Урок №1

Содержание

Введение в GDI++

1. Что такое GDI+?
2. Сравнительный анализ библиотек GDI+ и GDI
   a. Анализ принципов работы GDI
      - Контекст устройства.
      - Дескриптор.
      - Кисть.
      - Карандаш.
      - Шрифт.
      - Изображение.
   b. Сравнение GDI и GDI++
3. Пространство System.Drawing
4. Графические примитивы в GDI++
   a. Перья.
   b. Кисти.
   c. Шрифты.
   d. Изображения.
   e. Регионы.
   f. Траектории.
5. Системы координат.
6. Класс Graphics
   a. Цели и задачи класса Graphics.
   b. Способы получения доступа к объекту класса Graphics.
   c. Общий анализ методов и свойств класса.

7. Событие Paint.
8. Методы для вывода простейших графических примитивов
    a. Отображение точки.
    b. Отображение линии.
    c. Отображение прямоугольника.
    d. Отображение эллипса.

Введение

Итак, подошел важный момент в вашем изучении платформы **.Net Framework** это библиотека **GDI+** (Graphics Device Interface) **.**

Хотя приложения Web и Windows Forms позволяют строить мощные приложения, которые способны отображать данные из самых разнообразных источников, иногда этого бывает недостаточно. Например, мы можем пожелать нарисовать текст в определенном шрифте и цвете, отобразить изображение либо другую графику. Чтобы отобразить такого рода вывод, приложению необходимо проинструктировать операционную систему, что и как должно быть отображено. Этим и занимается **GDI+.**

1. Что такое GDI+?

GDI+ является частью операционной системы Windows XP, с помощью него мы можем разрабатывать Windows и Web приложения, позволяющие работать с векторной и растровой графикой, которые будут взаимодействовать с графическими устройствами, например: монитор компьютера, принтер или другие устройства отображения.

Приложения **GUI** (Graphical User Interface), взаимодействуя с этими устройствами, представляет данные в понятном для пользователя виде, используя при этом посредника, который принимает данные программы и направляет их устройству отображения. В результате мы получаем, к примеру, картинку на мониторе. Этим посредником и есть библиотека **GDI+.**

GDI+ не взаимодействует с устройствами отображения напрямую, а использует для этого драйвер устройства. Вывод простого текста, рисование линии или прямоугольника, печать - все это является примерами **GDI+.**

Рис. 1.1 отображает схему взаимодействия, описанную выше.

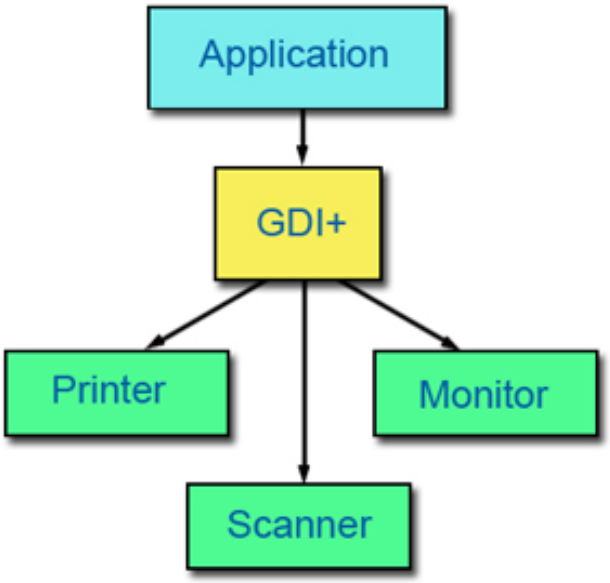


Рис. 1.1

Теперь рассмотрим подробней работу **GDI+**. Предположим, наше приложение рисует линию. Линия будет представлена как набор последовательных пикселов, у которой имеется начальная и конечная точка. Рисуя линию, монитору необходимо знать, в каком месте их рисовать. Как же приказать монитору нарисовать эти пиксели. Для этого мы используем в нашем приложении метод **DrawLine()**, библиотеки **GDI+**. **GDI+** дает инструкции операционной системе отобразить линию в виде последовательных пикселов.

2. Сравнительный анализ библиотек GDI+ и GDI.

а) Анализ принципов работы GDI

Для того чтобы получить какой - либо рисунок на экране монитора, нам необходимо такое аппаратное обеспечение как видеокарта. Компьютер дает для нее специфические команды, а та в свою очередь застав-

ляет монитор отобразить то, что нам необходимо. На рынке имеется огромное количество видеокарт разных производителей, которые имеют специфичный набор команд и возможностей. **GDI** позволяет нам абстрагироваться от этих ограничений, скрывая разницу между этими устройствами. **GDI** решает эти задачи самостоятельно, для нас нет необходимости писать код для определенного драйвера. Чтобы вывести изображение на принтер или монитор, нам нужно всего лишь вызвать соответствующие методы.

Контекст Устройства (**DC**) представляет объект Windows, который содержит набор графических объектов, информацию об их атрибутах рисования, а также определяет графические режимы устройства отображения.

Прежде чем что - либо отобразить, например, на мониторе, приложению необходимо получить этот контекст до того, как посылать выходной поток устройству. В Net.Framework это решается с помощью класса, **System.Drawing.Graphics** в который и помещен контекст устройства **DC**.

Благодаря взаимодействию **DC** наше приложение может быть оптимизировано, например, при прорисовке на экране конкретной области. С помощью контекста устройства мы можем определить в каких координатах и как отображать то, что нам необходимо.

Графические объекты представляются такими классами как **Pen -** для рисований линий, **Brush –** для рисования и заполнения форм, **Font -** для отображения текста, **Image** и другие.

b. Сравнение GDI и GDI+

GDI - это библиотека **Gdi32.dll,** она использовалась в ранних версиях Windows и базируется на старом Win32API с функциями языка C. Мы можем использовать ее функциональность в управляемом коде .NET. Чтобы применить библиотеку **GDI** в нашем приложении, мы должны им-

портировать ее с помощью типа **DLLImportAttribute**. Следующая запись показывает, как это сделать.

```
[System.Runtime.InteropServices.DllImportAttribute("gdi32.dll")]
```

После этого мы сможем использовать функциональность **gdi32.dll** библиотеки в нашем Net приложении. Вряд ли мы будем использовать ее в наших приложениях, ведь библиотека **GDI+** на управляемом языке решает многие задачи.

GDI+ является компонентом операционных систем WindowsXP и Windows Server2003. Это оболочка вокруг старой библиотеки **GDI,** она написана на языке C++ и представляет улучшенную производительность и более интуитивно понятную модель программирования, представляясь библиотекой **Gdiplus.dll.** Может применяться как в управляемом Net коде (заданной сборкой **Sysytem.Drawing.dll),** так и в неуправляемом.

Фактически библиотека NET.Framework **GDI+** также является оболочкой вокруг **GDI+** языка C++. Представляет более продвинутый API, включая автоматическое управление памятью, межъязыковую интеграцию, улучшенную безопасность, отладку, развертывание и многое другое. На рис. 2.1 представлены взаимоотношения библиотек **GDI** и **GDI+.**

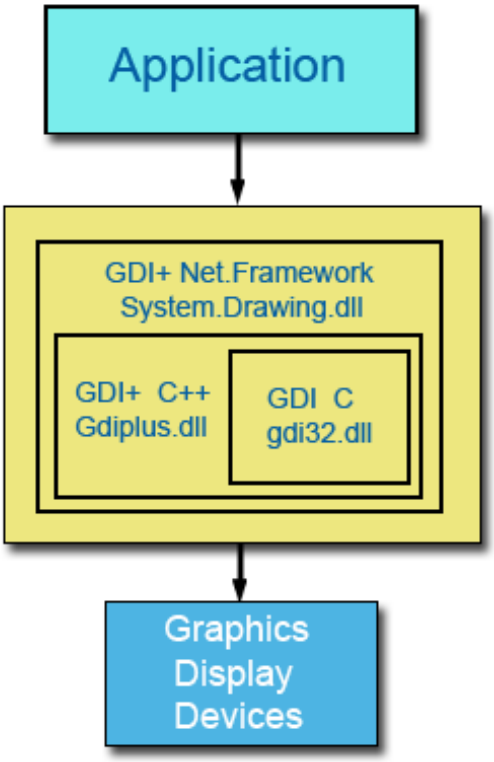


Рис.2.1 взаимоотношения библиотек **GDI** и **GDI+.**

3. Пространство System.Drawing.

Базовая функциональность управляемого **GDI+** представлена библиотекой NET.Framework, определенной в пространстве имен **System.Drawing**. В этом пространстве можно найти классы, представляющие изображения, кисти, перья, шрифты и другие типы, позволяющие работать с графикой. Дополнительная функциональность обеспечивается подпространствами: **System.Drawing.Desing, System.Drawing.Drawing2D, System.Drawing.Imaging, System.Drawing.Printing, System.Drawing.Text**. Следующая таблица представляет краткое описание этих пространств имен.

Таблица 3.1 Основные пространства имен GDI+.

