Основы разработки приложений с использованием Windows Forms

Урок №6

Использование возможностей GDI+

Содержание

1. 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+.**

1.1. Что такоеGDI+?

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.

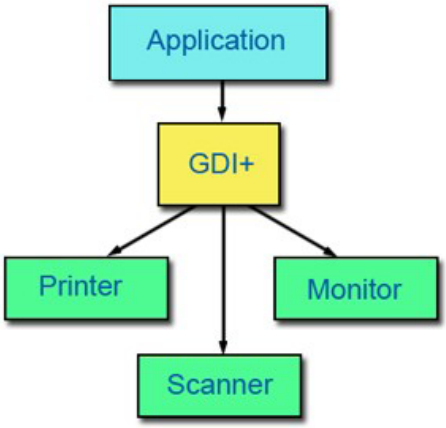

Рис. 1.1.

Теперь рассмотрим подробней работу **GDI+**. Предположим, наше приложение рисует линию. Линия будет представлена как набор последовательных пикселей, 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 пикселей.

2. Сравнительный анализ библиотек GDI+ и GDI

2.1. 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 заставляет 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.

2.2. 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 импортировать 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+.**

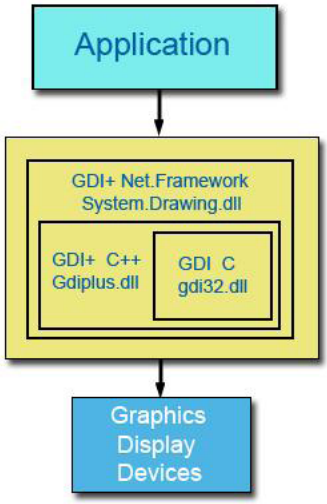

Рис.2.1. Взаимоотношения библиотек **GDI** и **GDI+.**

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
`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.

Пространство имен	Описание
`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 выборацвета, 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 Custom LineCap типы определяющие стили концов линий, LineJoin перечисления определяющие как линии будут соединятся между собой, PenAlignment перечисления определяющие как обьект Pen будет выравниваться относительно виртуальной линии, PenType перечисления определяющие заполнение линии;

Пространство имен	Описание
	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 типаGraphicsPath; 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

Пространство имен	Описание
	формат файла изображения, перечисление ImageFlags представляющее как данные о пикселах содержатся в изображении. Не поддерживается в Windows и ASP.NET сервисах.
`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();
}
```

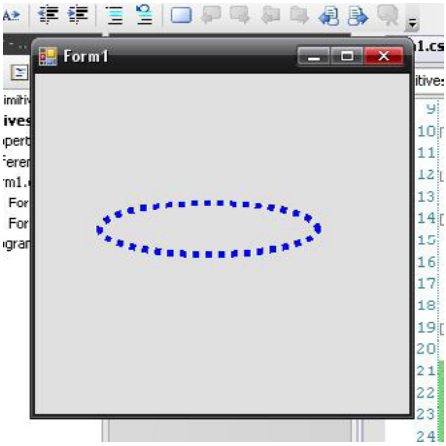

Рис. 4.1. Применение объекта **Pen**.

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();
}
```

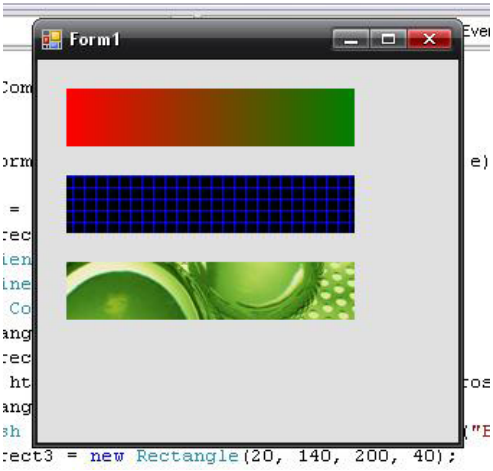

Рис. 4.2. Применение LinearGradientBrush, HatchBrush и TextureBrush.

Проект приведенный выше называется **Brushes-Example.**

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.

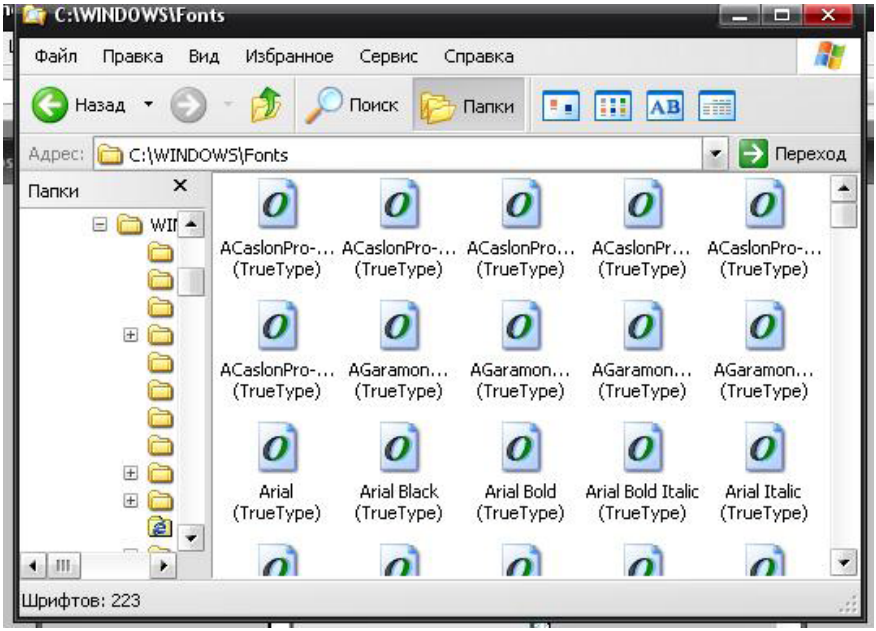

Рис. 4.3. Шрифты доступные в Windows.

