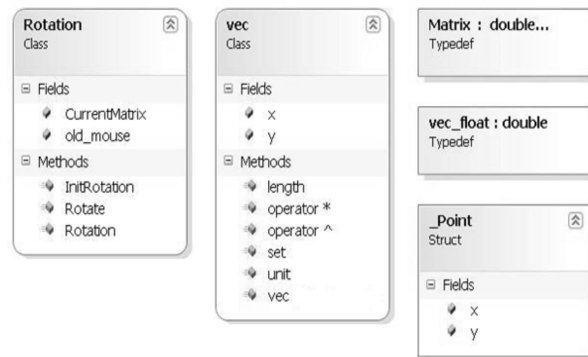


Рис. 5. Диаграмма классов

```
LRESULT CALLBACK WndProc(HWND hWnd,UINT messg, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    RECT Rect;
    HDC hdc, hCmpDC;
    HBITMAP hBmp;
    static Rotation *Ball;
    static int xx, yy;
     switch (messg)
    {
        case WM_CREATE:
        //InitRotation();     // функция закомментирована
        Ball = new Rotation();
        SetBall(Ball);
        break;
        case WM_SIZE:
                      GetClientRect(hWnd, &Rect);
                       xx = Rect.right - Rect.left;
                       yy = Rect.bottom - Rect.top;
//SetWindowSize(Rect.right - Rect.left, Rect.bottom - Rect.top);
                      SetWindowSize(xx, yy);
               break;
    case WM_LBUTTONDOWN:
Ball->InitRotation(LOWORD(lParam) - xx/2, HIWORD(lParam) - xx/2);
        break;
    case WM_MOUSEMOVE:
        if (UINT(wParam) & MK_LBUTTON)
```

```
            {
        Ball->Rotate(LOWORD(lParam) - xx/2, HIWORD(lParam) - xx/2);
                InvalidateRect(hWnd, NULL, FALSE);
            }
            break;
```

В оконную процедуру были добавлены 2-а события – нажатие на левую кнопку мыши и перемещение мыши при нажатой клавише. При нажатии на кнопку мышки функция Ball->InitRotation принимает параметры вектора, начало которого находится в середине окна, конец вектора – в указанной точке. При перемещении мышки функции Ball->Rotate передаются те же параметры текущего вектора.

Поскольку, событие WM_MOUSEMOVE вызывается циклически, пока происходит перемещение курсора, то функция Ball->Rotate вызывается с постоянно обновляемыми параметрами. При каждом цикле происходит расчет тригонометрических функций угла между предыдущим и текущим положениями вектора (см. рис. 5). Класс Rotation описывается в файлах rotation.h и rotation.cpp.

rotation.h

```
#include "matrix.h"
#include "vec.h"
class Rotation{
public:
    vec old_mouse;
    Matrix CurrentMatrix;
    Rotation();
    void InitRotation(int x, int y);
    void Rotate(int x, int y);};
```

rotation.cpp

```
#include "rotation.h"
Rotation::Rotation(){
 CurrentMatrix[0] = 1; CurrentMatrix[1] = 0; CurrentMatrix[2] = 0; CurrentMatrix[3] = 1;}
void   Rotation::InitRotation(int   x,   int   y){old_mouse.set(x,   y);old_mouse   =
old_mouse.unit();}
void Rotation::Rotate(int x, int y){
    vec new_mouse(x, y);
    vec_float sina, cosa;
    new_mouse = new_mouse.unit();
    sina = new_mouse ^ old_mouse;
    cosa = new_mouse * old_mouse;
    Matrix Rot;
    SetRotationMatrixbySinCos(sina, cosa, Rot);
    MultiplyMatrices(CurrentMatrix, CurrentMatrix, Rot);
    old_mouse = new_mouse;}
```

Конструктор Rotation инициализирует матрицу через значения тригонометрических функций при угле поворота 0. Он вызывается, когда объект этого класса создается в событии WM_CREATE. Функция InitRotation вызывается при нажатии мыши. При этом запоминаются нормализованные координаты текущего вектора. Функция Rotate вызывается при перемещении курсора. Получаем вектор, указывающий на текущее положение мыши. С помощью векторных операций реализуется расчет тригонометрических функций угла между этим вектором и вектором, который указывал на предыдущее положение мыши. Функция MultiplyMatrices реализует перемножение текущей матрицы поворота CurrentMatrix на предыдущую.