Пространство имен	Описание

Пространство имен	Описание
System.Drawing	Базовое пространство имен GDI+ определяет множество типов для основных операций визуализации, например Graphics определяет методы и свойства рисования на устройствах отображения, типы Point и Rectangle например инкапсулируют примитивы GDI+, класс Pen используется при рисовании линий и кривых, классы производные от абстрактного типа Brush используются для заполнения внутренних областей графических форм таких как прямоугольники и эллипсы. Не поддерживается в Windows и ASP.NET сервисах.
System.Drawing.Desing	Пространство имен содержащее типы, обеспечивающие базовую функциональность для разработки расширений пользовательского интерфейса времени выполнения и их размещение в панели инструментов ToolBox, также включает предопределенные диалоговые окна например: FontEditor представляет редактор выбора и конфигурирования объектом Font, ColorEditor представляет редактор для визуального выбора цвета, Тип ToolBoxItem базовый класс предназначен создания и визуального отображения ToolBox элемента на панели инструментов. Не поддерживается в Windows и ASP.NET сервисах.
System.Drawing.Drawing2D	Пространство имен используется для поддержки двумерной и векторной графики В свою очередь оно сгруппировано по категориям: а)типы кистей(PathGradientBrush и HatchBrush типы позволяющие заполнять геометрические формы повторяющемся узором либо градиентом) б)перечисления связанные с рисованием линий LineCap и CustomLineCap типы определяющие стили концов линий, LineJoin перечисления определяющие как линии будут соединятся между собой,

Пространство имен	Описание
	PenAlignment перечисления определяющие как обьект Pen будет выравниваться относительно виртуальной линии, PenType перечисления определяющие заполнение линии в)перечисления связанные с заполнением геометрических форм и путей HatchStyle перечисление определяющие стиль заполнения класса HatchBrush, Blend определяет смешивание для LinearGradientBrush, перечисления FillMode определяют стиль заполнения для типа GraphicsPath г)геометрические трансформации Matrix класс представляющий матрицу 3×3 хранящий информацию о трансформировании над векторной графикой, изображением или текстом, MatrixOrder перечисление определяет порядок трансформации. Не поддерживается в Windows и ASP.NET сервисах.
System.Drawing.Printig	Классы, имеющие отношение к сервису печати в Windows Forms, например PrintDocument, PrintSettings, PageSettings, печать производится с помощью вызова метода PrintDocument.Print() при этом срабатывает событие PrintPage, которое разработчик может перехватывать, PrinterResolution представляет разрешение, поддерживаемое принтером, перечисления PaperKind определяет стандартные размеры бумаги, например A3 или A4 и множество других типов. Не поддерживается в Windows и ASP.NET сервисах а также в приложениях ASP.NET
System.Drawing.Imaging	Содержит классы, позволяющие манипулировать графическими изображениями, например, класс ImageFormat определяет формат файла изображения, перечисление ImageFlags представляющее как данные о пикселах содержатся в изображении. Не поддерживается в Windows и ASP.NET сервисах.

Пространство имен	Описание
System.Drawing.Text	Пространство, которое содержит классы для управления шрифтами, например, InstalledFontCollection позволяет получить список шрифтов, установленных на данной системе, TextRenderingHint определяет качество визуализации текста, класс PrivateFontCollection обеспечивает доступ к семейству шрифтов клиентского приложения. Не поддерживается в Windows и ASP.NET сервисах.

Это был всего лишь краткий обзор пространства имен **System.Drawing**, впереди у вас детальное знакомство с аспектами использования данной технологии.

4. Графические примитивы в GDI+.

Pen применяется в **GDI+** для рисования линий, кривых и контуров. Имеет несколько перегруженных конструкторов, позволяющих задавать цвет и ширину кисти. Этот объект используется в методах рисования, объекта Graphics. Ниже представлен пример использования класса **Pen**.

```
private void Form1_Paint(object sender, PaintEventArgs e)
 {
     Graphics g = e.Graphics;
     Pen pn = new Pen(Brushes.Blue, 5);
     pn.DashStyle = System.Drawing.Drawing2D.DashStyle.Dot;
     g.DrawEllipse(pn, 50, 100, 170, 40);
     g.Dispose();
 }
```

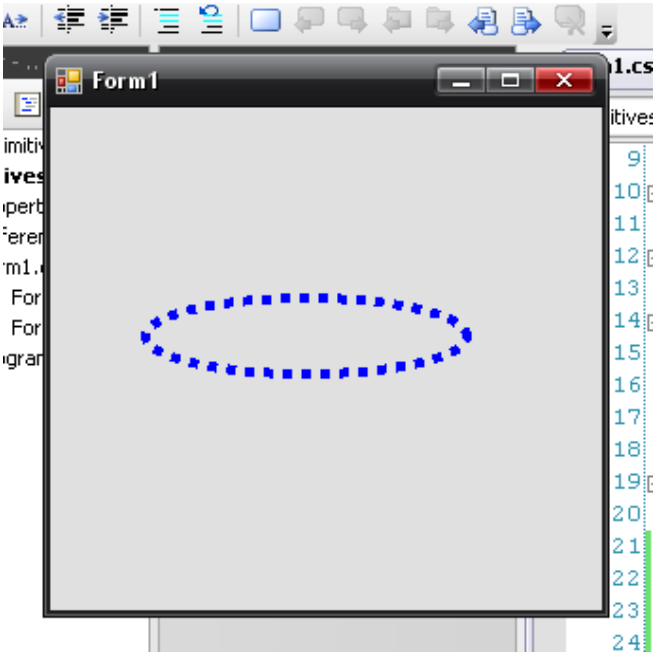


Рис. 4.1 Применение объекта **Pen**.

Проект, приведенный выше, называется **PenExample.**

Brush применяется в **GDI+** для заполнения внутренних областей графических форм таких, как прямоугольники или эллипсы.

Функциональность кисти обеспечивается пространством имен **System.Drawing** и **System.Drawing.Drawing2D**. Например, **Brush**, **SolidBrush**, **TextureBrush** и **Brushes** принадлежат пространству имен **System.Drawing,** а **HatchBrush** и **GradientBrush** заданы в пространстве имен **System.Drawing.Drawing2D**.

Класс **Brush** является абстрактным классом, поэтому не позволяет строить экземпляры непосредственно. Является базовым для таких типов, как например **LinearGradientBrush, HatchBrush, GradientBrush и** т.д.

LinearGradientBrush применяется тогда, когда необходимо смешать два цвета и получить градиентную заливку.

HatchBrush используется, когда необходимо залить геометрическую форму каким-либо узором.

TextureBrush позволяет заливать геометрические формы растровым изображением.

Ниже представлен пример использования кистей, описанных выше.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
            Graphics g = e.Graphics;
            Rectangle rect = new Rectangle(20, 20, 200, 40);
            LinearGradientBrush lgBrush =
                 new LinearGradientBrush(
                 rect, Color.Red, Color.Green, 0.0f, true);
            g.FillRectangle(lgBrush, rect);
            Rectangle rect2 = new Rectangle(20, 80, 200, 40);
            HatchBrush htchBrush = new HatchBrush(HatchStyle.Cross,
Color.Blue);
            g.FillRectangle(htchBrush, rect2);
            TextureBrush txBrush =
 new TextureBrush(new Bitmap("Background.bmp"));
            Rectangle rect3 = new Rectangle(20, 140, 200, 40);
            g.FillRectangle(txBrush, rect3);
            g.Dispose();
}
```

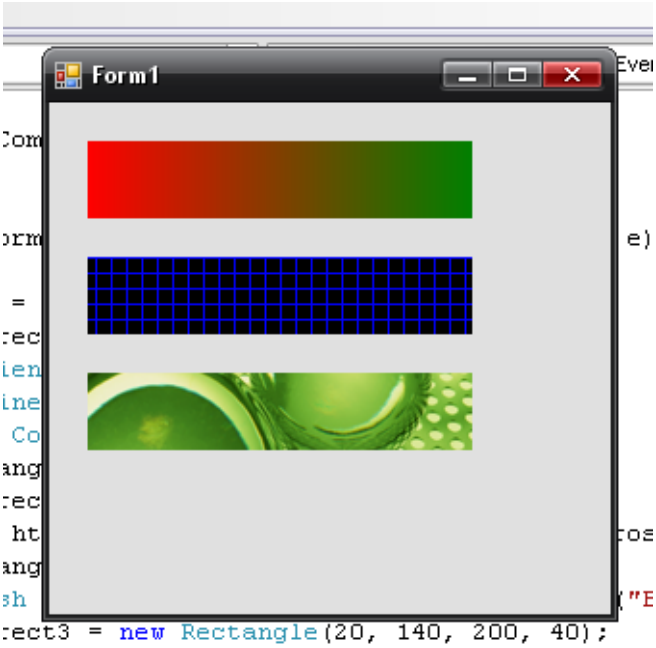


Рис. 4.2 Применение LinearGradientBrush, HatchBrush и TextureBrush.

Проект приведенный выше называется **BrushesExample.**

Font представляет шрифт, установленный на машине пользователя. **GDI** шрифты хранятся в системной директории **C:\WINDOWS\Fonts**. Шрифты делятся на растровые и векторные. Растровые шрифты визуализируются быстрее, но плохо поддаются трансформациям, например, масштабированию; векторные в отношениях к потреблению памяти более прожорливы, зато отлично трансформируются.

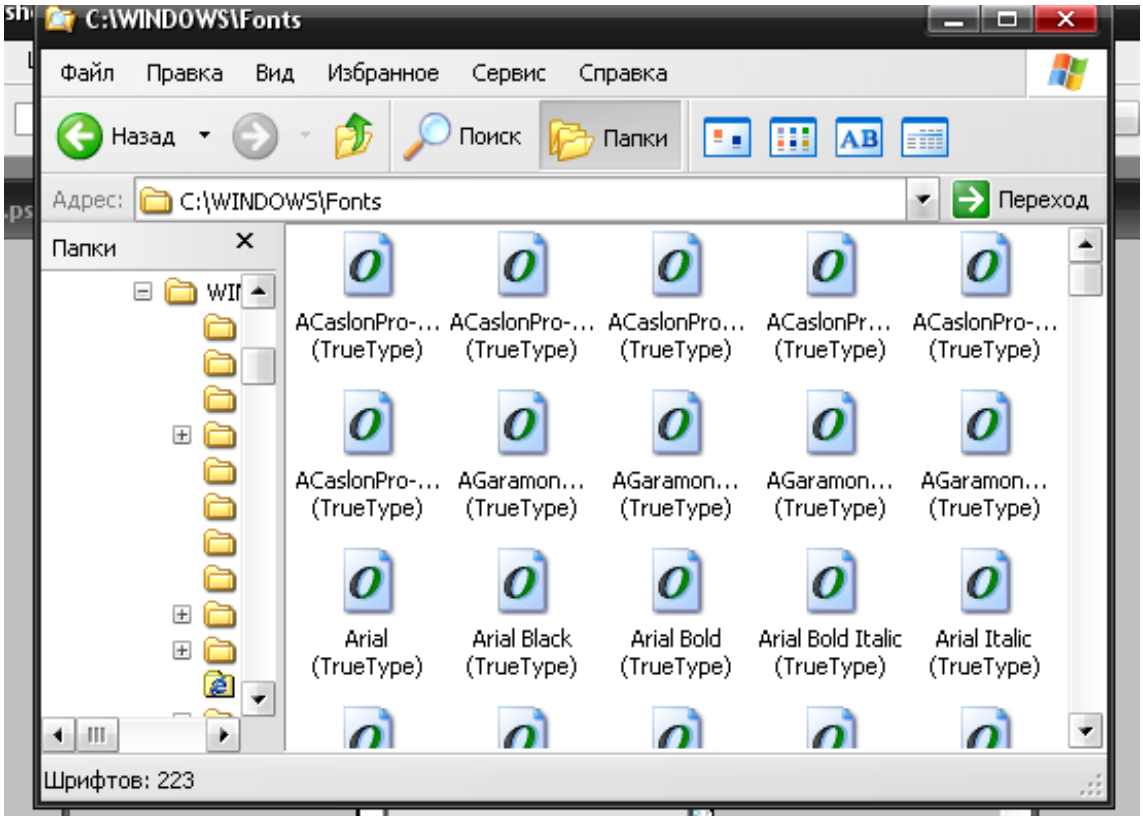


Рис. 4.3 Шрифты доступные в Windows.

В **GDI+** общая функциональность **Font** обеспечивается пространством имен **System.Drawing**, дополнительная функциональность пространством **System.Drawing.Text.**

Font используется в методе **Graphics.DrawString**, предназначенном для рисования текста. Конструктор этого класса имеет множество перегрузок, позволяющий задавать, например, гарнитуру, размер или стиль шрифта, экземпляр которого может быть создан различными способами, например:

```
Font f = new Font("Verdana", 12);
Font f = new Font("Verdana", 12, FontStyle.Bold);
Font f = new Font("Verdana", 12, FontStyle.Bold |
                                 FontStyle.Italic);
```

FontStyle перечисление, задает общий стиль для шрифта.

public enum **FontStyle**

{

Bold, Italic**,** Regular, Strikeout, Underline

}

Ниже представлен пример использования класса **Font**

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Font f = new Font("Verdana", 14, FontStyle.Bold |
                                     FontStyle.Italic);
    g.DrawString("Hello Font!", f,Brushes.Blue, 30, 55);
    g.Dispose();
}
```

Рис. 4.4 Применение объекта **Font.**

Проект приведенный выше называется **FontExample.**

Изображения, как и шрифты, деляться на два вида: растровые и векторные. Растровые представлены колекцией одного или больше пикселов, векторные же представлены коллекцией одного или больше векторов. Векторные изображения можно трансфосмировать без потери качества, в то время как при трансформации растровых изображений качество теряется.

Изображения в **GDI+** представлены абстрактным классом **Image,** определенным пространством имен **System.Drawing.** Размер изображения описывается такими свойсвами как **Widht, Height** и **Size,** разрешение - свойствами **HorizontalResolution** и **VerticalResolution.** Такие методы, как **FromImage()** и **FromStream(),** позволяют создать экземпляр объекта **Image** из указанного файла или потока. Непосредственно объект **Image** создать нельзя необходимо использовать другие способы получения, например так **Image im = new Bitmap("BgImage.bmp");.**

Рассмотрим пример с использование класса **Image.**

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Rectangle rect = this.ClientRectangle;
    Image im = new Bitmap("BgImage.bmp");
    g.DrawImage(im, rect);
    g.Dispose();
}
```

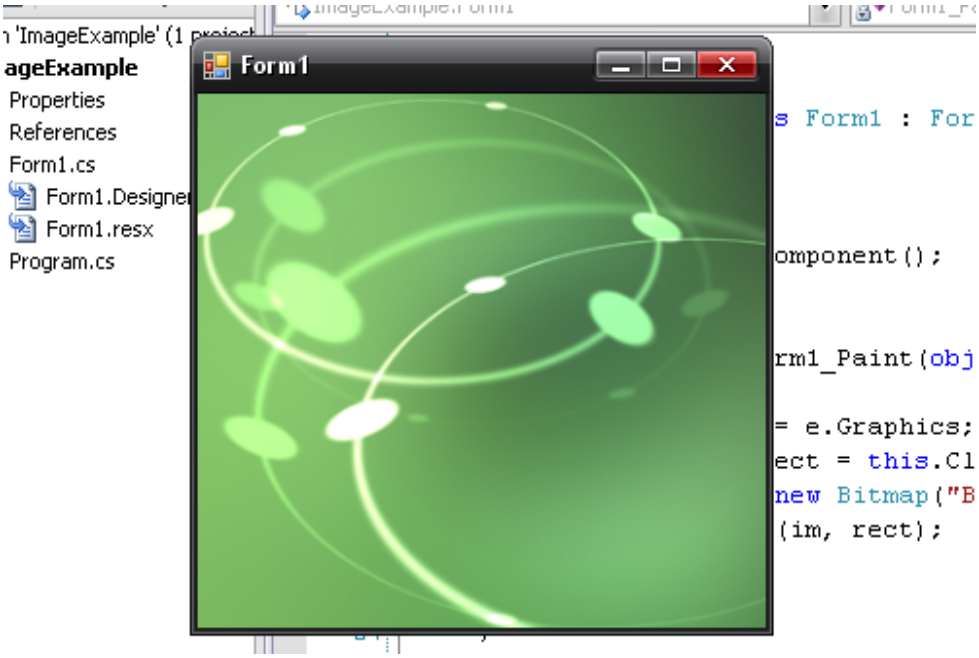


Рис. 4.5 Применение класса **Image.**

Проект приведенный выше называется **ImageExample.**

Регион – сущность, описывающая внутреннюю область замкнутых форм, в **GDI+** представлен классом **System.Drawing.Region.** С помощью него мы можем получать области пресечения, исключения и объединения, основанной на математической теории множеств. Для этих целей применяются следующие методы:

Complement - выполняет операцию объединения.

Exclude - выполняет операцию исключения.

Intersect - выполняет операцию пересечения.

Xor - выполняет операцию исключающая или (exclusive OR).

Ниже представлен пример с использованием класса регион.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.Clear(this.BackColor);
    // Создаем два прямоугольника
    Rectangle rect1 = new Rectangle(40, 40, 140, 140);
    Rectangle rect2 = new Rectangle(100, 100, 140, 140);
    // Создаем два региона
    Region rgn1 = new Region(rect1);
    Region rgn2 = new Region(rect2);
    g.DrawRectangle(Pens.Blue, rect1);
    g.DrawRectangle(Pens.Black, rect2);
    // определяем область пересеченияния
    rgn1.Intersect(rgn2);
    // заливаем ее красным цветом
    g.FillRegion(Brushes.Red, rgn1);
    g.Dispose();

}
```

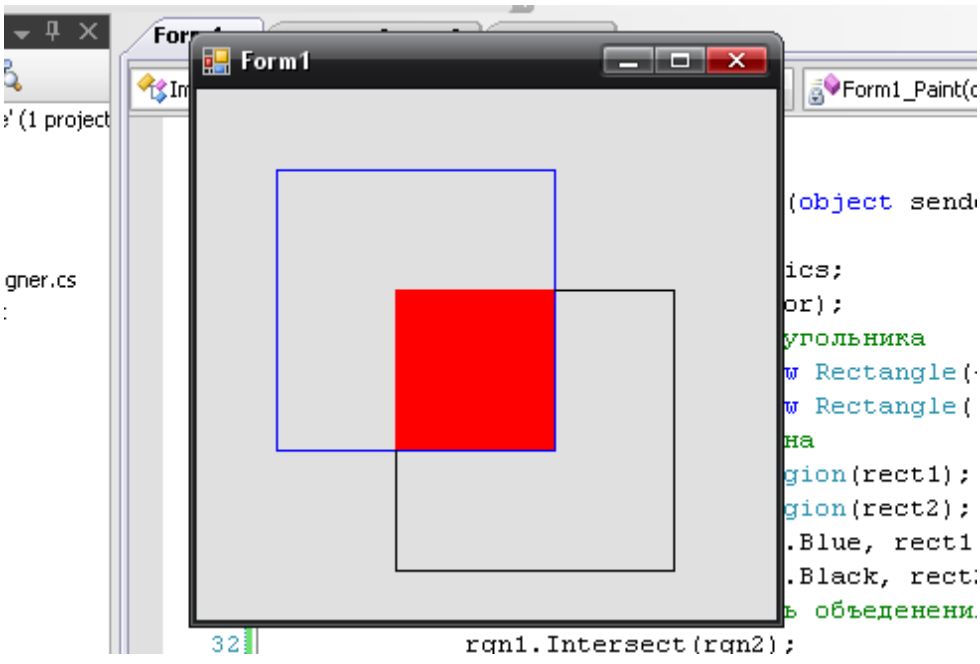


Рис. 4.6 Применение класса **Region.**

Проект называется **RegionExample.**

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

Объект **GraphicsPath** может формироваться путем последовательного объеденения геометрических форм, используя такие методы:

AddRectangle,

AddEllipse,

AddArc,

AddPolygon

Чтобы нарисовать траекторию, необходимо вызвать метод **Draw-Path** объекта **Graphics**.