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();
}
```

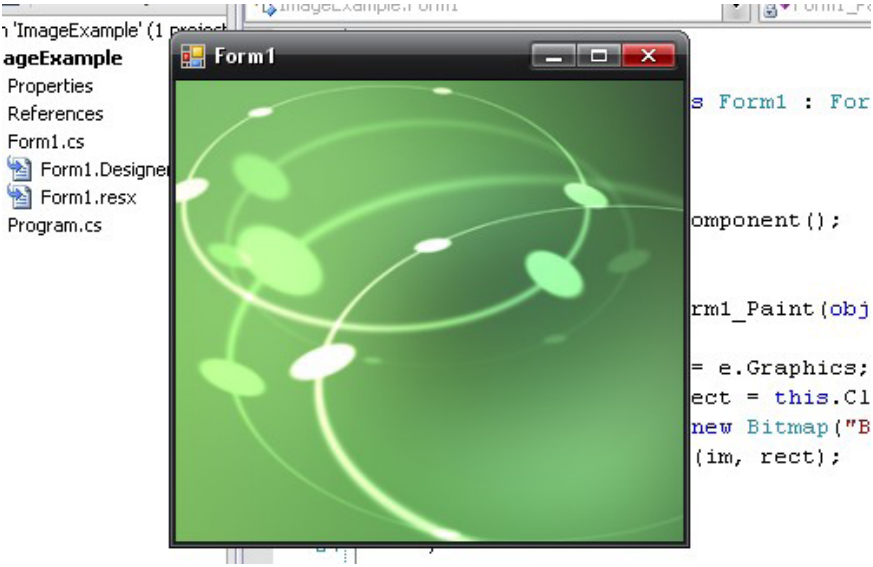

Рис. 4.5. Применение класса **Image.**

Проект приведенный выше называется **ImageExample.**

Регион — сущность, описывающая внутреннию 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.

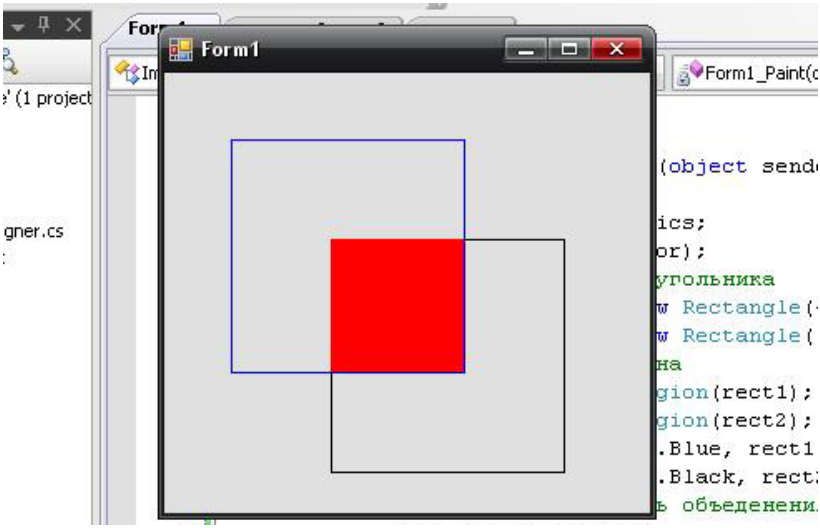

Рис. 4.6. Применение класса **Region.**

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 **DrawPath** 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();
}
```

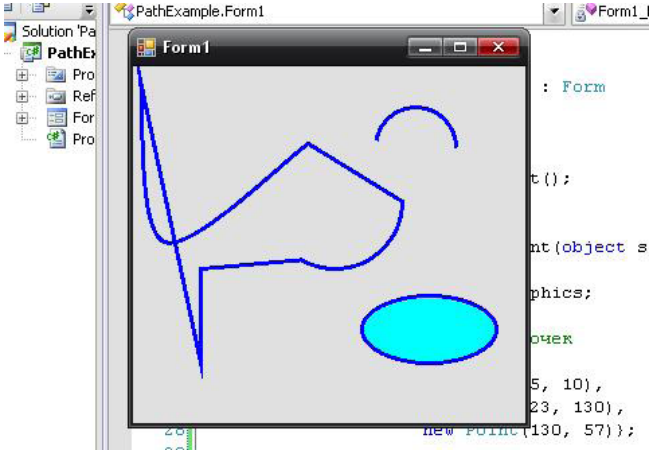

Рис. 4.7. Применение класса **GraphicsPath.**

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 позиция 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.

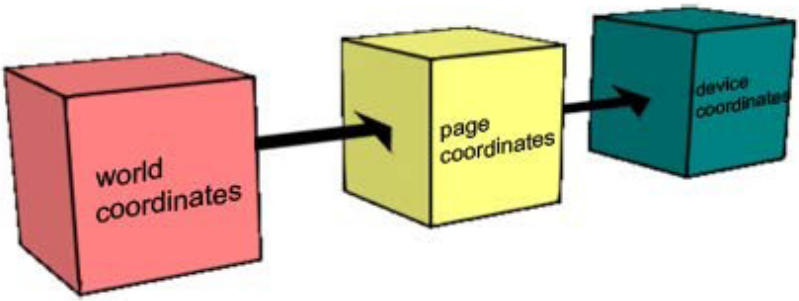

Рис. 5.1. Конвейер трансформаций координат.

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);

}
```

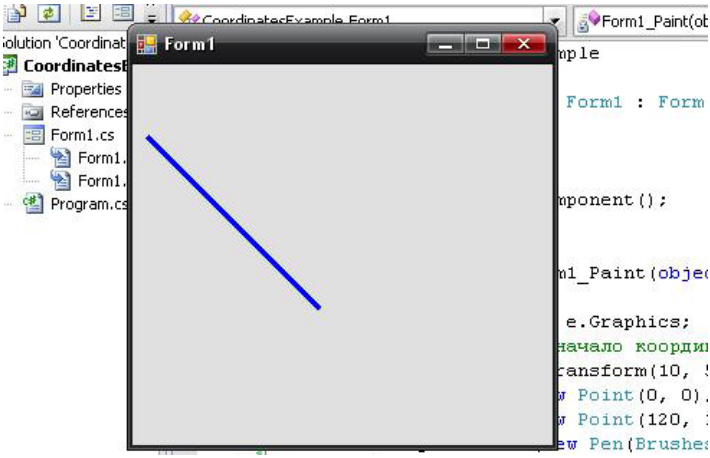

Рис. 5.2. Смещение начала координат страницы.

Проект называется **CoordinatesExample.**

6. Класс Graphics

6.1. 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()**.

6.2. 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.

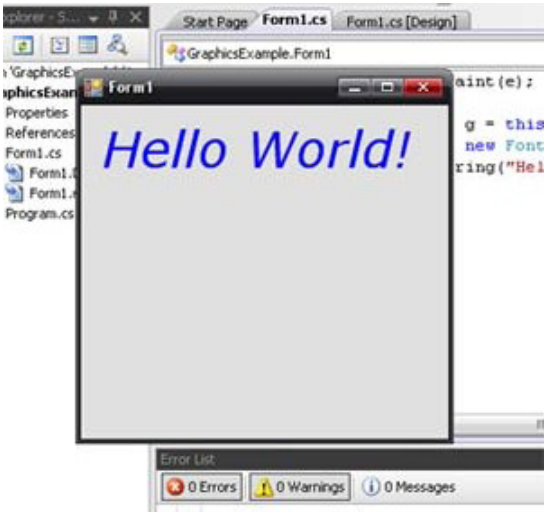

Рис 6.1.

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 помощью 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!" спомошью 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**.

6.3. 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

`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.

Методы	Описание
`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**
`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 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();
}
```

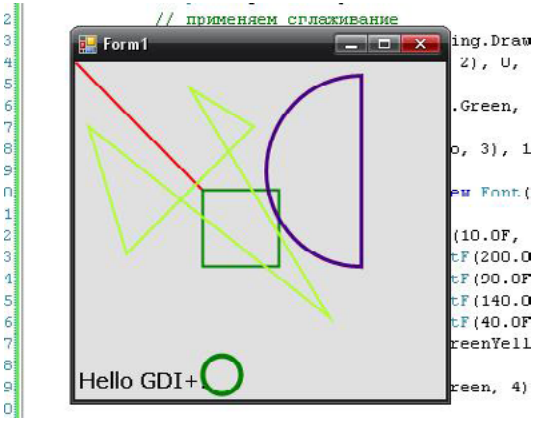

Рис. 6.2. Работа с методами **Graphics**

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 найти 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 7.1

