Белгородский Государственный Технологический Университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №3 по теме: «Аффинные преобразования на плоскости»

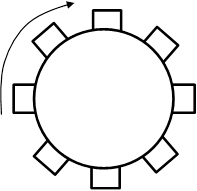
**Выполнил:**  
студент группы ПВ-41  
Адаменко И. И.

**Проверил:**  
старший преподаватель  
Осипов О. В.

Белгород  
2015

**Цель работы:** получение навыков построения аффинных преобразований на плоскости и написание графического приложения с использованием GDI в среде Microsoft Visual Studio.

# Задание

Разработать алгоритм и составить программу для построения на экране следующего изображения:

## Требования к программе

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, поворота, отражения).
2. В программе должна быть предусмотрена возможность ввода пользователем исходных данных ( — количество зубцов).
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 в 10 пикселей от границ и вертикальной линии и реагировать на изменение размера окна.
5. Раскрасить (залить) примитивы по собственному усмотрению.
6. Реализовать вращение фигуры и анимацию: периодическое появление (увеличение) вращающихся с разной скоростью случайных фигур внутри зубцов колеса с последующим постепенным их растворением.

# Теоретическая часть

Допустим на плоскости введена прямолинейная координатная система и дана точка . Сопоставим точке вторую точку с координатами . Координаты точек связаны следующими соотношениями (•):

Где: . Можно понимать, что в процессе данного преобразования точка перемещается в точку .

Таким образом, простейшие геометрические операции описываются следующими формулами:

Поворот вокруг точки (0, 0) на угол :

Растяжение или сжатие вдоль координатных осей:

Отражение (например, относительно оси X):

Перенос:

В курсе аналитической геометрии доказывается, что любое преобразование вида (•) можно представить, как последовательное исполнение простейших преобразований вида (1)–(4). Обычно используется матричная запись данных преобразований.

Для первого случая матрица имеет вид:

Для второго:

И так далее.

Однако, необходимо охватить матричным способом все четыре преобразования (в том числе и перенос). Это достигается следующим образом. Необходимо перейти к описанию произвольной точки плоскости не упорядоченной парой чисел, а тройкой чисел.

Пусть дано . Однородными координатами точки называется любая тройка одновременно неравных чисел , связанных следующими соотношениями:

При решении задач компьютерной графики однородные координаты обычно вводят так: (точке ставится в соответствие точка с изменёнными координатами).

При помощи троек однородных координат и матриц третьего порядка можно описать любое аффинное преобразование на плоскости, в том числе и перенос. Таким образом выражению (•) соответствует матричная запись

Элементы произвольной матрицы аффинного преобразования не несут явного смысла. Поэтому чтобы реализовать то или иное отображение необходимы специальные приёмы. Для того, чтобы выполнить последовательность преобразований необходимо перемножать матрицы.

Имеем матрицы третьего порядка для основных геометрических операций:

Поворот:

Растяжение и сжатие:

Отражение:

Перенос:

## Преобразования

В этой лабораторной работе использовались следующие матричные преобразования:

1. Созданные прямоугольники для зубцов подвергались следующим трансформациям:
   1. Перенос на необходимое расстояние от центра окружности к её краям.
   2. Перенос к центру окружности, поворот на необходимый градус относительно него, перенос обратно.

Таким образом, для каждого созданного прямоугольника совершаются следующие трансформации:

1. Все дополнительные фигуры (для анимации) создавались в центре зубцов, поэтому для каждой из них совершались следующие операции:
   1. Перенос к центру окружности.
   2. Растяжение.
   3. Поворот.
   4. Перенос обратно.

Таким образом, для каждой созданной фигуры совершаются следующие трансформации:

1. Все созданные фигуры (в том числе прямоугольники и исходная окружность) отражались на правую часть с помощью следующих операций:
   1. Отражение относительно оси X.
   2. Перенос на расстояние по оси X, равное двойному расстоянию от максимально удалённой точки исходного изображения.

