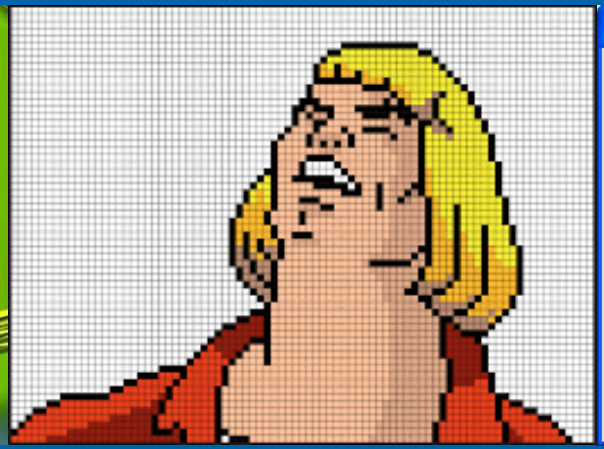
Grafiken mit C# und GDI+



eigene Zeichnungen mit GDI+

Möglichkeiten der Grafikbibliothek

Bilder darstellen und manipulieren mit der Image Klasse

Einschub: mehrere parallele Forms

- Das Erstellen von mehreren Forms innerhalb eines UI-Kontextes ist mit der Klasse **ApplicationContext** möglich

```
class Program
{
    static void Main(string[] args)
    {
        Application.EnableVisualStyles();
        Application.Run(new MultiFormContext(new MyForm("My First"), new MyForm("My Second")));
    }
}
```

```
class MyForm : Form
{
    public MyForm(string str)
    {
        this.Text = str;
    }
}
```

```
public class MultiFormContext : ApplicationContext
{
    private int openForms;
    public MultiFormContext(params Form[] forms)
    {
        openForms = forms.Length;
        foreach (var form in forms)
        {
            form.FormClosed += (s, args) =>
            {
                if (System.Threading.Interlocked.Decrement(ref openForms) == 0)
                    ExitThread();
            };
            form.Show();
        }
    }
}
```

Mit dieser Angepassten Version von **ApplicationContext** sind mehrere parallele WinForms innerhalb eines UI-Threads möglich.

Die Anwendung wird nach dem Schließen des zweiten Fensters auch geschlossen

GDI+

- steht für Graphics Device Interface
- ist der Nachfolger von GDI (seit Windows XP)
- hardwarebeschleunigtes Grafiksystem
- Alternativen: Quartz (Apple), Cairo (Linux)

GDI+ Eigenschaften

Darstellung von

- 2D Zeichnungen
- Grafiken (Rastergrafiken, …)
- Text

unterstützt:

- Texturen und Farbverläufe
- Gleitkomma – Koordinaten
- Bildbearbeitung
- Drehen, Schneiden, Spiegeln, …
- Alpha-Blending

Klasse Graphics

- bildet die Zeichenoberfläche
- besitzt keinen public Konstruktor für das Objekt
- Erstellung über den Aufruf von Steuerelementen:

```
Graphics g = this.CreateGraphics();
```

- **CreateGraphics** erzeugt Referenz auf die zugehörige Zeichenfläche
- Alternative: statische Methoden der Klasse
    - FromHDC()
    - FromHwnd()
    - FromImage()

Graphics Objekte können über 3 Arten erstellt werden:

- CreateGraphics:

```
public MyForm()
{
    Graphics MyGraphics = this.CreateGraphics();
}
```

- PaintEvent
  (Event registrieren)

```
private void MyForm_Paint(object sender, PaintEventArgs e)
{
    Graphics MyGraphics = e.Graphics;
}
```

- FromImage (statisch)

```
Image img = new Bitmap(400, 400);
Graphics MyGraphics = Graphics.FromImage(img);
```

Draw-Methoden

- stellt einen partiellen oder vollständigen Rand einer Fläche dar
- Art wird durch den Methodenname festgelegt
- Verlauf wird durch Parameter der Methode bestimmt
- Aussehen wird durch übergebenes **Pen**-Objekt festgelegt

