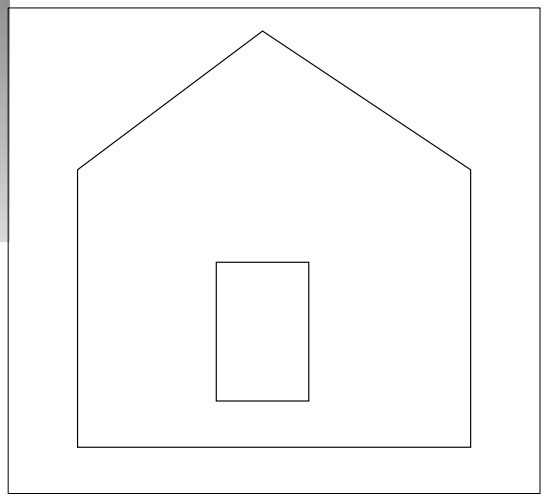
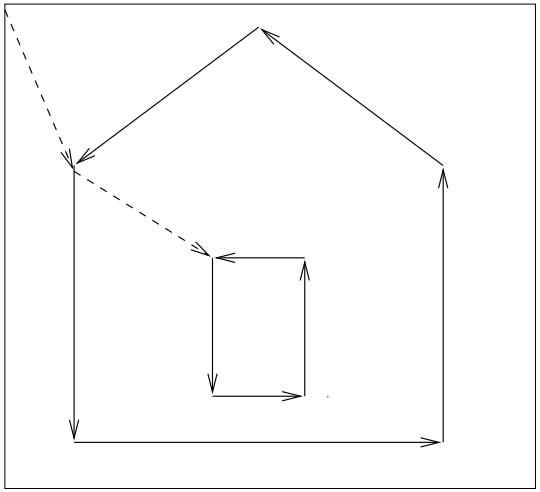
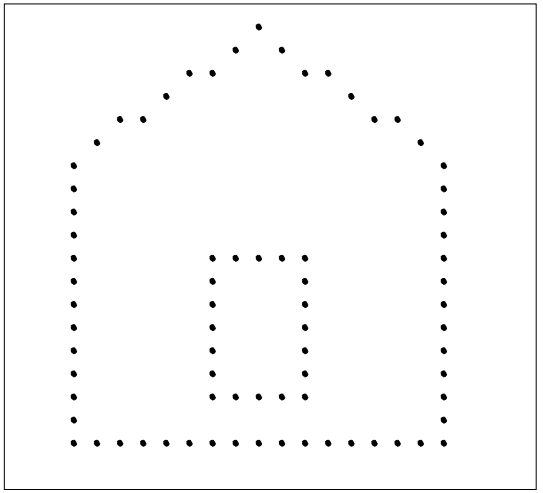
Darstellung eines Bildes



Originalbild



Vektorgrafik: Beschreibung mittels geometrischer Grundobjekte (z.B. Linienzüge, Kreise, Ellipsen,...)



Rastergrafik: Darstellung des Bildes mittels einer Pixelmatrix

Rastergrafik

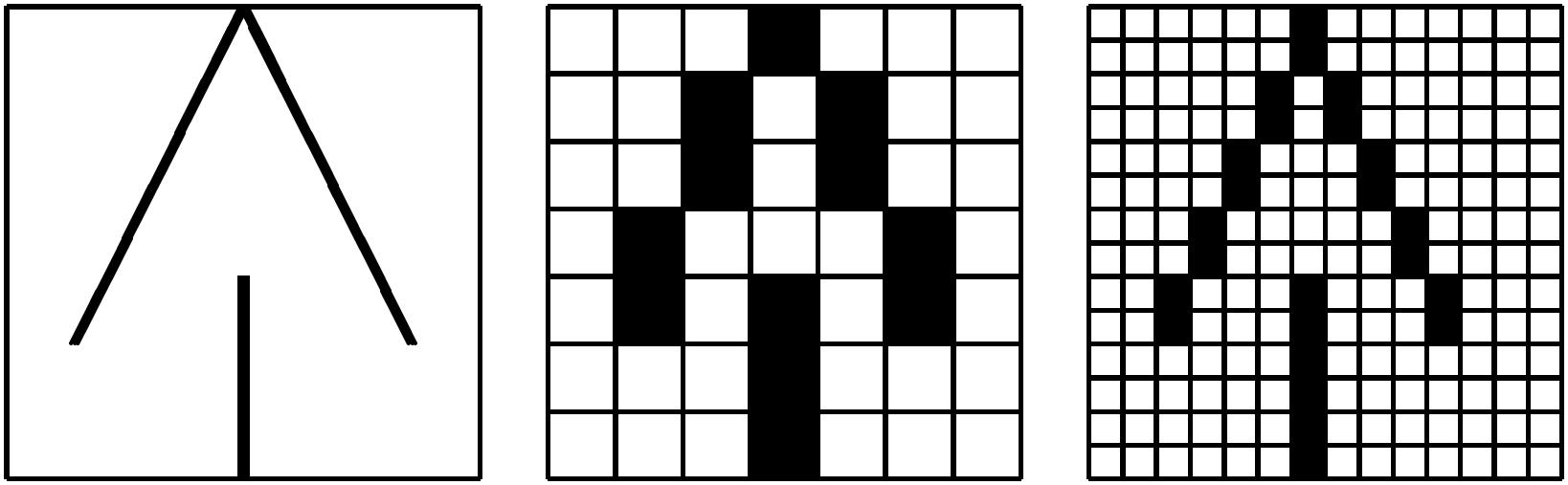
Rastergrafik bei einem Kathodenstrahlmonitor:

- Der Video-Controller durchläuft einen entsprechenden Bildschirmspeicher (zeilenweise von links nach rechts und rechts nach links).
- Bei jedem Pixel wird die Intensität des Strahls entsprechend den im Bildschirmspeicher angegebenen Informationen gewählt.
- $>$60Hz Wiederholungsrate des Bildes $\rightarrow$ kein Flackern beim Kathodenstrahlmonitor

Vektorgrafik

- skalierbar
- Umrechnung erforderlich zur Darstellung auf einem pixelorientierten Medium (Scan Conversion)
- Probleme beim Umrechnen durch **Aliasing-Effekte** (z.B. stufige Linien), die allgemein beim diskreten Abtasten eines kontinuierlichen Signals auftreten

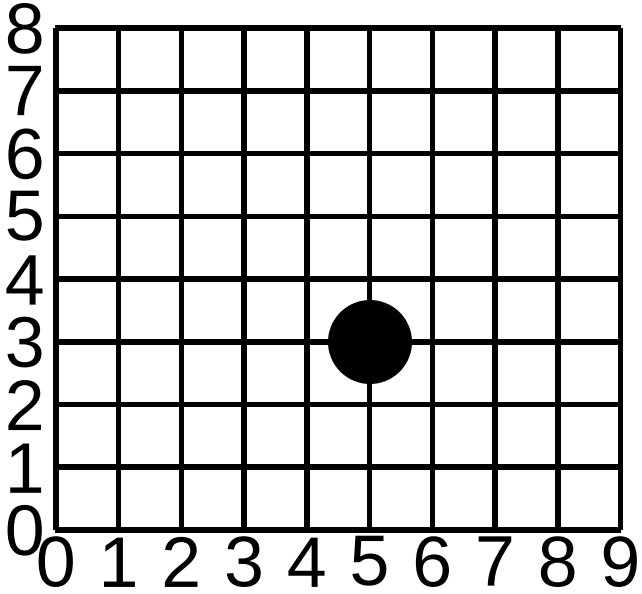
Rastergrafik und Skalierung



Eine Pfeilspitze in zwei Auflösungen dargestellt

Raster als Gitter

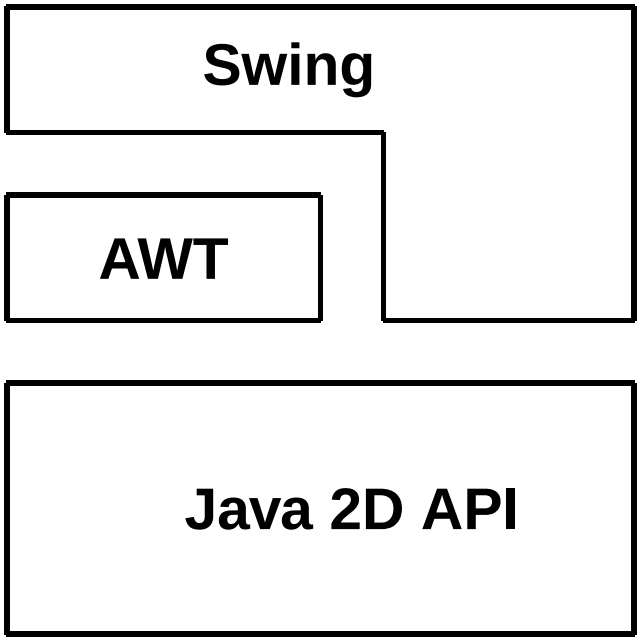
(Pixel liegen auf den Gitterpunkten)



Pixel mit den Koordinaten (5,3)

Teilweise werden die y-Koordinaten auch von oben nach unten nummeriert (z.B. bei Java).

Einstieg in Java 2D



AWT-Komponenten, die auf dem Bildschirm dargestellt werden, haben eine `paint()`-Methode, die ein `Graphics`-Objekt als Argument erhält.

Einstieg in Java 2D

In der Java 2D API erweitert die Klasse `Graphics2D` die Klasse `Graphics`.

Um die Möglichkeiten von Java 2D zu nutzen, muss nur das der `paint()`-Methode übergebene `Graphics`-Objekt innerhalb der `paint()`-Methode in ein `Graphics2D`-Objekt gecastet werden.

Das Fenster

```
import java.awt.*;

public class SimpleJava2DExample extends Frame
{
  SimpleJava2DExample()
  {
    addWindowListener(new MyFinishWindow());
  }

  public void paint(Graphics g)
  {
    Graphics2D g2d = (Graphics2D) g;
    g2d.drawString("Hello world!",30,50);
  }
```

Das Fenster

```
  public static void main(String[] argv)
  {
    SimpleJava2DExample f = new SimpleJava2DExample();
    f.setTitle("Das erste Java 2D Programm");
    f.setSize(250,80);
    f.setVisible(true);
  }
}
```

Fensterkoordinaten

- Koordinaten der linken oberen Ecke des Fensters: (0,0)
- Ausdehnung des Fensters nach rechts (hier 250 Pixel) und nach unten (hier 80 Pixel)
- y-Achse zeigt nach unten
- Auf den Rand des Fensters kann nicht gezeichnet werden.

Geometrische Grundobjekte

Punkte zur Definition anderer Objekte (z.B. Geradensegment zwischen zwei Punkten)

Kurven oder Kurvenzüge: Geradensegmente, quadratische oder kubische Kurven und daraus zusammengesetzte Formen

Flächen: geschlossener Kurvenzug

Geometrische Grundobjekte

Geradensegment: Verbindungslinie zwischen zwei Punkten

Polygonzug: Folge von aneinandergefügten Geradensegmenten

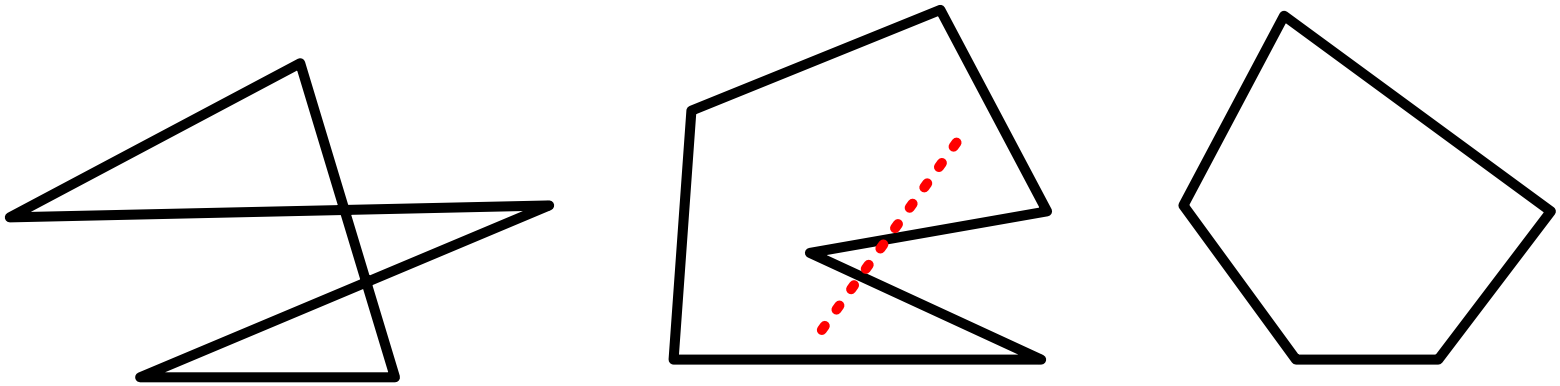
Polygon/geschlossener Polygonzug: Polygonzug, bei dem das letzte Geradensegment im Anfangspunkt des ersten endet.