файл vec.h

```
#include <math.h>
typedef double vec_float;
class vec{
public:
    vec_float x, y;
    vec(){}  vec(vec_float xx, vec_float yy) {x = xx; y = yy;}
   void set(vec_float xx, vec_float yy){x = xx; y = yy;}
    vec_float operator * (vec t) // скалярное произведение
    {return x * t.x + y * t.y;}
    vec_float operator ^ (vec t) // векторное произведение
    {return x * t.y - y * t.x;}
    vec_float length() // длина вектора
    {return sqrt(x * x + y * y);}
    vec unit() // нормализация вектора
    { vec_float l = length(); if (l == 0.0f) return vec(0.0f, 0.0f); return vec(x / l, y
/ l);}
};
```

В файлы matrix.h и matrix.cpp добавляется новая функция SetRotationMatrixbySinCos, которая передает матрице текущие значения синуса и косинуса угла между векторами.

matrix.h

```
#include "geometry.h"
typedef double Matrix[4];
void SetRotationMatrix(double alpha, Matrix &matrix);
void SetRotationMatrixbySinCos(double sinalpha, double cosalpha, Matrix &matrix);
void MultiplyMatrices(Matrix &dest, Matrix &left, Matrix &right);
void ApplyMatrixtoPoint(Matrix rot, _Point &point);
```

matrix.cpp

```
…
void SetRotationMatrixbySinCos(double sinalpha, double cosalpha, Matrix &matrix){
matrix[0] = cosalpha; matrix[1] = -sinalpha; matrix[2] = sinalpha; matrix[3] = cosalpha;}
…
```

Изменения в файлах draw:

draw.h

```
#include "rotation.h"
void Draw(HDC hdc);
void SetWindowSize(int _Width, int _Height);
void SetBall(Rotation *_Ball);
//void InitRotation();   // функция закомментирована
```

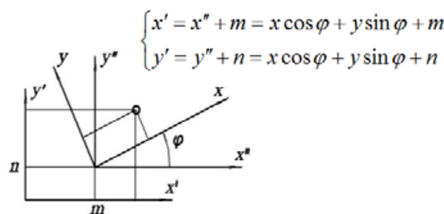
draw.cpp

```
#include <windows.h>
#include "geometry.h"
#include "matrix.h"
#include "draw.h"
int Width, Height; Matrix current_rot; const int MARGIN = 10;
Rotation *Ball;
void SetBall(Rotation *_Ball){Ball = _Ball;}
…
```

Ко всем точкам применяется преобразование, но теперь оно берется из объекта Ball:

```
ApplyMatrixtoPoint(Ball->CurrentMatrix, triangle[i]);
//ApplyMatrixtoPoint(current_rot, triangle[i]);//закомментирована
```

Выше был рассмотрен поступательный процесс создания программной системы, обеспечивающей преобразование вращения (см. рис. 3). Сохраняя специфику создания программы, обобщим ее до преобразований группы движений (рис. 6) и аффинных преобразований. Ниже приводится матричное представление системы уравнений, описывающих группу движений (3) и аффинные преобразования (4).



$$(x' \; y') = (x \; y) \begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix} + (m \; n) \quad (3)$$

$$(x' \; y') = (x \; y) \begin{pmatrix} a & d \\ b & e \end{pmatrix} + (c \; f) \quad (4)$$

Рис. 6. Преобразование точки в прямоугольной декартовой СК

Для использования в программе аффинные преобразования представим в однородных координатах, которые позволяют получать сложное преобразование из нескольких элементарных (композицию преобразований) путем перемножения матриц.

$$\begin{aligned} x' &= ax + by + c \\ y' &= dx + ey + f \\ 1' &= 0x + 0y + 1 \end{aligned}; \qquad (x' \; y' \; 1) = (x \; y \; 1) \begin{pmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{pmatrix} \quad (5)$$

Программа базируется на рассмотренном выше приложении, описывающем преобразование вращения. Принципиальное отличие лишь в том, что вместо систем уравнений (1) и (2) и соответствующей им матрицы с 4-я коэффициентами используется матрица аффинных преобразований (5). Кроме этого реализуется композиция преобразований. В соответствии с этими особенностями программа была модифицирована, осуществлена дальнейшая ее структуризация, созданы новые классы (рис. 7).

Класс Engine объединяет функцию формирования изображения (Draw) и объекты классов Action и Viewport, обеспечивающие действия по перестройке изображения, вызванной событиями нажатия на кнопки, перемещения мышки и изменения размеров окна.