Draw – Methoden:

- DrawArc
- DrawBezier
- DrawBeziers
- DrawCurve
- DrawClosedCurve
- DrawEllipse
- DrawIcon
- DrawIconUnstretched
- DrawImage
- DrawImageUnscaled
- DrawLine
- DrawLines
- DrawPath
- DrawPie
- DrawPolygon
- DrawRectangle
- DrawRectangles
- DrawString

Fill - Methoden

- stellt eine Fläche dar
- Art wird durch den Methodennamen festgelegt
- Größe und Position legen die Parameter der Methode fest
- Aussehen wird durch ein übergebenes **Brush**-Objekt festgelegt

Fill – Methoden

- FillClosedCurve
- FillEllipse
- FillPath
- FillPie
- FillPolygon
- FillRectangle
- FillRectangles
- FillRegion

Hilfsklassen und wichtige Funktionen

- Clear: Löscht den Inhalt und stellt eine Hintergrundfarbe dar
- Flush: führt sämtliche Zeichenoperationen bis zum Ende aus

Hilfsklassen:

- Pen
- Pens
- Brush
- SolidBrush
- TextureBrush
- Brushes

Mehr Hilfsklassen in System.Drawing.Drawing2D:

- LinearGradientBrush
- HatchBrush
- PathGradientBrush

Komplexbeispiel 1: zwinkerndes Smiley

- Gesicht soll in der Form dargestellt werden
- alle 0.5 Sekunden wird das rechte Auge geöffnet bzw. wieder geschlossen
- Zeitsteuerung über die Komponente **Timer**

- Konstruktor der Form enthält lediglich die Properties zur Form selbst

- **Graphics** Referenz und **bool**-Wert (Auge geschlossen?) anlegen

```
private Graphics g;
private bool EyeShut = false;

public MyForm()
{
    InitializeComponent();
    this.ClientSize = new Size(400, 400);
    this.BackColor = Color.White;
    this.Text = "Smiley";
}
```

```
private void MyForm_Load(object sender, EventArgs e)
{
    g = this.CreateGraphics();
    g.SmoothingMode = SmoothingMode.HighQuality;
    MyTimer.Enabled = true;
    MyTimer.Interval = 500;
}

private void MyTimer_Tick(object sender, EventArgs e)
{
    EyeShut ^= true;
    //this.Invalidate();
    this.Refresh();
}
```

- Methode für das **Load** Event sollte Graphics-Objekt erstellen
- **Timer** könnte auch in den Konstruktor (ist unabhängig von der graphischen Oberfläche)
- **Smoothing-Mode**: kontrolliert die Qualität bzw. **Anti-Aliasing**

```
private void MyForm_Paint(object sender, PaintEventArgs e)
{
    g.Clear(Color.White);
    Pen myBlackPen = new Pen(Color.Black, 3.5f);
    Pen myRedPen = new Pen(Color.Red, 6.5f);
    //Rand:
    g.FillEllipse(Brushes.Yellow, 50, 50, 300, 300);
    g.DrawEllipse(myBlackPen, 50, 50, 300, 300);
    //Mund
    g.DrawArc(myRedPen, 50, 0, 300, 300, 45, 90);
    //Nase
    g.FillPie(Brushes.Blue, 100, 70, 200, 180, 75, 30);
    //linkes Auge
    g.FillEllipse(Brushes.White, 100, 120, 80, 60);
    g.DrawEllipse(myBlackPen, 100, 120, 80, 60);
    g.FillEllipse(Brushes.Black, 120, 130, 40, 40);
    //rechtes Auge
    if(EyeShut)
    {
        g.FillEllipse(Brushes.White, 220, 120, 80, 60);
        g.DrawEllipse(myBlackPen, 220, 120, 80, 60);
        g.FillEllipse(Brushes.Black, 240, 130, 40, 40);
    }
    else
    {
        g.DrawLine(myBlackPen, 220, 160, 300, 160);
    }
}
```

