Урок №3: Системы координат и преобразования, регионы, траектория.

Содержание

1. Понятие системы координат

Прежде чем перейти к рассмотрению операций преобразования, мы должны четко понимать, что собой представляют системы координат в GDI++. Каждая геометрическая форма, отображенная на клиентской области, имеет свои координаты. Например, линия представлена двумя координатами A (x1, y1) и B (x2, y2).

2. Типы систем координат

Из первого урока мы вкратце ознакомились, с понятием координатных систем **GDI++.** Напомним, что в **GDI++** существует три координатные системы: **Мировые** – позиция точки в пикселях относительно верхнего левого угла документа; **Страничные** - позиция точки в пикселях относительно верхнего левого угла клиентской области; **Устройства** – подобны страничным за исключением того, что могут быть представлены не только пикселями, но и другими единицами измерения, дюймами или миллиметрами.

а) **Мировые координаты.**

Мировые координаты находятся на верхней ступени иерархии координат **GDI++**. Представляют абстракцию размеров, не зависящую от единиц измерения. Когда мы рисуем линию с помощью вызова метода **Graphics.DrawLine(pen,0,0,150,120)**. (0,0) и (150, 120) - это точки, определяющие начало (0 единиц по горизонтали и 0 по вертикали) и конец линии (150 единиц по горизонтали и 120 по вертикали). Они представлены в мировом пространстве координат. Исходя из этого все методы **GDI++,** для рисования геометрических фигур принимают мировые координаты.

в) Страничные координаты.

Ниже в иерархии находятся **страничные** координаты, которые представляют собой смещение по отношению к мировым. Эту систему координат удобно применять тогда, когда мы хотим выполнить смещение, не изменяя **мировые** координаты. Например, мы хотим сместить прямоугольник на 10 пикселов вправо и на 30 пикселов вниз. Начало страничных координат изменится и будет находиться в новой точке мировых координат (10,30) рис 2.1.

Рассмотрим описанный пример. В Visual Studio создайте новый Windows Forms проект, назовите его **PageCoordinates**. Сгенерируйте для формы метод обработчика события **Paint,** заполните его следующей программной логикой.

```
        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            // смещаем начало координат для страницы относительно
мировых
            g.TranslateTransform(10, 30);
            // рисуем прямоугольник
            g.DrawRectangle(new Pen(Brushes.Blue,4), 0, 0, 80, 80);
        }
```

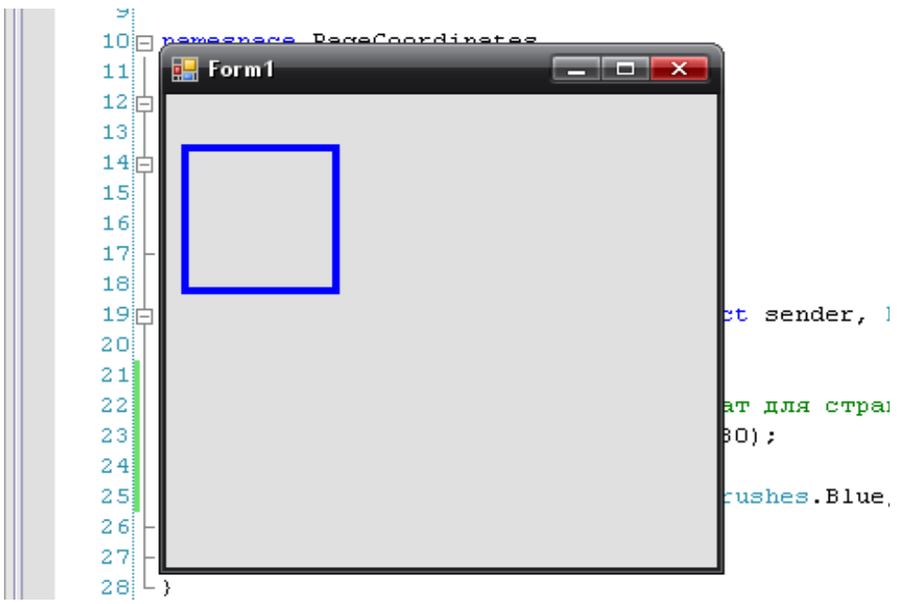

Рис. 2.1 Смещение страничных координат.

Проект называется **PageCoordinates.**

c) Координаты устройства

Наконец существуют координаты **устройства**. Координаты устройства представляют, где и как графические примитивы будут отображаться на устройствах вывода, таких как принтер или монитор. Фактически эти координаты представляют результат применения **страничных** координат к **мировым.**

Как было описано в первом уроке, координаты проходят конвейер трансформаций, прежде чем будут отображены на устройстве вывода. Если к графическим объектам не применяются какие-либо трансформации, то все описанные выше координатные системы совпадают, и начало всех трех координат находится в точке (0,0) .

3. Преобразование координат

a) Что такое преобразование координат?

Прежде чем нарисовать графический примитив с помощью **GDI++,** координатные системы проходят через конвейер преобразований (трансформаций). Первым в конвейере стоят мировые преобразования (**world transformation)** здесь происходит конвертирование мировых координат в страничные. Далее следуют страничные преобразования (**page transformation)** на этом этапе происходит конвертирование страничных координат в координаты устройства.

Преобразования над графическими примитивами в GDI++ разделены на две основные категории: **глобальные** преобразования и **локальные**. Дополнительно существуют так же **смешанные** преобразования. Далее мы подробно рассмотрим эти категории.

b) Матричное представление преобразования

Прежде чем приступить к разбору этой темы мы должны понимать, что такое преобразования (трансформации). Благодаря тому, что в **GDI++** существует преобразования, описанные выше, мы можем выполнять такие трансформации над графическими примитивами, как перемещение, масштабирование, вращение, сжатие и т.д.

1. Что такое матрица

Матрица - это набор сгруппированных однотипных чисел, представленных рядами m и столбцами n, фактически это тот же самый массив. На рисунке ниже показаны примеры различных матриц.

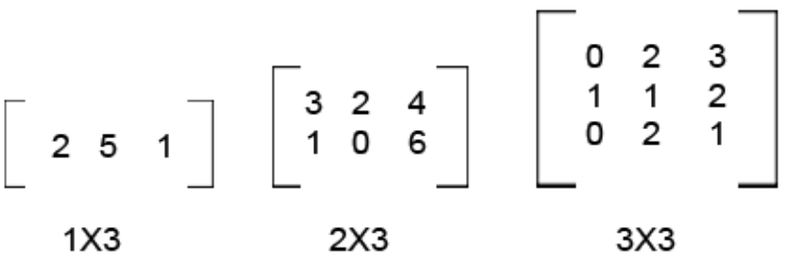

Рис. 3.1 Различные виды матриц.