Класс Action объединяет действия и данные, связанные с аффинными преобразованиями. Функция Rotate (Translate) устанавливает преобразования вращения (перемещения), опираясь на текущее и предыдущее по-ложения мыши. Функция InitAction запоминает координаты курсора в объекте класса vec. Эти функции вызываются при событиях нажатия кнопки мышки и перемещения курсора. Функция Transform устанавливает преобразования через инициализацию коэффициентов матрицы

Класс Viewport объединяет действия и данные, связанные с изменением размеров окна, Кроме этого, в класс включена функция T, которая выполняет преобразование точек фигуры из экранных координат экрана, в логические координаты, а также, функция T_inv, которая преобразует координаты курсора из экранных координат в логические. Функция T_inv отсутствовала в приложении, рассмотренном ранее, поскольку в нем реализовывалось лишь преобразование вращения, а значение угла поворота не зависело от СК. В этом приложении с помощью курсора задается и преобразование перемещения, а оно зависит от СК.

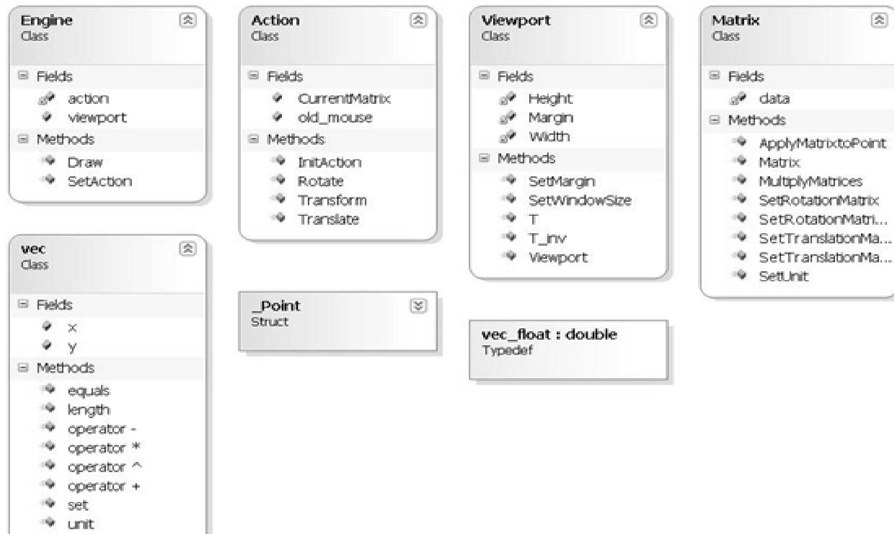


Рис. 7. Диаграмма классов приложения

В класс Matrix была добавлена функция SetTranslationMatrix, в которой непосредственно инициализируются коэффициенты матрицы преобразований.

В класс Vec добавлено переопределение операций для суммирования и вычитания векторов. Эти операции используются в программе для определения преобразования перемещения, опираясь на текущее и предыдущее положения курсора при перемещении мыши.

Практически не меняя структуры программы, обобщим ее использование до аффинных преобразований 3D пространства, где переход из одной прямолинейной координатной системы к другой описывается в общем случае системой уравнений (6):

$$\begin{aligned} x^* &= \alpha_1 x + \alpha_2 y + \alpha_3 z + \lambda \\ y^* &= \beta_1 x + \beta_2 y + \beta_3 z + \mu ; \\ z^* &= \gamma_1 x + \gamma_2 y + \gamma_3 z + \nu \end{aligned} \qquad \left(x^* y^* z^* 1\right) = \left(x\ y\ z\ 1\right) A, \quad A = \begin{pmatrix} \alpha_1 & \beta_1 & \gamma_1 & 0 \\ \alpha_2 & \beta_2 & \gamma_2 & 0 \\ \alpha_3 & \beta_3 & \gamma_3 & 0 \\ \lambda & \mu & \nu & 1 \end{pmatrix} \tag{6}$$

Матрицы, соответствующие базовым геометрическим преобразованиям, приводятся на рис. 8 и 9.

Рис. 8. Матрицы базовых геометрических преобразований

$$\left(x'y'z'1\right) = \left(x\ y\ z\ 1\right)\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & r \\ 0 & 0 & 0 & 1 \end{pmatrix} = \left(x\ y\ 0\ rz+1\right) = \left(\frac{x}{rz+1}\ \frac{y}{rz+1}\ 0\ 1\right)$$

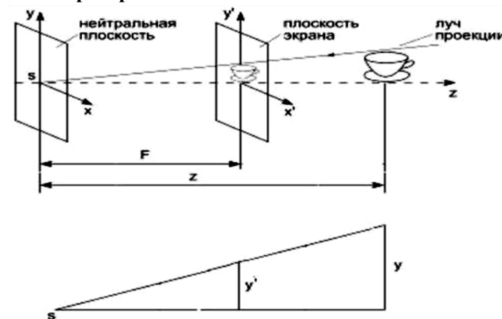


Рис. 9. Графическая интерпретация коэффициента *r* в матрице центрального проецирования

Разработанная программа тестировалась на примере задачи геометрических преобразований рупорной антенны, положение которой определяется в пространстве 5 параметрами (рис. 10). При запуске программы плоскость внешнего раствора рупора совпадает с картинной плоскостью (рис. 11). Необходимо обеспечить при последовательном нажатии на клавиши 1…5,0 композицию матричных преобразований *T=T1T2T3T4T5T0* (рис. 12). При нажатии на клавишу 0 создается центральная проекция.