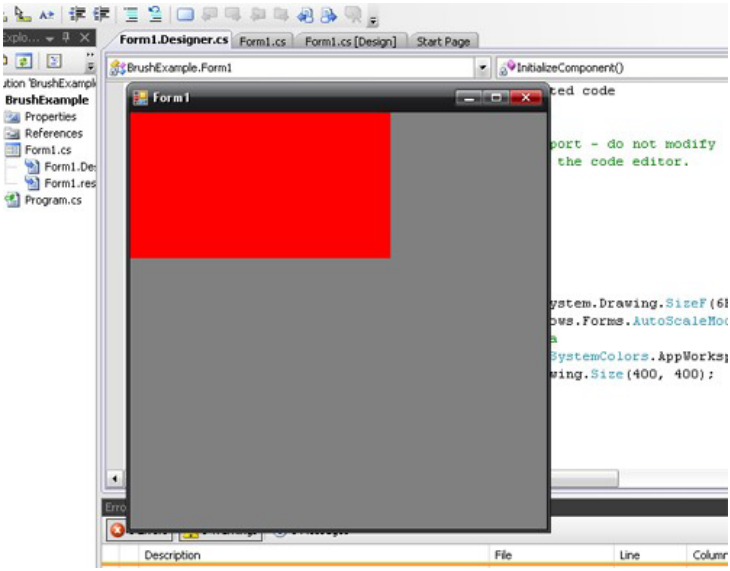

Рис. 7.1.

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. 7.2.

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

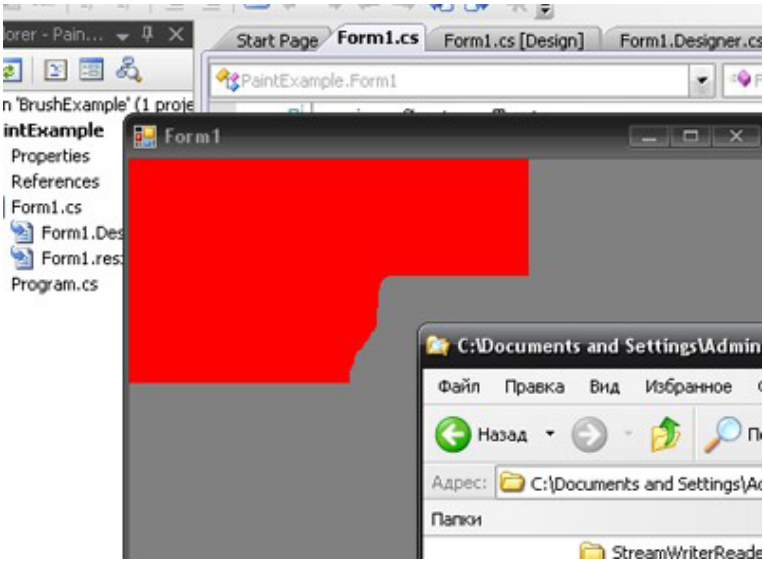

Рис. 7.2.

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.

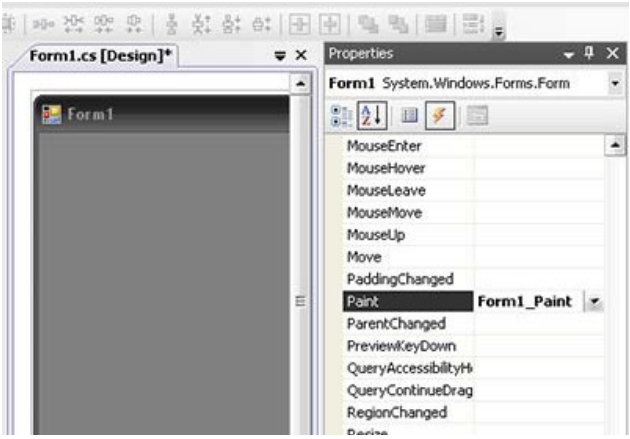

Рис 7.3.

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 7.3.

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 7.4.

Очевидно, что событие **Paint** срабатывает, когда проводник перекрывает любую область формы.

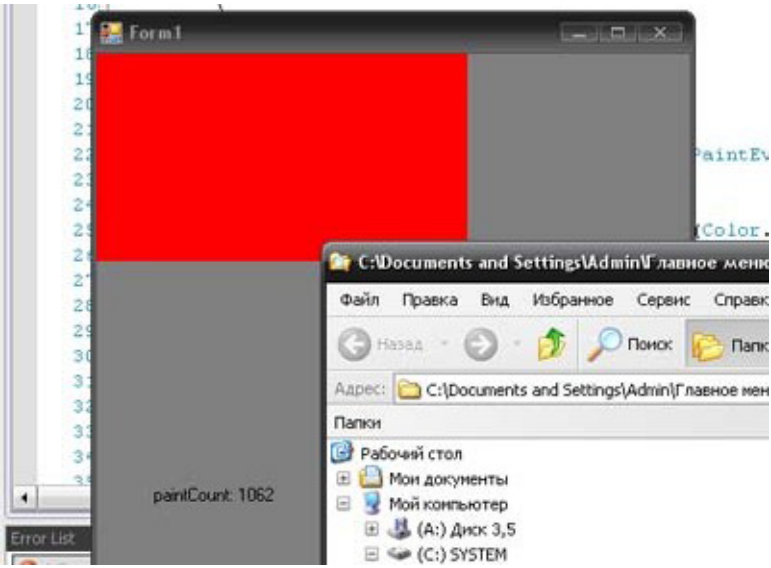

Рис 7.4.

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 нашегопрямоугольника
    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.**

8. Методы для вывода простейших графических примитивов

8.1. 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.**

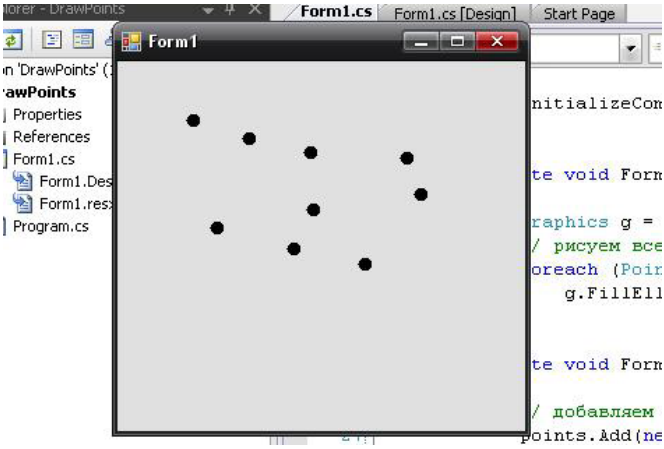

Рис. 8.1. Отображение точки.