В GDI++ матрица представлена объектом **Matrix. Matrix -** это класс, представляющий матрицу 3×2 хранящий информацию о трансформировании геометрических примитивов, таких как (изображение, линия, точка, текст и т.д.). Прежде чем использовать класс **Matrix** в приложении, необходимо добавить пространство имен **System.Drawing.Drawing2D.**

Экземпляр класса **Matrix** может быть создан различными способами, например:

```
Matrix M1 = new Matrix();
Matrix M2 = new Matrix(1, 3, 3, 1, 0, 2);
Matrix M3 =
     new Matrix(0.0f, 1.0f, -1.0f, 2.0f, 0.0f, 1.0f);
```

Входящие параметры представляют матрицу, и каждый элемент массива несет в себе информацию о преобразовании. Объект **Matrix** может бать наполнен информацией и с помощью методов экземпляра, таких как **Scale(), Shear(), Rotate()** и т.д.

Рассмотрим пример, в котором будет изменяться матрица, представленная свойством **Transform** объекта рисования **Graphics.** В Visual Studio создайте проект Windows Forms. Назовите его **MatrixAndTransform.** Добавьте к форме два элемента **Label,** которые будут отображать значение объекта **Matrix** до и после перемещения линии. В метод обработчика события **Paint** добавьте следующую логику.

```
        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            string matrix1 = null;
            System.Drawing.Drawing2D.Matrix X1 = g.Transform;
            // производим чтение из матрицы X1
            for (int i = 0; i < X1.Elements.Length; i++)
            {
                matrix1 += X1.Elements[i].ToString();
                matrix1 += ", ";
            }
            lblBefore.Text = "before - " + matrix1.Trim(' ', ',');
            // рисуем линию
            g.DrawLine(new Pen(Brushes.Red, 3), 0, 10, 100, 50);
            // смещаем начало координат для страницы относительно
мировых
            g.TranslateTransform(10, 50);

            string matrix2 = null;
            System.Drawing.Drawing2D.Matrix X2 = g.Transform;
            // производим чтение из матрицы X1
            for (int i = 0; i < X2.Elements.Length; i++)
            {
                matrix2 += X2.Elements[i].ToString();
                matrix2 += ", ";
            }
            lblAfter.Text = "after - " + matrix2.Trim(' ', ',');
            // рисуем линию
            g.DrawLine(new Pen(Brushes.Red, 3), 0, 10, 100, 50);

        }
```

В этом примере мы видим, что информация о преобразовании сохраняется в классе **Matrix** рис. 1.1.

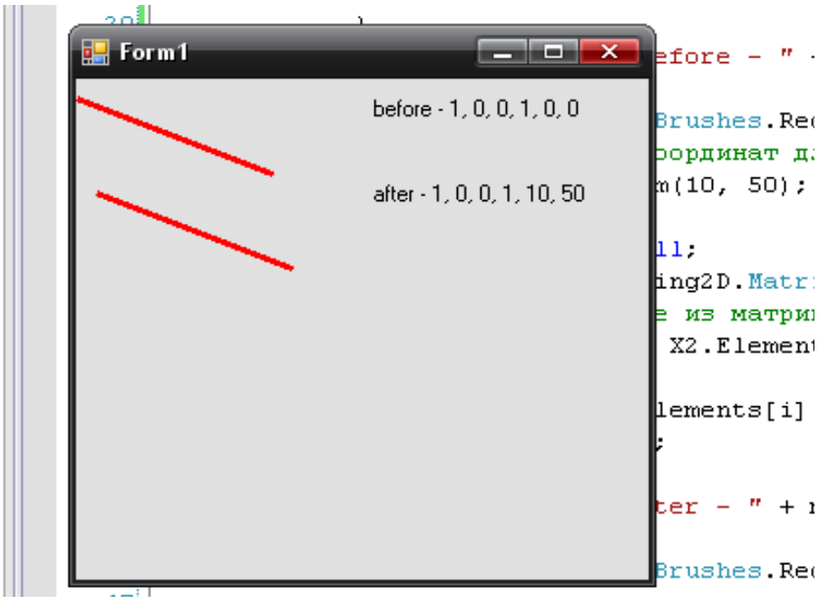

Рис. 1.1 Значения Matrix до и после трансформации.

Проект называется **MatrixAndTransform.**

2. Операции с матрицами

Мы можем выполнять над матрицами такие математические операции, как сложение. Рисунок ниже иллюстрирует сложение матриц.

$$\begin{bmatrix} 5 & 4 \end{bmatrix} + \begin{bmatrix} 20 & 30 \end{bmatrix} = \begin{bmatrix} 25 & 34 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 1 & 3 \end{bmatrix} + \begin{bmatrix} 2 & 4 \\ 1 & 5 \\ 0 & 6 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 1 & 7 \\ 1 & 9 \end{bmatrix}$$

Рис. 3.2 Сложение матриц.

Умножение матриц

Мы можем выполнять операции умножения над матрицами. Рисунок ниже иллюстрирует умножение матриц.

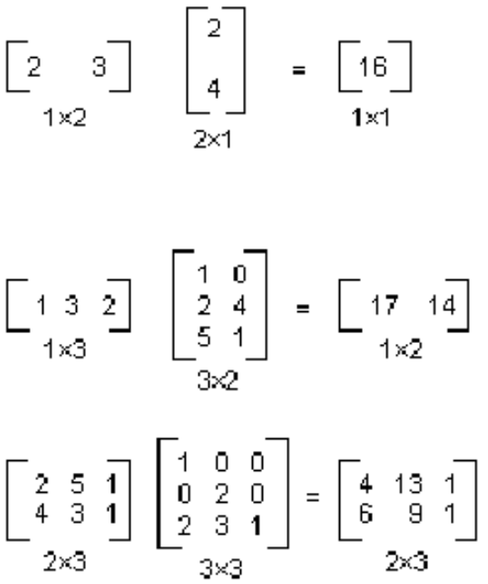

Рис. 3.3 Умножение матриц.

В **GDI++** матрицы могут быть перемножены с помощью метода **Multiply** объекта **Matrix,** причем результат сохраняется в матрице, вызвавшей этот метод. Рассмотрим пример перемножения двух матриц. В Visual Studio Создайте проект Windows Forms. Добавьте к форме элемент **MenuStrip** с одним **ToolStripMenuItem** назовите его **mnuMultiply,** для свойства **Text** установите "Multiply". Так же добавьте к форме три элемента **Label,** которые будут отображать матрицу и два для операций умножения. Перемножение матриц производим в методе обработчика события **Click** элемента **ToolStripMenuItem.** Результат выводим с помощью **lblMatrix3.** Ниже представлен пример кода описанного выше.

```
using System;
using System.Windows.Forms;
using System.Drawing.Drawing2D;

namespace MatrixMultiply
{
    public partial class Form1 : Form
    {
        Matrix X =
                  new Matrix(2.0f, 1.0f, 1.0f, 2.0f, 0.0f, 1.0f);
        Matrix Y =
                 new Matrix(0.0f, 1.0f, -2.0f, 0.0f, 2.0f, 0.0f);
        public Form1()
        {
```

```
            InitializeComponent();
            // инициализируем элементы Lebel
            InitializeLebels();
        }

        private void InitializeLebels()
        {
            string matrix1 = null;
            string matrix2 = null;
            // создаем новые временные матрицы
            Matrix m1 = new Matrix();
            m1 = X;
            Matrix m2 = new Matrix();
            m2 = Y;
            // производим чтение из матрицы
            for (int i = 0; i < m1.Elements.Length; i++)
            {
                matrix1 += m1.Elements[i].ToString();
                matrix1 += ", ";
            }
            lblMatrix1.Text = matrix1.Trim(' ', ',');
            // производим чтение из матрицы
            for (int i = 0; i < m2.Elements.Length; i++)
            {
                matrix2 += m2.Elements[i].ToString();
                matrix2 += ", ";
            }
            lblMatrix2.Text = matrix2.Trim(' ', ',');
        }

        private void mnuMultiply_Click(object sender, EventArgs e)
        {
            string matrix3 = null;
            X.Multiply(Y, MatrixOrder.Append);
            // производим чтение из матрицы
            for (int i = 0; i < X.Elements.Length; i++)
            {
                matrix3 += X.Elements[i].ToString();
                matrix3 += ", ";
            }
            lblMatrix3.Text = matrix3.Trim(' ',',');
        }
    }
}
```