- Gesicht wird zusammen mit der Form gezeichnet
- Event: **Paint**
- neues Zeichnen erfolgt durch Refresh()

Koordinatentransformationen

- Transformationen werden durch Matrixmultiplikationen realisiert
- je nach Struktur der Matrix gibt es eine spezielle Transformation
- Wdh.: Matrixmultiplikation → Tafel

Lineare Transformationen

- Skalierung
- Drehung
- Spiegelung

Affine Transformation

- Kombination aus linearer Transformation und Verschiebung
- durch Hilfskonstruktion in einer Matrixmultiplikation durchführbar
- Nutzung in GDI+ durch Klasse **Matrix**

```
g = this.CreateGraphics();
Matrix matrix = new Matrix(2, 0, 0, 1, 0, 0);
g.Transform = matrix;
```

Koordinatensysteme in C#

GDI+ beinhaltet drei Koordinatensysteme:

- Welt → Koordinatensystem des Steuerelements
- Seite → Koordinatensystem der kompletten Form
- Gerät → Koordinatensystem des vollständigen Desktops

Komplexbeispiel 2: Funktionsplotter

Graph der Funktion $f(x) = \frac{1}{3}x^3 - x^2 - \frac{1}{3}x + 1$ soll geplottet werden

- Intervall: von -2 bis +4 (Wertebereich: [-5, 5])
- Gitternetz im Raster (Abstand jeweils 1)
- Hervorheben der x- und y- Achse
- Anfangsgröße des Fensters: 300x500
- bei Skalierung des Fensters soll die Darstellung zentriert erfolgen

```
private Graphics g;
private Point[] fktPts;
Matrix matrix;
private float scale;

public MyForm()
{
    InitializeComponent();
    this.ClientSize = new Size(300, 500);
    this.BackColor = Color.White;
    scale = 50;
    matrix = new Matrix(1, 0, 0, -1, 2, 5);

    // Funktionswerte:
    fktPts = new Point[7];
    Func<int, int> f = x => (x * x * x - 3 * x * x - x + 3) / 3;
    for (int i = 0; i < fktPts.Length; ++i)
    {
        fktPts[i] = new Point(i - 2, f(i - 2));
    }
}
```

- **fktPts** beinhaltet die Werte der Funktion
- Funktion ist durch **Func Delegate** realisiert
- Skalierung entspricht 50 zum Beginn

```
private void MyForm_Paint(object sender, PaintEventArgs e)
{
    g = e.Graphics;
    // Spiegelung an der x-Achse wegen mathematischer Orientierung
    g.Transform = matrix;
    // Skalierung auf das Fenster:
    g.PageScale = scale;
    Pen gridPen = new Pen(Color.Black, 1 / scale);
    Pen axisPen = new Pen(Color.Black, 2 / scale);
    Pen plotPen = new Pen(Color.Blue, 2 / scale);
    for (int i = -5; i <= 5; ++i)
    {
        g.DrawLine(gridPen, -2, i, 4, i);
    }
    for (int i = -2; i <= 4; ++i)
    {
        g.DrawLine(gridPen, i, -5, i, 5);
    }
    g.DrawLine(axisPen, -2, 0, 4, 0);
    g.DrawLine(axisPen, 0, -5, 0, 5);
    g.DrawCurve(plotPen, fktPts);
}
```