Проект называется **DrawPoints.**

8.2. 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();
}
```

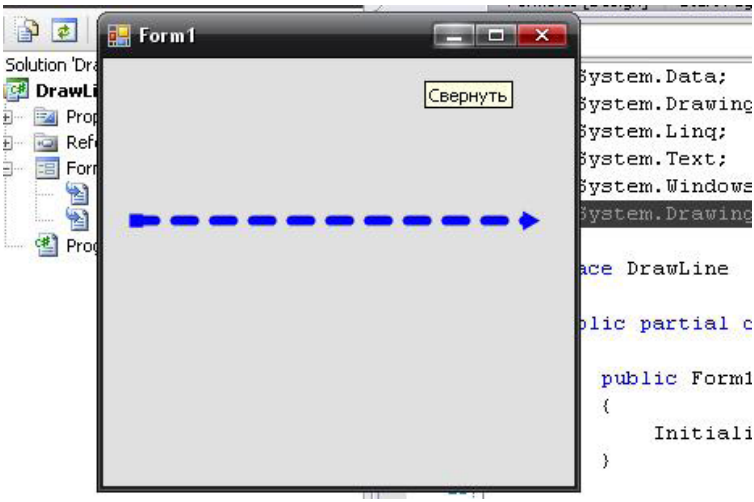

Рис. 8.3. Отображение линии.

Проект называется **DrawLine.**

8.3. Отображение прямоугольника

Существует несколько способов создания прямоугольника, **Rectangle**. Мы можем создать прямоугольник, предав четыре значения в качестве параметров конструктору структуры **Rectangle,** которые представляют 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 пара метр методов принимает объект **Pen** либо **Brush,** второй структуру **Rectangle** который представляет позицию и габаритыпрямоугольника.

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);
}
```

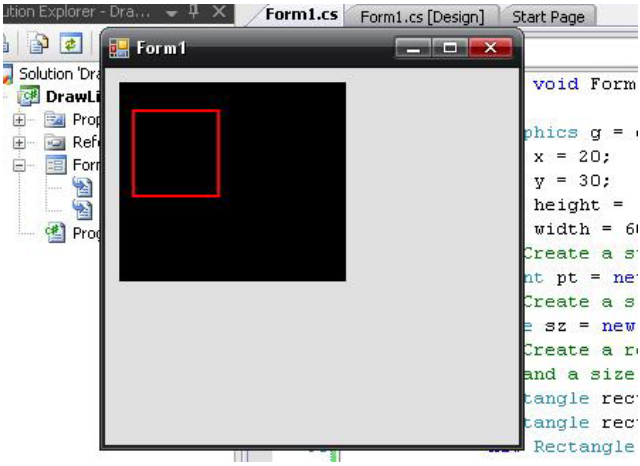

Рис. 8.4. Отображение прямоугольника.

Проект называется **DrawRectangle.**

8.4. 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);
}
```

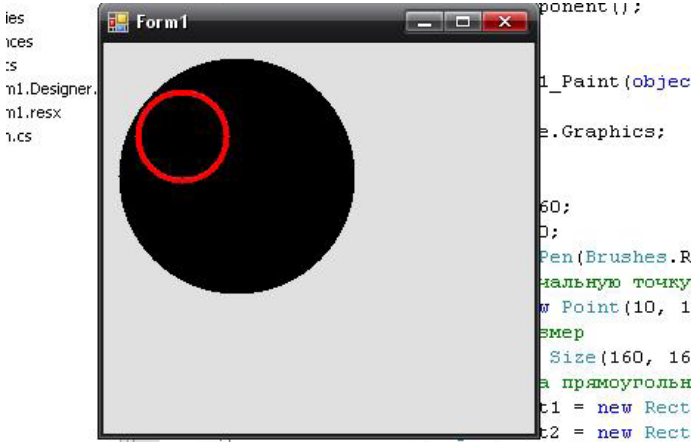

Рис. 8.5. Отображение эллипса.

Проект называется **DrawEllipse.**

9. Структуры Color, Size, Rectangle, Point

9.1. Структура Color

Цвета в GDI+ представлены экземплярами структуры **System.Drawing.Color.**

Первый способ создания структуры **Color** — указать значения для красного, синего и зеленого цветов, вызывая статистическую функцию **Color.FromArgb()**.

```
Color pink = Color.FromArgb(241, 105, 190);
```

Три параметра являются соответственно количествами красного, синего и зеленого цвета. Существует ряд других перегружаемых методов для этой функции, некоторые из них также позволяют определить так называемую альфа-смесь. Альфа-смешивание позволяет рисовать полупрозрачными тонами, комбинируя с цветом, который уже имеется на экране, что может создавать красивые эффекты и часто используется в играх.

Создание структуры с помощью **Color.FromArgb()** является наиболее гибкой техникой, так как она, по сути, означает, что можно определить любой цвет, который различает человеческий глаз. Однако если требуется простой, стандартный, хорошо известный цвет, такой как красный или синий, то значительно легче просто назвать требуемый цвет. В связи с этим Microsoft предоставляет также большое число статистических свойств в **Color**, каждое из которых возвращает именованный

цвет. Существует несколько сотен таких цветов. Полный список подан в документации MSDN. Он включает все простые цвета: **Red**, **Green**, **Black** и т.д, а также такие, как **DarkOrchid**, **LightCoral** и т.д.