Если стоит задача последовательно создать несколько траекторий, необходимо вызвать метод **StartFigure()** объекта **GraphicsPath,** далее сформировать траекторию с помощью методов **AddXXX()**. По умолчанию все траектории являются открытыми, для того чтобы явно закрыть траекторию, необходимо вызвать метод **CloseFigure().**

Ниже приведен пример с использованием класса **GraphicsPath.**

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

    //Создаем массив точек
    Point[] points = {
            new Point(5, 10),
            new Point(23, 130),
            new Point(130, 57)};

    GraphicsPath path = new GraphicsPath();
    //рисуем первую траекторию
    path.StartFigure();
    path.AddEllipse(170, 170, 100, 50);
    // заливаем траекторию цветом
```

```
            g.FillPath(Brushes.Aqua, path);
            //рисуем вторую траекторию
            path.StartFigure();
            path.AddCurve(points, 0.5F);
            path.AddArc(100, 50, 100, 100, 0, 120);
            path.AddLine(50, 150, 50, 220);
            // Закрываем траекторию
            path.CloseFigure();
            //рисуем четвертую траекторию
            path.StartFigure();
            path.AddArc(180, 30, 60, 60, 0, -170);

            g.DrawPath(new Pen(Color.Blue, 3), path);
            g.Dispose();
        }
```

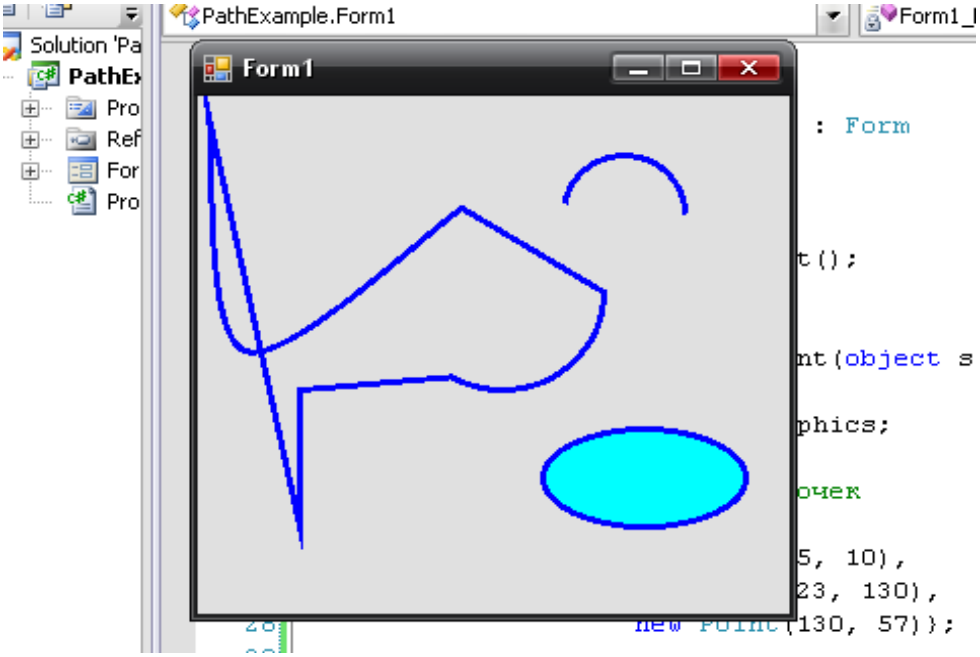


Рис. 4.6 Применение класса **GraphicsPath.**

Проект называется **PathExample**.

5. Системы координат

В **GDI+** существуют три типа координатных систем: мировые(**world**) – это позиция точки измеренная в пикселях относительно верхнего левого угла документа, страничные(**page**) - это пози-

ция точки измеренная в пикселях относительно верхнего левого угла клиентской области, устройства(**device**) – подобны страничным, за исключением того, что позиция точки может определяется например, дюймами или миллиметрами.

Перед тем как графическая форма будет нарисована на поверхности с помощью **GDI+**, она проходит через конвейер трансформаций, сначала мировые координаты преобразуются в страничные (**world transformation)**, затем страничные преобразуются в координаты устройства (**page transformation)** . На рисунке 5.1 показан конвейер трансформаций координат.

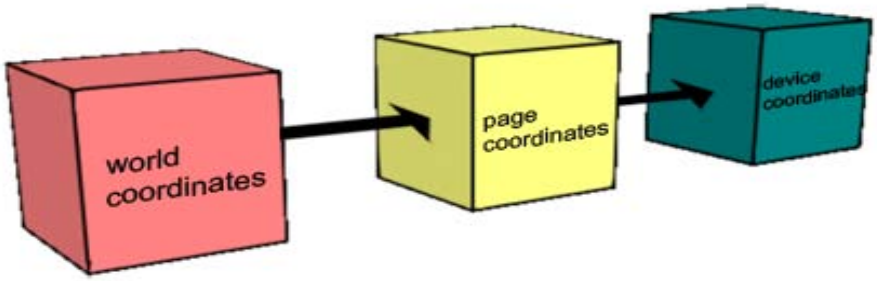


рис. 5.1 Конвейер трансформаций координат.

Координаты устройства представляют, как графические объекты будут отображаться на устройствах вывода, например монитор или принтер. Координаты устройства по умолчанию установлены в **Pixel** и совпадают со страничными. Они могут быть изменены с помощью свойства **PageUnit** класса **Graphics** например:

```
Graphics g = e.Graphics;
g.PageUnit = GraphicsUnit.Inch;
```

Координаты графических объектов могут быть заданны структурой **Point** или могут быть переданы в методах рисования в качестве параметров. **Point** – структура, находящаяся в пространстве

System.Drawing, представляет упорядоченную пару целых чисел — координат **X** и **Y**, определяющую точку на двумерной плоскости.

По умолчанию начало координат всех трех координатных систем расположено в точке (0,0) которая расположена в верхнем левом углу клиентской области и задается в пикселях. Мы можем изменить начало координат страницы с помощью метода **TranslateTransform** объекта **Graphics.** Пример представлен ниже.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    // сдвигаем начало координат страницы
    g.TranslateTransform(10, 50);
    Point A = new Point(0, 0);
    Point B = new Point(120, 120);
    g.DrawLine(new Pen(Brushes.Blue,3), A, B);

}
```

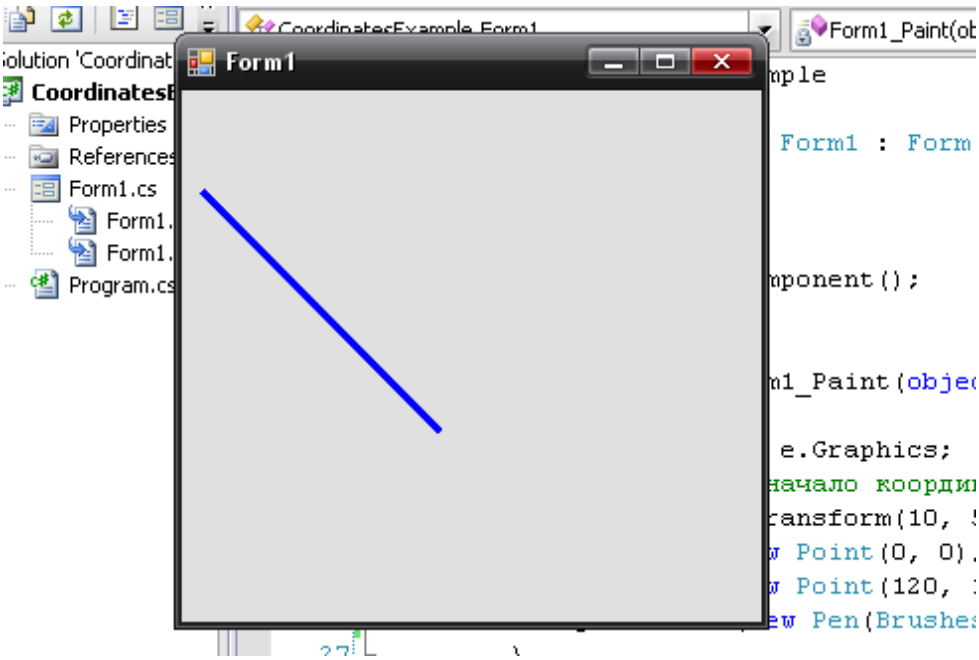


рис. 5.2 Смещение начала координат страницы.

Проект называется **CoordinatesExample.**

6. Класс Graphics

а) Цели и задачи класса Graphics

Объект **Graphics** является сердцем **GDI+,** в Net.Framework представлен классом **System.Drawing.Graphics.** Он обеспечивает основную функциональность визуализации. Как упоминалось 2-й главе **Graphics**, ассоциируется с определенным контекстом устройства. Содержит методы и свойства для рисования графических объектов, например, метод **DrawLine()** рисует линию, **DrawPoligon()** рисует полигон, чтобы нарисовать изображение или иконку, применяются методы **DrawImage()** и **DrawIcon().**

б) Способы получения доступа к объекту класса Graphics

Прежде чем приступить к рисованию линии или текста с помощью **GDI+,** мы должны получить экземпляр класса **Graphics.** Этот объект можно рассматривать как инструмент художника для рисования. Для рисования мы должны вызвать методы этого объекта.

Рассмотрим способы получения объекта **Graphics** на простом примере приложения, Windows Forms, которое будет рисовать надпись на форме “Hello World!”.Создайте новый проект в Visual Studio, назовите его GraphicsExample.

Способы получения объекта **Graphics**:

1 Получение экземпляра объекта **Graphics** из входного параметра **PaintEventArgs** события Paint. В конструкторе формы сгенерируйте обработчик события **Paint.**

```
public Form1()
```

```
        {
            InitializeComponent();
            this.Paint += new PaintEventHandler(Form1_Paint);

        }


        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            Font f = new Font("Verdana", 30,FontStyle.Italic);
            g.DrawString("Hello World!", f, Brushes.Blue, 10, 10);

        }
```

Рисунок ниже показывает работу нашего приложения.