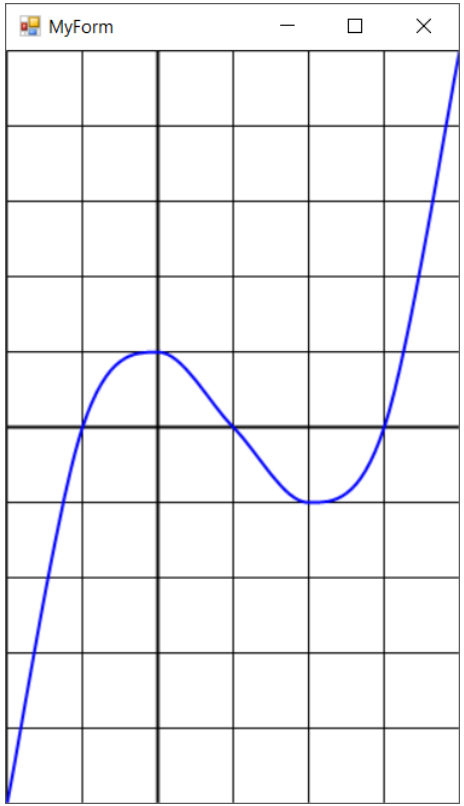
Änderung der Grafik anhand der Fenstergröße

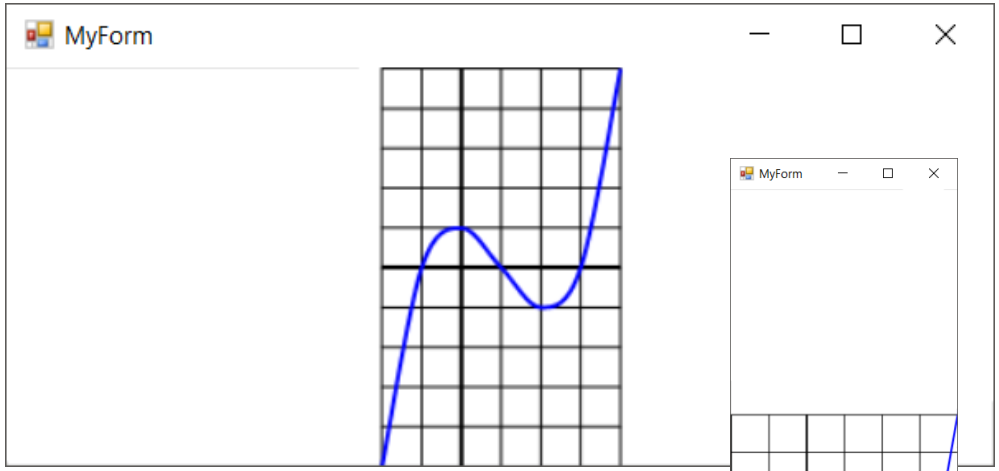
```
private void MyForm_ClientSizeChanged(object sender, EventArgs e)
{
    Size newSize = this.ClientSize;
    if (newSize.Height / 10 < newSize.Width / 6)
    {
        scale = newSize.Height / 10.0F;
    }
    else
    {
        scale = newSize.Width / 6.0F;
    }
    matrix = new Matrix(1, 0, 0, -1, (newSize.Width / scale) / 2 - 1, (newSize.Height / scale) / 2);
    this.Refresh();
}
```

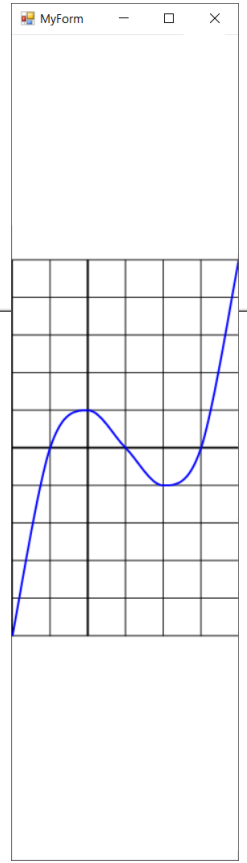
- Anpassung erfolgt durch das **ClientSizeChanged**-Event

- Skalierungswert und Translationswerte müssen angepasst werden

Funktionsplotter







- durch Anpassung ist der Plot immer gut sichtbar
- keine Verzerrung durch zentrale Skalierung

Bildklassen

- zentrale abstrakte Klasse für Pixelgrafiken: **Image**

- Implementierungen als Rückgabe statischer Methoden:
    - FromFile()
    - FromHBitmap()
    - FromStream()

