6 Graphik - GDI

Oft benötigt man für eine graphische Benutzeroberfläche mehr als die vom Framework zur Verfügung gestellten Steuerelemente. Dann muss das Programm selber zeichnen. Das betrifft nicht nur den Bildschirm, auch für die Ausgabe auf einen Drucker oder ein File will man zeichnen. Wann immer man zeichnen will, ist man mit einigen Grundelementen konfrontiert, Farben (*colors*), Pinsel (*brushes*), Stifte (*pens*) und Schriften (*fonts*) und mit Dingen, die man zeichnen will, Formen (*shapes*), Bilder (*images*) und Texte (*strings*). Die Klassen fürs Zeichnen sind im Namespace `System.Drawing` und verwenden GDI+ (Graphics Device Interface+), den Nachfolger von GDI. Das ganze Konzept bemüht sich um eine geräteunabhängige abstrakte Sicht auf die Zeichenfläche, die durch die Klasse Graphics repräsentiert wird.

6.1 Graphics-Objekt und Paint Event

6.1.1 In einem Control zeichnen

Für eine Form, ein Panel aber auch für andere Controls kann man sich das zugehörige Graphics-Objekt mit der Methode `CreateGraphics()` erzeugen. Dann kann man mit Methoden wie `DrawEllipse`, `FillEllipse`, `DrawRectangle`, `FillRectangle`, `DrawLine` usw. zeichnen.

Beispiel:

Das Beispiel hat nur einen Button in der Mitte einer Form. Der Code für den Click-Event des Buttons ist:

```
private void btnDraw_Click(object sender, System.EventArgs e)
{
    ellipse = !ellipse;
    using (Graphics g = this.CreateGraphics())
    {
       if (ellipse)
       {
          g.FillEllipse(Brushes.Azure, this.ClientRectangle);
       }
       else
       {
          g.FillEllipse(SystemBrushes.Control, this.ClientRectangle);
       }
    }
}
```

Diese Programmzeilen zeichnen und löschen abwechselnd eine Ellipse, welche die Form an den Rändern berührt.

Das `using`-Statement wird hier verwendet, weil sichergestellt werden sollte, dass die vom Graphics Objekt verwendeten Windows-System Ressourcen in jedem Fall wieder freigegeben werden. Das `using`-Statement garantiert das sowohl im Normalfall als auch im Falle einer Exception die Freigabe des Objektes mit Dispose();

6.1.2 Das Paint-Event verwenden:

Wenn man die Größe der Form ändert oder die Form von einem anderen Fenster vorübergehend überdeckt wird, dann verhält sich unsere Zeichnung nicht wie gewohnt. Offensichtlich fehlt das dann notwendige automatische Zeichnen.

Das System verschickt an eine Form und alle Controls der Form das Paint-Event, wenn ein Neuzeichnen erforderlich ist. Auf diesen Event muss das Programm reagieren:

```
private void MyForm_Paint(object sender, PaintEventArgs e)
{
    if (!ellipse) return;
    Graphics g = e.Graphics;
    g.FillEllipse(Brushes.Azure, this.ClientRectangle);
}
```

Der Hintergrund ist zu diesem Zeitpunkt schon neu gezeichnet, d.h. wir müssen nur dann zeichnen, wenn die Ellipse vorhanden sein soll.

6.1.3 Das Paint-Event auslösen:

Es ist nicht sinnvoll den Zeichencode zweimal zu programmieren. Eine nahe liegende Lösung ist es das Paint-Event anzufordern. Dies erfolgt durch die Methode `Invalidate(true)`. Allerdings wird das Paint-Event vom System mit relative niedriger Priorität behandelt, was zu Verzögerungen führen kann. Notfalls kann man das verhindern, indem man zusätzlich `Update` aufruft. Die Methode `Refresh` kombiniert `Invalidate` + `Update`.

Allerdings verhält sich die Form immer noch eigenartig. Das liegt daran, dass bei einer Vergrößerung nur der neue Bereich neu gezeichnet wird und bei einer Verkleinerung die Form selber nicht neu gezeichnet wird. Dieses Verhalten kann man durch `SetStyle` ändern.