Polygone

wichtige Zusatzeigenschaften von Polygonen:

Überschneidungsfreiheit der Kanten

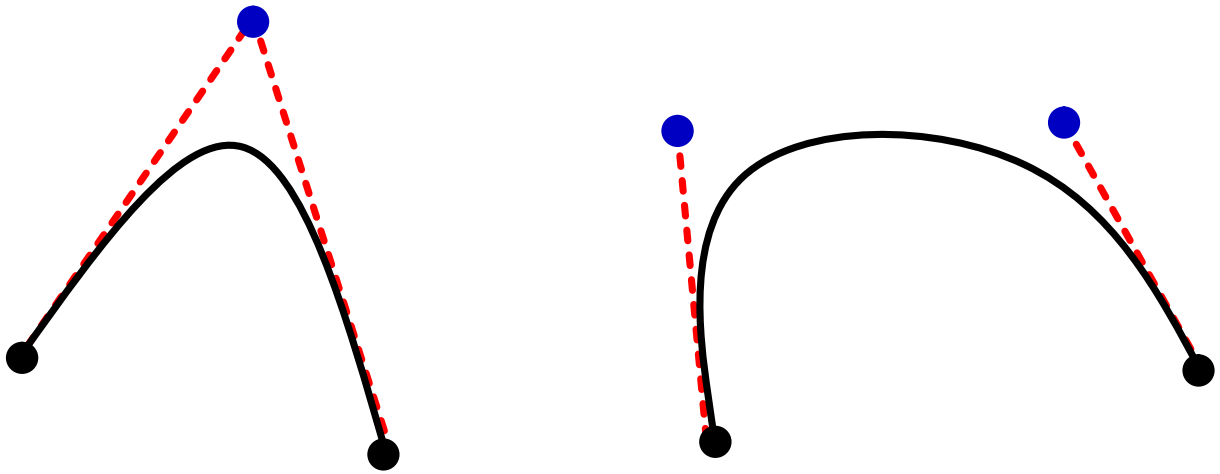
Konvexität: Mit je zwei Punkten liegt auch die Verbindungsstrecke zwischen diesen beiden Punkten vollständig innerhalb des Polygons. Diese Definition der Konvexität gilt für beliebige Flächen oder Körper.



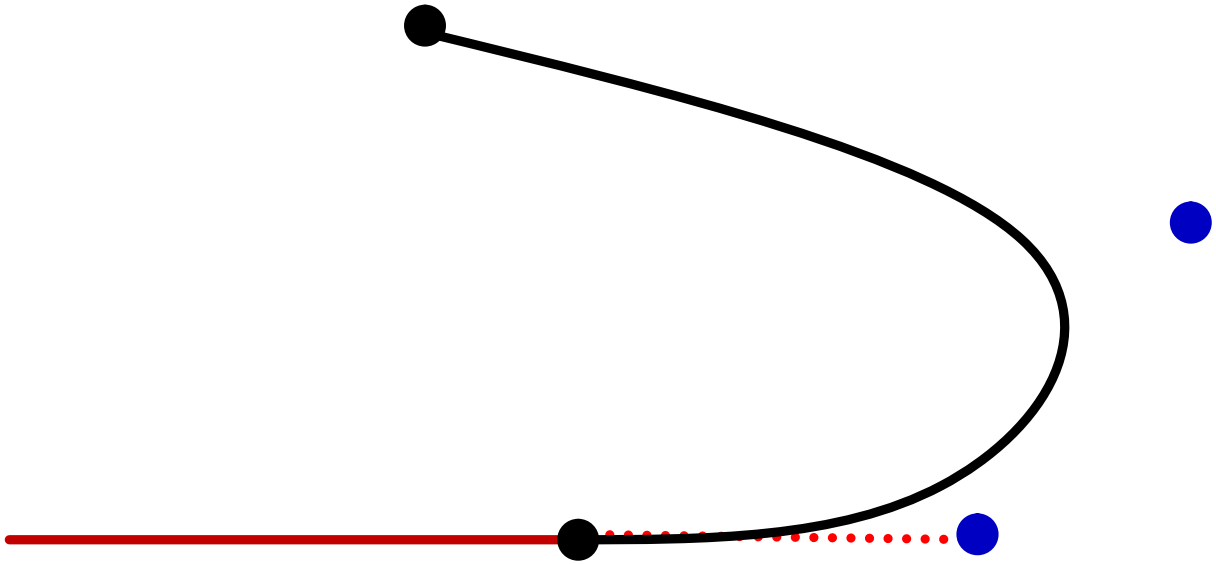
Parametrische Kurven

quadratische Kurven: Anfangs-, End- und ein Kontrollpunkt

kubische Kurven: Anfangs-, End- und zwei Kontrollpunkte



Parametrische Kurven

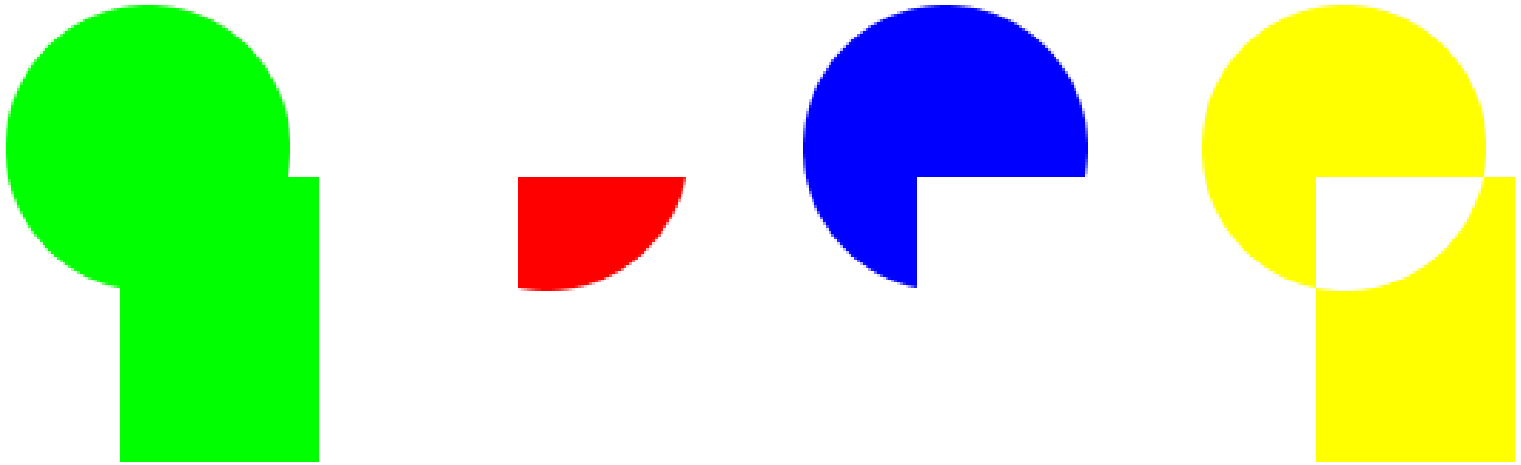


Glattes Anfügen einer kubischen Kurve an eine Gerade

Definition von Flächen

- geschlossene Kurvenzüge
- elementare geometrische Objekte wie Kreise, Ellipsen, (achsenparallele) Rechtecke
- Verformung von Flächen mittels geometrischer Transformationen
- Anwendung von mengentheoretischen Operationen wie Vereinigung, Durchschnitt, Differenz und symmetrische Differenz auf bereits erzeugte Flächen

Mengenoperationen



Vereinigung, Durchschnitt, Differenz und symmetrische Differenz eines Kreises und eines Rechtecks

Geometrische Objekte in Java 2D

- Die abstrakte Klasse `Shape` mit ihren Unterklassen stellt eine Reihe von (zweidimensionalen) geometrischen Grundobjekten zur Verfügung.
- Shapes können in reellen Koordinaten (`float` oder `double`) angegeben.
- Shapes werden erst gezeichnet, wenn in der `paint`-Methode die `draw`- oder `fill`-Methode mit dem entsprechenden Shape aufgerufen wird: `g2d.draw(shape)` bzw. `g2d.fill(shape)`

Geometrische Objekte in Java 2D

Die abstrakte Klasse `Point2D` ist keine Unterklasse von `Shape`.

Punkte können in Java nicht gezeichnet werden.

Sie dienen nur zu Beschreibung anderer Objekte.

Unterklassen von `Point2D`: `Point2D.Float` und `Point2D.Double`

Unterklassen von Shape

Linie (Geradensegment): Angabe des Anfangs- und Endpunktes der Linie:

```
Line2D.Double line =
        new Line2D.Double(x1,y1,x2,y2);
```

quadratische Kurve: Angabe eines Anfangs-, End- und Kontrollpunktes

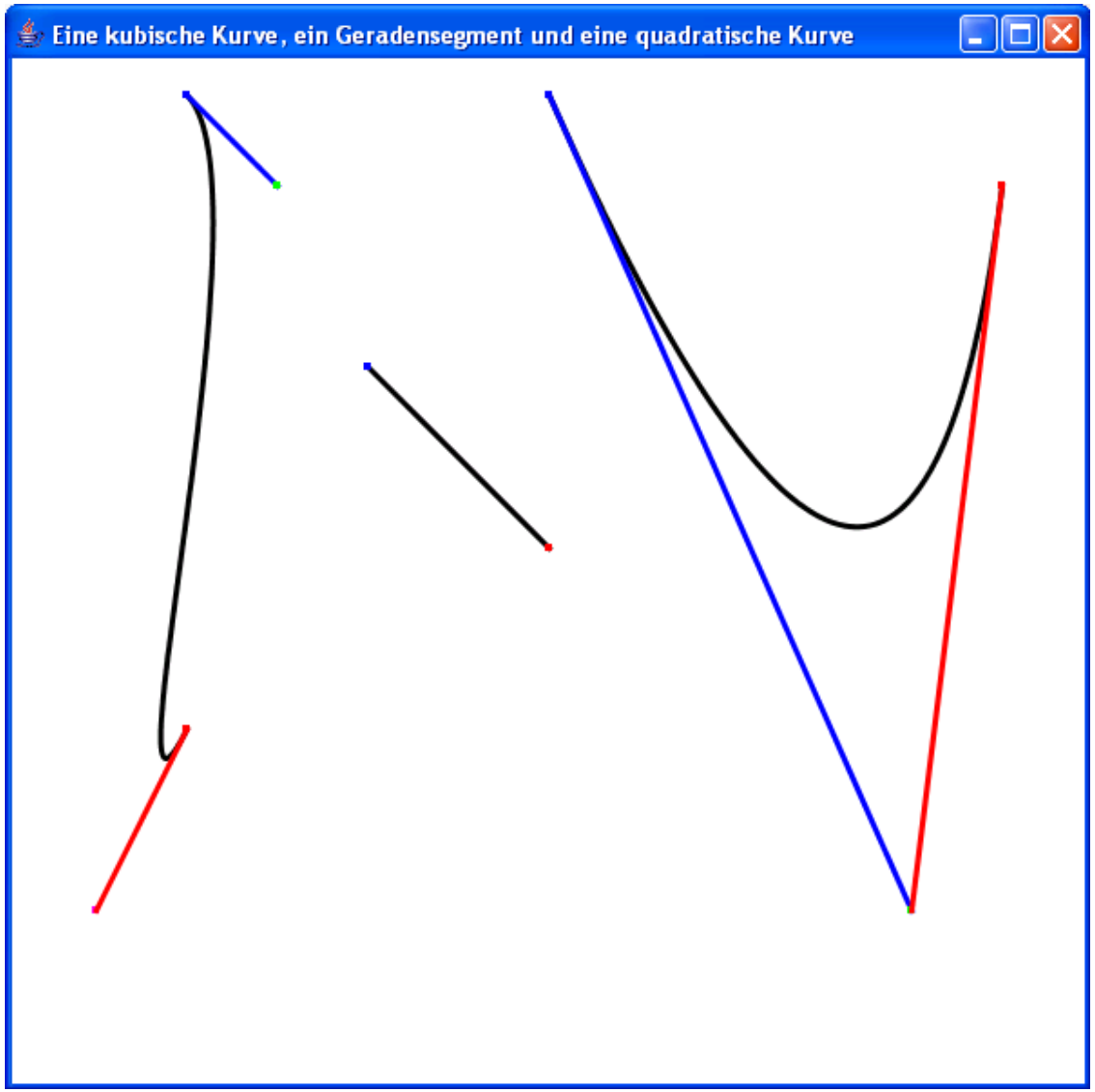
```
QuadCurve2D.Double qc =
  new QuadCurve2D.Double(x1,y1,
                         ctrlx,ctrly,
                         x2,y2);
```