- Darstellung von Image über **Graphics** oder **PictureBox**

- Klasse **Bitmap**: ermöglicht das Lesen und Schreiben einzelner Pixel

Beispiel: Rotation

In einem Fenster mit schwarzem Hintergrund soll

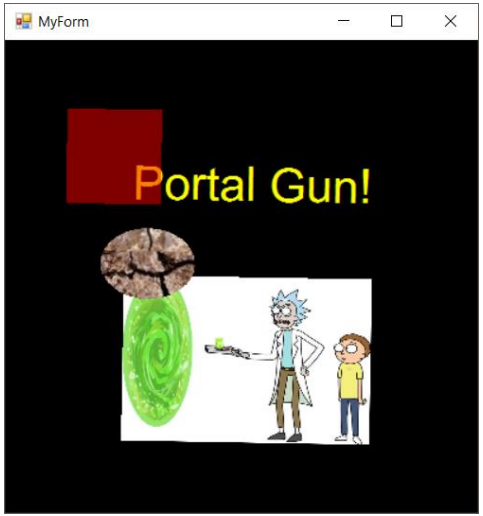
- eine skalierten Bilde
- eine mit Textur gefüllte Ellipse
- eine Zeichenkette
- ein halb-transparentes Rechteck

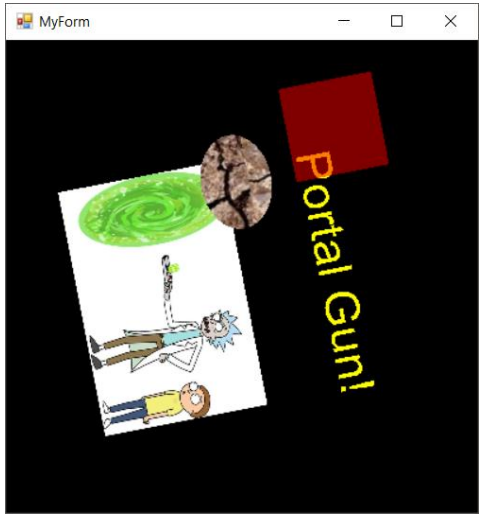
dargestellt werden und um die Bildmitte rotieren ohne zu flackern

```
private Image img;
private Image texture;
private TextureBrush textureBrush;
private SolidBrush solidBrush;
private Matrix matrix;

public MyForm()
{
    InitializeComponent();
    this.ClientSize = new Size(400, 400);
    this.DoubleBuffered = true;
    this.BackColor = Color.Black;

    img = Image.FromFile("PortalGun.jpg");
    texture = Image.FromFile("Textur.jpg");
    textureBrush = new TextureBrush(texture, new Rectangle(0, 0, 180, 180));
    solidBrush = new SolidBrush(Color.FromArgb(128, Color.Red));
    matrix = new Matrix(); // identische Abbildung

    myTimer.Enabled = true;
    myTimer.Interval = 25;
}
```

```
private void MyTimer_Tick(object sender, EventArgs e)
{
    matrix.Translate(-200.0F, -200.0F, MatrixOrder.Append);
    matrix.Rotate(1.0F, MatrixOrder.Append);
    matrix.Translate(200.0F, 200.0F, MatrixOrder.Append);
    this.Refresh();
}

private void MyForm_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.Transform = matrix;
    // Format: 3/2
    g.DrawImage(img, 100, 200, 210, 140);
    g.FillEllipse(textureBrush, new Rectangle(80, 160, 80, 60));
    g.DrawString("Portal Gun!", new Font("Arial", 30), Brushes.Yellow, 100, 100);
    g.FillRectangle(solidBrush, new Rectangle(50, 60, 80, 80));
}
```

Ergebnis





Zusammenfassung

- GDI+ stellt eine breite Palette an Werkzeugen für das Zeichen bereit
- Darstellung von Bildern und Manipulation sind ebenfalls möglich
- Manipulation von Darstellungen erfolgt über affine Transformationen
- GDI+ in Mono leider noch sehr instabil