Таким образом, для всех фигур совершаются следующие трансформации:

# Практическая часть

## Код программы

Main.cs — код основной программы:

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Drawing;
4. **using** System.Windows.Forms;
6. **namespace** lab3
7. {
8. **public** partial **class** Main : Form
9. {
10. **private** Graphics g;
11. **private** **float** space = 10;
12. **private** **float** lineWidth = 2;
13. **private** **float** squareSize = 30;
14. **private** **int** squaresCount = 1;
15. **private** **float** angle = 0;
16. **private** List<SBlinking> blinkingFigures = **new** List<SBlinking>();
18. **public** Main()
19. {
20. InitializeComponent();
21. **this**.SetStyle(
22. ControlStyles.DoubleBuffer |
23. ControlStyles.UserPaint |
24. ControlStyles.AllPaintingInWmPaint |
25. ControlStyles.ResizeRedraw,
26. **true**
27. );
28. **this**.UpdateStyles();
29. }
31. **private** **void** Canvas\_Paint(**object** sender, PaintEventArgs e)
32. {
33. g = e.Graphics;
34. var bounds = g.ClipBounds;
35. var width = bounds.Width / 2 - space - lineWidth;
36. var height = bounds.Height - lineWidth;
37. var radius = (width < height ? width : height) / 2;
39. **if** (radius < squareSize \* 2) **return**;
41. var center = **new** PointF(width / 2, height / 2);
43. var circle = **new** SCircle(center, radius - squareSize);
44. circle.FillColor = Color.Pink;
45. circle.Pen = **new** Pen(Color.Peru, 2F);
47. var rectangles = **new** List<S>();
48. var figures = **new** List<SBlinking>();
50. **for** (**int** i = 0; i < squaresCount; i++)
51. {
52. var rectangle = **new** SRectangle(center, squareSize, squareSize)
53. \* **new** MTranslate(-squareSize / 2, -radius + 2)
54. \* **new** MTranslate(-center.X, -center.Y)
55. \* **new** MRotate(i \* 360 / squaresCount + angle)
56. \* **new** MTranslate(center.X, center.Y);
58. rectangle.Pen = **new** Pen(Color.Transparent, 0F);
59. rectangle.FillColor = Color.Peru;
61. rectangle.Draw(g);
63. var leftBlinking = **new** PointF(rectangle.Center.X + squareSize / 2,  
     rectangle.Center.Y + squareSize / 2);
65. var figure = GetBlinking(i, leftBlinking.X, leftBlinking.Y);
67. var figureCenter = figure.Center;
69. figure  = figure \* **new** MTranslate(-figureCenter.X, -figureCenter.Y)
70. \* **new** MScale(figure.CurrentScale, figure.CurrentScale)
71. \* **new** MRotate(angle \* figure.Speed)
72. \* **new** MTranslate(figureCenter.X, figureCenter.Y);
74. figure.Pen = **new** Pen(Color.Transparent, 0F);
75. figure.FillColor = Color.FromArgb(Convert.ToInt32(figure.Opacity \* 255),
76. figure.FillColor.R,
77. figure.FillColor.G,
78. figure.FillColor.B);
79. figure.Draw(g);
81. rectangles.Add(rectangle);
82. figures.Add(figure);
83. }
85. circle.Draw(g);
87. var translateReflection = width \* 2 + space \* 2;
89. CreateReflection(g, translateReflection, rectangles);
90. CreateReflection(g, translateReflection, figures);
92. circle \*= (**new** MReflect('x'));
93. circle \*= (**new** MTranslate(translateReflection, 0));
94. circle.Draw(g);
95. }
97. **private** **void** CreateReflection(Graphics g, **float** translate, List<S> figures)
98. {
99. **foreach** (var figure **in** figures)
100. {
101. var tmp = figure;
102. tmp \*= (**new** MReflect('x'));
103. tmp \*= (**new** MTranslate(translate, 0));
104. tmp.Draw(g);
105. }
106. }
108. **private** **void** CreateReflection(Graphics g, **float** translate, List<SBlinking> figures)
109. {
110. **foreach** (var figure **in** figures)
111. {
112. var tmp = figure;
113. tmp \*= (**new** MReflect('x'));
114. tmp \*= (**new** MTranslate(translate, 0));
115. tmp.Draw(g);
116. }
117. }
119. **private** SBlinking GetBlinking(**int** index, **float** x, **float** y)
120. {
121. var sb = **new** SBlinking();
123. **if** (index >= blinkingFigures.Count || blinkingFigures.Count == 0)
124. {
125. sb = SBlinking.GetRandom();
127. sb.FillColor = Helper.GetRandomColor();
128. **while** (sb.FillColor == Color.Peru)
129. {
130. sb.FillColor = Helper.GetRandomColor();
131. }
133. var rnd = **new** Random();
134. var r = rnd.Next(200) + 1;
136. sb.Speed = (**float**)r / 10;
138. blinkingFigures.Add(sb);
139. }
141. sb = blinkingFigures[index];
143. **for** (**int** i = 0; i < sb.Points.Count; i++)
144. {
145. var point = sb.Points[i];
147. point.X = x - sb.PointsInitial[i].X;
148. point.Y = y - sb.PointsInitial[i].Y;
150. sb.Points[i] = point;
151. }
153. blinkingFigures[index] = sb;
155. **return** sb;
156. }
158. **private** **void** timer\_Tick(**object** sender, EventArgs e)
159. {
160. angle += 3;
162. **foreach** (var figure **in** blinkingFigures)
163. {
164. figure.Tick();
165. }
167. Canvas.Refresh();
168. }
170. **private** **void** squaresCountUpDown\_ValueChanged(**object** sender, EventArgs e)
171. {
172. squaresCount = Convert.ToInt32((sender **as** NumericUpDown).Value);
173. }
174. }
175. }