Unterklassen von Shape

kubische Kurve: Angabe eines Anfangs-, End- und zweier Kontrollpunkte

```
CubicCurve2D.Double cc =
new CubicCurve2D.Double(x1,y1,
                        ctrlx1,ctrly1,
                        ctrlx2,ctrly2,
                        x2,y2);
```

CurveDemo.java



Unterklassen von Shape

general path: Ein `GeneralPath` ist ein Kurvenzug, der aus Linien, quadratischen und kubischen Kurven zusammengesetzt wird.

```
GeneralPath gp = new GeneralPath();

gp.moveTo(50,50);
gp.lineTo(50,200);
gp.quadTo(150,500,250,200);
gp.curveTo(350,-100,150,150,100,100);
gp.lineTo(50,50);
```

Unterklassen von Shape

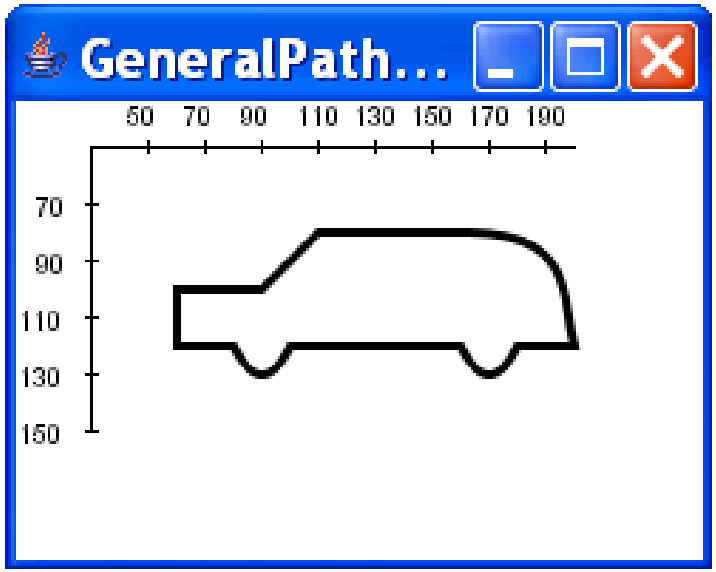
- Ein `GeneralPath` sollte mit `moveTo` beginnen, was bewirkt, dass keine Verbindungslinie/-kurve zu dem entsprechenden Punkt gezogen wird.
- `lineTo` konstruiert eine Linie vom (bisher) letzten Punkt des `GeneralPath` zum angegebenen Endpunkt.
- `quadTo` und `curveTo` konstruieren eine quadratische bzw. kubische Kurve vom (bisher) letzten Punkt zum angegebenen Endpunkt mit dem bzw. den entsprechenden Kontrollpunkten.

GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

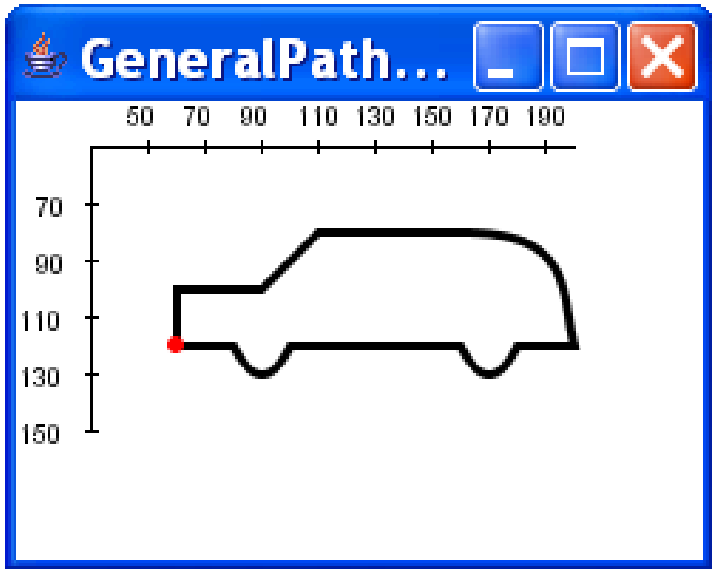


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

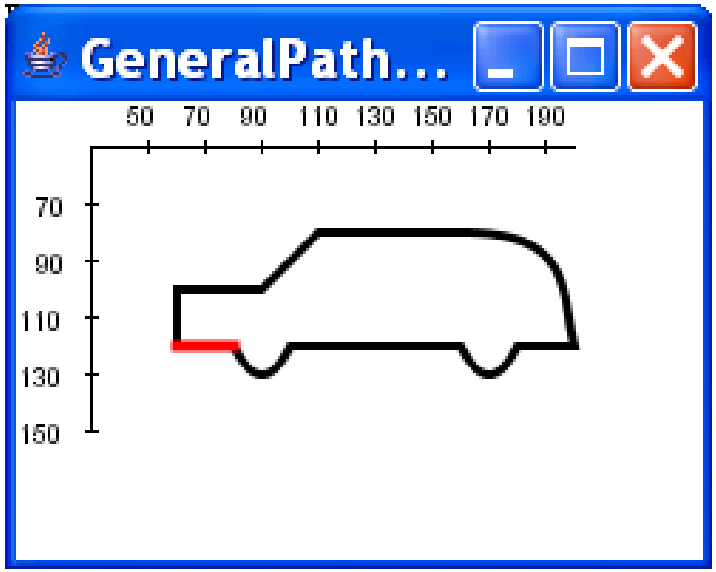


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

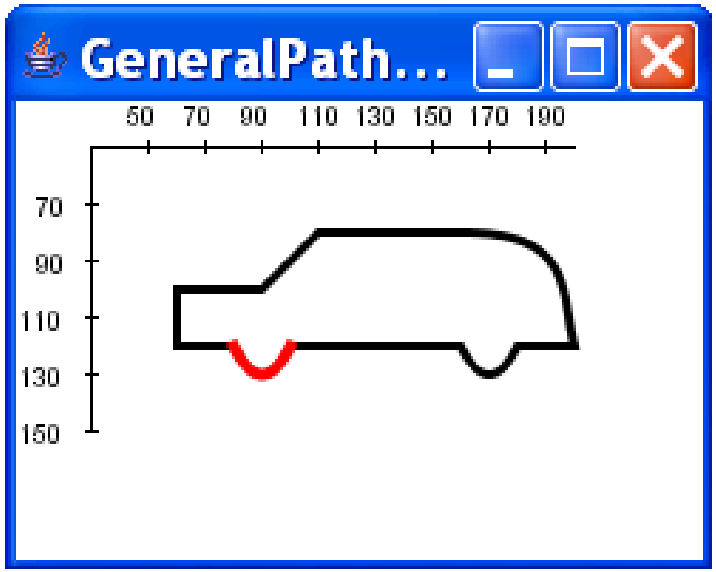


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

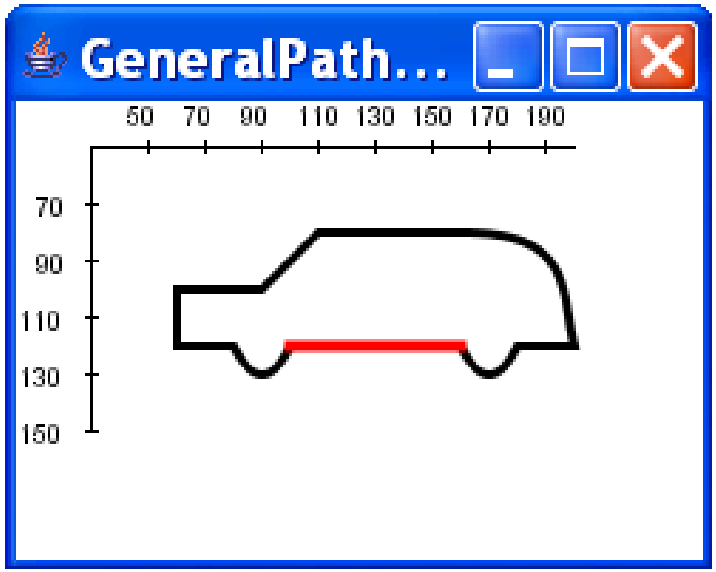


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

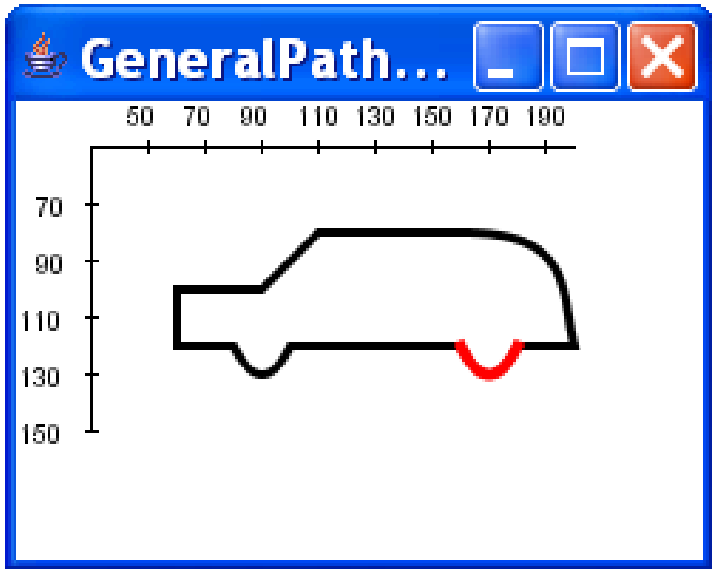


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

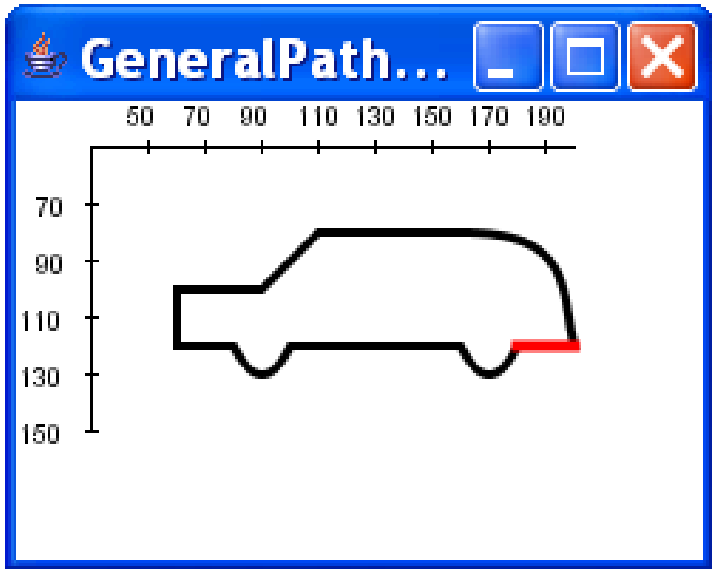


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

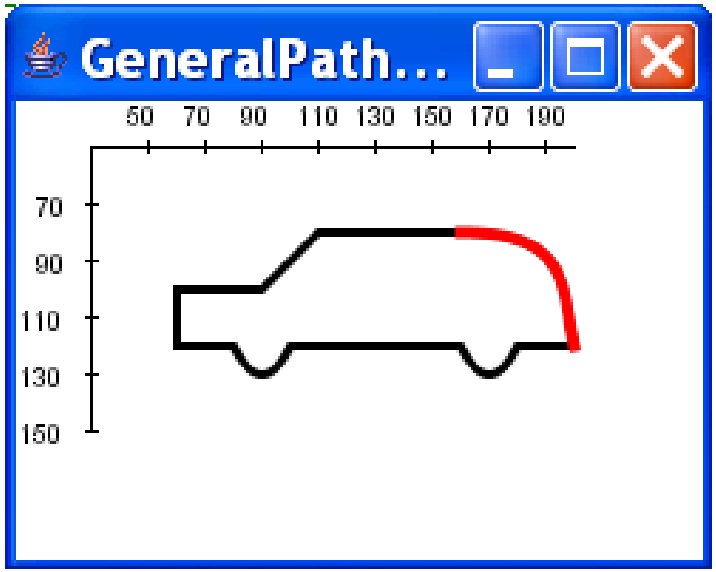


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

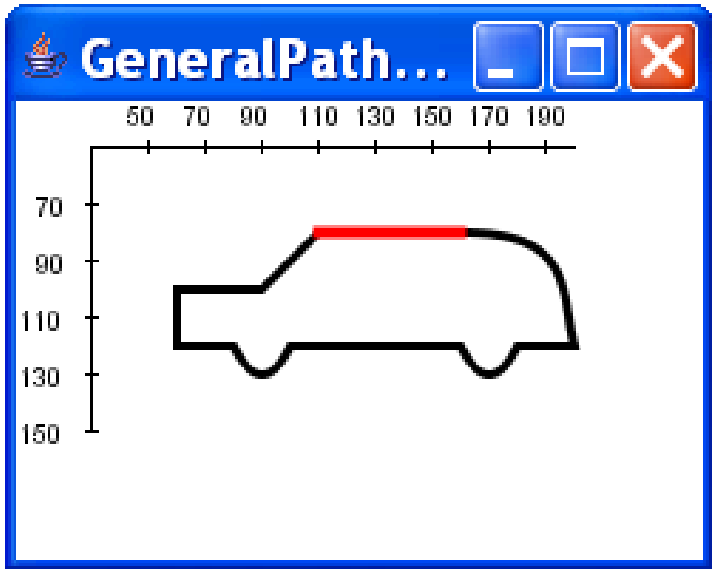


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

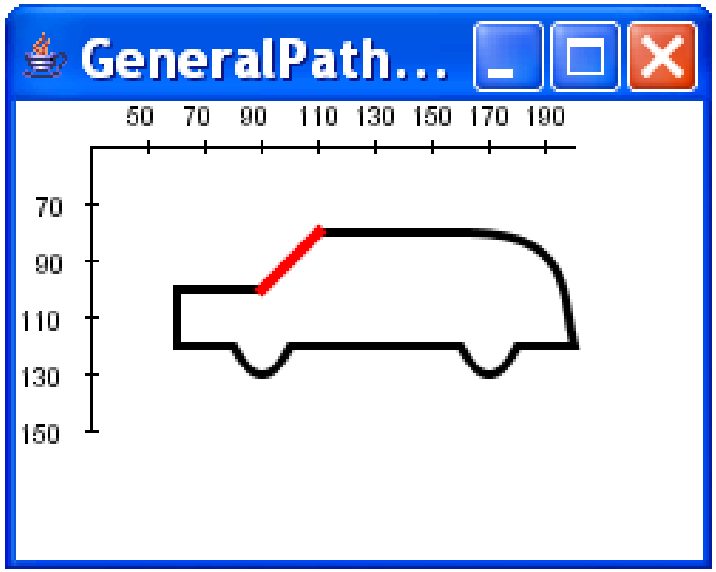


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

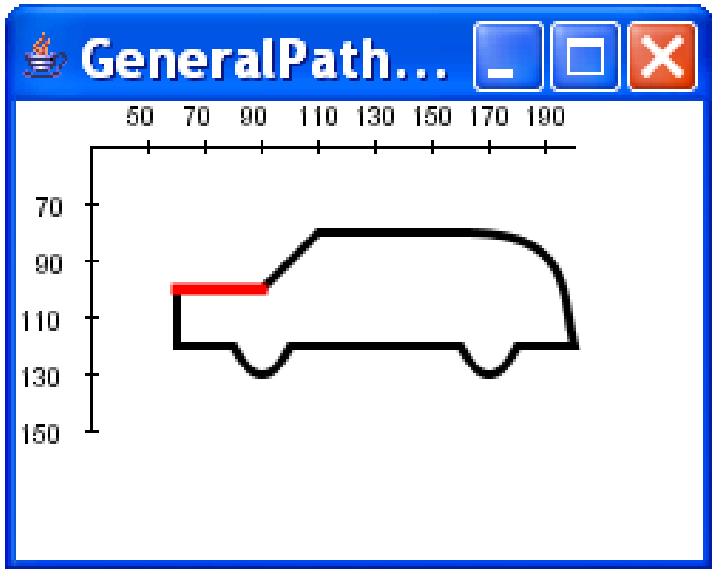


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

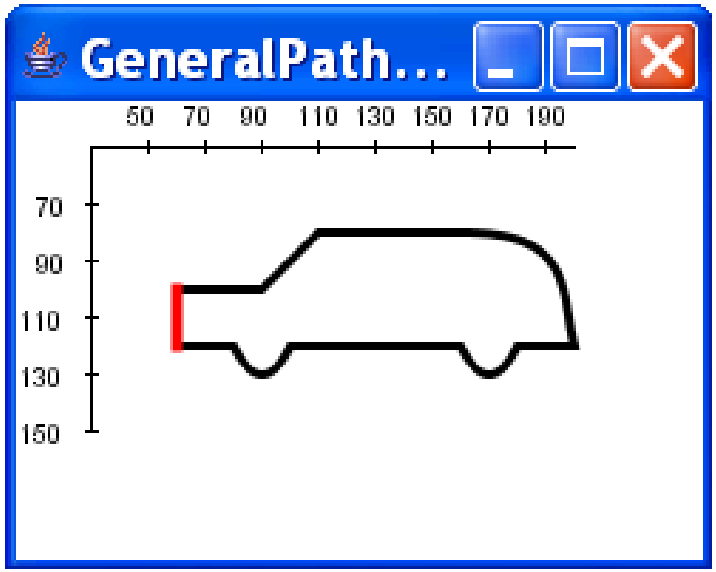


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```



Achsenparallele Rechtecke

Rechtecke: Mit

```
Rectangle2D.Double r2d =
  new Rectangle2D.Double(
            x,y,width,height);
```

kann ein Rechteck erzeugt werden,

- dessen obere linke Ecke die Koordinaten `(x,y)` hat,
- dessen Breite `width` und
- dessen Höhe `height` beträgt.

Kreise und Ellipsen

Kreise und Ellipsen: Mit

```
Ellipse2D.Double elli =
new Ellipse2D.Double(x,y,width,height);
```

kann eine Ellipse erzeugt werden, die in ein `Rectangle2D` mit den Parametern `x,y,width,height` eingepasst wird.

Mittels

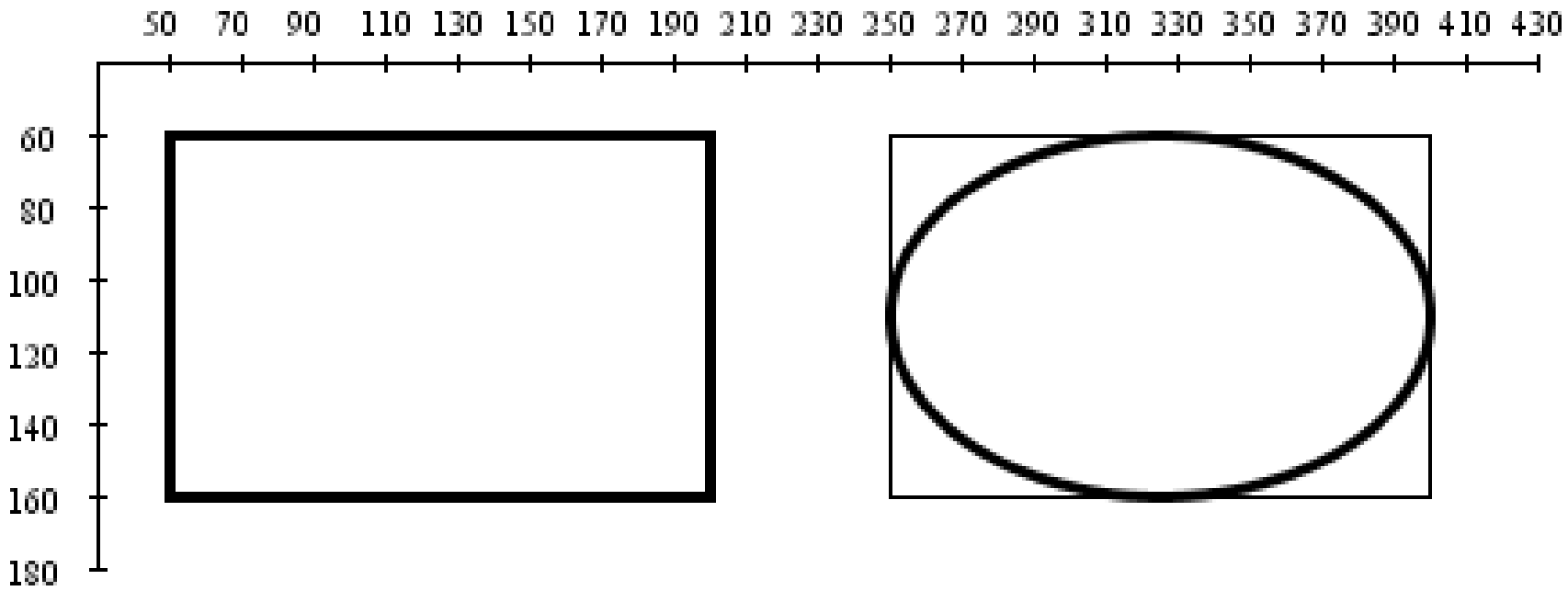
```
Ellipse2D.Double elli = new
Ellipse2D.Double(x-r,y-r,2*r,2*r);
```

wird ein Kreis mit dem Mittelpunkt `(x,y)` und dem Radius `r` erzeugt.

Beispiel: Rechteck und Ellipse

```
Rectangle2D.Double r2d =
  new Rectangle2D.Double(50,60,150,100);

Ellipse2D.Double elli =
  new Ellipse2D.Double(250,60,150,100);
```



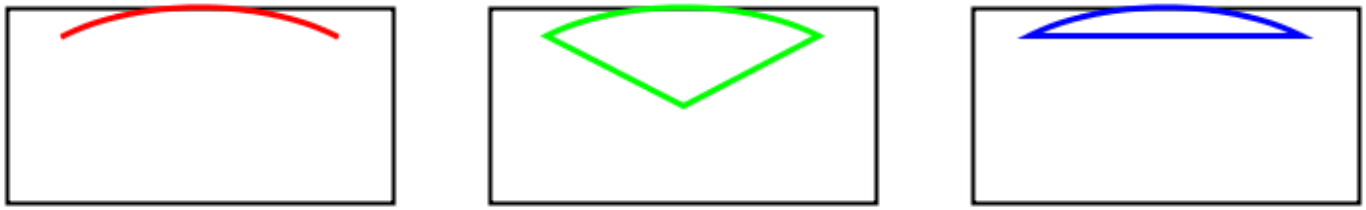
Ellipsenbögen

Bögen (Teil einer Ellipse) erzeugt man mittels

```
Arc2D.Double arc = new Arc2D.Double(
                   rect,startAngle,angle,type);
```

- `rect` ist das umschreibende `Rectangle2D` der Ellipse.
- `startAngle` ist der Winkel (in Grad) bezogen auf einen Kreis, bei dem der Bogen beginnen soll.
- `angle` ist der Öffnungswinkel, d.h., der Bogen wird von `startAngle` bis `startAngle + angle` gezeichnet (bezogen auf einen Kreis).
- `type` gibt den Typ (OPEN, PIE, CHORD) des Bogens an.

ArcExampleColour.java



```
Rectangle2D.Double rect1 =
 new Rectangle2D.Double(50,50,200,100);
Arc2D.Double arcOpen =
 new Arc2D.Double(rect1,45,90,Arc2D.OPEN);
Rectangle2D.Double rect2 =
 new Rectangle2D.Double(300,50,200,100);
Arc2D.Double arcPie =
 new Arc2D.Double(rect2,45,90,Arc2D.PIE);
Rectangle2D.Double rect3 =
 new Rectangle2D.Double(550,50,200,100);
Arc2D.Double arcChord =
 new Arc2D.Double(rect3,45,90,Arc2D.CHORD);
```

`Area`-*Objekte*

Area: Die Klasse `Area` ermöglicht beliebige Kombinationen von Shapes.