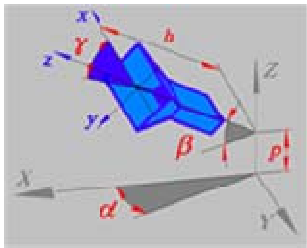


Рис. 10. Тестовое задание для программы

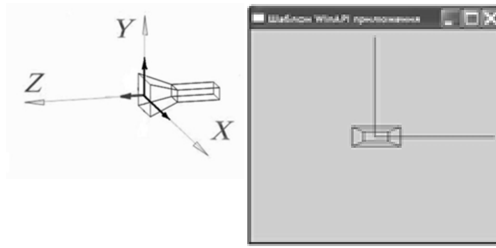


Рис. 11. Исходное положение рупорной антенны

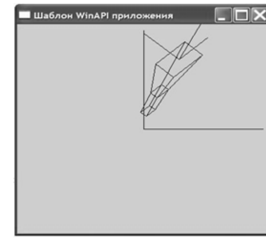


Рис. 12. Конечное положение рупорной антенны

Программа обеспечивает возможность вращения антенны вокруг оси *Y* при движении курсора мышки. Также обеспечивается возможность вращения антенны вокруг оси *z* в локальной системе координат при движении курсора мышки за счет пересчета точек объекта в новой системе координат. С помощью таймера можно запустить запуск вращения антенны вокруг этих осей при нажатии на клавишу P (пуск) и остановку – при нажатии на клавишу S (stop).

В программе реализуются 2 способа определения ортогональной проекции объекта:

- вектор проецирования и плоскость проекций неподвижны, объект перемещается (рис. 13а);
- вектор проецирования и плоскость проекций перемещаются, объект неподвижен (рис. 13б).

В программе реализуются оба способа. Первый способ очевиден, он реализуется через последовательность соответствующих преобразований (см. рис. 8) относительно глобальной СК. Для реализации 2-го способа необходимо:

- определить новую СК, плоскость *XY* которой будет перпендикулярна вектору проецирования;
- пересчитать точки объекта в новой системе координат;
- обнулить координату *z*.

Вектор проецирования можно задавать с помощью двух углов (рис. 14а) – углом вращения относительно оси *X* и углом вращения относительно оси *Y*. Эти же углы определяют положение новой СК (рис. 14б).

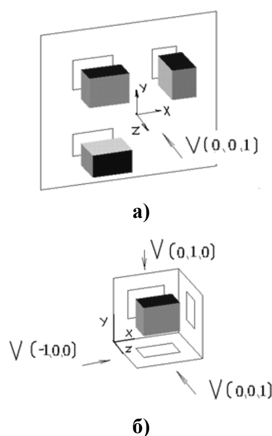


Рис. 13. Определение проекции: объект перемещается (а); вектор перемещается (б)

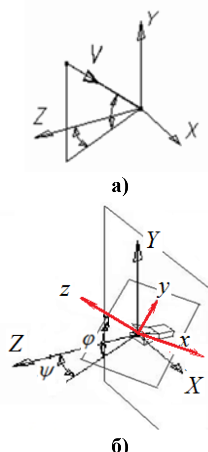


Рис. 14. Определение положения вектора (а) и новой СК (б)

$$R_y^{-1}=\begin{pmatrix}\cos\psi & 0 & \sin\psi & 0\\ 0 & 1 & 0 & 0\\ -\sin\psi & 0 & \cos\psi & 0\\ 0 & 0 & 0 & 1\end{pmatrix}\quad R_x^{-1}=\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & \cos\varphi & -\sin\varphi & 0\\ 0 & \sin\varphi & \cos\varphi & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

$$B^{-1}=R_y^{-1}R_x^{-1}$$

$$\begin{pmatrix}\cos\psi & \sin\psi\sin\varphi & \sin\psi\cos\varphi & 0\\ 0 & \cos\varphi & -\sin\varphi & 0\\ -\sin\psi & \cos\psi\sin\varphi & \cos\psi\cos\varphi & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

Рис. 15. Получение матрицы для пересчета точек объекта в новой системе координат

Положение новой СК относительно старой задается композицией 2-х матричных преобразований – вращения относительно оси X и вращения относительно оси Y (см. рис. 8). Для пересчета точек объекта в новой системе координат необходимо получить обратную матрицу, как композицию обратных матриц 2-х вращений (рис.15). Эти преобразования в программе обеспечиваются нажатием клавиш 6 и 7.