```
Color navy = Color.Navy;
Color silver = Color.Silver;
Color sringGreen = Color.SpringGreen;
```

Структура **Color** содержит следующие методы:

Имя	Описание
`public static Color FromArgb(int argb)`	Создает структуру **Color** на основе 32-разрядного значения ARGB.
`public static Color FromArgb(int alpha, Color baseColor)`	Создает структуру **Color** из указанной структуры **Color**, но с новым определенным значением альфа. Хотя и этот метод позволяет передать 32-разрядное значение для значения альфа, оно ограничено 8 разрядами. Значение альфа для нового цвета **Color** допустимые от 0 до 255.
`public static Color FromArgb (int red, int green, int blue)`	Создает структуру **Color** из указанных 8-разрядных значений цветов (красный, зеленый, синий). Значение альфа неявно определено как 255 (полностью непрозрачно). Хотя и этот метод позволяет передать 32-разрядное значение для каждого компонента цвета, значение каждого из них ограничено 8 разрядами.
`public static Color FromArgb(int alpha,int red,int green,int blue)`	Создает структуру **Color** из четырех значений компонентов ARGB (альфа, красный, зеленый и синий).

Имя	Описание
	Хотя и этот метод позволяет передать 32-разрядное значение для каждого компонента, значение каждого из них ограничено 8 разрядами.
`public static Color FromKnownColor (KnownColor color)`	Создает структуру Color из указанного, предварительно определенного цвета.
`public static Color FromName (string name)`	Создает структуру Color из указанного имени предопределенного цвета.
`public static Color FromSysIcv (int icv)`	Получает цвет, определенный системой. Для получения системного цвета используется целое значение.
`public float GetBrightness()`	Получает значение яркости (оттенок-насыщенность-яркость (HSB)) для данной структуры Color.
`public float GetHue()`	Получает значение оттенка (оттенок-насыщенность-яркость (HSB)) в градусах для данной структуры Color.
`public float GetSaturation()`	Получает значение насыщенности (оттенок-насыщенность-яркость (HSB)) для данной структуры Color.
`public int ToArgb()`	Возвращает 32-разрядное значение ARGB этой структуры Color.
`public KnownColor ToKnownColor()`	Возвращает значение KnownColor этой структуры.
`public override string ToString()`	Преобразует структуру Color в удобную для восприятия строку.

Операторы

Имя	Описание
`public static bool operator == (Color left, Color right)`	Проверяет эквивалентность двух указанных структур Color.
`public static bool operator != (Color left, Color right)`	Проверяет различие двух указанных структур Color.

Поля

Имя	Описание
`public static readonly Color Empty`	Представляет цвет, являющийся ссылкой null

Свойства

Имя	Описание
`public byte A { get; }`	Получает значение альфа-компонента этой структуры Color.
`public byte B { get; }`	Получает значение синего компонента этой структуры Color.
`public byte G { get; }`	Получает значение зеленого компонента этой структуры Color.
`public bool IsEmpty { get; }`	Определяет, является ли эта структура Color не инициализированной.
`public bool IsKnownColor { get; }`	Возвращает значение, показывающее, является ли структура Color предопределенным цветом. Предварительно определенные цвета, представленные элементами перечисления KnownColor.
`public bool IsNamedColor { get; }`	Получает значение, указывающее, является ли структура Color имено-

Имя	Описание
	ванным цветом или элементом перечисления KnownColor.
`public bool IsSystemColor { get; }`	Возвращает значение, показывающее, является ли структура Color системным цветом. Системным является цвет, который используется в элементе отображения Windows. Системные цвета, представленные элементами перечисления KnownColor.
`public string Name { get; }`	Возвращает имя данного цвета Color.
`public byte R { get; }`	Получает значение красного компонента этой структуры Color.

9.2. Структура Size

Структура Size в GDI+ используется для представления размера, выраженного в пикселях. Она определяет как ширину, так и высоту.

Конструктор данной структуры перегружен:

Имя	Описание
`public Size(Point pt)`	Инициализирует новый экземпляр класса Size из указанного объекта Point.
`public Size(int width, int height)`	Инициализирует новый экземпляр класса Size из указанных размеров.

Методы

Имя	Описание
`public static Size Add(Size sz1, Size sz2)`	Прибавляет ширину и высоту одной структуры **Size** к ширине и высоте другой структуры **Size**.
`public static Size Ceiling(SizeF value)`	Преобразует указанную структуру **SizeF** в структуру **Size**, округляя значения структуры **Size** до ближайшего большего целого числа.
`public override bool Equals(Object obj)`	Проверяет, совпадают ли размеры указанного объекта Size с размерами объекта Size.
`protected virtual void Finalize()`	Позволяет объекту Object попытаться освободить ресурсы и выполнить другие операции очистки, перед тем как объект Object будет утилизирован в процессе сборки мусора.
`public static Size Round(SizeF value)`	Преобразует указанную структуру SizeF в структуру Size, округляя значения структуры SizeF до ближайших целых чисел.
`public static Size Subtract(Size sz1, Size sz2)`	Вычитает ширину и высоту одной структуры Size из ширины и высоты другой структуры Size.
`public override string ToString()`	Создает удобную для восприятия строку, представляющую размер Size.
`public static Size Truncate(SizeF value)`	Преобразует указанную структуру SizeF в структуру Size, округляя значения структуры SizeF до ближайших меньших целых чисел.

Операторы

Имя	Описание
`public static Size operator +(Size sz1, Size sz2)`	Прибавляет ширину и высоту одной структуры Size к ширине и высоте другой структуры Size.
`public static bool operator == (Size sz1, Size sz2)`	Проверяет равенство двух структур Size.
`public static explicit operator Point (Size size)`	Преобразует указанный размер Size в точку Point.
`public static implicit operator SizeF (Size p)`	Преобразует указанный размер Size в точку SizeF.
`public static bool operator != (Size sz1, Size sz2)`	Проверяет, различны ли две структуры Size.
`public static Size operator – (Size sz1, Size sz2)`	Вычитает ширину и высоту одной структуры Size из ширины и высоты другой структуры Size.

Поля

Имя	Описание
`public static readonly Size Empty`	Инициализирует новый экземпляр класса Size.

Свойства

Имя	Описание
`public int Height { get; set; }`	Получает или задает вертикальный компонент этого размера Size.
`public bool IsEmpty { get; }`	Проверяет, равны ли 0 ширина и высота размера Size.
`public int Width { get; set; }`	Получает или задает горизонтальный компонент размера Size.

9.3. Структура Point

В GDI+ структура **Point** используется для представления отдельной точки. Это точка, расположенная на двумерной плоскости, которая определяет единственный пиксель. Многим функциям, использующимся в GDI+, таким как **DrawLine()**, **Point** передается в качестве аргумента. Для структуры определены конструкторы:

```
int x = 15, y = 20;
Point p0 = new Point();
Point p1 = new Point(x);
Point p2 = new Point(x, y);
Point p3 = new Point(new Size(x, y));
```

В структуре **Point** определены следующие методы:

Имя	Описание
`public static Point Add(Point pt, Size sz)`	Добавляет заданный размер **Size** к точке **Point**.
`public static Point Ceiling(PointF value)`	Преобразует указанную структуру **PointF** в структуру **Point**, округляя значения **PointF** до ближайшего большего целого числа.
`public override bool Equals(Object obj)`	Определяет, содержит объект ли **Point** те же координаты, что и указанный объект **Object**, или нет.
`protected virtual void Finalize()`	Позволяет объекту **Object** попытаться освободить ресурсы и выполнить другие операции очистки, перед тем как объект **Object** будет утилизирован в процессе сборки мусора.

Имя	Описание
`public void Offset(Point p)`	Смещает точку **Point** на указанную точку **Point**
`public void Offset(int dx, int dy)`	Смещает точку **Point** на указанное значение (dx , dy)
`public static Point Round(PointF value)`	Преобразует указанный объект **PointF** в **Point**, округляя значения **Point** до ближайших целых чисел.
`public static Point Subtract(Point pt, Size sz)`	Возвращает результат вычитания заданного размера Size из заданной точки Point.
`public override string ToString()`	Преобразует объект **Point** в строку, доступную для чтения.
`public static Point Truncate(PointF value)`	Преобразует указанную точку **PointF** в точку **Point** путем усечения значений объекта **Point**.

Операторы

Имя	Описание
`public static Point operator + (Point pt, Size sz)`	Смещает точку **Point** на заданное значение **Size**.
`public static bool operator == (Point left, Point right)`	Сравнивает два объекта **Point**. Результат определяет, равны значения свойств X и Y двух объектов **Point** или нет.
`public static explicit operator Size (Point p)`	Преобразует заданную структуру **Point** в структуру **Size**.
`public static implicit operator PointF (Point p)`	Преобразует заданную структуру **Point** в структуру **PointF**.

Имя	Описание
`public static bool operator != (Point left, Point right)`	Сравнивает два объекта Point. Результат показывает неравенство значений свойств X или Y двух объектов Point.
`public static Point operator (Point pt, Size sz)`	Смещает Point на отрицательное значение, заданное Size.

Поля

Имя	Описание
`public static readonly Point Empty`	Представляет объект Point, у которого значения X и Y установлены равными нулю.

Свойства

Имя	Описание
`public bool IsEmpty { get; }`	Получает значение, определяющее, пуст ли класс Point.
`public int X { get; set; }`	Получает или задает координату X точки Point.
`public int Y { get; set; }`	Получает или задает координату Y точки Point.

9.4. Структура Rectangle

Эта структура используется в GDI+ для задания координат прямоугольников. Структура **Point** описывает верхний левый угол прямоугольника, а структура **Size** — его размеры. У **Rectangle** есть два конструктора. Одному в качестве аргументов передаются координаты

X, Y, ширина и высота. Второй конструктор принимает структуры **Point** и **Size**.