- Mittels `Area a = new Area(Shape s)` wird ein Area-Objekt in Form des entsprechenden Shapes erzeugt.

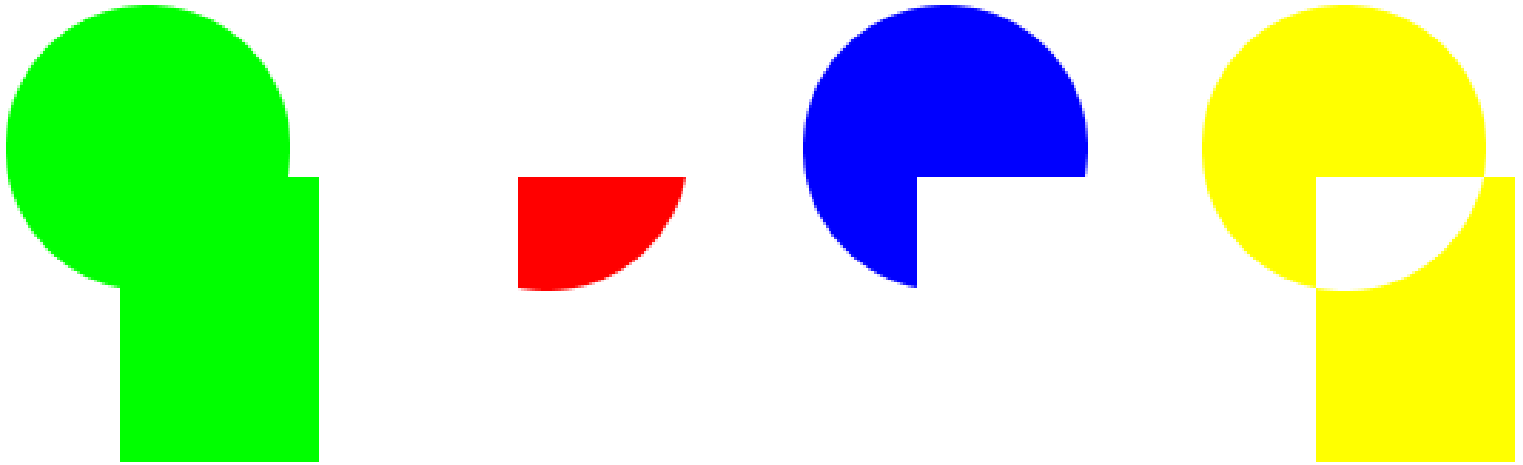
`a.add(b)` vereinigt Area `a` mit Area `b`.

`a.intersect(b)` schneidet Area `a` mit Area `b`.

`a.subtract(b)` liefert Area `a` ohne Area `b`.

`a.exclusiveOr(b)` liefert die Vereinigung von Area `a` mit Area `b` ohne den Durchschnitt.

AreaExamplesColour.java



Geometrische Transformationen (2D)

Skalierung: Streckung/Stauchung in x- und y-Richtung:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x \cdot x \\ s_y \cdot y \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

Stauchung in x-Richtung $\Leftrightarrow |s_x| < 1$

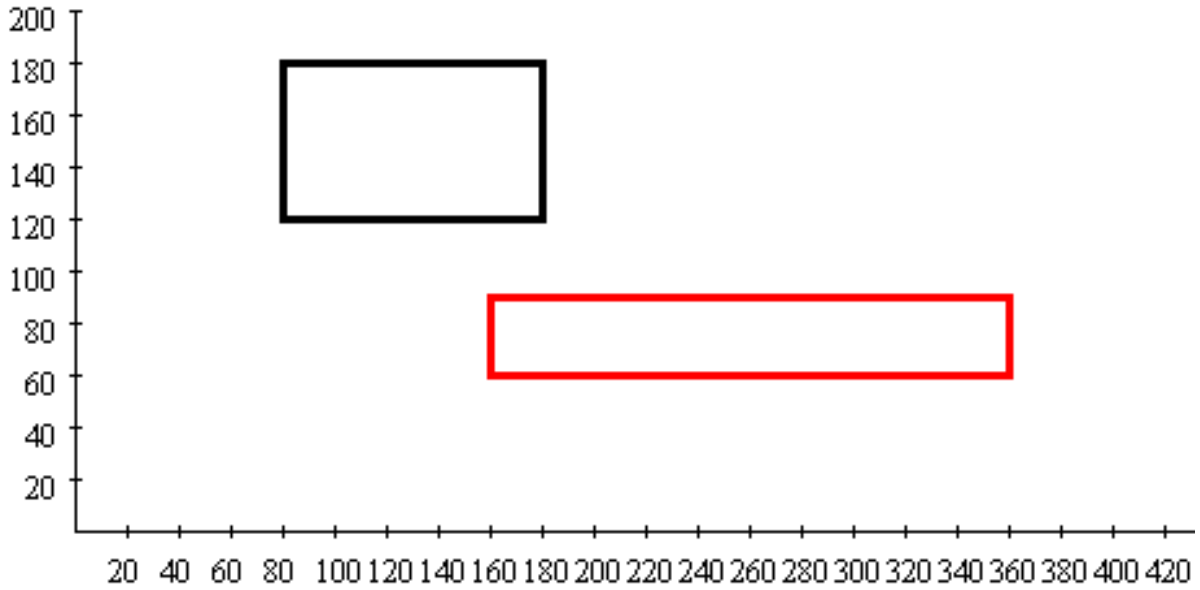
Streckung in x-Richtung $\Leftrightarrow |s_x| > 1$

Stauchung in y-Richtung $\Leftrightarrow |s_y| < 1$

Streckung in y-Richtung $\Leftrightarrow |s_y| > 1$

Bei negativen Werten s_x bzw. s_y wird zusätzlich noch an der entsprechenden Achse gespiegelt.

ScalingExampleColour.java



Skalierung mit $s_x = 2$, $s_y = 0.5$

Es wird immer bzgl. des Koordinatenursprungs skaliert. Das bedeutet, dass neben der Stauchung bzw. Streckung des Objektes noch eine Verschiebung des Objektes hinzukommt, sofern es nicht im Koordinatenursprung zentriert ist.

Geometrische Transformationen (2D)

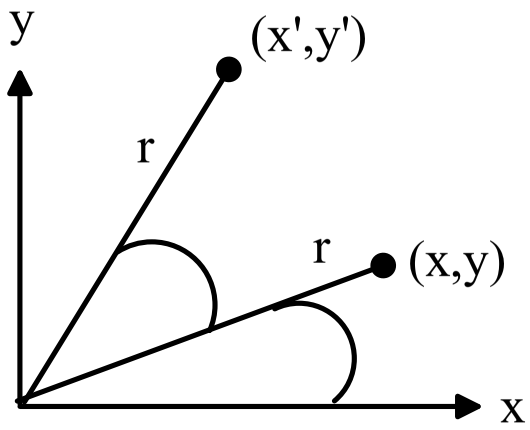
Rotation: um den Ursprung entgegen den Uhrzeigersinn um einen Winkel θ:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \cdot \cos(\theta) - y \cdot \sin(\theta) \\ x \cdot \sin(\theta) + y \cdot \cos(\theta) \end{pmatrix}$$

$$= \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

Im Gegensatz zu den Bögen wird der Winkel bei Rotationen in Java 2D im Bogenmaß angegeben!

Herleitung der Rotationsmatrix



Polarkoordinaten: $x = r \cdot \cos(\varphi), \quad y = r \cdot \sin(\varphi)$

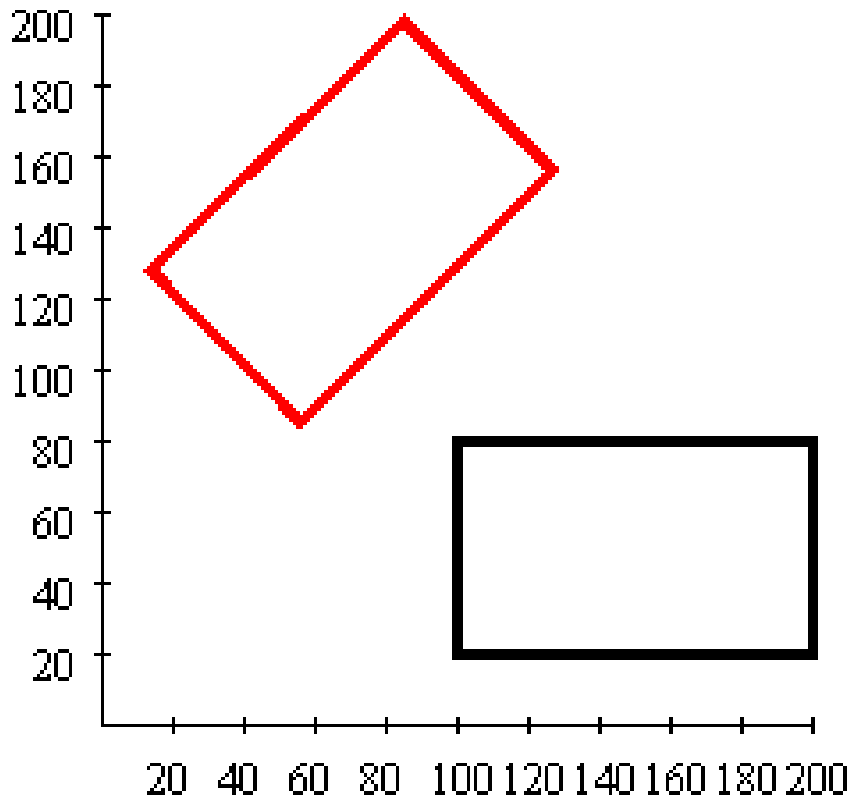
$$x' = r \cdot \cos(\varphi + \theta) = r \cdot \cos(\varphi) \cdot \cos(\theta) - r \cdot \sin(\varphi) \cdot \sin(\theta)$$

$$= x \cdot \cos(\theta) - y \cdot \sin(\theta)$$

$$y' = r \cdot \sin(\varphi + \theta) = r \cdot \cos(\varphi) \cdot \sin(\theta) + r \cdot \sin(\varphi) \cdot \cos(\theta)$$

$$= x \cdot \sin(\theta) + y \cdot \cos(\theta)$$

RotationExampleColour.java



Rotation mit $\theta = \pi/4$

Rotation

- Es wird immer bzgl. des Koordinatenursprungs rotiert.
  Das bedeutet, dass neben der Eigendrehung des Objektes noch eine Verschiebung des Objektes hinzukommt, sofern es nicht im Koordinatenursprung zentriert ist.
- Bei der Darstellung in Java 2D erscheint eine Rotation um einen positiven Winkel als Drehung im Uhrzeigersinn, da die y-Achse nach unten orientiert ist.

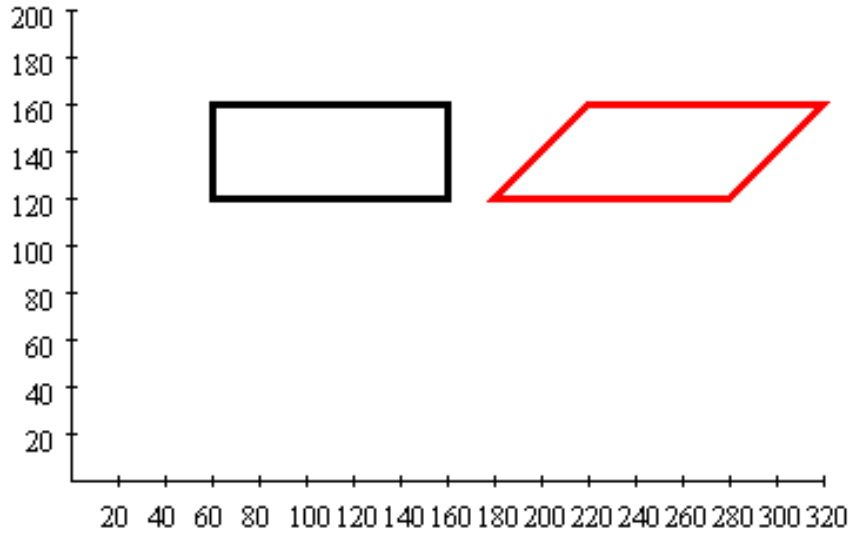
Geometrische Transformationen (2D)

Scherung: Verzerrung eines elastischen Körpers:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x + s_x \cdot y \\ y + s_y \cdot x \end{pmatrix} = \begin{pmatrix} 1 & s_x \\ s_y & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

- Ist $s_y = 0$ spricht man von einer Scherung in x-Richtung.
- Ist $s_x = 0$ spricht man von einer Scherung in y-Richtung.