Рис 9.1

2 Аналогичного результата можно добиться следующим образом. Переопределите виртуальный метод **OnPaint()** базового класса **Control.**

```
        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            Graphics g = e.Graphics;
            Font f = new Font("Verdana", 30, FontStyle.Italic);
            g.DrawString("Hello World!", f, Brushes.Blue, 10, 10);
```

```
        }
```

3 Следующий способ получения экземпляра объекта **Graphics** мы можем достичь, вызвав метод формы **this.CreateGraphics().** Исправьте код написанный выше.

```
        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            Graphics g = this.CreateGraphics();
            ...

        }
```

4 Последний способ, который мы рассмотрим с вами, - это получение **Graphics** с помощью объекта, производного от абстрактного класса **Image**. Создайте новый проект в Visual Studio, назовите его GraphicsExample2. Перетащите на форму кнопку из панели инструментов, назовите ее **btnSave,** сгенерируйте для нее метод обработчика события **Click**, в этом методе мы выполним загрузку изображения, затем с помошью GDI+ нарисуем на нем текст, возьмем его в рамку и сохраним под другим именем. Файл изображения создайте в графическом редакторе, например Photoshop, и поместите его в папку **bin->Debug** приложения.

```
        private void btnSave_Click(object sender, EventArgs e)
        {
            try
            {
                Bitmap myBitmap = new Bitmap(@"Background.bmp");
              // получаем объект Graphics
                Graphics gFromImage = Graphics.FromImage(myBitmap);
                Font f = new Font("Verdana", 8, FontStyle.Italic);
                string helloStr = "Hello World!";
              // меряем "Hello World!" с помошью метода  MeasureString
                SizeF sz = gFromImage.MeasureString(helloStr, f);
                gFromImage.DrawString("Hello World!", f, Brushes.Blue,
10, 10);
                gFromImage.DrawRectangle(new Pen(Color.Red, 2),
```

```
                10.0F, 10.0F, sz.Width, sz.Height);
            // сохроняем изображение на диск
              myBitmap.Save(@"NewBackground.bmp");
              Rectangle regionRec = new Rectangle(new Point(0,0),
myBitmap.Size);
              myBitmap.Dispose();
              gFromImage.Dispose();
            // этот метод выполняет перерисовку клиентской области
              this.Invalidate(regionRec);
          }
          catch { }

        }
```

На случай отсутствия файла Background.bmp поместим наш код в блок try-catch. Метод **Graphics.FromImage** отвечает за создание экземпляра класса **Graphics,** метод **Invalidate** является перегруженным, может не иметь параметров и предназначен для принудительной перерисовки клиентской области. Область прорисовки задаем структурой **Rectangle** и паредаем его в качестве параметра методу **Invalidate().** Так как объект **Graphic** использует различные неуправляемые ресурсы, рекомендуется применять метод **Dispose().**

Теперь добавьте обработчик события **Paint** и заполните его как представлено ниже.

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

                Bitmap myBitmap = new Bitmap(@"NewBackground.bmp");
                Graphics g = e.Graphics;
                g.DrawImage(myBitmap, 0, 0, 300, 200);
                myBitmap.Dispose();
                g.Dispose();
            }
            catch { }
```

```
        }
```

Запустите приложение, нажмите на кнопку **Save.** Зайдите в папку Debug приложения, заметьте что добавился новый файл, под названием **NewBackground.bmp**.

c) Общий анализ методов и свойств класса Graphics

Методы класса **Graphics** разделены на три категории: методы заполнения внутренних областей геометрических форм **FillXXX(),** методы рисования **DrawXXX(),** разносторонние методы, например, такие как **Clear(), Save(), MeasureString(), и т.д..**

Таблица 6.1 Основные методы рисования **Graphics.**

Методы	Описание
DrawArc() DrawBezier() DrawBeziers() DrawCurve() DrawClosedCurve() DrawIcon() DrawEllipse() DrawLine() DrawLines() DrawPie() DrawPath() DrawRectangle() DrawString() DrawImage()	Методы применяются для рисования графически объектов, почти все методы перегружены и могут принимать различное количество параметров.

Таблица 6.2 Методы заполнения внутренних областей **Graphics.**

Методы	Описание

Методы	Описание
FillClosedCurve() FillEllipse() FillPath() FillPie() FillPoligon() FillRectangle() FillRegion() FillRectangles()	Методы применяются для заполнения внутренних областей графических форм, почти все методы перегружены и могут принимать различное количество параметров.

Таблица 6.3 Основные разносторонние методы **Graphics.**

Методы	Описание
Clear()	Приводит к очистке объекта Graphics и заливает его определенным цветом, который передается через входящий параметр.
ExcludeClip()	Обновляет область отсечения, исключая регион определенной структурой Rectangle.
AddMetafileComment()	Добавляет комментарии к метафайлу.
FromImage() FromHdc() FromHwnd()	Методы позволяющие создать объект Graphics, например из изображения, точечного рисунка или GUI элемента.
GetNearestColor()	Возвращает ближайший цвет, определенный структурой Color
IntersectClip()	Обновляет область отсечения объекта Graphics определенной текущей областью отсечения и структурой Rectangle.
IsVisible()	Возвращает true, если точка находится внутри видимой области отсечения.
MeasureString()	Измеряет размер строки, исходя из определенного шрифта и возвращает SizeF

Методы	Описание
MultiplyTransform()	Перемножает глобальные трансформации объекта Graphics и Matrix
ResetClip()	Восстанавливает область отсечения объекта Graphics по умолчанию
ResetTransform()	Восстанавливает матрицу трансформаций объекта Graphics
Restore()	Восстанавливает состояние объекта Graphics, которое определяется входящим параметром GraphicsState
RotateTransform() ScaleTransform()	Применяет вращение или масштабирование к матрице трансформаций
TransformPoints()	Трансформирует массив точек из одной координатной системы в другую

Таблица 6.4 Основные Свойства **Graphics.**

Методы	Описание
Clip ClipBounds VisibleClipBoud IsClipEmpty IsVisibleClipEmpty	Свойства позволяющие работать с областью отсечения объекта Graphics
InterpolationMode	Возвращает или устанавливает режим интерполяции
DpiX DpiY	Возвращает горизонтальное или вертикальное разрешение объекта Graphics
CompositionMode CompositionQuality	Свойства позволяющие управлять качеством визуализации композиции
Transform	Возвращает или устанавливает мировые трансформации, заданные типом Matrix
TextRenderingHint	Возвращает или устанавливает режим визуализации для текста
TextContrast	Возвращает или устанавливает величину

Методы	Описание
	Гамма коррекции для текста
VisibleClipBounds	Возвращает границы видимой области отсечения
SmoothingMode	Возвращает или устанавливает качество визуализации для объекта Graphics

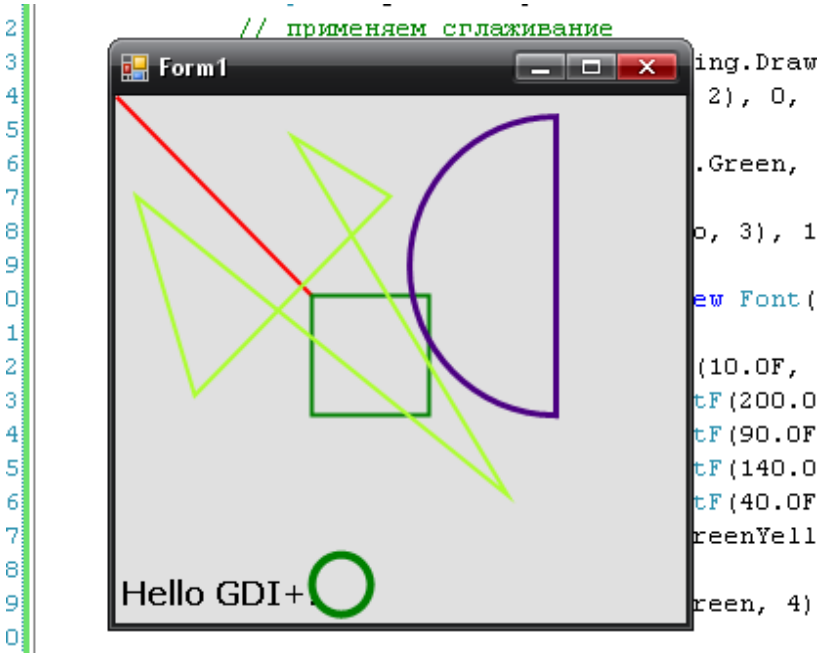
Большинство методов рисования являются перегруженными и могут принимать различное количество параметров. Например, метод **DrawLine** имеет следующие перегрузки:

```
public void DrawLine(Pen, Point, Point);

public void DrawLine(Pen, PointF, PointF);

public void DrawLine(Pen, int, int, int, int);

public void DrawLine(Pen, float, float, float, float);
```

Рассмотрим пример рисования различных геометрических форм. В Visual Studio создайте новый проект **Windows Forms**, назовите его DrawMetods. Сгенерируйте для формы метод обработчика события **Pain**, заполните его программной логикой предложенной ниже.

```
        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            // применяем сглаживание
            g.SmoothingMode =
System.Drawing.Drawing2D.SmoothingMode.HighQuality;
            g.DrawLine(new Pen(Color.Red, 2), 0, 0, 100, 100);
            // рисуем прямоугольник
            g.DrawRectangle(new Pen(Color.Green, 2), new
Rectangle(100, 100, 60, 60));
            // рисуем пирог
            g.DrawPie(new Pen(Color.Indigo, 3), 150, 10, 150, 150, 90,
180);
            // рисуем текст
            g.DrawString("Hello GDI+!", new Font("Verdana",
12,FontStyle.Bold), Brushes.Black, 0, 240);
            // рисуем полигон
            PointF[] pArray = {new PointF(10.0F, 50.0F),
                               new PointF(200.0F, 200.0F),
```

```
                                    new PointF(90.0F, 20.0F),
                                    new PointF(140.0F, 50.0F),
                                    new PointF(40.0F, 150.0F)};
            g.DrawPolygon(new Pen(Color.GreenYellow, 2), pArray);
            // ресуем элипс
            g.DrawEllipse(new Pen(Color.Green, 4), 100, 230, 30, 30);
            g.Dispose();

        }
```



На рис. 6.2 Работа с методами **Graphics**

7. Событие Paint

Чтобы понять работу события **Paint,** давайте рассмотрим следующий пример.

Запустите Visual Studio, выберите **File->New->Project** отметьте тип проекта Windows Forms Application, назовите его **PaintExample** нажмите ok.

В методе InitializeComponent() файла Fosm1.Desiner.cs который можно ныйти в окне Solution Explorer меняем цвет фона и размер окна.