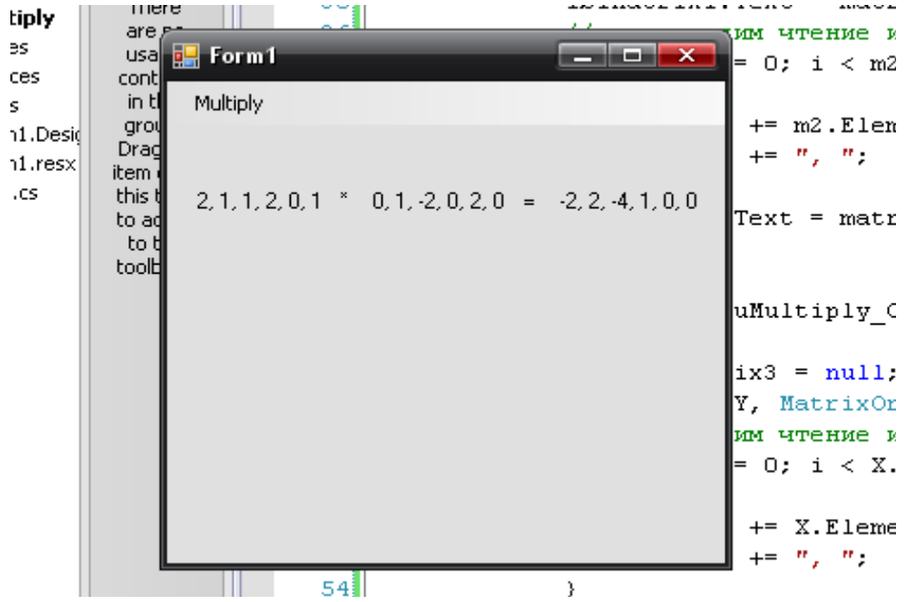

Рис. 3.4 Результат перемножения матриц.

Проект называется **MatrixMultiply.**

Инвертирование матриц

Матрица может быть инвертирована. Для этого необходимо воспользоваться методом **Invert** класса **Matrix.** Рассмотрим приложение, которое будет производить инвертирование матрицы. В Visual Studio создайте проект Windows Forms. Добавьте к форме элемент **MenuStrip** с одним **ToolStripMenuItem,** назовите его **mnuInvert,** для свойства **Text** установите “Invert”. Также добавьте к форме элемент **Label,** который будет отображать матрицу. Событие **Click** элемента **ToolStripMenuItem** будет выполнять инвертирование объекта **Matrix**. Ниже представлен пример кода описанного выше.

```
using System;
using System.Windows.Forms;
using System.Drawing.Drawing2D;

namespace MatrixInvert
{
    public partial class Form1 : Form
    {
        Matrix X = new Matrix(1, 0, 3, 4, 0, 2);
```

```
        public Form1()
        {
            InitializeComponent();
            InitializeLebels();
        }

        private void InitializeLebels()
        {
            string matrix1 = null;
            // производим чтение из матрицы
            for (int i = 0; i < X.Elements.Length; i++)
            {
                matrix1 += X.Elements[i].ToString().Replace(',','.');
                matrix1 += ", ";
            }
            lblMatrix1.Text = matrix1.Trim(' ', ',');

        }

        private void mnuInvert_Click(object sender, EventArgs e)
        {
            // инвертируем Матрицу
            X.Invert();
            InitializeLebels();
        }
    }
}
```

Проект называется **MatrixInvert**

Другие операции с матрицами

Над матрицами могут быть выполнены и другие операции преобразования. Для этого в объекте Matrix существуют следующие методы: **Rotate, RotateAt, Scale, Shear, и Translate.** Все они, включая **Invert** и **Multiply,** являются методами смешанного преобразования, которое будет рассмотрено позже.

Рассмотрим метод **Scale.** Этот метод предназначен для масштабирования матрицы, принимает два аргумента с плавающей точкой, которые определяют коэффициент масштабирования **(Scale Foctor)** по двум осям **X и У.**

Методы **Rotate и RotateAt** предназначены для вращения матрицы на определенный угол. Метод **RotateAt** принимает, дополнительный па-

раметр, экземпляр структуры **PointF**. Он определяет положение точки, относительно которой будет происходить вращение.

Метод **Shear** предназначен для сдвига матрицы. Принимает два параметра с плавающей точкой, которые определяют коэффициент сдвига по оси X и У**.**

Метод **Translate** выполняет перемещение матрицы. Он принимает два аргумента с плавающей точкой, которые представляют смещение по оси X и У.

Все перечисленные методы имеют пару перегрузок и могут принимать дополнительный аргумент, перечесление **MatrixOrder. MatrixOrder** задает порядок следования преобразований. Он является важным в преобразованиях. Имеет два значения: **Append** – новая операция будет добавлена после предыдущей; **Prepend** - новая операция будет добавлена до предыдущей.

Рассмотрим на примере перечисленные методы. В Visual Studio создайте проект Windows Forms. Добавьте к форме элемент **MenuStrip** с двумя **ToolStripMenuItem,** назовите его **mnuOperations.** Назовите **ToolStripMenuItem -** элементы **mnuItemReset** и **mnuItemSelOperation**. Одна из них будет сбрасывать преобразования в матрице, а другая будет дополнительно иметь разворачивающееся меню. В нем будут кнопки выполняющие различные операции над матрицей. Так же нам понадобится создать перечисление **MatrixOperations**, которое будет представлять операцию преобразования на данный момент.

```
// представляет операцию преобразования
public enum MatrixOperations
{
    Reset,
    Rotate,
    RotateAt,
    Scale,
    Shear,
    Translate
}
```

Объявим приватные переменные внутри тела класса **Form1**

```
        private  Graphics g;

        private  MatrixOperations mxOperation = MatrixOperations.Reset;
```