ShearingExampleColour.java



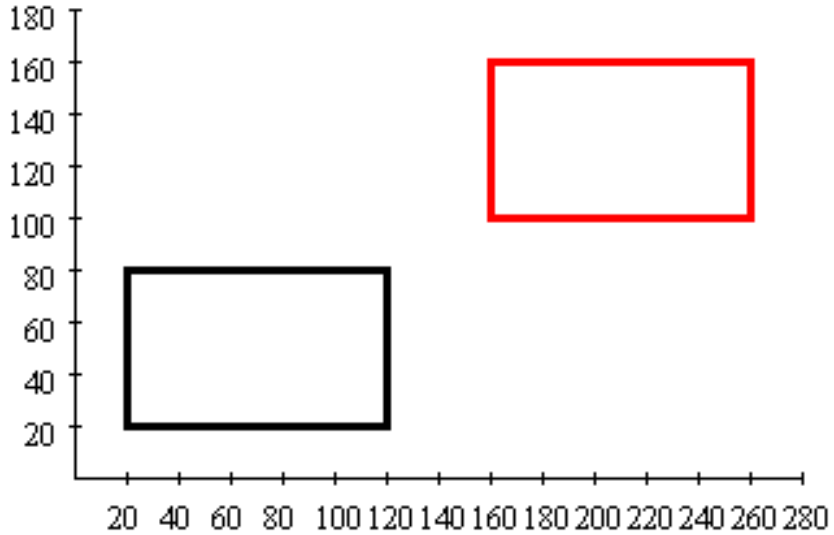
Scherung mit $s_x = 1$, $s_y = 0$

Die Scherung wird immer bzgl. des Ursprungs ausgeführt. Das bedeutet wiederum eine eventuelle Verschiebung des gescherten Objektes.

Geometrische Transformationen (2D)

Translation: Verschiebung um einen Vektor d:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix}$$



$d_x = 140,\ d_y = 80$

Geometrische Transformationen (2D)

- Eine Translation lässt sich nicht in der Form $\mathbf{v}' = A\mathbf{v}$ darstellen, da $\mathbf{0} + \mathbf{d} \neq A \cdot \mathbf{0}$ für jede Translation $\mathbf{d} \neq \mathbf{0}$ gilt.
- Alle anderen besprochenen geometrischen Transformationen lassen sich in Matrixform $v' = A \cdot v$ schreiben.
- Transformationen der Form $\mathbf{v}' = A\mathbf{v}$ heißen **lineare Transformationen**.
- Transformationen der Form $\mathbf{v}' = A\mathbf{v} + \mathbf{d}$ heißen **affine Transformationen**.

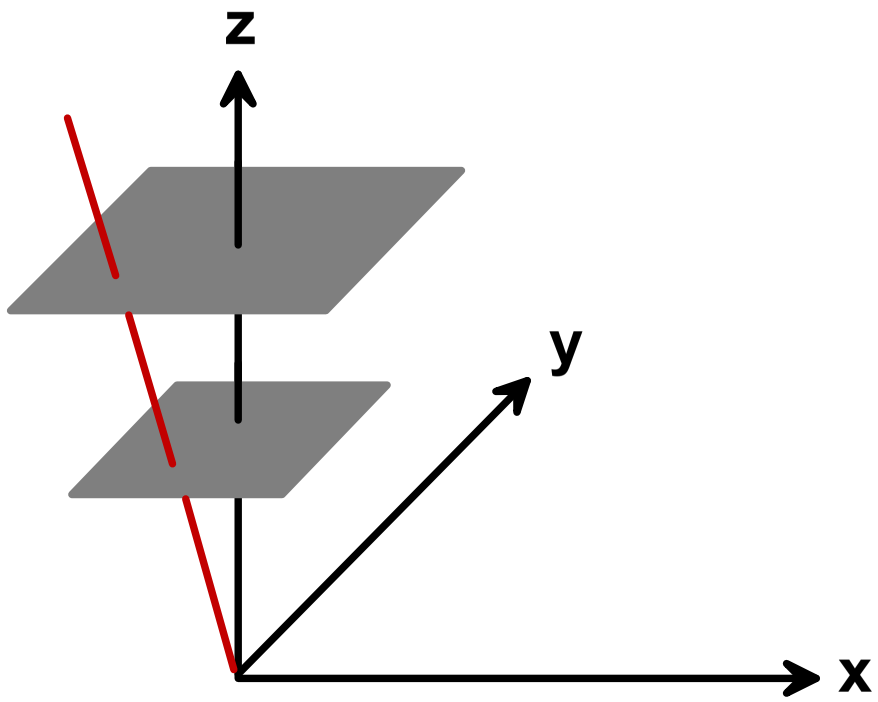
Homogene Koordinaten

Um beliebige affine Transformationen als Matrixmultiplikation darstellen zu können, verwendet man **homogene Koordinaten**.

(mehrfache) Einbettung der Ebene in den $\mathbb{R}^3 \backslash (\mathbb{R}^2 \times \{0\})$:

Der Punkt (x, y, z) (mit $z \neq 0$) in homogenen Koordinaten wird mit dem Punkt $\left(\frac{x}{z}, \frac{y}{z}\right)$ in der Ebene (in gewöhnlichen Koordinaten) identifiziert.

Homogene Koordinaten



Speziell wird der Punkt $(x, y, 1)$ mit dem Punkt (x, y) identifiziert.

(Formal: Zwei Vektoren/Punkte $\mathbf{v}, \mathbf{v}' \in \mathbb{R}^3 \backslash (\mathbb{R}^2 \times \{0\})$ sind äquivalent, falls ein $\lambda \in \mathbb{R}$ existiert mit $\mathbf{v}' = \lambda \mathbf{v}$.)

Translation in hom. Koordinaten

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Translationsmatrix: $T(d_x, d_y) = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix}$

Transformationsmatrizen

Transformation	Kurzform	Matrix
Translation	$T(d_x, d_y)$	$\begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix}$
Skalierung	$S(s_x, s_y)$	$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Transformationsmatrizen

Rotation $R(\theta)$

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Scherung $S(s_x, s_y)$

$$\begin{pmatrix} 1 & s_x & 0 \\ s_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Verkettung von Transformationen

Die Hintereinanderausführung von geometrischen Transformationen lässt sich in homogenen Koordinaten mittels Matrixmultiplikation realisieren.

Alle bisherigen Transformationsmatrizen haben die Form

$$\begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}$$

Als Produkt zweier solcher Matrizen ergibt sich wiederum eine Matrix dieser Form.

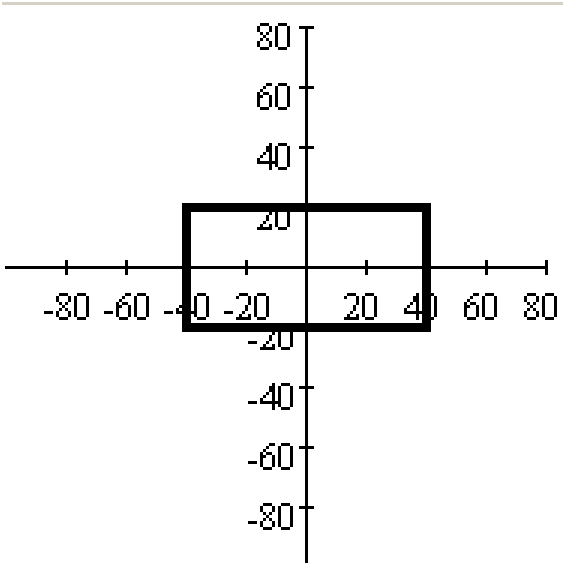
Verkettung von Transformationen

Rotation und Translation erhalten Längen und Winkel. Die Skalierung und die Scherung erhalten im allgemeinen weder Längen noch Winkel, aber zumindest Parallelität von Linien.

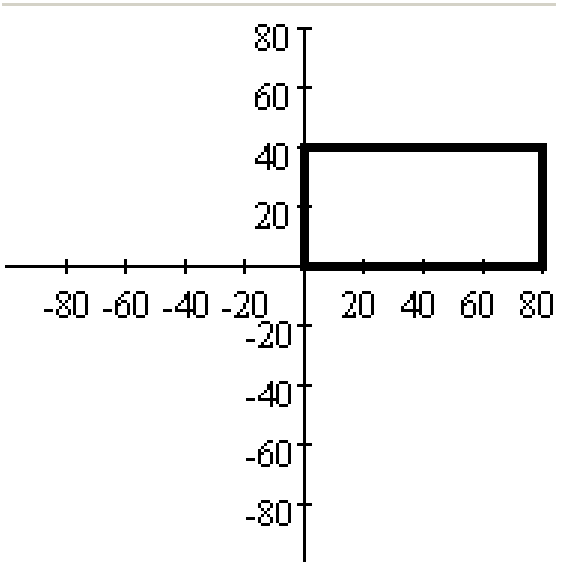
Nicht-Kommutativität der Matrixmultiplikation: Bei der Hintereinanderschaltung von geometrischen Transformationen führt die Ausführungsreihenfolge im allgemeinen eine Rolle. Ausnahmen sind: Hintereinanderschaltung gleichartiger Transformationen (Rotation und Rotation, Translation und Translation, Skalierung und Skalierung) und Rotation mit Skalierung mit gleicher Skalierung der x- und y-Achse.