Das fertige Programm (Ausschnitt):

```
public class MyForm : System.Windows.Forms.Form
{
    bool ellipse = false;
    private System.Windows.Forms.Button btnDraw;
    private System.ComponentModel.Container components = null;

    public MyForm()
    {
       InitializeComponent();
       this.SetStyle(ControlStyles.ResizeRedraw, true);
    }

    private void btnDraw_Click(object sender, System.EventArgs e)
    {
       ellipse = !ellipse;
       Invalidate();
    }

    private void MyForm_Paint(object sender, PaintEventArgs e)
    {
       if (!ellipse) return;
       Graphics g = e.Graphics;
       g.FillEllipse(Brushes.Azure, this.ClientRectangle);
    }
}
```

6.2 Zeichenwerkzeuge

6.2.1 Farben (*Colors*)

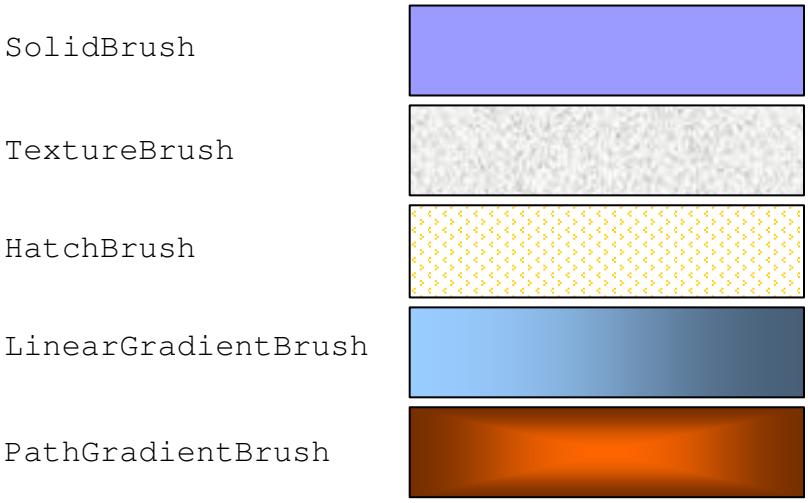
Eine Farbe ist durch vier Werte definiert, die Menge an rot, grün und blau und die Undurchsichtigkeit (*opacity*). Alle vier Werte sind vom Typ `byte` und haben Werte von 0 bis 255.

Man kann Farben durch Angabe der RGB-Werte und des Alpha-Wertes für die Opazität definieren oder man kann aus zwei Gruppen von vordefinierten Farben auswählen.

```
Color red = Color.FromArgb(255, 0, 0);
Color black = Color.FromArgb(0, 0, 0);
Color white = Color.FromArgb(255, 255, 255);
Color red25percent = Color.FromArgb(255*1/4, 255, 0, 0);
Color red = Color.Red;
Color background = SystemColors.Control;
```

6.2.2 Pinsel (*Brushes*)

Pinsel bestimmen, wie die verschiedenen Formen, die man zeichnen kann, ausgefüllt werden. Es gibt etwa die Möglichkeiten, wie man sie von der Gestaltung der Diagramme in Excel kennt.



Für den `SolidBrush` gibt es vom Framework verwaltete Typen in den vordefinierten Farben. Man referenziert sie mit

```
Brush b;
b = System.Drawing.Brushes.xxx;
b = System.Drawing.SystemBrushes.xxx;
```

`TextureBrushes` muss man sich erzeugen und für das Muster eine Graphikdatei angeben. Auch die restlichen drei muss man erzeugen und verwalten. Die zugehörigen Klassen findet man im Namespace System.Drawing.Drawing2D. Die Details werden im Konstruktor definiert. Für selber erzeugte und verwaltete Typen empfiehlt sich die Verwendung der `using`-Anweisung.

```
void MyPaint(Graphics g)
{
    int x = 0;
    int y = 0;
    int width = this.ClientRectangle.Width;
    int height = this.ClientRectangle.Height/3;
```

```
    Brush brush = Brushes.Beige;
    g.FillRectangle(brush, x, y, width, height);
    y += height;
    brush = Brushes.IndianRed;
    g.FillRectangle(brush, x, y, width, height);
    y += height;
    using (Brush myBrush = new SolidBrush(Color.FromArgb(200, 200, 200)) )
    {
       g.FillRectangle(myBrush, x, y, width, height);
    }
}
```