С помощью конструкции **switch** в событии **Pain** формы обработаем перечисление и вызовем соответствующий метод преобразования.

```
        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            // обработаем перечисление и вызовем соответствующий метод
преобразования
            switch (mxOperation)
            {
                case MatrixOperations.Rotate:
                    MatrixRotate();
                    break;
                case MatrixOperations.Scale:
                    MatrixScale();
                    break;
                case MatrixOperations.Reset:
                    MatrixReset();
                    break;
                case MatrixOperations.Shear:
                    MatrixShear();
                    break;
                case MatrixOperations.Translate:
                    MatrixTranslate();
                    break;
                default:
                    MatrixReset();
                    break;
            }

        }
```

В методах обработчика события **Click** будет происходить установка **MatrixOperations,** а также принудительная прорисовка клиентской области с помощью метода **Invalidate**.

```
        #region ClickEvents
        private void mnuRotate_Click(object sender, EventArgs e)
        {
            mxOperation = MatrixOperations.Rotate;
            this.Invalidate();
        }
        // ниже следуют другие методы
         ...
```

```
        #endregion
```

Различные операции с матрицами будут выполняться с помощью приватных методов **MatrixReset, MatrixTranslate, MatrixShear, MatrixScale, MatrixRotate,** заключённых в регион **TransformationMethods.**

```
        #region  TransformationMethods
        private void MatrixReset()
        {
            if (g != null)
            {
                // сбрасываем преобразования в матрице
                g.Transform.Reset();
            }
        }

        private void MatrixTranslate()
        {
            g = this.CreateGraphics();
            g.FillRectangle(Brushes.Blue, 60, 60, 110, 40);
            // Создаем матрицу
            Matrix X = new Matrix();
            // Применяем перемещение
            X.Translate(40.0f, 80.0f, MatrixOrder.Append);
            g.Transform = X;
            g.FillRectangle(Brushes.Blue, 60, 60, 110, 40);
        }

         // ниже следуют другие методы преобразования
         ...

       #endregion
```

Скриншот ниже демонстрирует операции вращения слева, и масштабирования справа.

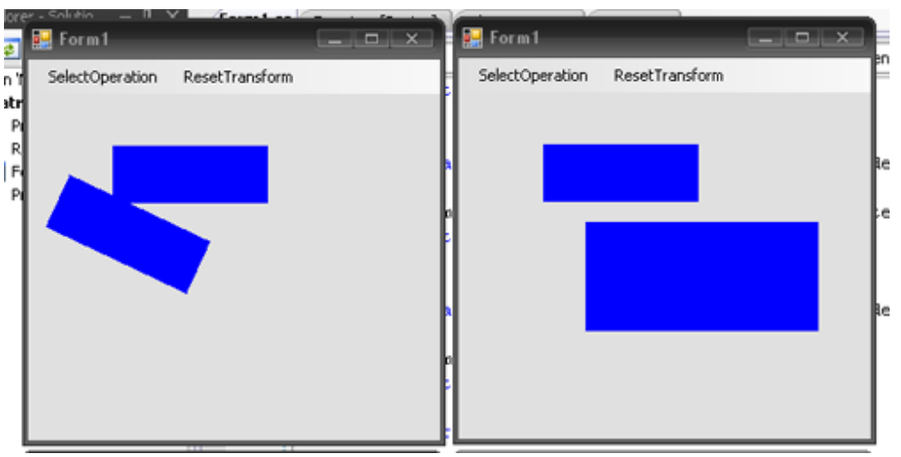

Рис. 3.5 Вращение и масштабирование матрицы.

Операции сдвига слева, и перемешения справа.

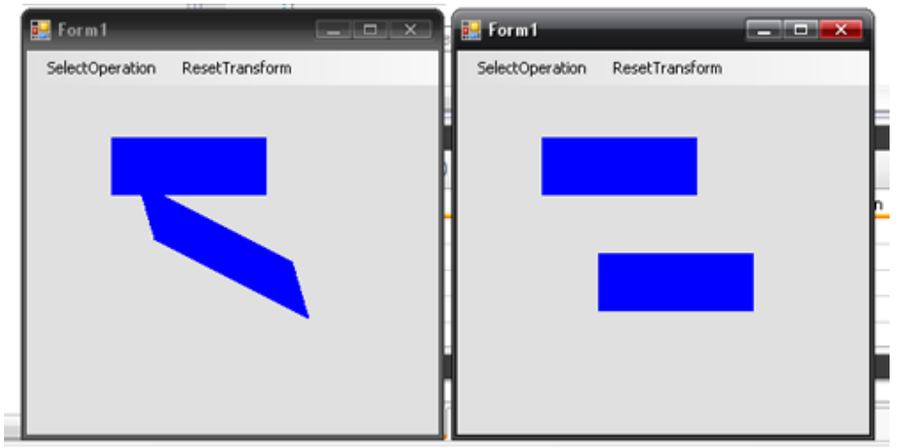

Рис. 3.6 Сдвиг и перемещение матрицы.

Проект называется **MatrixOperations.**

в) Глобальное преобразование.

Глобальное преобразование(Global Transformations или World Transformations) - это преобразование, которое было выполнено ко всем графическим примитивам, отображенным с помощью объекта Graphics. С помощью свойства **Graphics.Transform** можно установить глобальное преобразование. Это свойство представляет объект **Matrix.** Вся информация о преобразовании сохраняется в этом свойстве.

Например, давайте построим несколько графических примитивов, линию, эллипс, прямоугольник и посмотрим, как они будут выглядеть до и после глобального преобразования.

Пример ниже демонстрирует глобальное преобразование.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Rectangle rect = new Rectangle(30, 30, 60, 60);
    Pen penB = new Pen(Brushes.Blue,2);
    Pen penR = new Pen(Brushes.Red,2);
    // примитивы до глобального преобразования
    g.DrawRectangle(penB, rect);
    g.DrawLine(penR, 30, 200, 200, 170);
```

```
            g.FillEllipse(Brushes.Brown, new Rectangle(100, 30, 100,
100));

            Matrix X = new Matrix();
            X.Scale(1.4f, 1.4f, MatrixOrder.Append);
            X.RotateAt(-10,new PointF(0.0f,0.0f),MatrixOrder.Append);
            g.Transform = X;
            // примитивы после глобального преобразования
            g.DrawRectangle(penB, rect);
            g.DrawLine(penR, 30, 200, 200, 170);
            g.FillEllipse(Brushes.Brown, new Rectangle(100, 30, 100,
100));

        }
```

Метод **Scale()** выполняет равномерный масштаб на 40%, а метод **RotateAt()** выполняет вращение на -10 градусов относительно точки с координатами (0,0). Так как мы выполняем несколько преобразований подряд, то такие преобразования будут считаться глобыльными и **смешанными**. Как альтернатива смешанное преобразование может осуществляться и с помощью вызова методов **RotateTransform, ScaleTransform и** др. объекта **Graphics** например:

```
                ...