```
    private void InitializeComponent()
    {
        this.SuspendLayout();
        //
```

```
        // Form1
        //

        ...

        // меняем цвет фона и размер окна
        this.BackColor = System.Drawing.SystemColors.AppWorkspace;
        this.ClientSize = new System.Drawing.Size(400, 400);
    }
```

Затем добавляем в конструктор формы файла Form1.cs следующую логику.

```
        public Form1()
        {
            InitializeComponent();
            this.Show();
            Graphics g = this.CreateGraphics();
            SolidBrush redBrush = new SolidBrush(Color.Red);
            Rectangle rect = new Rectangle(0, 0, 250, 140);
            g.FillRectangle(redBrush, rect);
        }
```

Теперь запустите приложение, результат будет выглядеть, как показано на рисунке 4.1

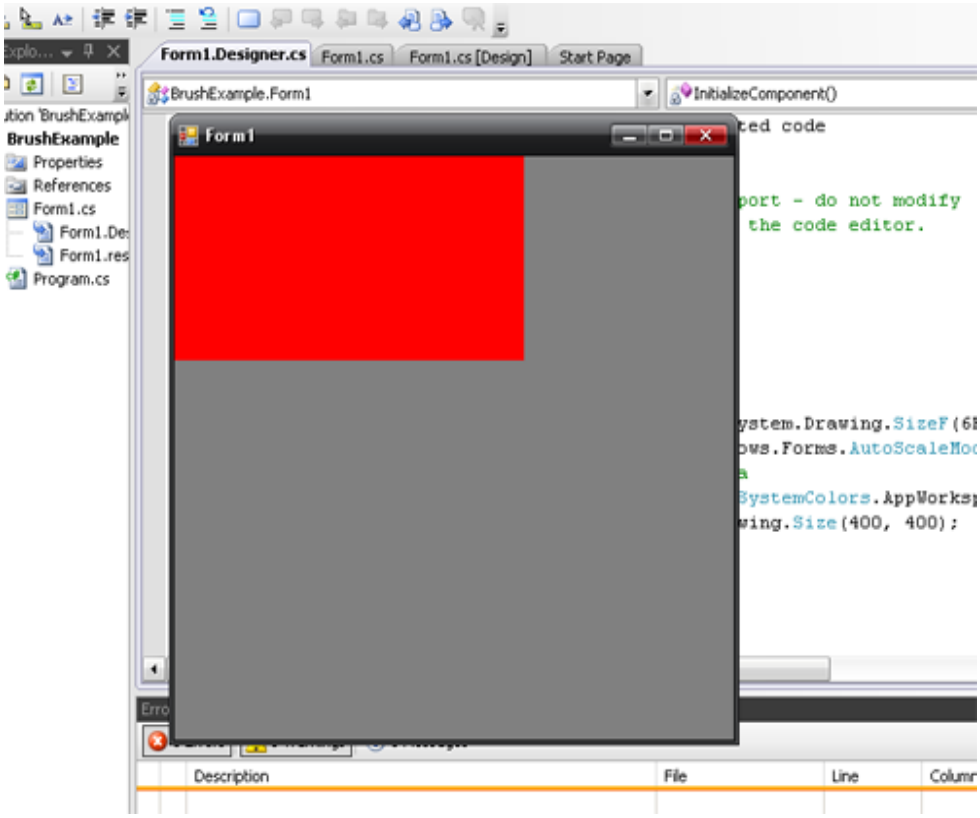


рис. 4.1

Теперь попробуйте минимизировать окно с помощью кнопки вверху слева и восстановить его, мы видим, что наш красный прямоугольник исчезает и больше не отображается. Проблема также возникнет, если мы попытаемся разместить другое окно над нашим, наш квадрат частично сотрется рис 4.2.

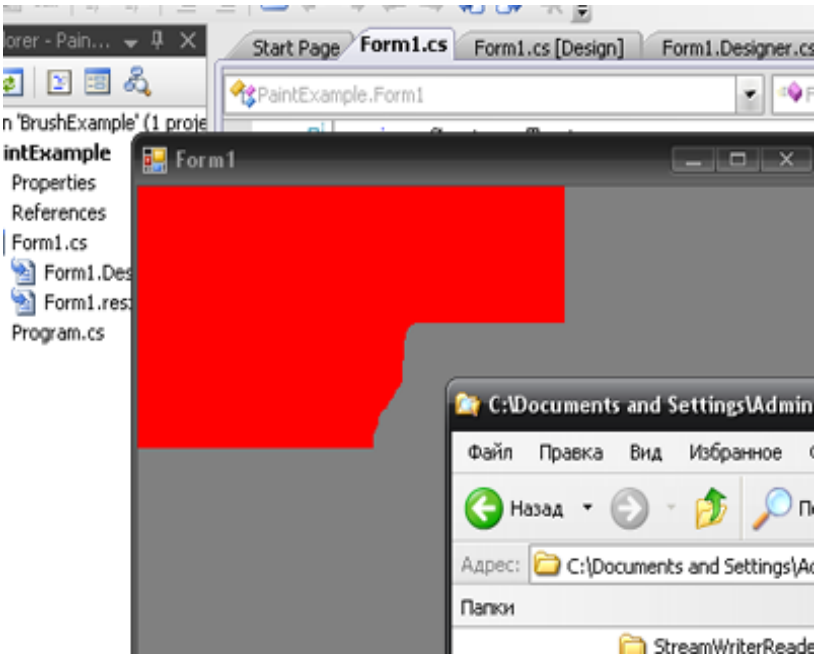


Рис. 4.2

Что же происходит? Дело в том, что Windows скрывает невидимую часть окна и при этом освобождает память. На рабочем столе у нас может быть открыто несколько десятков окон. Если бы Windows сохранял всю визуальную информацию, то память видеокарты была бы сильно загружена.

Проблема в том что, мы разместили код, который рисует прямоугольник, в теле конструктора и тот срабатывает только один раз при вызове приложения. Для того чтобы решить данную проблему, нашей форме необходимо событие, которое прорисовывало форму, когда это необходимо.

Для этого в Windows Forms предусмотрено событие **Paint**, Windows возбуждает это событие, форма производит прорисовку.

Код, рассмотренный выше, не является правильным и представлен лишь в целях демонстрации, как происходит прорисовка формы.

Теперь давайте исправим наш код. Удалите запись, которую написали в конструкторе. Переопределите виртуальный метод класса **Control OnPaint().**

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();

    }
    protected override void OnPaint(PaintEventArgs e)
    {
        base.OnPaint(e);
        Graphics g = e.Graphics;
        SolidBrush redBrush = new SolidBrush(Color.Red);
        Rectangle rect = new Rectangle(0, 0, 250, 140);
        g.FillRectangle(redBrush, rect);

    }
}
```

Запустите приложение. Как видите, теперь все работает отлично и проблемы, которые были раньше исчезли.

Как альтернативный способ получения метода события мы можем достичь следующим способом: В Visual Studio зайдите в дизайнере формы **Form1.cs[Desing]*,** правой клавиши мыши вызовите контекстное меню и выберите **Properties,** найдите в этом окне событие **Paint** и щелкните на нем два раза левой клавишей мыши рис 7.1.

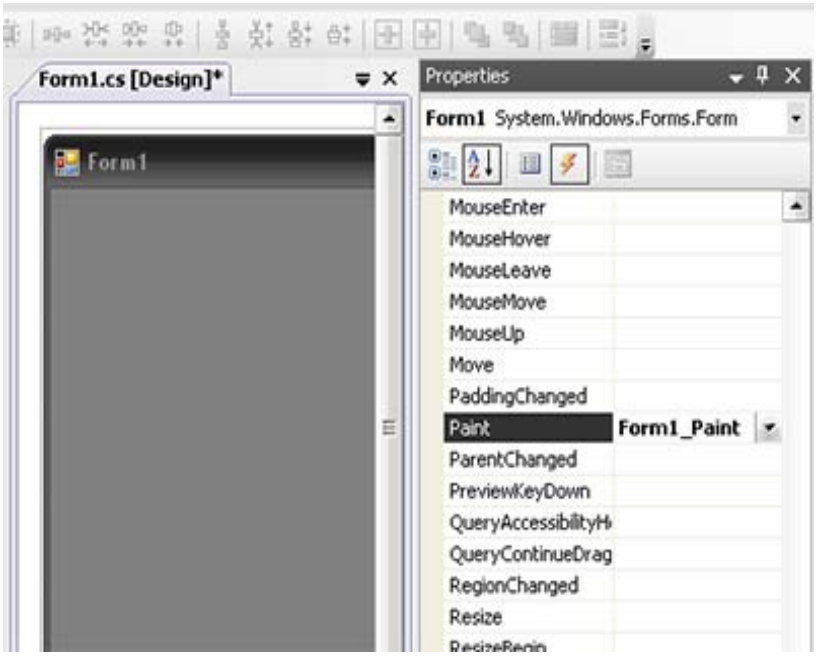


рис 7.1

Заметьте. что метод обработчика события был автоматически сгенерирован.

```
public partial class Form1 : Form
 {
     public Form1()
     {
         InitializeComponent();

     }

     private void Form1_Paint(object sender, PaintEventArgs e)
     {

     }
 }
```

Теперь заполните его программной логикой предложенной ниже.

```
        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            SolidBrush redBrush = new SolidBrush(Color.Red);
            Rectangle rect = new Rectangle(0, 0, 250, 140);
            g.FillRectangle(redBrush, rect);

        }
```

Вам необходимо запомнить, что событие **Paint** генерируется всегда, когда необходима прорисовка формы.