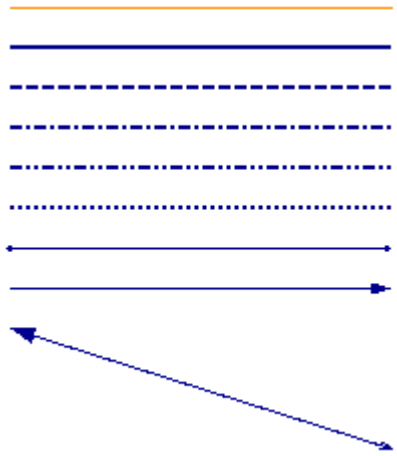
6.2.3 Stifte (*Pens*)

Stifte bestimmen den Rahmen der verschiedenen Formen. Wie bei den Pinseln gibt es auch bei den Stiften einen vordefinierten Stift für jede vordefinierte Farbe. Er hat die Stärke 1 und zeichnet eine durchgängige Linie. Die Eigenschaften der vordefinierten Stifte kann man nicht ändern.

Wichtige Pen Eigenschaften:

`Width`	Breite des Stifts, Typ float
`Color`	Farbe
`DashStyle`	Typ der Line, Voreinstellung ist `Solid` (durchgehende Linie), man kann aber auch wählen: `Dash` (strichliert), `DashDot` (strichpunktiert), `DashDotDot` (Strich - Punkt - Punkt), `Dot` (punktiert) und `Custom`.
`Alignment`	Gibt die Lage der Linie bezüglich der gewählten Koordinaten an, wenn die Breite größer als 1 ist.
`LineJoin`	Legt die Gestaltung der Ecken fest.
`StartCap, EndCap`	Gestaltung von Anfang und Ende einer Linie.

Beispiele für Linien, die mit verschiedenen Stiften gezeichnet wurden:



```
using System.Drawing.Drawing2D;
...
private void MyForm_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    int x1, y1, x2, y2;
    int dy = 20;
    x1 = 10; y1 = 10; x2 = 200; y2 = 10;
    Pen p = Pens.DarkOrange;
    g.DrawLine(p, x1, y1, x2, y2);
    using (Pen myPen = new Pen(Color.DarkBlue, 2.0f))
    {
       y1 += dy; y2 += dy;
       g.DrawLine(myPen, x1, y1, x2, y2);
       myPen.DashStyle = DashStyle.Dash;

       y1 += dy; y2 += dy;
       g.DrawLine(myPen, x1, y1, x2, y2);
       myPen.DashStyle = DashStyle.DashDot;

       y1 += dy; y2 += dy;
       g.DrawLine(myPen, x1, y1, x2, y2);
       myPen.DashStyle = DashStyle.DashDotDot;

       y1 += dy; y2 += dy;
       g.DrawLine(myPen, x1, y1, x2, y2);
       myPen.DashStyle = DashStyle.Dot;

       myPen.Width = 1.0f;
       myPen.DashStyle = DashStyle.Solid;
       myPen.StartCap = LineCap.DiamondAnchor;
       myPen.EndCap = LineCap.ArrowAnchor;
       y1 += dy; y2 += dy;
       g.DrawLine(myPen, x1, y1, x2, y2);

       myPen.StartCap = LineCap.NoAnchor;
       myPen.EndCap = LineCap.Custom;
       myPen.CustomEndCap =
          new System.AdjustableArrowCap(3.0f, 5.0f, true);
       y1 += dy; y2 += dy;
       g.DrawLine(myPen, x1, y1, x2, y2);

       myPen.StartCap = LineCap.Custom;
       myPen.CustomStartCap = new AdjustableArrowCap(4.0f, 6.0f, true);
       myPen.EndCap = LineCap.Custom;
       myPen.CustomEndCap = new AdjustableArrowCap(4.0f, 6.0f, false);
       y1 += dy; y2 += dy;
       g.DrawLine(myPen, x1, y1, x2, y2 + 60);

}
```