        //X.Scale(1.4f, 1.4f, MatrixOrder.Append);
        //X.RotateAt(-10, new PointF(0.0f, 0.0f), MatrixOrder.Append);
        //g.Transform = X;
          g.RotateTransform(-10, MatrixOrder.Append);
          g.ScaleTransform(1.4f, 1.4f);

                ...
```

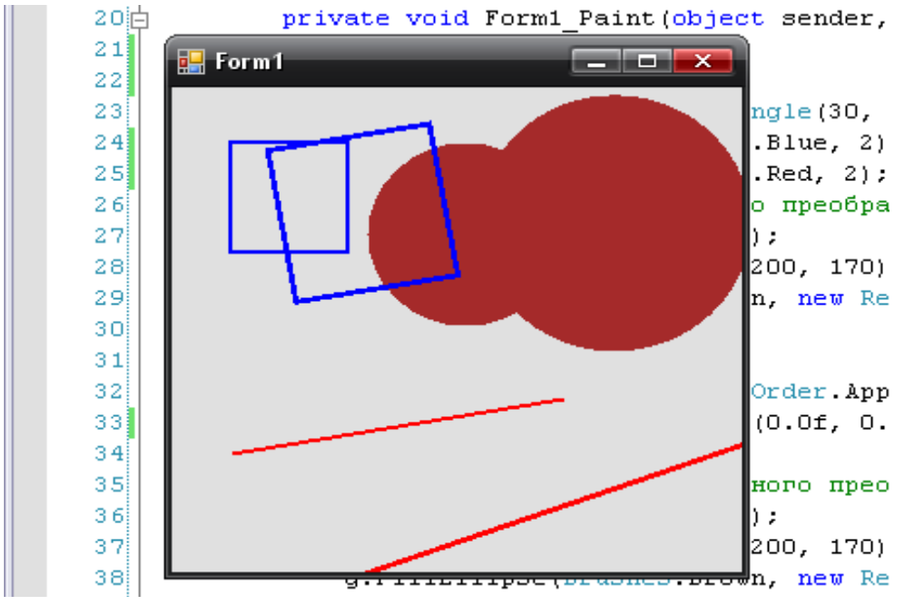

Рис. 3.4 Глобальное преобразование объектов.

Проект называется **GlobalTransformation.**

d. Локальное преобразование

В отличие от глобального преобразования, **локальное** (Local Transformations) применяется к отдельным элементам рисования, полученных с помощью объекта **Graphics.** Хорошим примером **локального** преобразования является визуализация объектов с помощью класса **GraphicsPath.** Рассмотрим пример **локального** преобразования. В Visual Studio создайте проект Windows Forms. Заполните следующей логикой событие **Paint.**

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    using (Graphics g = e.Graphics)
    {
        // создаем путь
        GraphicsPath path = new GraphicsPath();
        path.AddLine(20, 50, 200, 50);
        path.AddArc(50, 50, 100, 50, 10, -220);

        Matrix X = new Matrix();
        // вращаем на 20 градусов
        X.Rotate(20);
```

```
                // применяем преобразование к  GraphicsPath
                path.Transform(X);
                // Визуализируем прямоугольник
                g.DrawRectangle(new Pen(Brushes.Green,2), 10, 10, 60,
60);
                // Визуализируем путь
                g.DrawPath(new Pen(Color.Blue, 4), path);
                // Освобождаем ресурсы
            }

        }
```

Из этого примера видно, что преобразование применяется только к объекту **GraphicsPath.**

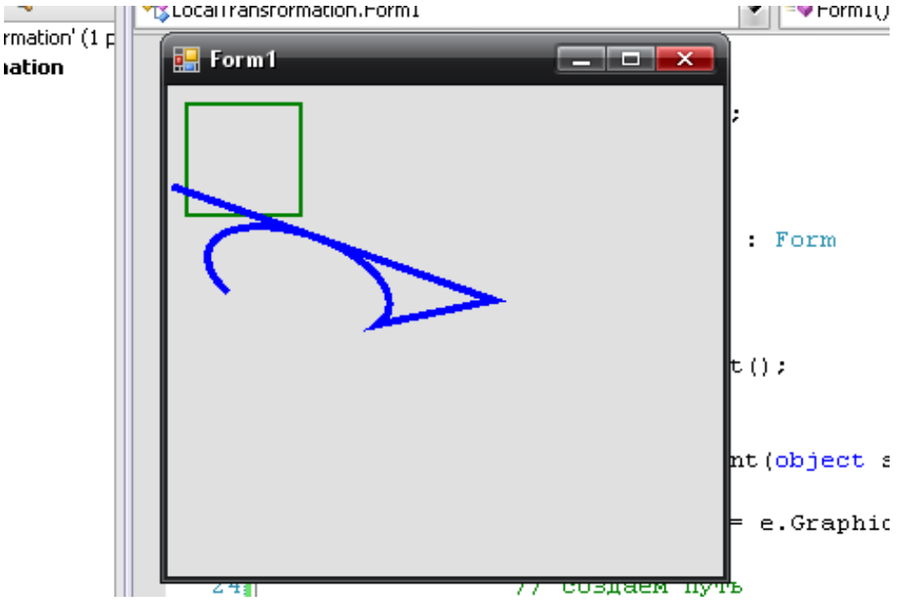

Рис. 3.5 Локальное преобразование объекта **path**.

Проект называется **LocalTransformation.**

4. Примеры использования преобразований

Рассмотрим пример преобразования изображений. Трансформация изображений выполняется так же просто, как и трансформация выше описанных примитивов. Пример ниже демонстрирует преобразование изображения.

```
        private void Form1_Paint(object sender, PaintEventArgs e)
```

```
        {
            using (Graphics g = e.Graphics)
            {
                try
                {
                    // рисуем изображение без трасформаций
                    g.DrawImage(new Bitmap(@"Boxs.bmp"), 0, 0, 100,
100);
                }
                catch { }
                // Создаем матрицу
                Matrix X = new Matrix();
                // Установка трансформаций
                X.RotateAt(45,new Point(150,150));
                X.Translate(100, 100);
                g.Transform = X;
                try
                {
                    // рисуем изображение
                    g.DrawImage(new Bitmap(@"Rings.bmp"), 0, 0, 100,
100);
                }
                catch { }

                // Сброс трансформаций
                X.Reset();
                // Установка трансформаций
                X.RotateAt(25, new Point(50, 150));
                X.Translate(150, 10);
                X.Shear(0.5f, 0.3f);
                g.Transform = X;
                try
                {
                    // рисуем изображение
                    g.DrawImage(new Bitmap(@"Cells.bmp"), 0, 0, 100,
100);
                }
                catch { }

            }

        }
```

Файлы изображений создайте в программе Adobe Photoshop, и поместите их в папку приложения, ImageTransformation\bin\Debug. Для освобождения неуправляемых ресурсов, поместим объект **Graphics** в блок using. Рисунок ниже показывает работу приложения.