Reihenfolgeabhängigkeit

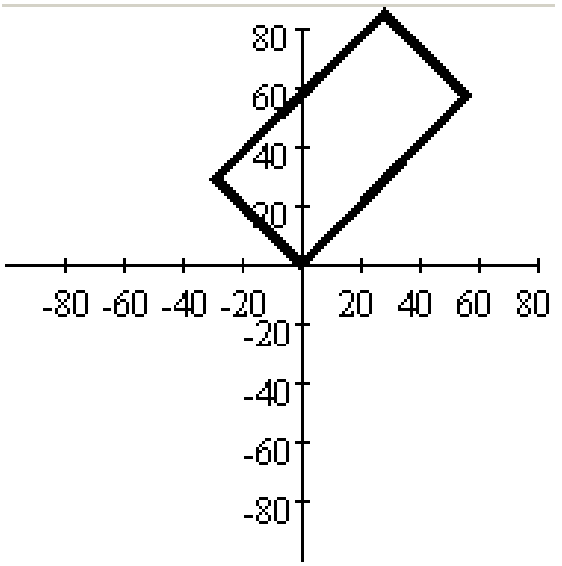
$$R(45^\circ) \cdot T(40, 20)$$



Original



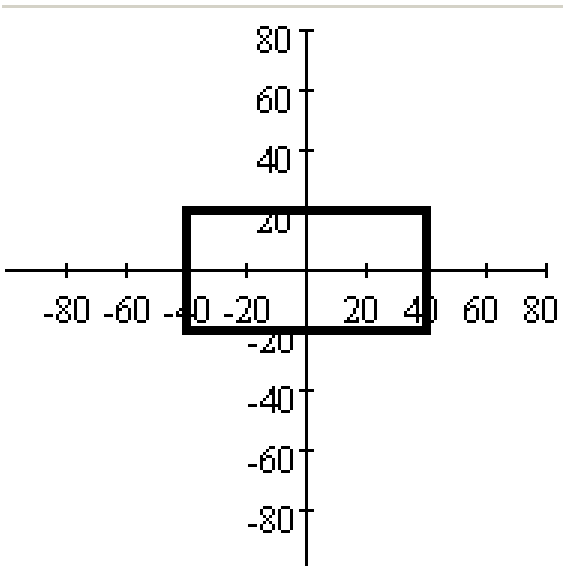
Translation



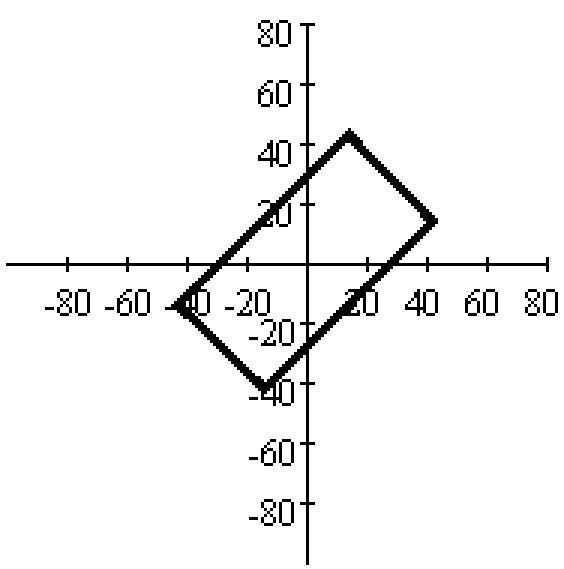
1. Translation,
2. Rotation

Reihenfolgeabhängigkeit

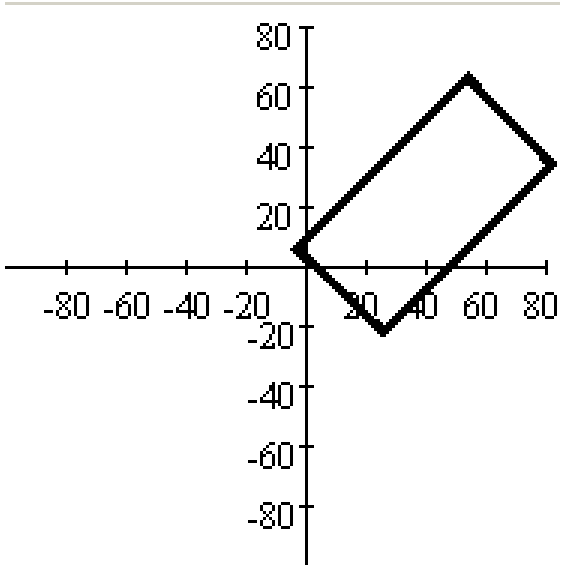
$$T(40, 20) \cdot R(45^\circ)$$



Original



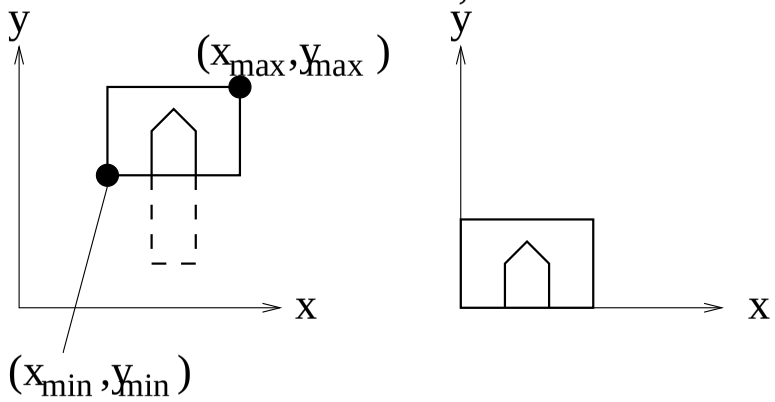
Rotation

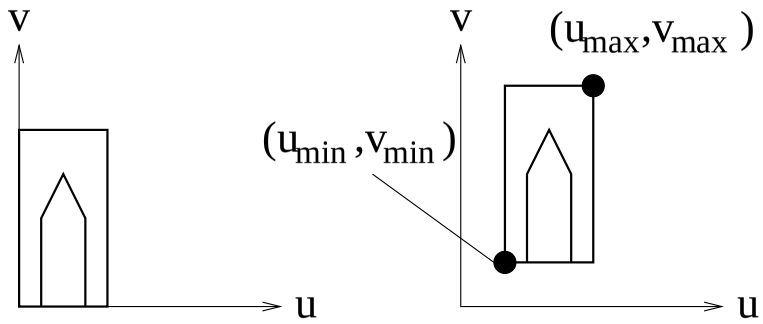


1. Rotation,
2. Translation

Welt- $\rightarrow$ Fensterkoordinaten

Fenster in Weltkoordinaten, im Weltkoordinatenursprung





skaliertes Fenster im Koordinaten-urspring

Endposition des Fensters

Welt- $\rightarrow$ Fensterkoordinaten

Die gesuchte Abbildung lautet:

$$
\begin{aligned}
M_{WV} &= T(u_{\text{min}}, v_{\text{min}}) \cdot \\
&\quad S\left(\frac{u_{\text{max}} - u_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}, \frac{v_{\text{max}} - v_{\text{min}}}{y_{\text{max}} - y_{\text{min}}}\right) \cdot \\
&\quad T(-x_{\text{min}}, -y_{\text{min}})
\end{aligned}
$$

Rotation und Skalierung

Rotation um den Punkt (x_0, y_0):

$$R(\theta, x_0, y_0) \;=\; T(x_0, y_0) \cdot R(\theta) \cdot T(-x_0, -y_0)$$

Skalierung bzgl. des Punktes (x_0, y_0)

$$S(s_x, s_y, x_0, y_0) \;=\; T(x_0, y_0) \cdot S(s_x, s_y) \cdot T(-x_0, -y_0)$$

Spiegelung der y-Achse

- normalerweise bei Bildschirmfensterkoordinaten:
  - Koordinatenursprung in der linken, oberen Ecke
  - y-Achse zeigt nach unten
- (manchmal) gewünscht:
  - Koordinatenursprung in der linken, unteren Ecke
  - y-Achse zeigt nach oben

anzuwendende Transformation bei Fenster der Höhe h:

$$T(0, h) \cdot S(1, -1)$$

Transformationen in Java 2D

Klasse `AffineTransform`: Grundlage der affinen Transformationen in homogenen Koordinaten in Java 2D

Konstruktoren:

- `AffineTransform id = new AffineTransform()` erzeugt die identische Abbildung, die durch die Einheitsmatrix kodiert wird.
- explizite Angabe der Matrix:

  `new AffineTransform(a,b,c,d,e,f)`

  Dabei sind $a, \ldots, f$ die sechs (`Double`-)Parameter der Transformationsmatrix.

Transformationen in Java 2D

Rotation:

- Mit `at.setToRotation(angle)` und `at.setToRotation(angle,x,y)` wird die Transformation `at` als eine Rotation um den Winkel `angle` um den Ursprung bzw. um den Punkt `(x,y)` definiert.
- `at.rotation(angle)` und `at.rotation(angle,x,y)` fügen an die Transformation `at` eine entsprechende Rotation an (als Matrixmultiplikation von rechts, so dass die Rotation vor der ursprünglichen Transformation `at` ausgeführt wird).

Transformationen in Java 2D

Skalierung:

- `at.setToScale(sx,sy)` definiert die Transformation `at` als eine Skalierung mit den Skalierungsfaktoren `sx` für die x- und `sy` für die y-Achse bzgl. des Ursprungs.
- `at.scale(sx,sy)` fügt an die Transformation `at` eine entsprechende Skalierung an.

Scherung:

- `at.setToShear(sx,sy)` definiert die Transformation `at` als eine Scherung mit den Scherungswerten `sx` für die x- und `sy` für die y-Achse bzgl. des Ursprungs.
- `at.shear(sx,sy)` fügt an die Transformation `at` eine entsprechende Scherung an.

Transformationen in Java 2D

Translation:

- Die Methode `at.setToTranslation(dx,dy)` definiert die Transformation `at` als eine Translation um den Vektor $(dx, dy)^\top$.
- Die Methode `at.translate(dx,dy)` fügt an die Transformation `at` eine entsprechende Translation an.

Transformationen in Java 2D

Hintereinanderschaltung von Transformationen:

- Mit `at1.concatenate(at2)` wird die affine Transformation `at2` an die affine Transformation `at1` angefügt (im Sinne einer Matrixmultiplikation von rechts, so dass zuerst `at2` und dann die ursprüngliche Transformation `at1` ausgeführt wird).
- Mit `at1.preConcatenate(at2)` wird die affine Transformation `at2` vor die affine Transformation `at1` geschaltet (im Sinne einer Matrixmultiplikation von rechts, so dass zuerst die ursprüngliche Transformation `at1` und dann `at2` ausgeführt wird).

Transformationen des Bildes

Wendet man eine Transformation `at` mittels `g2d.transform(at)` auf das Graphics2D-Objekt `g2d` an, wird das gesamte Bild transformiert.

Dies kann z.B. dafür benutzt werden, um die Problematik, dass die y-Achse bei Fenstern nach unten zeigt, zu umgehen.

```
AffineTransform yUp =
                    new AffineTransform();
yUp.setToScale(1,-1);
AffineTransform translate =
                    new AffineTransform();
translate.setToTranslation(0,windowHeight);
yUp.preConcatenate(translate);
```

Transformationen eines Objektes

Viele Objekte lassen sich relativ einfach aus elementaren Objekten durch Transformationen erzeugen.

Soll beispielsweise ein Rechteck gezeichnet werden, dessen Mittelpunkt im Punkt (x, y) liegt und das nicht achsenparallel ist, sondern einen Winkel von $45°$ zu den Koordinatenachsen bildet, so kann ein Rechteck im Ursprung erzeugt werden, auf das zunächst eine Rotation und dann eine Translation angewendet wird.

Transformation von Objekten

Anwendung einer Transformation `affTrans` auf ein `Shape s`:

```
Shape transformedShape =
    affTrans.createTransformedShape(s);
```

Entsprechend für eine `Area a`:

```
Area transformedArea =
    affTrans.createTransformedArea(a);
```

Bewegte Objekte

Um Objekte zu bewegen (animierte Grafik), können ebenfalls Transformationen verwendet werden.

Die Transformationen müssen die Bewegung des Objektes beschreiben,

- das Objekt wird gezeichnet,
- das weiterbewegte Objekt mittels der Transformationen berechnet,
- das alte Objekt bzw. das Fenster gelöscht und
- das neu berechnete Objekt und ggf. auch alle anderen gezeichnet.

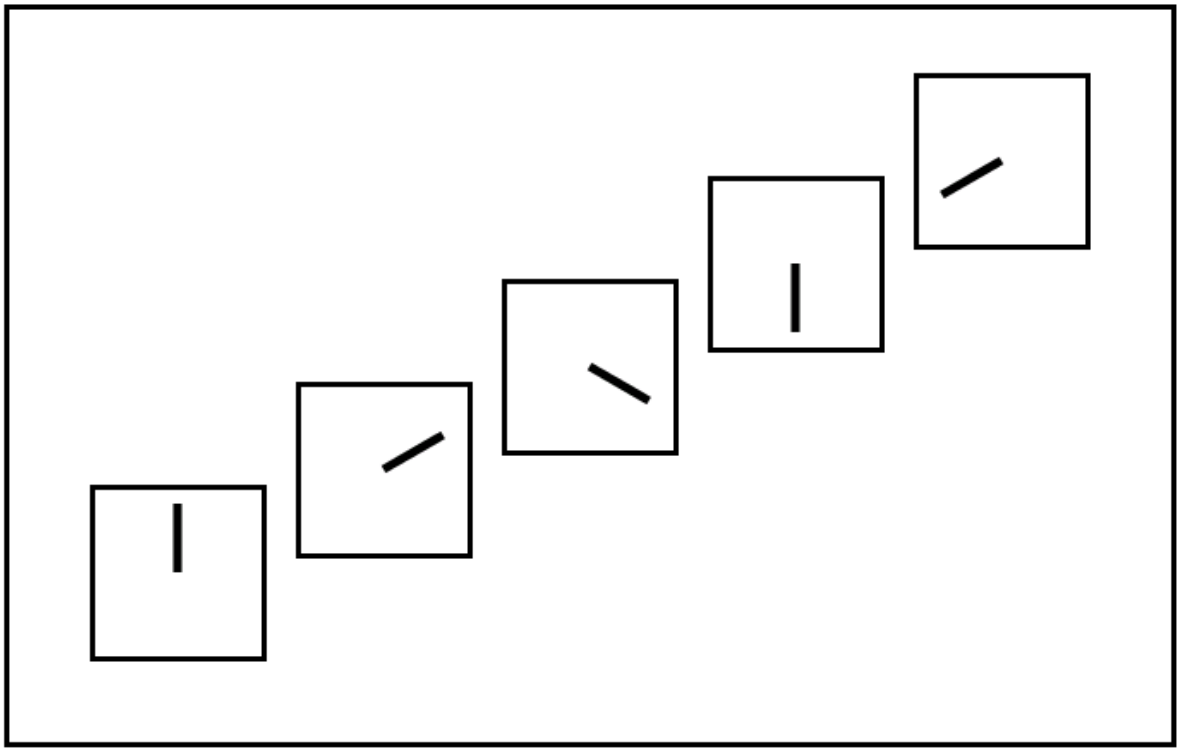
Bewegte Objekte

Strategie 1: Erzeuge das Objekt im Ursprung und führe Buch über die während der Bewegung akkumulierten Transformationen. Wende jedesmal die in geeigneter Reihenfolge akkumulierten Transformationen auf das Objekt im Ursprung an.