6.3 Koordinaten, Punkte, Rechtecke

6.3.1 Koordinaten und Einheiten

Betrachtet man die Wirkung einer Anweisung wie

```
g.DrawLine(myPen, x1, y1, x2, y2 + 60);
```

so wird schnell klar, dass beim Zeichnen am Bildschirm das Koordinatensystem den Nullpunkt in der linken oberen Ecke des Controls hat, die x-Koordinaten wie üblich von links

nach rechts die y-Koordinaten aber von oben nach unten ansteigen. Das Graphics-Objekt, wie es für Controls in einer Form verwendet wird, interpretiert Koordinaten und Längenangaben in Pixel.

Grundsätzlich gibt es aber verschiedene Einheiten, die durch die Property `PageUnit` festgelegt werden. Die unterschiedlichen Einheiten werden über die `Enumeration GraphicsUnit` festgelegt:

Enumerations Name	Einheit
Display	Pixel für Bildschirm 1/100 inch für Drucker
Document	1/300 inch
Inch	1 inch
Millimeter	1 mm (Millimeter)
Pixel	1 Pixel (Geräteeinheit)
Point	1 pt = 1/72 inch
World	Weltkoordinaten (world unit)

Für Zeichenoperationen am Bildschirm ist Pixel die voreingestellte Einheit, für ein Graphics Objekt, wie man es fürs Drucken verwendet, ist Display mit einer Einheit von 1/100 inch die voreingestellte Einheit. Die Umrechung von realen Koordinaten auf Pixel am Bildschirm hängt von mehreren Systemeinstellungen, wie z.B. dem Schriftgrad, der Auflösung des Bildschirms und der Bildschirmgröße ab. Sie dürfen daher nicht erwarten, dass z.B. eine Linie mit 100 Millimeter-Einheiten auf einem Bildschirm wirklich 100 mm sind. Am Drucker stimmt das eher.

Einheiten:

1 inch oder 1 Zoll entspricht 25.4 mm, das Einheitenzeichen ist ″ (Unicode 2033 (hex)) oder in. 1 Point ist 1/72 inch oder 0.35 mm

Wir werden später noch auf die Bedeutung der unterschiedlichen PageUnits und deren unterschiedliche Einheiten zurückkommen.

6.3.2 Punkte (Points)

Viele Zeichenoperationen benötigen einen oder mehrere Punkte, bzw. sehr oft ein Array mit Points. Im Framework gibt es zwei Datentypen für Punkte. Den Typ `Point` und den Typ `PointF`. Beides sind Strukturen mit je einem Datenfeld für x und y. Beim `Point` sind das zwei ganze Zahlen (`int`), beim `PointF` sind es zwei Werte vom Typ `float`. Eine Struktur hat vieles mit einer Klasse gemeinsam, es gibt jedoch einige Einschränkungen. Strukturen sind Wertetypen und werden am Stack abgelegt.

Die Koordinaten x und y sind über die Eigenschaften X und Y gesetzt oder abgefragt werden.

Beispiel: Zwei Punkte erzeugen und in der `DrawLine` Methode verwenden:

```
Point pt1 = new Point(10, 20);
Point pt2 = new Point(100, 120);
g.DrawLine(pen, pt1, pt2);
```

Koordinaten eines Punktes ändern:

```
pt1.X = 200;
```

```
pt1.Y = 300;
```

Ein Array mit Punkten erzeugen und verwenden:

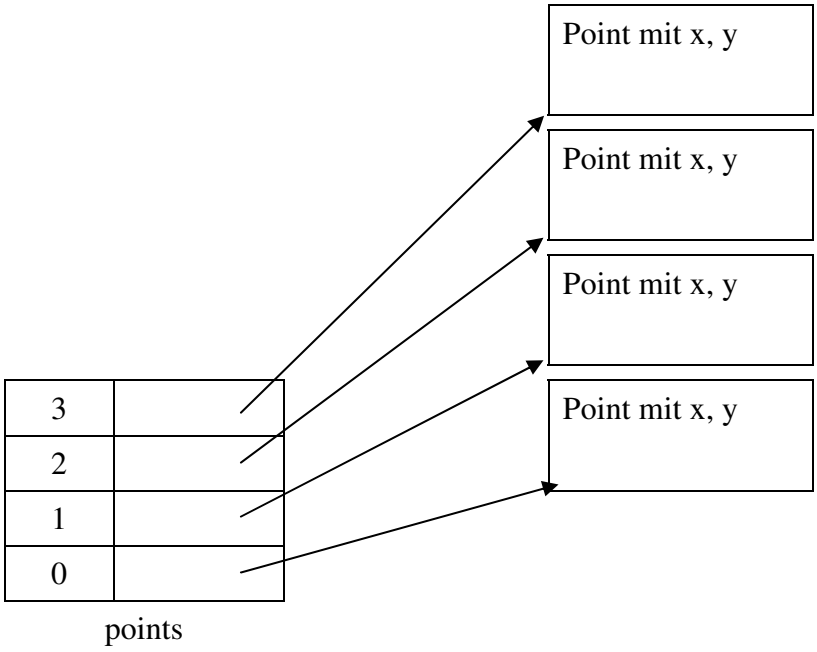
```
void DrawPolygon()
{
   Graphics g = panel.CreateGraphics();
   Pen pen = new Pen(Color.Red, 1.5f);

   Point [] points = new Point[4];       // array for 4 Point objects
   points[0] = new Point( 50,  50);
   points[1] = new Point(150,  50);
   points[2] = new Point(120, 250);
   points[3] = new Point( 70, 150);
   g.DrawPolygon(pen, points);
   for (int i = 0; i < points.Length; i++)
   {
      g.DrawEllipse(pen, points[i].X - 2, points[i].Y - 2, 4, 4);
   }
   pen.Dispose();
}
```

Mit der Zeile

```
Point [] points = new Point[4];
```

erzeugt man nur ein Array, das Referenzen auf 4 Point Objekte aufnehmen kann. Die Punkte selber sind damit noch nicht erzeugt. Diese werden mit den nächsten Anweisungen erzeugt. Im Speicherbild muss man sich das so vorstellen:



`DrawPolygon` verbindet die Punkte mit geraden Linien. Der letzte Punkt wird am Ende mit dem ersten Punkt verbunden.

Die for-Schleife sorgt dann mit dem DrawEllipse dafür, dass die einzelnen Punkt noch mit kleinen Punktsymbolen (Kreise) gezeichnet werden, die aus einer Füllung und einer Randlinie bestehen.

6.3.3 Größe (Size)

`Size` ist eine Struktur, die ebenfalls zwei Werte speichert, welche eine Größe in der Form Länge oder Breite (`Width`) und Höhe (`Height`) definieren. Auch hier gibt es eine `SizeF` Variante, sowie Operatoren und Methoden, welche z.B. von `SizeF` auf `Size` umwandeln.

6.3.4 Rechtecke (Rectangle)

Die Struktur `Rectangle` speichert 4 Ganzzahlwerte, welche die Position (*Location*) und die Größe eines Rechtecks definieren. `RectangleF` verwendet dafür 4 Werte des Typs `float`.

Wichtige Eigenschaften:

X und Y sind die Koordinaten der linken oberen Ecke, `Width` und `Height` sind die Breite und Höhe des Rechtecks. Einige weitere Eigenschaften wie `Top`, `Bottom` etc. sind nur in der `get` - Form (*readonly*) implementiert. Auch einige interessante Methoden sind verfügbar. Auch der Gleichheits- und Ungleichheitsoperator steht zur Verfügung. Ein Blick in die Hilfe ist allemal ratsam.