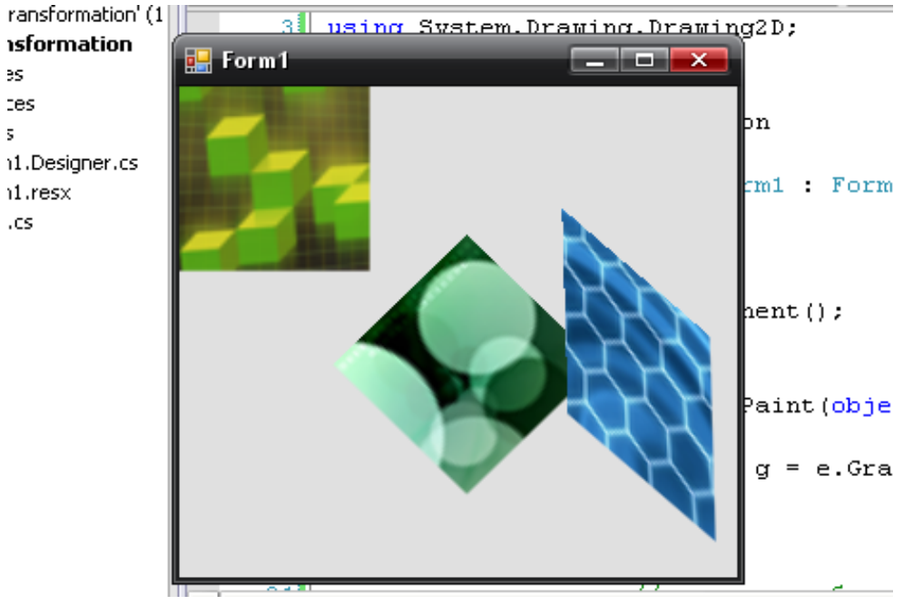

Рис. 4.1 Преобразования изображений.

Проект называется **ImageTransformation.**

5. Понятие региона

В первом уроке мы с вами поверхностно ознакомились с понятием региона. Напомним, что **Регион** это сущность, описывающая внутреннюю область замкнутых форм. В **GDI++** представлен классом **Sytem.Drawing.Region.** Регион имеет две области - клиентскую и неклиентскую. Неклиентская область - это та область, в которой не разрешено рисование графических объектов, например: меню управления или полосы прокруток. Клиентская область, наоборот, позволяет отображать графические объекты **GDI++.**

Класс **Region** имеет методы, описанные в таблице 5.1.

Таблица 5.1 Основные методы класса **Region**.

Методы	Описание
Union	Выполняет операцию объединения.
Exclude	Выполняет операцию исключения.
Intersect	Выполняет операцию пересечения.

Методы	Описание
Xor	Выполняет операцию исключающая или, (exclusive OR).
GetRegionData	Возвращает объект RegionData который описывает этот регион
Clone	Создает точную копию региона
IsEmpty	Возвращает true, если регион пуст иначе false
IsVisible	Возвращает true, если заданный прямоугольник содержится в пределах региона
MakeEmpty	Помечает регион как пустой
MakeInfinite	Делает регион бесконечно большим
Transform	Устанавливает преобразование с помощью Matrix
Translate	Смещает координаты региона

а) Создание региона

Конструктор класса **Region** имеет пять перегрузок. Экземпляр класса **Region** может быть создан различными способами. Его конструктор может принимать следующие аргументы: **Rectangle, RectangleF, GraphicsPath,** или **RegionData.** Ниже показаны способы получения экземпляра класса **Region.**

```
Rectangle rect =
          new Rectangle(30, 20, 50, 80);
RectangleF rectF =
          new Rectangle(20, 30, 60, 90);
GraphicsPath path = new GraphicsPath();
path.AddRectangle(rect);

// Создание региона
Region rgn = new Region();
// Создание региона из RectangleF
Region rectFRgn = new Region(rectF);
// Создание региона из Rectangle
Region rectRgn = new Region(rect);
// Создание региона из GraphicsPath
Region pathRgn= new Region(path);
// Создание RegionData из pathRgn
RegionData regionData = pathRgn.GetRegionData();
// Создание региона из RegionData
Region fromDataRgn = new Region(regionData);
```

б) Операции с регионами

С помощью таких методов объекта **Region** как **Complement, Union, Exclude, Intersect и Xor** мы можем получать области пресечения, исключения и объединения, основанной на математической теории множеств.

7. Примеры использования регионов

Рассмотрим на примере различные операции с регионами. В Visual Studio создайте проект Windows Forms. Добавьте к форме элемент **MenuStrip** с одним **ToolStripMenuItem,** назовите его **mnuOperations.** Назовите его **mnuItemSelOperation**. Он будет дополнительно иметь разворачивающееся меню. В нем будут кнопки выполняющие различные операции над регионами. Создадим перечисление **RegionOperations**, которое будет представлять операцию над регионом.

```
public enum RegionOperations
{
    Complement,
    Intersect,
    Union,
    Exclude,
    Xor,
    No

}
```

Объявим приватные переменные внутри тела класса **Form1**

```
private  Graphics g;

private  MatrixOperations mxOperation = MatrixOperations.Reset;
```

Сгенерируйте метод обработчика события **Pain.** Объявите в нем метод **DrawRegionOperations();.**

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    DrawRegionOperations();
}
```

Наведите на нем курсор мыши. Правой клавишей вызовите контекстное меню, из списка выберите **GenerateMethod,** будет сгенериван метод. Заполнитеего логикой предложенной ниже.

```
        private void DrawRegionOperations()
        {
            g = this.CreateGraphics();
            // Создаем два прямоугольника
            Rectangle rect1 = new Rectangle(100, 100, 120, 120);
            Rectangle rect2 = new Rectangle(70, 70, 120, 120);
            // Создаем два региона
            Region rgn1 = new Region(rect1);
            Region rgn2 = new Region(rect2);
            // рисуем прямоугольники
            g.DrawRectangle(Pens.Green, rect1);
            g.DrawRectangle(Pens.Black, rect2);

            // обработаем перечисление и вызовем соответствующий метод
            switch (rgnOperation)
            {
                case RegionOperations.Union:
                    rgn1.Union(rgn2);
                    break;
                case RegionOperations.Complement:
                    rgn1.Complement(rgn2);
                    break;
                case RegionOperations.Intersect:
                    rgn1.Intersect(rgn2);
                    break;
                case RegionOperations.Exclude:
                    rgn1.Exclude(rgn2);
                    break;
                case RegionOperations.Xor:
                    rgn1.Xor(rgn2);
                    break;
                default:
                    break;
            }
            // Рисуем регион
            g.FillRegion(Brushes.Blue, rgn1);
            g.Dispose();
        }
```

В методах обработчика события **Click** производим установку перечисления **RegionOperations,** а также вызываем метод **Invalidate()**.

```
        #region ClickEvents
        private void mnuComplement_Click(object sender, EventArgs e)
        {
            rgnOperation = RegionOperations.Complement;
            this.Invalidate();
```

```
        }

        private void mnuIntersect_Click(object sender, EventArgs e)
        {
            rgnOperation = RegionOperations.Intersect;
            this.Invalidate();
        }

          ...