Helper.cs — класс с дополнительными функциями:

1. **using** System;
2. **using** System.Drawing;
4. **namespace** lab3
5. {
6. **class** Helper
7. {
8. **public** **static** Color GetRandomColor()
9. {
10. var randomGen = **new** Random();
11. var names = (KnownColor[])Enum.GetValues(**typeof**(KnownColor));
12. KnownColor randomColorName = names[randomGen.Next(names.Length)];
14. **return** Color.FromKnownColor(randomColorName);
15. }
16. }
17. }

S.cs — родительский класс, реализующий работу создаваемых фигур:

1. **using** System.Collections.Generic;
2. **using** System.Drawing;
4. **namespace** lab3
5. {
6. **class** S
7. {
8. **public** List<PointF> Points { **get**; **set**; }
9. **public** PointF Center
10. {
11. **get**
12. {
13. var min = **new** PointF(Points[0].X, Points[0].Y);
14. var max = **new** PointF(Points[0].X, Points[0].Y);
16. **for** (**int** i = 0; i < Points.Count; i++)
17. {
18. **if** (Points[i].X < min.X) min.X = Points[i].X;
19. **if** (Points[i].Y < min.Y) min.Y = Points[i].Y;
20. **if** (Points[i].X > max.X) max.X = Points[i].X;
21. **if** (Points[i].Y > max.Y) max.Y = Points[i].Y;
22. }
24. **return** **new** PointF((max.X + min.X) / 2, (max.Y + min.Y) / 2);
25. }
26. }
27. **public** Pen Pen { **get**; **set**; }
28. **public** Color FillColor { **get**; **set**; }
30. **public** S()
31. {
32. Pen = **new** Pen(Color.Black, 2F);
33. Points = **new** List<PointF>();
34. }
36. **public** S(List<PointF> points)
37. {
38. Pen = **new** Pen(Color.Black, 2F);
39. Points = points;
40. }
42. **public** **void** Draw(Graphics g)
43. {
44. var pointsArray = Points.ToArray();
45. g.DrawPolygon(Pen, pointsArray);
47. **if** (FillColor != **null**)
48. {
49. g.FillPolygon(**new** SolidBrush(FillColor), pointsArray);
50. }
51. }
53. **public** S Clone()
54. {
55. var clone = **new** S();
57. **for** (**int** i = 0; i < **this**.Points.Count; i++)
58. {
59. clone.Points.Add(**new** PointF(**this**.Points[i].X, **this**.Points[i].Y));
60. }
62. clone.Pen = **new** Pen(**this**.Pen.Color, **this**.Pen.Width);
63. clone.FillColor = FillColor;
65. **return** clone;
66. }
68. **public** **static** S **operator** \*(S s, M m)
69. {
70. var newS = s.Clone();
72. **for** (**int** i = 0; i < s.Points.Count; i++)
73. {
74. var point = **new** **float**[] { s.Points[i].X, s.Points[i].Y, 1 };
76. point = m.VectorMult(point);
78. newS.Points[i] = **new** PointF(point[0], point[1]);
79. }
81. **return** newS;
82. }
83. }
84. }

SCircle.cs — дочерний для S класс, реализующий особенности работы фигур-окружностей:

1. **using** System.Collections.Generic;
2. **using** System.Drawing;
4. **namespace** lab3
5. {
6. **class** SCircle : S
7. {
8. **public** SCircle() : **base**() { }
10. **public** SCircle(PointF center, **float** radius)
11. {
12. Pen = **new** Pen(Color.Black, 2F);
14. Points = **new** List<PointF> {
15. **new** PointF(center.X - radius, center.Y - radius),
16. **new** PointF(center.X + radius, center.Y - radius),
17. **new** PointF(center.X + radius, center.Y + radius),
18. **new** PointF(center.X - radius, center.Y + radius)
19. };
20. }
22. **new** **public** **void** Draw(Graphics g)
23. {
24. var rect = **new** RectangleF(Points[0].X,
25. Points[0].Y,
26. Points[1].X - Points[0].X,
27. Points[2].Y - Points[0].Y);
28. g.DrawEllipse(Pen, rect);
30. **if** (FillColor != **null**)
31. {
32. g.FillEllipse(**new** SolidBrush(FillColor), rect);
33. }
34. }
36. **new** **public** SCircle Clone()
37. {
38. var clone = **new** SCircle();
40. **for** (**int** i = 0; i < **this**.Points.Count; i++)
41. {
42. clone.Points.Add(**new** PointF(**this**.Points[i].X, **this**.Points[i].Y));
43. }
45. clone.Pen = **new** Pen(**this**.Pen.Color, **this**.Pen.Width);
46. clone.FillColor = FillColor;
48. **return** clone;
49. }
51. **public** **static** SCircle **operator** \*(SCircle s, M m)
52. {
53. var newS = s.Clone();
55. **for** (**int** i = 0; i < s.Points.Count; i++)
56. {
57. var point = **new** **float**[] { s.Points[i].X, s.Points[i].Y, 1 };
59. point = m.VectorMult(point);
61. newS.Points[i] = **new** PointF(point[0], point[1]);
62. }
64. **return** newS;
65. }
66. }
67. }

SRectangle.cs — дочерний для S класс, реализующий особенности работы фигур-прямоугольников:

1. **using** System.Collections.Generic;
2. **using** System.Drawing;
4. **namespace** lab3
5. {
6. **class** SRectangle : S
7. {
8. **private** SRectangle() : **base**() { }
10. **public** SRectangle(PointF topLeft, **float** width, **float** height)
11. {
12. Pen = **new** Pen(Color.Black, 2F);
14. Points = **new** List<PointF> {
15. **new** PointF(topLeft.X, topLeft.Y),
16. **new** PointF(topLeft.X + width, topLeft.Y),
17. **new** PointF(topLeft.X + width, topLeft.Y + height),
18. **new** PointF(topLeft.X, topLeft.Y + height),
19. };
20. }
22. **new** **public** SRectangle Clone()
23. {
24. var clone = **new** SRectangle();
26. **for** (**int** i = 0; i < **this**.Points.Count; i++)
27. {
28. clone.Points.Add(**new** PointF(**this**.Points[i].X, **this**.Points[i].Y));
29. }
31. clone.Pen = **new** Pen(**this**.Pen.Color, **this**.Pen.Width);
32. clone.FillColor = FillColor;
34. **return** clone;
35. }
37. **public** **static** SRectangle **operator** \*(SRectangle s, M m)
38. {
39. var newS = s.Clone();
41. **for** (**int** i = 0; i < s.Points.Count; i++)
42. {
43. var point = **new** **float**[] { s.Points[i].X, s.Points[i].Y, 1 };
45. point = m.VectorMult(point);
47. newS.Points[i] = **new** PointF(point[0], point[1]);
48. }
50. **return** newS;
51. }
52. }
53. }

SBlinking.cs — дочерний для S класс, реализующий особенности работы дополнительных фигур:

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Drawing;
5. **namespace** lab3
6. {
7. **class** SBlinking : S
8. {
9. **public** **float** LifeTime { **get**; **set**; }
10. **public** **float** OpacityTime { **get**; **set**; }
11. **public** **float** CurrentTime { **get**; **set**; }
12. **public** **float** Opacity { **get**; **set**; }
13. **public** **float** Scale { **get**; **set**; }
14. **public** **float** CurrentScale { **get**; **set**; }
15. **public** **float** Speed { **get**; **set**; }
16. **public** List<PointF> PointsInitial { **get**; **set**; }
18. **public** SBlinking()
19. : **base**()
20. {
21. PointsInitial = **new** List<PointF>();
22. }
24. **public** SBlinking(**float** lifetime, **float** opacitytime)
25. : **base**()
26. {
27. LifeTime = lifetime;
28. OpacityTime = opacitytime;
29. Opacity = 1;
30. Scale = 1 / (lifetime - opacitytime);
31. CurrentScale = Scale;
32. Speed = 1;
33. PointsInitial = **new** List<PointF>();
34. }
36. **public** SBlinking(**float** lifetime, **float** opacitytime, List<PointF> points)
37. : **base**(points)
38. {
39. LifeTime = lifetime;
40. OpacityTime = opacitytime;
41. Opacity = 1;
42. Scale = 1 / (lifetime - opacitytime);
43. CurrentScale = Scale;
44. Speed = 1;
45. PointsInitial = **new** List<PointF>();
47. **foreach** (var point **in** points)
48. {
49. PointsInitial.Add(**new** PointF(point.X, point.Y));
50. }
51. }
53. **public** **static** SBlinking GetRandom()
54. {
55. var list = **new** List<SBlinking>
56. {
57. **new** SBlinking(200, 120,
58. **new** List<PointF>
59. {
60. **new** PointF(15, 5),
61. **new** PointF(25, 23),
62. **new** PointF(5, 23),
63. }
64. ),
65. **new** SBlinking(100, 30,
66. **new** List<PointF>
67. {
68. **new** PointF(7, 7),
69. **new** PointF(23, 7),
70. **new** PointF(23, 23),
71. **new** PointF(7, 23),
72. }
73. ),
74. **new** SBlinking(150, 100,
75. **new** List<PointF>
76. {
77. **new** PointF(15, 7),
78. **new** PointF(23, 15),
79. **new** PointF(23, 23),
80. **new** PointF(7, 23),
81. **new** PointF(7, 15),
82. }
83. ),
84. **new** SBlinking(120, 100,
85. **new** List<PointF>
86. {
87. **new** PointF(12, 7),
88. **new** PointF(18, 7),
89. **new** PointF(18, 12),
90. **new** PointF(23, 12),
91. **new** PointF(23, 18),
92. **new** PointF(18, 18),
93. **new** PointF(18, 23),
94. **new** PointF(12, 23),
95. **new** PointF(12, 18),
96. **new** PointF(7, 18),
97. **new** PointF(7, 12),
98. **new** PointF(12, 12),
99. }
100. ),
101. **new** SBlinking(80, 40,
102. **new** List<PointF>
103. {
104. **new** PointF(7, 7),
105. **new** PointF(23, 7),
106. **new** PointF(23, 15),
107. **new** PointF(15, 15),
108. **new** PointF(15, 23),
109. **new** PointF(7, 23),
110. }
111. ),
113. };
115. var rnd = **new** Random();
116. var r = rnd.Next(list.Count);
118. **return** list[r];
119. }
121. **public** **void** Tick()
122. {
123. CurrentTime++;
125. **if** (CurrentTime > LifeTime)
126. {
127. CurrentTime = 0;
128. Opacity = 1;
129. CurrentScale = Scale;
130. } **else** **if** (CurrentTime + OpacityTime > LifeTime)
131. {
132. Opacity -= 1 / OpacityTime;
133. }
134. **else**
135. {
136. CurrentScale += Scale;
137. }
138. }
140. **new** **public** SBlinking Clone()
141. {
142. var clone = **new** SBlinking();
144. **for** (**int** i = 0; i < **this**.Points.Count; i++)
145. {
146. clone.Points.Add(**new** PointF(**this**.Points[i].X, **this**.Points[i].Y));
147. clone.PointsInitial.Add(**new** PointF(**this**.PointsInitial[i].X,  
      **this**.PointsInitial[i].Y));
148. }
150. clone.Pen = **new** Pen(**this**.Pen.Color, **this**.Pen.Width);
151. clone.FillColor = FillColor;
152. clone.LifeTime = LifeTime;
153. clone.CurrentTime = CurrentTime;
154. clone.Opacity = Opacity;
155. clone.Scale = Scale;
156. clone.CurrentScale = CurrentScale;
157. clone.Speed = Speed;
159. **return** clone;
160. }
162. **public** **static** SBlinking **operator** \*(SBlinking s, M m)
163. {
164. var newS = s.Clone();
166. **for** (**int** i = 0; i < s.Points.Count; i++)
167. {
168. var point = **new** **float**[] { s.Points[i].X, s.Points[i].Y, 1 };
170. point = m.VectorMult(point);
172. newS.Points[i] = **new** PointF(point[0], point[1]);
173. }
175. **return** newS;
176. }
177. }
178. }