Теперь давайте проследим, как и сколько раз происходит событие **Paint,** сделать это довольно просто добавим в наш код счетчик, который будет вести подсчет событий **Paint** а также можно будет наблюдать когда оно происходит.

```
public partial class Form1 : Form
{
    private int paintCount;
    public Form1()
    {
        InitializeComponent();
        paintCount = 0;

    }

     ...
```

Проинициализируем его в конструкторе нулевым значением, подсчет ведем в методе **Form1_Paint** и выводим значение на форму с помощью элемента **Label.**

```
    private void Form1_Paint(object sender, PaintEventArgs e)
    {
        ...
        this.label1.Text =
              String.Format("paintCount: {0}", paintCount++);

    }
```

Запустите приложение. Видно, что при запуске счетчик равен 1. В меню пуск выберите проводник и наведите на нашу форму, заметьте, что счетчик увеличивается только когда мы сдвигаем проводник вниз рис 7.2.

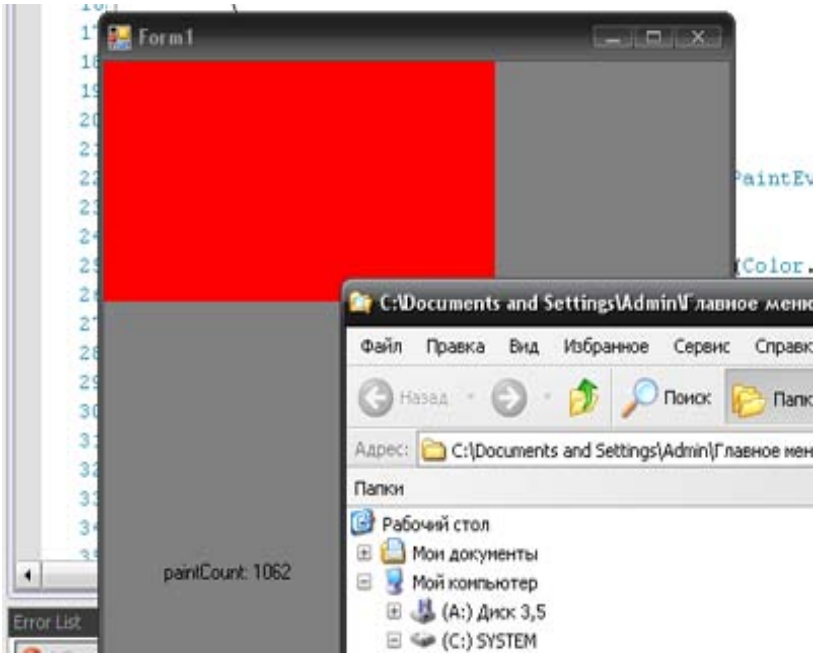


Рис 7.2

Очевидно, что событие **Paint** срабатывает, когда проводник перекрывает любую область формы. Возникает вопрос, зачем нам перерисовывать всю форму, если необходимо перерисовать только красный прямоугольник?

В **GDI+** применяется такая терминология, как область отсечения (**Clipping Region**). Она определяет то место на форме, где будет происходить перерисовка. С помощью нее можно уведомить контекст устройства DC, какую область необходимо перерисовать.

Аргумент **PaintEventArgs** события **Paint** может получить доступ к области отсечения **ClipRectangle**, который представлен экземпляр структуры **System.Drawing.Rectangle.** Структура имеет четыре свойства: **Top, Bottom, Left, Right,** описывающие вертикальные и горизонтальные координаты.

Теперь давайте изменим наш код.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
     // с помощью этого условия определим зону отсечения
     //   140 и 250 это размеры нашего прямоугольника
  if (e.ClipRectangle.Top < 140 && e.ClipRectangle.Left < 250)
    {
        Graphics g = e.Graphics;
```

```
                SolidBrush redBrush = new SolidBrush(Color.Red);
                Rectangle rect = new Rectangle(0, 0, 250, 140);
                g.FillRectangle(redBrush, rect);

                this.label1.Text =
                    String.Format("paintCount: {0}", paintCount++);
            }
        }
```

Запустите программу. Заметьте, что счетчик будет увеличиваться только тогда, когда проводник будет находится над нашим прямоугольником. Можно сказать, что мы только что повысили производительность нашего приложения.

Данный проект называется **PaintExample.**

9. Методы для вывода простейших графических примитивов.

a. Отображение точки

Рассмотрим пример построения точки на клиентской области. Метода для рисования точки в классе **Graphics** не существует. Чтобы нарисовать точку, можно залить внутреннюю область таких геометрических форм, как прямоугольник или эллипс и использовать такие методы, как **FillRectangle()** или **FillEllipse()** объекта **Graphics**.

Создайте новый Windows Forms проект, назовите его **DrawPoints**. Сгенерируйте для формы метод обработчика события **Paint,** заполните его следующей программной логикой.

```
    public partial class Form1 : Form
    {
        List<Point> points = new List<Point>();
        public Form1()
        {
            InitializeComponent();
        }
```

```
        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            // рисуем все точки в коллекции
            foreach (Point p in points)
                g.FillEllipse(Brushes.Black, p.X, p.Y, 10F, 10F);
        }

        private void Form1_MouseClick(object sender, MouseEventArgs e)
        {
            // добавляем в коллекцию новую точку
            points.Add(new Point(e.X, e.Y));
            // выполняем перерисовку клиентской области
            Invalidate();
        }

    }
```

Чтобы множество точек отображались на клиентской области, мы создаем коллекцию, наполняем ее нажатием левой кнопки мыши. С помощью аргумента **MouseEventArgs** метода обработчика события **MouseClick** передаем координаты конструктору класса **Point,** экземпляр которого передаем методу **Add(),** который наполняет коллекцию **points.** Прорисовку всех точек выполняем в методе обработчика события **Paint.**

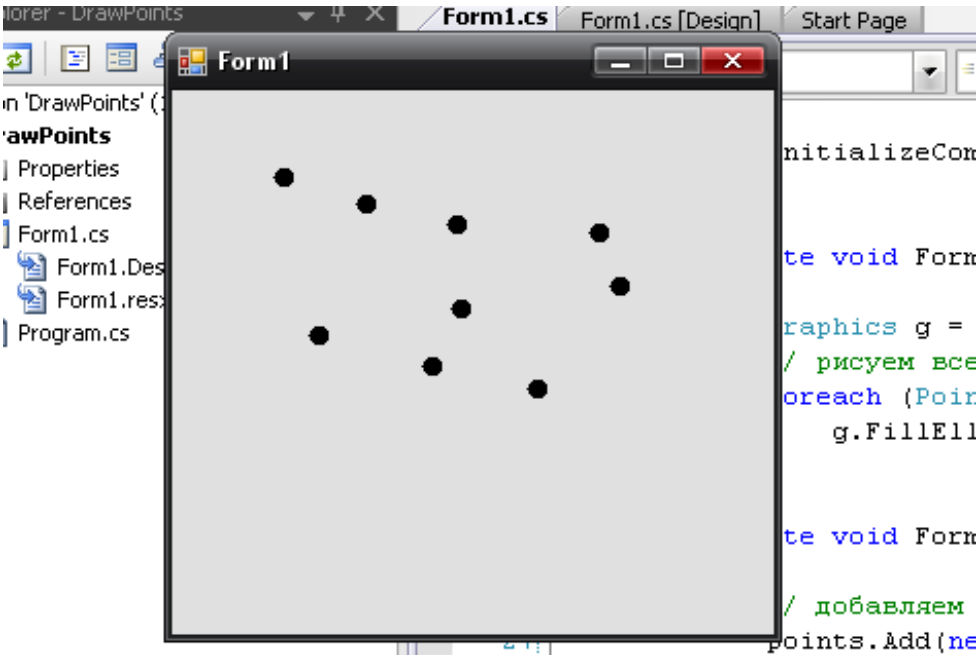


Рис. 8.1 Отображение точки.

Проект называется **DrawPoints.**

b. Отображение линии.

В предыдущих главах вы уже наблюдали, как строится линия, мы использовали метод **DrawLine()** объекта **Graphics**. Линии могут иметь различные стили. Например, мы можем нарисовать штрихпунктирную линию со стрелкой на конце(**Cap**).

Линия состоит из трех частей: тело линии, начальный и конечный **Cap.** Часть, которая соединяет концы линии, называется телом(**Body**).

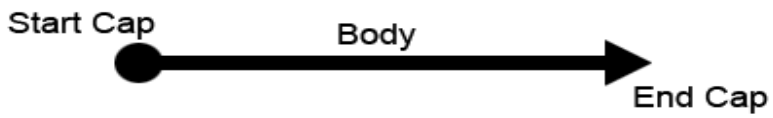


Рис 8.2

Стиль линии определяется классом **Pen,** с помощью его свойств и методов мы можем определить, как будут выглядеть концы и тело линии. Например, свойство **StartCap** устанавливает стиль для начальной точки линии, с помощью **DashStyle** можно задать пунктирный узор для тела линии, **DashCap** позволяет задать, как будет представлены концы пунктирных линий, например: **Flat**, **Round** или **Triangle**.

Рассмотрим пример построения стилизованной линии. Создайте новый Windows Forms проект, назовите его **DrawLine**. Добавьте пространство имен **System.Drawing.Drawing2D**. Сгенерируйте для формы метод обработчика события **Paint,** заполните его следующей программной логикой.

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

```
            Graphics g = e.Graphics;
            g.SmoothingMode = SmoothingMode.HighQuality;
            Pen bluePen = new Pen(Color.Blue, 6);
            // Устанавливаем стиль для концов и тела линии
            bluePen.StartCap = LineCap.SquareAnchor;
            bluePen.EndCap = LineCap.ArrowAnchor;
            bluePen.DashStyle = DashStyle.Dash;
            bluePen.DashCap = DashCap.Round;
            g.DrawLine(bluePen, 20, 100, 270, 100);
            bluePen.Dispose();
            g.Dispose();