        #endregion
```

Скриншот ниже демонстрирует операции Complement слева, и Union справа.

Рис. 5.1

Рисунок 5.2 демонстрирует операции Intersect слева, и Exclude справа.

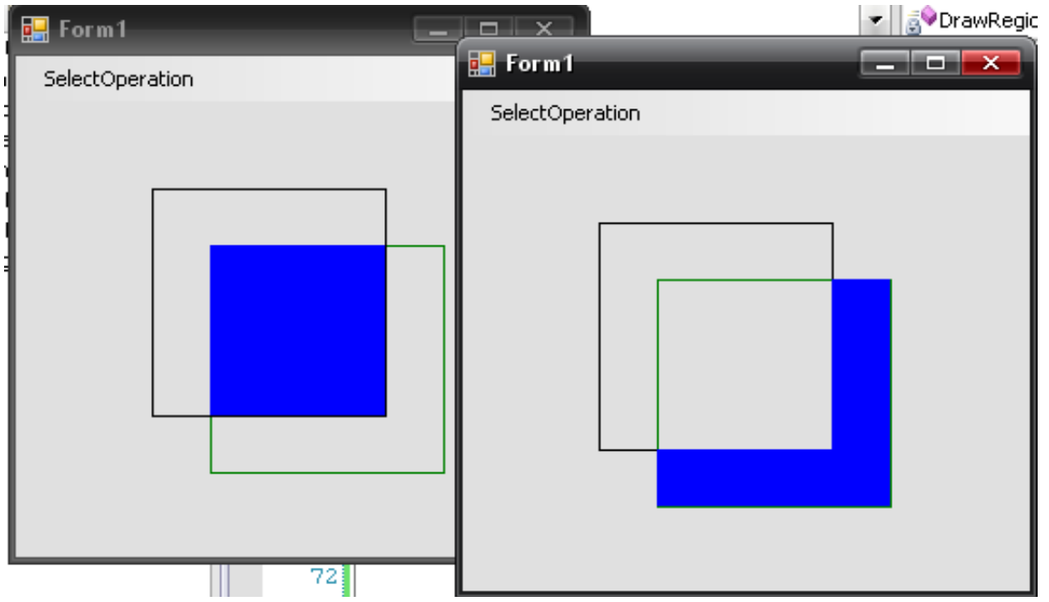

Рис. 5.2

Проект называется **RegionOperations.**

8. Понятие траектории

Траектория – это линия, по которой движется объект, изменяя свое положение в пространстве. Если перемещение объекта прямолинейное, то траектория будет прямая линия. Траектории могут быть как замкнутые, так и открытые.

9. Класс GraphicsPath

Траектории в **GDI++** представлены классом **GraphicsPath** пространства имен **System.Drawing.Drawing2D. GraphicsPath** представляет набор линий, кривых, включая графические объекты такие как прямоугольники, элипсы и текст. Траектории используются, например, для рисования контуров, заполнения внутренних областей, и при создании области отсечения.

а) Создание траектории

GraphicsPath может быть сформирован из нескольких графических объектов. Для этого используются такие методы, как **AddRectangle, AddArc, AddEllipse** и т.п.

Конструктор класса **GraphicsPath** имеет шесть перегрузок.

```
public GraphicsPath();
public GraphicsPath(FillMode fillMode);
public GraphicsPath(Point[] pts,byte[] types);
public GraphicsPath(PointF[] pts,byte[] types);
public GraphicsPath(Point[] pts,byte[] types,FillMode fillMode);
public GraphicsPath(PointF[] pts,byte[] types,FillMode fillMode);
```

Входящие аргументы **pts,** представляет массив точек, **types** массив перечесления **PathPointType**, определяющий какого типа должна быть точка в траектории. Ниже представлен пример формирования описанного массива.

```
byte[] types = {(byte)PathPointType.Start,
                (byte)PathPointType.Bezier,
                    (byte)PathPointType.DashMode };
```

Перечесление **FillMode** имеет два значения **Alternate и Winding.** Оно определяет то, как будут заполнена внутренняя область траектории. По умолчанию используется **Alternate.**

По умолчанию траектория рисуется одной формой. Чтобы отобразить несколько форм траектории необходимо вызвать метод экземпляра **GraphicsPath.StartFigure(),** далее сформировать траекторию с помощью методов **AddXXX()**.

б) Отображение траектории

После добавление различных геометрических примитивов к **GraphicsPath,** мы можем выполнить визуализацию этого объекта используя например методы**: DrawPath** или **FillPath.** Ниже представлен пример рисование траектории.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    using (Graphics g = e.Graphics)
    {
```

```
                GraphicsPath path = new GraphicsPath();

                path.AddLine(20, 20, 170, 20);
                path.AddLine(20, 20, 20, 100);
                // рисуем новую фигуру
                path.StartFigure();
                path.AddLine(240, 140, 240, 50);
                path.AddLine(240, 140, 80, 140);
                path.AddRectangle(new Rectangle(30, 30, 200, 100));
                // рисуем  path
                Pen redPen = new Pen(Color.Red, 2);
                g.FillPath(new SolidBrush(Color.Bisque), path);
                g.DrawPath(redPen, path);
            }

        }
```

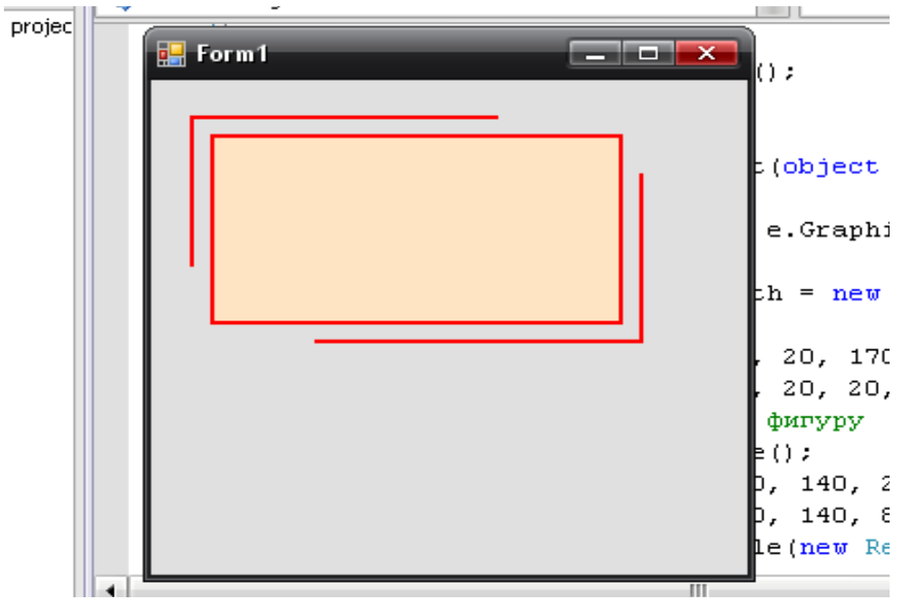

Рис. 9.1 Визуализация траектории.