Strategie 2: Führe Buch über die Position (Position und Orientierung, evtl. auch Skalierung) des Objektes und wende die Transformationen entsprechend einzeln an.

Beispiel für Strategie 1

sich bewegende Uhr:



Beispiel für Strategie 1

Bewegung der Uhr in jedem Einzelschritt:

$$T_{\text{Uhr,Schritt}} = T(2,1)$$

Rotation des Sekundenzeigers in jedem Einzelschritt:

$$T_{\text{Zeiger,Schritt}} = R(-\pi/180)$$

Beispiel für Strategie 1

$$T^{(\text{neu})}_{\text{Uhr,gesamt}} = T_{\text{Uhr,Schritt}} \cdot T^{(\text{alt})}_{\text{Uhr,gesamt}}$$

$$T^{(\text{neu})}_{\text{Zeiger,Gesamtrotation}} = T_{\text{Zeiger,Schritt}} \cdot T^{(\text{alt})}_{\text{Zeiger,Gesamtrotation}}$$

$$T_{\text{Zeiger,gesamt}} = T_{\text{Uhr,gesamt}} \cdot T_{\text{Zeiger,Gesamtrotation}}$$

Interpolatoren

Ziel: kontinuierlicher Übergang von einem Anfangs- in einen Endzustand

einfachster Fall: Übergang von Position $\mathbf{p}_0$ nach $\mathbf{p}_1$

Punkte $\mathbf{p}_\alpha$ auf der Verbindungsstrecke ergeben sich durch Konvexkombinationen:

$$\mathbf{p}_\alpha = (1-\alpha)\cdot\mathbf{p}_0 + \alpha\cdot\mathbf{p}_1, \qquad \alpha \in [0,1].$$

Interpolation von Transformationen

Konvexkombination zweier affiner Transformationen, die durch die Matrizen M_0 und M_1 gegeben sind:

$$M_\alpha \;=\; (1-\alpha)\cdot M_0 + \alpha \cdot M_1, \qquad \alpha \in [0,1].$$

Anwendung: Umwandlung zweier Objekte ineinander, die durch die Transformationen M_0 und M_1 aus dem selben Grundobjekt entstanden sind.

Interpolation von Transformationen

Verwende zur Berechnung der Konvexkombinationen die den beiden Transformationen
$M_0 =$`initialTransform` und
$M_1 =$`finalTransform` zugeordneten Matrizen.

```
double[] initialMatrix = new double[6];
initialTransform.getMatrix(initialMatrix);

double[] finalMatrix = new double[6];
finalTransform.getMatrix(finalMatrix);
```

(vgl. `ConvexCombTransforms.java`)

Einfache Objektinterpolation

Zwei Objekte S und S' werden durch jeweils n Punkte $(x_1, y_1), \ldots, (x_n, y_n)$ bzw. $(x'_1, y'_1), \ldots, (x'_n, y'_n)$ und Verbindungslinien (Geradenstücke, quadratische oder kubische Kurven) festgelegt, die diese Punkte verwenden.
Dabei tauchen in beiden Objekten jeweils korrespondierende Verbindungslinien auf, d.h., wenn das Objekt S beispielsweise die quadratische Kurve beinhaltet, die durch die Punkte (x_1, y_1), (x_3, y_3) und (x_8, y_8) definiert wird, dann beinhaltet das Objekt S' die quadratische Kurve, die durch die Punkte (x'_1, y'_1), (x'_3, y'_3) und (x'_8, y'_8) definiert wird.

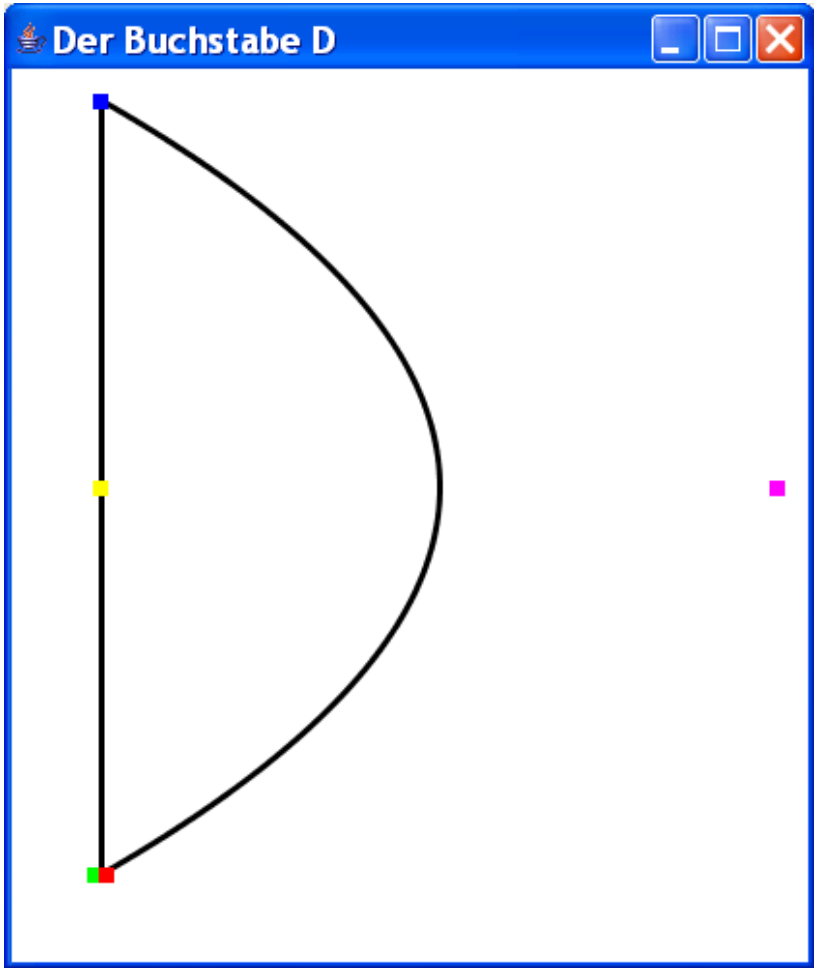
Einfache Objektinterpolation

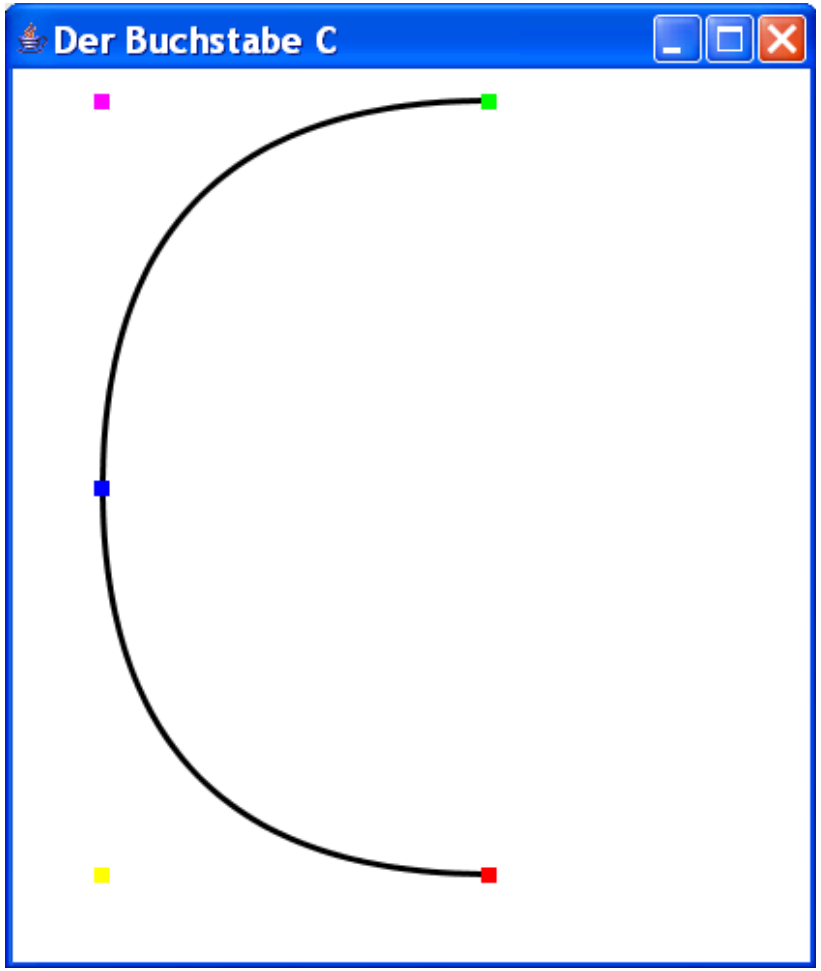
Beispiel: Die Objekte S und S' sind die Buchstaben D bzw. C, die durch die Punkte

$$
\begin{array}{lcllcl}
(x_1, y_1) & = & (50, 50) & (x'_1, y'_1) & = & (50, 250) \\
(x_2, y_2) & = & (50, 450) & (x'_2, y'_2) & = & (250, 50) \\
(x_3, y_3) & = & (400, 250) & (x'_3, y'_3) & = & (50, 50) \\
(x_4, y_4) & = & (50, 450) & (x'_4, y'_4) & = & (250, 450) \\
(x_5, y_5) & = & (50, 250) & (x'_5, y'_5) & = & (50, 450)
\end{array}
$$

und jeweils zwei quadratische Kurven, die jeweils den 1.,3. und 2. bzw. 1.,5. und 4. Punkt verwenden.

Einfache Objektinterpolation





Einfaches Objektmorphing

Für den Zwischenschritt $\alpha \in [0,1]$ werden die Konvexkombinationen

$$(\tilde{x}_i, \tilde{y}_i) \;=\; (1-\alpha)\cdot(x_i, y_i) + \alpha \cdot (x'_i, y'_i)$$

der Punkte (x_i, y_i) und (x'_i, y'_i) ($i = 1, \ldots, n$) berechnet und die entsprechenden Verbindungslinien mit den Punkten $(\tilde{x}_i, \tilde{y}_i)$ gezeichnet.

(vgl. `DToCMorphing.java`)

Rundungsfehlerbeispiel

Ein Sekundenzeiger mit einer Länge von 100 Pixeln rotiert um den Koordinatenursprung iterativ in Einzelschritten von jeweils $6°$.

Anwendung auf die Pixelwerte (Integer-Arithmetik):
Resultat nach einer Minute: $(95, -2)$

Rundungsfehlerbeispiel

Zeit	x	y
	double	
1 Minute	99.9999999999973	−4.8572257327350600E-14
2 Minuten	99.9999999999939	−9.2981178312356860E-14
3 Minuten	99.9999999999906	−1.3739009929736312E-13
4 Minuten	99.9999999999876	−1.4571677198205180E-13
5 Minuten	99.9999999999857	−2.2204460492503130E-13
6 Minuten	99.9999999999829	−2.9143354396410360E-13
7 Minuten	99.9999999999803	−3.1641356201816960E-13
8 Minuten	99.9999999999771	−3.7331249203020890E-13
9 Minuten	99.9999999999747	−4.2604808569990380E-13
10 Minuten	99.9999999999715	−4.5657921887709560E-13
8 Stunden	99.9999999986587	−2.9524993561125257E-11

Rundungsfehlerbeispiel

Zeit	x	y
	`float`	
1 Minute	100.00008	−1.1175871E-5
2 Minuten	100.00020	−1.4901161E-5
3 Minuten	100.00032	−1.8626451E-5
4 Minuten	100.00044	−1.1920929E-5
5 Minuten	100.00056	−8.9406970E-6
6 Minuten	100.00068	−3.1292439E-5
7 Minuten	100.00085	−5.3644180E-5
8 Minuten	100.00100	−7.2270630E-5
9 Minuten	100.00108	−8.0466270E-5
10 Minuten	100.00113	−8.4191560E-5
8 Stunden	100.00328	−1.9669533E-4