```
int x = 15, y = 20, h = 70, w = 200;
Rectangle rect0 = new Rectangle(x, y, w, h); //или
Rectangle rect1 = new Rectangle(new Point(x, y),
                  new Size(w, h));
```

Методы

Имя	Описание
`public static Rectangle Ceiling(RectangleF value)`	Преобразует указанную структуру RectangleF в структуру Rectangle, округляя значения RectangleF до ближайшего большего целого числа.
`public bool Contains(Point pt)`	Определяет, содержится ли заданная точка в структуре Rectangle.
`public bool Contains(Rectangle rect)`	Этот метод возвращает значение true, если прямоугольная область, представленная параметром rect, полностью содержится в структуре Rectangle; в противном случае — значение false.
`public bool Contains(int x, int y)`	Определяет, содержится ли заданная точка в структуре Rectangle.
`public override bool Equals(Object obj)`	Проверяет, является ли obj структурой Rectangle с таким же расположением и размером, что и структура Rectangle.
`protected virtual void Finalize()`	Позволяет объекту Object попытаться освободить ресурсы и выполнить другие операции очистки, перед тем как объект Object будет утилизирован в процессе сборки мусора.

Имя	Описание
`public static Rectangle FromLTRB (int left, int top, int right, int bottom)`	Создает структуру Rectangle с заданным положением краев.
`public void Inflate(Size size)`	Увеличивает этот Rectangle на указанную величину size.
`public void Inflate (int width, int height)`	Увеличивает этот Rectangle на указанную величины width и height.
`public static Rectangle Inflate (Rectangle rect, int x, int y)`	Создает и возвращает развернутую копию указанной структуры Rectangle. Копия увеличивается на указанную величину. Исходная структура Rectangle остается без изменений.
`public void Intersect(Rectangle rect)`	Заменяет данный Rectangle его пересечением с указанным прямоугольником Rectangle.
`public static Rectangle Intersect(Rectangle a, Rectangle b)`	Возвращает третью структуру Rectangle, представляющую собой пересечение двух других структур Rectangle. Если пересечение отсутствует, возвращается пустая структура Rectangle.
`public bool IntersectsWith (Rectangle rect)`	Определяет, пересекается ли данный прямоугольник с прямоугольником rect.
`public void Offset(Point pos) public void Offset(int x, int y)`	Этот метод изменяет положение левого верхнего угла в горизонтальном направлении, используя координату указанной точки по оси X, и в вертикальном направлении, используя координату этой точки по оси Y.

Имя	Описание
`public static Rectangle Round (RectangleF value)`	Преобразует указанный RectangleF в Rectangle, округляя значения RectangleF до ближайших целых чисел.
`public override string ToString()`	Преобразует атрибуты этого прямоугольника Rectangle в удобную для восприятия строку.
`public static Rectangle Truncate (RectangleF value)`	Преобразует указанный прямоугольник RectangleF в Rectangle путем усечения значений RectangleF.
`public static Rectangle Union (Rectangle a, Rectangle b)`	Получает структуру Rectangle, содержащую объединение двух структур прямоугольника Rectangle.

Операторы

Имя	Описание
`public static bool operator == (Rectangle left, Rectangle right)`	Этот оператор возвращает значение true, если у двух структур Rectangle равны свойства X, Y, Width и Height.
`public static bool operator != (Rectangle left, Rectangle right)`	Этот оператор возвращает значение true, если значения каких-либо из свойств X, Y, Width или Height двух структур Rectangle не совпадают; в противном случае — значение false.

Поля

Имя	Описание
`public static readonly Rectangle Empty`	Представляет структуру Rectangle, свойства которой не инициализированы.

Свойства

Имя	Описание
`public int Bottom { get; }`	Возвращает координату по оси Y, являющуюся суммой значений свойств Y и Height данной структуры Rectangle.
`public int Height { get; set; }`	Возвращает или задает высоту в структуре Rectangle.
`public bool IsEmpty { get; }`	Проверяет, все ли числовые свойства этого прямоугольника Rectangle имеют нулевые значения.
`public int Left { get; }`	Возвращает координату по оси X левого края структуры Rectangle.
`public Point Location { get; set; }`	Возвращает или задает координаты левого верхнего угла структуры Rectangle.
`public int Right { get; }`	Возвращает координату по оси X, являющуюся суммой значений свойств X и Width данной структуры Rectangle.
`public Size Size { get; set;}`	Возвращает или задает размер этого прямоугольника Rectangle.
`public int Top { get; }`	Возвращает координату по оси Y верхнего края структуры Rectangle.
`public int Width { get; set; }`	Возвращает или задает ширину структуры Rectangle.
`public int X { get; set; }`	Возвращает или задает координату по оси X левого верхнего угла структуры Rectangle.
`public int Y { get; set; }`	Возвращает или задает координату по оси Y левого верхнего угла структуры Rectangle.

10. Кисти

10.1. Пространство Drawing2D

Пространство имен **System.Drawing.Drawing2D** предоставляет расширенные функциональные возможности векторной и двухмерной графики. Оно обеспечивает возможность устанавливать специальные «наконечники» для перьев, создавать кисти, которые рисуют не сплошной полосой, а текстурами, производить различные векторные манипуляции с графическими объектами.

Категория классов	Сведения
Графика и графические контуры	Классы GraphicsState и GraphicsContainer содержат информацию отчета о текущем объекте Graphics. Классы GraphicsPath представляют наборы линий и кривых. Классы GraphicsPathIterator и PathData предоставляют подробные сведения о содержимом объекта GraphicsPath.
Типы, относящиеся к матрице и преобразованию	Класс Matrix представляет матрицу для геометрических преобразований. Перечисление MatrixOrder задает порядок для матричных преобразований.
Классы Brush	Классы PathGradientBrush и HatchBrush позволяют произвести заливку форм, соответственно, с

Категория классов	Сведения
	использованием градиента или шаблона штриховки.
Перечисление, связанное с линиями	Перечисления LineCap и CustomLineCap позволяют задать стили завершения для линии. Перечисление LineJoin позволяет указать, как две линии объединяются в контур. Перечисление PenAlignment позволяет определить выравнивание наконечника при рисовании линии. Перечисление PenType задает шаблон, с помощью которого следует выполнить заливку линии.
Перечисления, связанные с заполнением фигур и контуров	Перечисление HatchStyle задает стили заполнения для объекта HatchStyle. Класс Blend задает шаблон смешивания для объекта LinearGradientBrush. Перечисление FillMode задает стиль заполнения для объекта GraphicsPath.

10.2. Класс Brush

Кисти (*brush*) предназначены для «закрашивания» пространства между линиями. Можно определить для кисти цвет, текстуру или изображение. Сам класс **Brush** является абстрактным, и создавать объекты этого класса нельзя. Класс **Brush** находится в пространстве имен **System.Drawing**. Производные классы **TextureBrush**, **HatchBrush** и **LinearGradientBrush** находятся в пространстве имен

System.Drawing.Drawing2D. Рассмотрим более подробно наследники класса **Brush**.

10.3. Сплошная кисть, класс SolidBrush

SolidBrush заполняет фигуру сплошным цветом. Свойство **Color** позволяет получить или задать цвет объекта **SolidBrush**.

10.4. Текстурная кисть, класс TextureBrush

TextureBrush позволяет заполнить фигуру рисунком, хранящемся в двоичном представлении. Этот класс не может быть унаследован. При создании такой кисти также требуется задавать обрамляющий прямоугольник и режим обрамления. Обрамляющий прямоугольник определяет, какую порцию рисунка мы должны использовать при рисовании, использовать весь рисунок целиком совершенно необязательно. Приведем некоторые члены данного класса:

Имя	Описание
`public Image Image { get; }`	Получает объект Image, связанный с объектом TextureBrush.
`public Matrix Transform {get; set;}`	Получает или задает копию объекта Matrix, определяющую локальное геометрическое преобразование для изображения, связанного с объектом TextureBrush.
`public WrapMode WrapMode {get; set;}`	Получает или задает перечисление WrapMode, задающее режим переноса для объекта TextureBrush.

Методы

Имя	Описание
`public void MultiplyTransform (Matrix matrix)` `public void MultiplyTransform (Matrix matrix, MatrixOrder order)`	Умножает объект Matrix, представляющий локальное геометрическое преобразование объекта TextureBrush, на указанный объект Matrix с помощью добавления указанного объекта Matrix в начало.