        }
```

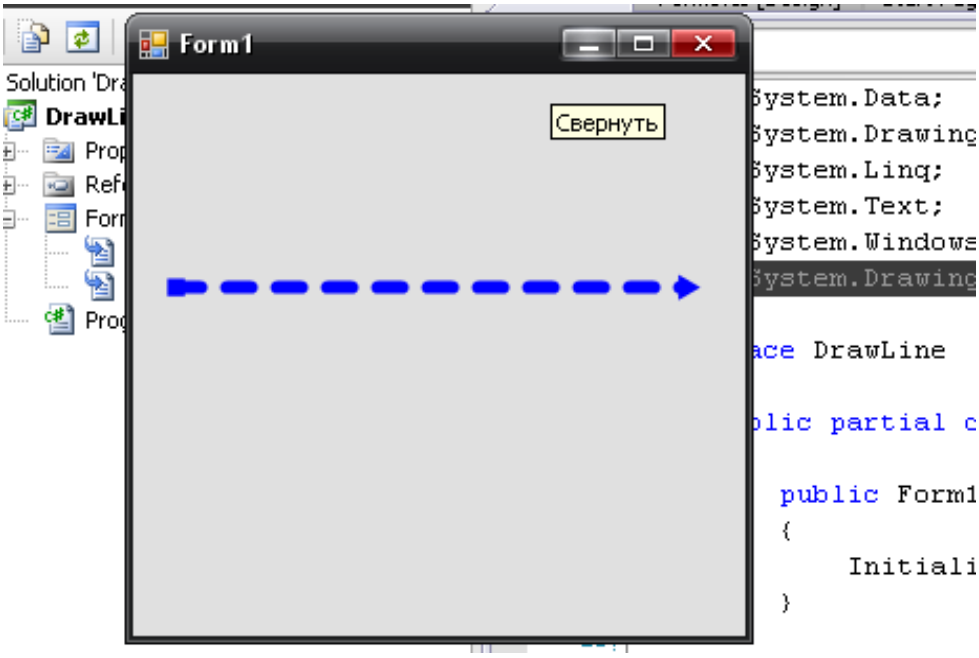


Рис. 8.3 Отображение линии.

Проект называется **DrawLine.**

c. Отображение прямоугольника.

Существует несколько способов создания прямоугольника, **Rectangle**. Мы можем создать прямоугольник, предав четыре значения в качестве параметров конструктору структуры **Rectangle,** которые пред-

ставляют начальную точку и размер, или передав такие структуры как **Point** и **Size**.

Прямоугольник так же может быть создан с помощью структуры **RectangleF,** который является зеркальным отражением структуры **Rectangle,** включает те же свойства и методы, за исключением того, что **RectangleF** принимает значения с плавающей точкой.

Чтобы отобразить прямоугольник на клиентской области, экземпляр **Rectangle** необходимо передать в качестве параметра методу **DrawRectangle()** или **FillRectangle()** объекта **Graphics.** Первый параметр методов принимает объект **Pen** либо **Brush,** второй структуру **Rectangle** который представляет позицию и габариты прямоугольника.

Рассмотрим пример рисования прямоугольника. Создайте новый Windows Forms проект, назовите его **DrawRectangle**. Сгенерируйте для формы метод обработчика события **Paint,** заполните его следующей программной логикой.

```
        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            int x = 20;
            int y = 30;
            int height = 60;
            int width = 60;
            // Создаем начальную точку
            Point pt = new Point(10, 10);
            // Создаем размер
            Size sz = new Size(160, 140);
            // Создаем два прямоугольника
            Rectangle rect1 = new Rectangle(pt, sz);
            Rectangle rect2 = new Rectangle(x, y, width, height);
            // Отображаем прямоугольники
            g.FillRectangle(Brushes.Black, rect1);
            g.DrawRectangle(new Pen(Brushes.Red, 2), rect2);

        }
```

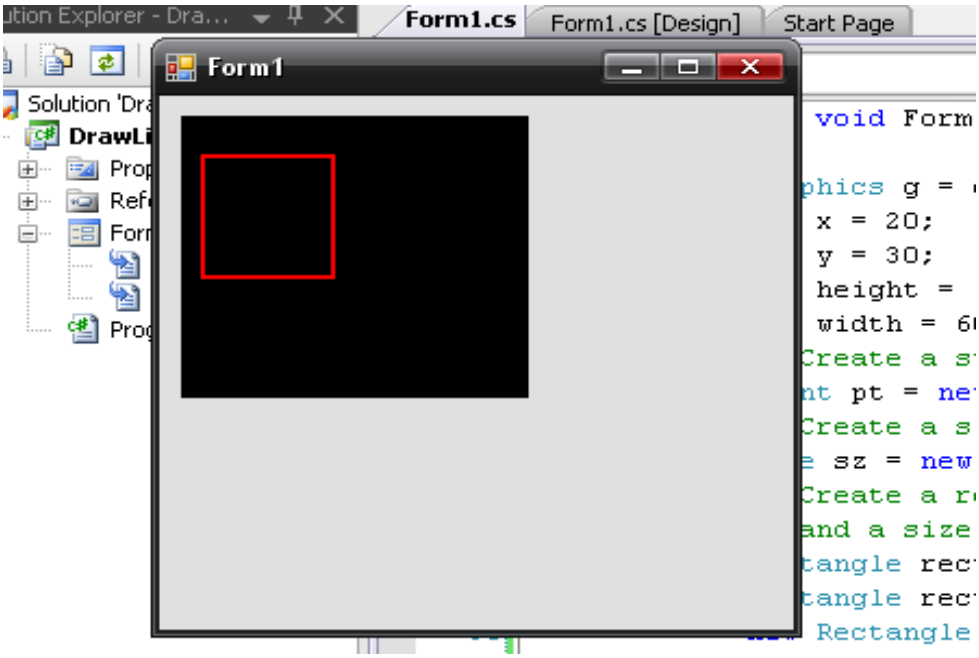


Рис. 8.4 Отображение прямоугольника.

Проект называется **DrawRectangle.**

d. Отображение эллипса

Так же как и прямоугольник, эллипс можно отобразить несколькими способами. Рисование производится методами **DrawEllipse()** или **FillEllipse()** объекта **Graphics.** Первый параметр методов принимает объект **Pen** либо **Brush,** второй структуру **Rectangle,** который представляет позицию и габариты эллипса.

Рассмотрим пример рисования эллипса. Создайте новый Windows Forms проект, назовите его **DrawEllipse**. Сгенерируйте для формы метод обработчика события **Paint,** заполните его следующей программной логикой.

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

```
            int x = 23;
            int y = 33;
            int height = 60;
            int width = 60;
            Pen pn = new Pen(Brushes.Red, 4);
            // Создаем начальную точку
            Point pt = new Point(10, 10);
            // Создаем размер
            Size sz = new Size(160, 160);
            // Создаем два прямоугольника
            Rectangle rect1 = new Rectangle(pt, sz);
            Rectangle rect2 = new Rectangle(x, y, width, height);
            g.FillEllipse(Brushes.Black, rect1);
            g.DrawEllipse(pn, rect2);

        }
```

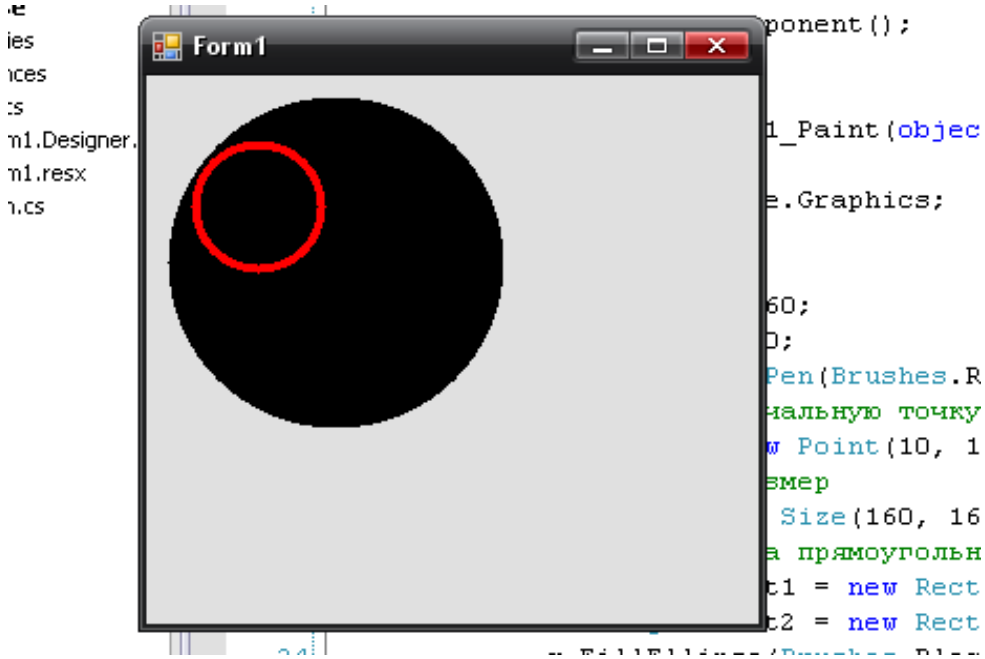


Рис. 8.5 Отображение эллипса.

Проект называется **DrawEllipse.**

Домашнее задание.

1. Создайте приложение, которое будет рисовать шахматную доску и шахматные фигуры на клиентской области формы. Для каждой фигуры должно отображаться контекстное меню.

2. Создайте приложение, с помощью которого можно рисовать различные геометрические формы на клиентской области. Приложение должно иметь в вверху панель инструментов с кнопками, которые позволяли заходить в настройки геометрического примитива и рисовать их. Нарисованные примитивы должны отображаться на клиентской области формы. Так же добавьте возможность сохранения композиции на жесткий диск в любом формате.