Проект называется **PathRendering.**

c) Преобразование траекторий

Преобразования траекторий выполняется также как и с другими графическими примитивами **GDI++.** Мы можем выполнять как локальное преобразование, так и глобальное. Давайте рассмотрим преобразование траектории. Для этого возьмем готовый проект **PathRendering** приведенный выше и откорректируем код в событии **Paint**

```
            ...
        path.AddLine(240, 140, 80, 140);
        path.AddRectangle(new Rectangle(30, 30, 200, 100));
        // локальное преобразование траектории
        Matrix X = new Matrix();
        X.RotateAt(45, new PointF(60.0f, 100.0f));
        path.Transform(X);
        // рисуем  path
        Pen redPen = new Pen(Color.Red, 2);
        g.FillPath(new SolidBrush(Color.Bisque), path);

         ...
```

Рисунок ниже демонстрирует локальное преобразование траектории. За более детальной информацией, по преобразованию графических объектов обратитесь в главу **3**.

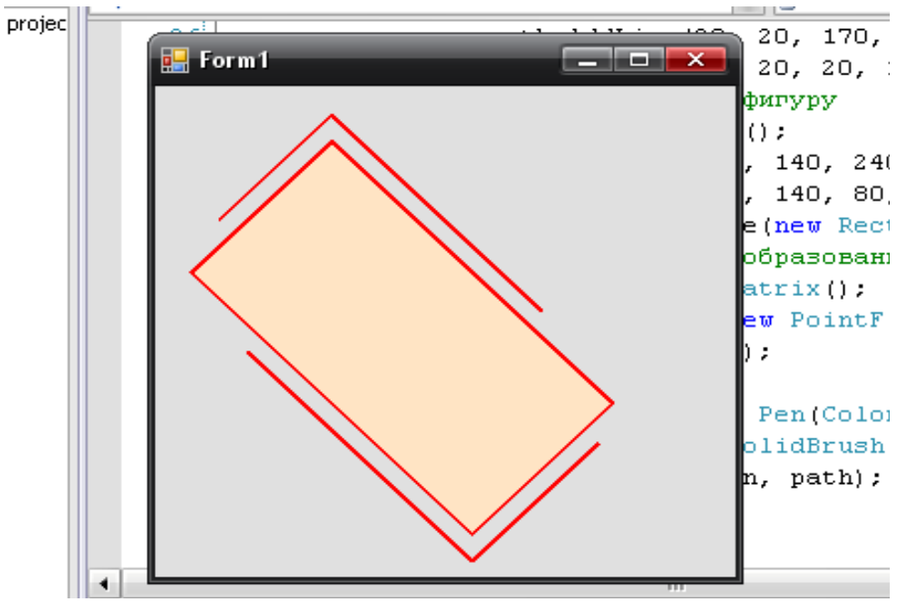

Рис. 9.2 Преобразование траектории.

Проект называется **PathRendering.**

11. Примеры использования траекторий

Рассмотрим прремер с траекторией. В Visual Studio создайте проект Windows Forms. Назовите его **PathExample.** Сгенерируйте для формы метод обработчика события **Paint.** Заполните его логикой представленной ниже.

```
        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            using (Graphics g = e.Graphics)
            {
                g.SmoothingMode = SmoothingMode.AntiAlias;
                // Создаем траекторию
                GraphicsPath path = new GraphicsPath();
                Rectangle rect = new Rectangle(20, 20, 150, 150);
                path.AddRectangle(rect);

                // Создаем градиентную кисть
                PathGradientBrush pgBrush =
                    new PathGradientBrush(path.PathPoints);
                // Устанавливаем цвет кисти
                pgBrush.CenterColor = Color.Red;
                pgBrush.SurroundColors = new Color[] { Color.Blue };
                // Создаем объект Matrix
                Matrix X = new Matrix();
                // Translate
                X.Translate(30.0f, 10.0f, MatrixOrder.Append);
                // Rotate
                X.Rotate(10.0f, MatrixOrder.Append);
                // Scale
                X.Scale(1.2f, 1.0f, MatrixOrder.Append);
                // Shear
                X.Shear(.2f, 0.03f, MatrixOrder.Prepend);

                // Применяем преобразование к траектории и кисти
                path.Transform(X);
                pgBrush.Transform = X;
                // Выполняем визуализацию
                g.FillPath(pgBrush, path);
                // Отображаем текст
                path.AddString("Hello Path!", FontFamily.Families[0],
0, 37, new Point(50, 220), null);
                g.DrawPath(new Pen(Brushes.Chocolate, 2), path);
            }

        }
```

Здесь мы используем класс **PathGradientBrush.** С помощью него мы можем заполнять градиентом внутренние области траектории. Сойства **CenterColor и SurroundColors** устанавливает цвета гадиета. Так же отобразим текст в виде оконтовки, для этого добавим его в объект **GraphicsPath.** В начале кода используем перечисление **SmoothingMode,** позволяющее определить режим сглаживания.

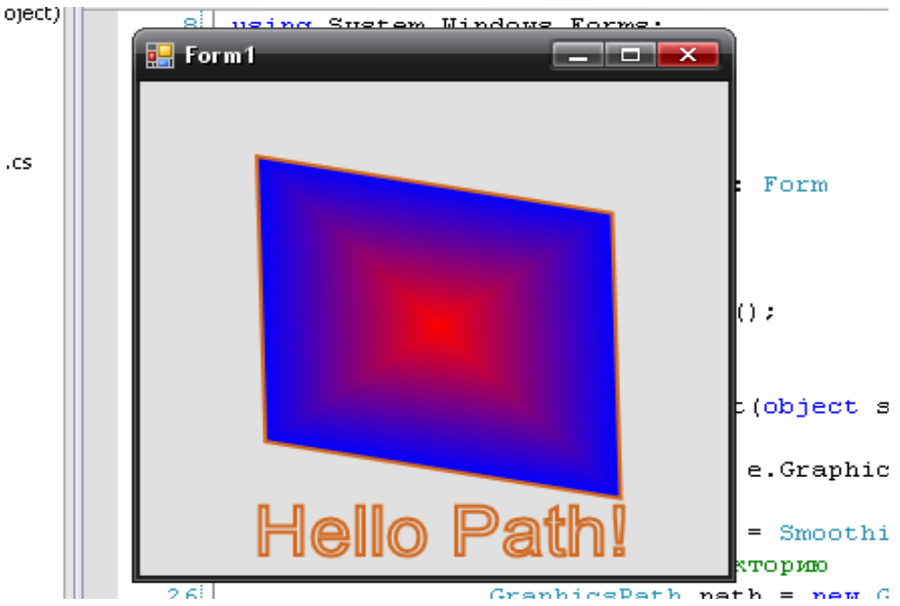

Рис. 10.1 Траектория и градиентная кисть

Проект называется **PathExample.**

Домашнее задание

1. Создайте приложение Windows Forms, которое будет рисовать график на клиентской области формы. Этот график должен отображать колебания курса доллара за последний год. Колебания курса определите сами. График должен иметь две шкалы.
2. Используя GDI++**,** разработайте логотип вымышленной компании**.** Данный логотип будет динамически загружаться в приложение. Текст компании должен иметь оконтовку.
3. Разработайте приложение Windows Forms, которое будет отображать часы на клиентской области. Часы должны иметь стрелки: часы, минуты, секунды.
4. Разработайте приложение Windows Forms «лабиринт». Обязательные требования: он должен быть проходимым и должен иметь 2 выхода.