`public void ResetTransform()`	Возвращает свойству Transform объекта TextureBrush единичное значение.
`public void RotateTransform (float angle)`	Поворачивает локальное геометрическое преобразование объекта TextureBrush на заданную величину. Этот метод добавляет поворот перед преобразованием.
`public void RotateTransform (float angle, MatrixOrder order)`	Поворачивает локальное геометрическое преобразование объекта TextureBrush на заданную величину в указанном порядке.
`public void ScaleTransform (float sx, float sy)` `public void ScaleTransform (float sx, float sy, MatrixOrder order)`	Изменяет масштаб локального геометрического преобразования объекта TextureBrush на заданные значения. Этот метод вставляет изменение масштаба перед преобразованием. sx — Величина изменения масштаба по оси X. sy — Величина изменения масштаба по оси Y. order — Перечисление MatrixOrder, задающее порядок использования матрицы изменения масштаба (в начале или в конце).

Имя	Описание
`public void TranslateTransform (float dx, float dy)` `public void TranslateTransform (float dx, float dy, MatrixOrder order)`	Выполняет перенос локального геометрического преобразования объекта TextureBrush на заданные значения в указанном порядке. dx — Величина переноса преобразования по оси X. dy — Величина переноса преобразования по оси Y. order — Порядок применения преобразования (в начале или в конце).

10.5. Кисть с насечками, класс HatchBrush

Более сложное «закрашивание» можно произвести при помощи производного от **Brush** класса **HatchBrush**. Этот тип позволяет закрасить внутреннюю областью объекта при помощи большого количества штриховки, определенных в перечислении **HatchStyle** (имеется 54 уникальных стиля штриховки). Основной цвет задает цвет линий; цвет фона определяет цвет интервалов между линиями. Этот класс не может быть унаследован.

Свойства (выборочно)

Имя	Описание
`public Color BackgroundColor { get; }`	Получает цвет интервалов между линиями штриховки, нарисованными данным объектом HatchBrush.
`public Color ForegroundColor { get; }`	Получает цвет линий штриховки, нарисованных данным объектом HatchBrush.
`public HatchStyle HatchStyle { get; }`	Получает стиль штриховки для данного объекта HatchBrush.

10.6. Градиентная кисть, класс LinearGradientBrush

LinearGradientBrush содержит кисть, которая позволяет рисовать плавный переход от одного цвета к другому, причем первый цвет переходит во второй под определенным углом. Углы при этом задаются в градусах. Угол, равный 0°, означает, что переход от одного цвета к другому осуществляется слева направо. Угол, равный 90°, означает, что переход от одного цвета к другому осуществляется сверху вниз. Этот класс не может быть унаследован. Приведем некоторые свойства **LinearGradientBrush:**

Свойства

Имя	Описание
`public Blend Blend { get; set; }`	Получает или задает объект Blend, определяющий позиции и коэффициенты, задающие настраиваемый спад для градиента.
`public bool GammaCorrection { get; set; }`	Получает или задает значение, указывающее, включена ли гамма-коррекция для этого объекта LinearGradientBrush.
`public ColorBlend InterpolationColors { get; set; }`	Получает или задает перечисление WrapMode, задающее режим переноса для объекта TextureBrush.
`public Color[] LinearColors { get; set; }`	Получает или задает начальный и конечный цвета градиента.
`public RectangleF Rectangle { get; }`	Получает прямоугольную область, которая определяет начальную и конечную точки градиента.

10.7. Кисть с использованием траектории, класс PathGradientBrush

PathGradientBrush позволяет создавать сложный эффект затенения, при котором используется изменение цвета от середины рисуемого пути к его краям.

Свойства (выборочно)

Имя	Описание
`public Color CenterColor { get; set; }`	Получает или задает цвет в центре градиента контура.
`public PointF CenterPoint { get; set; }`	Получает или задает центральную точку градиента контура.
`public PointF FocusScales { get; set; }`	Получает или задает точку фокуса для градиентного перехода.
`public ColorBlend InterpolationColors { get; set; }`	Получает или задает объект ColorBlend, определяющий многоцветный линейный градиент.
`public Color [] SurroundColors { get; set; }`	Получает или задает массив цветов, соответствующих точкам контура, заполняемого объектом PathGradientBrush.

11. Перья, класс Pen

Обычное применение объектов **Pen** (перьев) заключается в рисовании линий. Как правило, объект **Pen** используется не сам по себе: он передается в качестве параметра многочисленным методам вывода, определенным в классе **Graphics**.

В этом классе предусмотрены несколько перегруженных конструкторов, при помощи которых можно задать исходный цвет и толщину пера. Большая часть возможностей **Pen** определяется свойствами этого класса.

Свойства (выборочно)

Имя	Описание
`public PenAlignment Alignment {get; set;}`	Это свойство определяет, каким образом объект **Pen** рисует замкнутые кривые и многоугольники. Перечислением **PenAlignment** задаются пять значений; однако только два значения — **Center** и **Inset** — изменяют внешний вид рисуемой линии. Значение **Center** устанавливается по умолчанию для этого свойства, и им задается центрирование ширины пера по контуру кривой или многоугольника. Значение **Inset** этого свойства соответствует размещению пера по всей ширине внутри границы кривой или многоугольника. Три другие значения, **Right**, **Left** и **Outset** задают центрированное перо.

Имя	Описание
	Перо Pen, для которого выравнивание установлено равным **Inset**, выдает недостоверные результаты, иногда рисуя в позиции вставки, а иногда — в центрированном положении. К тому же, вложенное перо невозможно использовать для рисования составных линий, и оно не позволяет рисовать пунктирные линии с оконечными элементами **Triangle**.
`public Brush Brush { get; set; }`	Получает или задает объект Brush, определяющий атрибуты объекта Pen.
`public float[] CompoundArray { get; set; }`	Получает или задает массив значений, определяющий составное перо. Составное перо рисует составную линию, состоящую из параллельных линий и разделяющих их промежутков.
`public CustomLineCap CustomStartCap { get; set; }`	Получает или задает настраиваемое начало линий, нарисованных при помощи объекта Pen.
`public DashStyle DashStyle { get; set; }`	Получает или задает стиль, используемый для пунктирных линий, нарисованных при помощи объекта Pen.
`public PenType PenType { get; }`	Получает или задает стиль линий, нарисованных с помощью объекта Pen.

Перечисление **DashStyle**

Имя члена	Описание
`Solid`	Задает сплошную линию.
`Dash`	Задает линию, состоящую из штрихов.

Имя члена	Описание
Dot	Задает линию, состоящую из точек.
DashDot	Задает штрих-пунктирную линию.
DashDotDot	Задает линию, состоящую из повторяющегося шаблона “штрих-две точки”.
Custom	Задает пользовательский тип пунктирных линий.

Кроме класса **Pen** в GDI+ также можно использовать коллекцию заранее определенных перьев (коллекция **Pens**). При помощи статистических свойств коллекции **Pens** можно мгновенно получить уже готовое перо, без необходимости создавать его вручную. Однако все типы **Pen**, которые создаются при помощи коллекции **Pens**, имеют одинаковую ширину равную 1. Изменить ее нельзя.

Для работы с «наконечниками» перьев служит перечисление **LineCap**.

Значения перечисления **LineCap**

Значение	Описание
ArrowAnchor	Линии оканчиваются стрелками
DiamondAnchor	Лини оканчиваются ромбами
Flat	Стандартное прямоугольное завершение линий
Round	Лини на концах скруглены
RoundAnchor	На концах линий — «шары»
Square	На концах линий — квадраты в толщину линии

Значение	Описание
`SquareAnchor`	На концах линий — квадраты большего размера, чем толщина линий.
`Triangle`	Треугольное завершение линий.

Желательно вызывать метод **Dispose()** для создаваемых объектов **Brush** и **Pen**, либо использовать для работы с ними конструкцию **using**, в противном случае приложение может исчерпать ресурсы системы.

12. Примеры использования кистей и перьев

Рассмотрим пример программы с использованием кистей и перьев разных видов. Создадим статический класс **Cub**, реализовывающий прорисовку создание графического куба и разные методы заливки граней. В методе **BrushesExampleMethod(Graphics g)** показана работа с кистями. Создадим структуру **Color** с помощью метода **Color.FromArgb():**