M.cs — базовый класс, реализующий работу с матрицами:

1. **using** System.Collections.Generic;
3. **namespace** lab3
4. {
5. **class** M
6. {
7. **protected** **float**[,] Matrix { **get**; **set**; }
9. **public** M()
10. {
11. Matrix = **new** **float**[,] {
12. {0, 0, 0},
13. {0, 0, 0},
14. {0, 0, 0},
15. };
16. }
18. **public** M(**float**[,] matrix)
19. {
20. Matrix = matrix;
21. }
23. **public** **float**[] VectorMult(**float**[] vector)
24. {
25. var result = **new** List<**float**>();
27. **for** (**int** i = 0; i < Matrix.GetLength(0); i++)
28. {
29. var res = 0F;
31. **for** (**int** j = 0; j < Matrix.GetLength(1); j++)
32. {
33. res += vector[j] \* Matrix[j, i];
34. }
36. result.Add(res);
37. }
39. **return** result.ToArray();
40. }
41. }
42. }

MReflect.cs — дочерний для M класс, реализующий работу матриц для операции отражения:

1. **namespace** lab3
2. {
3. **class** MReflect : M
4. {
5. **public** MReflect(**char** type)
6. {
7. **float** x, y, z;
8. x = y = z = 1;
10. **switch** (type)
11. {
12. **case** 'x': x = -1;
13. **break**;
14. **case** 'y': y = -1;
15. **break**;
16. **case** 'z': z = -1;
17. **break**;
18. }
20. **this**.Matrix = **new** **float**[,] {
21. { x, 0, 0 },
22. { 0, y, 0 },
23. { 0, 0, z }
24. };
25. }
26. }
27. }

MRotate.cs — дочерний для M класс, реализующий работу матриц для операции поворота:

1. **using** System;
3. **namespace** lab3
4. {
5. **class** MRotate : M
6. {
7. **public** MRotate(**double** deg)
8. {
9. var phi = Math.PI \* deg / 180;
10. var cos = (**float**)Math.Cos(phi);
11. var sin = (**float**)Math.Sin(phi);
13. **this**.Matrix = **new** **float**[,] {
14. { cos, sin, 0 },
15. { -sin, cos, 0 },
16. { 0, 0, 1 },
17. };
18. }
19. }
20. }

MScale.cs — дочерний для M класс, реализующий работу матриц для операции растяжения/сжатия:

1. **namespace** lab3
2. {
3. **class** MScale : M
4. {
5. **public** MScale(**float** a, **float** b)
6. : **base**(**new** **float**[,] {
7. { a, 0, 0 },
8. { 0, b, 0 },
9. { 0, 0, 1 }
10. })
11. {
12. // nothing here :)
13. }
14. }
15. }

MTranslate.cs — дочерний для M класс, реализующий работу матриц для операции переноса:

1. **namespace** lab3
2. {
3. **class** MTranslate : M
4. {
5. **public** MTranslate(**float** x, **float** y)
6. : **base**(**new** **float**[,] {
7. { 1, 0, 0 },
8. { 0, 1, 0 },
9. { x, y, 1 }
10. })
11. {
12. // nothing here :)
13. }
14. }
15. }

## Скриншоты

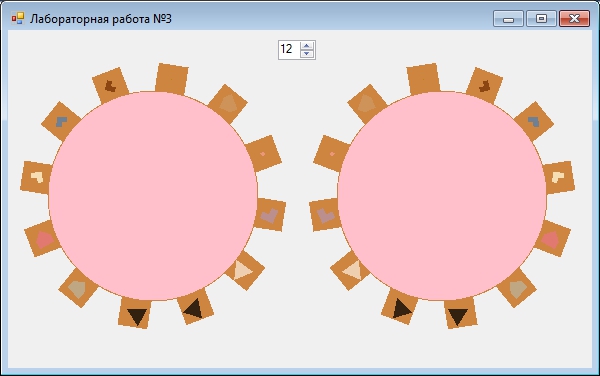


Рисунок 1. Рабочая программа