В программе также обеспечена возможность изменения углов вектора проецирования (см. рис. 14а) с помощью курсора мышки. При движении вверх (вниз) меняется угол вращения вокруг оси *X*, при движении вправо (влево) – угол вращения вокруг оси *Y*. Для решения этой задачи используется композиция двух обратных матриц (см. рис.15). Значения косинусов и синусов в матрице преобразований (см. рис. 15) заменены на соответствующие значения перемещений курсора мышки (t_y и t_x):

$$\sin\psi=t_x \quad \cos\psi=\sqrt{1-t_x^2} \quad \sin\varphi=t_y \quad \cos\varphi=\sqrt{1-t_y^2}$$

Фрагменты программной реализации приводятся ниже. При движении мышки с нажатой клавишей Ctrl управление передается функции (файл main.cpp) :

```
action->Translate(mouse_point.x, mouse_point.y);
```

Определение функции (файл action.cpp) содержит вызов функции

```
Tr.SetTranslationMatrix(delta.x, delta.y);
```

Этой функции передаются перемещения курсора мышки (*ty* и *tx*). В соответствии с принятыми обозначениями в определении этой функции (файл matrix.cpp) инициализируются коэффициенты матрицы:

```
void Matrix::SetTranslationMatrix(double tx, double ty){
    SetUnit();// инициализация единичной матрицы
double txx = tx*tx;
double txy = tx*ty;
double tyy = ty*ty;
double tsx = sqrt (1 - txx);
double tsy = sqrt (1 - tyy);
    data[0][0] = tsx;      // a          //  a    c    p    0  //
    data[0][1] = -txy;     // c          //  b    d    q    0  //
    data[0][2] = tx*tsy;   // p          //  h    f    r    0  //
    //data[0][3] = 1.0;    // 0          //  m    n    l    1  //
    data[1][1] = tsy;      // d
    data[1][2] = ty;      // q
    data[2][0] = -tx;       // h
    data[2][1] = -tsx*ty;   // f
    data[2][2] = tsx*tsy;   // r
}
```

Выводы

Разработана архитектура программы для решения задач, связанных с аффинными преобразованиями пространства. Описан процесс создания программы на базе математического аппарата аффинных преобразований 2D и 3D пространства, отталкиваясь от элементарных графических возможностей, предоставляемых в оконной (физической) системе координат.

Программа протестирована на примере задачи сканирования рупорной антенны. При решении этой задачи было продемонстрировано многообразие возможностей геометрических преобразований и эффективность управления ими.

Универсальность архитектуры программы подтверждается тем, что она легко модифицируется под решение ряда других задач – восстановление параметров пространственного объекта по изображению [4], реконструкции трехмерной модели по изображениям [5].

Литература

1. Херн Д. Компьютерная графика и стандарт OpenGL / Дональд Херн, М. Паулин Бейкер. – М. : Вильямс, 2005. – 1168 с.

2. Bradsky G. Learning OpenCV / Gary Rost Bradsky, Adrian Kaehler. – O'Reilly, 2008. – 556 p.

3. Свирневский Н.С. Имитация полета крылатой ракеты на основе объектно-ориентированной методологии проектирования программной системы / Н. С. Свирневский // Вісник Хмельницького національного університету. Технічні науки. – 2012. – № 6. – С. 203–207.

4. Свирневский Н.С. Восстановление параметров пространственного объекта по изображению / Н.С. Свирневский // Вісник Хмельницького національного університету. Технічні науки. – 2011. – № 6. – С. 74–78.

5. Свирневский Н.С. Алгоритм реконструкции трехмерной модели по изображениям / Н. С. Свирневский // Вісник Хмельницького національного університету. Технічні науки. – 2017. – № 2. – С. 103–109.

6. Свирневский Н.С. Основы разработки графических приложений / Н.С. Свирневский, С.С. Ковальчук. – Хмельницкий : ХНУ, 2015. – 270 с.

Отримана/Received : 14.3.2017 р. Надрукована/Printed :9.6.2017 р.
Рецензент: д.т.н., проф. Сорокатый Р.В.