```
Color pink = Color.FromArgb(241, 105, 190);

//создаем сплошную кисть цвета pink
SolidBrush sldBrush = new SolidBrush(pink);

//закрасим кистью sldBrush прямоугольник, у которого
//верхний левый угол имеет координаты (300;150), а высоту
//и широту 70
g.FillRectangle(sldBrush, 300, 150, 70, 70);
```

Рассмотрим применение штриховки **hBrush**. В качестве параметров конструктору передаем стиль штриховки, цвет линий и цвет промежутков между линиями:

```
HatchBrush hBrush = new HatchBrush(HatchStyle.
    NarrowVertical, Color.Pink, Color.Blue);
g.FillRectangle(hBrush, 370, 150, 70, 70);
```

Создадим градиентную кисть **lgBrush**. Структура **Rectangle** представляет границы линейного градиента:

```
LinearGradientBrush lgBrush = new LinearGradientBrush(new
    Rectangle(0, 0, 20, 20), Color.Violet,
    Color.LightSteelBlue, LinearGradientMode.Vertical);
g.FillRectangle(lgBrush, 300, 220, 70, 70);
```

Для заливки следующего прямоугольника будем использовать встроенную кисть из коллекции **Brushes:**

```
g.FillRectangle(Brushes.Indigo, 370, 220, 70, 70);
```

Воспользуемся текстурной кистью. Рисунок для кисти загрузим из файла:

```
TextureBrush tBrush = new TextureBrush(Image.
        FromFile(@"Images\charp.jpg"));
g.FillRectangle(tBrush, 370, 290, 70, 70);
```

В методе **DrawLeftAndUpperBound** класса **Cub** рисуем левую и верхнюю грань куба:

```
public static void DrawLeftAndUpperBound(Graphics g)
{
Point[] p = { new Point(240, 110), new Point(440, 110),
            new Point(510, 150), new Point(300, 150) };
TextureBrush tBrush = new TextureBrush(Image.
            FromFile(@"Images\0073.jpg"));
g.FillPolygon(tBrush, p);

Point[] p1 = { new Point(240, 110), new Point(300, 150),
            new Point(300, 360), new Point(240, 310) };
HatchBrush hBrush = new HatchBrush(HatchStyle.
            DashedDownwardDiagonal,
            Color.Violet, Color.White);
g.FillPolygon(hBrush, p1);
```

Работа с перьями демонстрируется в методе **PensExampleMethod** класса **Cub:**

```
public static void PensExampleMethod(Graphics g)
{
//рисуем вертикальные линии сплошным пером
    Pen p = new Pen(Color.White, 1);
    for (int i = 0; i < 4; i++)
    {
        g.DrawLine(p,300+70*i,150,300+70*i,360);
    }
//рисуем горизонтальные линии пунктирным пером
    p = new Pen(Color.White, 1);
    p.DashStyle = DashStyle.DashDot;
    for (int i = 0; i < 4; i++)
    {
        g.DrawLine(p, 300, 150 + 70 * i, 510, 150 + 70 * i);
    }
//рисуем косые линии пером, на концах которого ромбы
    p = new Pen(Color.White, 1);
    p.StartCap = LineCap.DiamondAnchor;
    p.EndCap = LineCap.DiamondAnchor;
    for (int i = 0; i < 3; i++)
    {
        g.DrawLine(p, 240+70*i, 110 , 300+70*i, 150);
    }
}
```

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.