B2. 2D-Computergrafik mit Java

B2.1 Grundbegriffe der 2D-Computergrafik
B2.2 Einführung in das Grafik-API "Java 2D"
B2.3 Eigenschaften von Grafik-Objekten
B2.4 Integration von 2D-Grafik in Programmoberflächen
B2.5 Wichtige Grafik-Operationen

Literatur:

J. Knudsen, Java 2D Graphics, O'Reilly 1999
http://java.sun.com/products/java-media/2D/

Ausfüllen von Formen

- Prinzipielle Varianten für die Füllung einer Form (**`paint`**):
  - Massive Farben *(solid colors)*
  - Verläufe *(gradient paints)*
    - » Aus Punkt- und Farb-Parametern berechnet
  - Texturen *(textures)*
    - » Auf der Basis eines gegebenen Bildes
  - Selbstdefinierte Füllungen (**`Paint`**-Interface implementieren)
- Die aktuelle Füllungsregel ist Bestandteil des internen Zustands eines **`Graphics2D`**-Objektes
  - Setzen mit **`setPaint(Paint p)`**
  - **`Paint`** ist formal eine Schnittstelle, die z.B. von **`Color`** implementiert wird

Farbverläufe

- Ein Farbverlauf *(gradient paint)* wird festgelegt durch
  - Eine Linie, gegeben durch Startpunkt und Endpunkt
    - » definiert Richtungsvektor des Verlaufs sowie Punkte festgelegter Farbwerte
  - Zwei Farbwerte für Start- und Endpunkt
  - Information, ob der Verlauf *zyklisch* oder *azyklisch* ist
    - » Zyklischer Verlauf wiederholt sich im Bereich ausserhalb der Linie
    - » Azyklischer Verlauf erhält Randfarben im Bereich ausserhalb der Linie

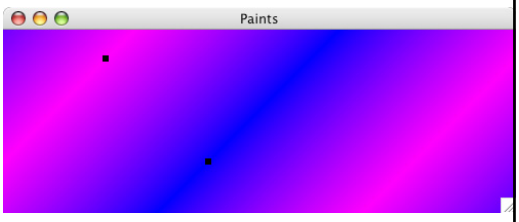
- Java2D:

```
public GradientPaint
  (float x1, float y1, Color color1,
   float x2, float y2, Color color2,
   boolean cyclic)
```

  - Alternative Varianten, z.B. mit `Point2D`-Objekten anstelle der Koordinaten
  - `GradientPaint` implementiert die `Paint`-Schnittstelle
  - Standardtyp von Farbverläufen ist azyklisch

Beispiel: Zyklischer Verlauf

```
private void drawPoint(Graphics2D g2, Point2D p) {
    double ptsize = 3.0;
    double x = p.getX();
    double y = p.getY();
    g2.setPaint(Color.black);
    g2.fill(
      new Rectangle2D.Double(
        x-ptsize, y-ptsize,
        2*ptsize, 2*ptsize));
}
```




```
public void paint(Graphics g) {
   Graphics2D g2 = (Graphics2D)g;

   Point2D p1 = new Point2D.Double(100, 50);
   Point2D p2 = new Point2D.Double(200, 150);

   g2.setPaint(new GradientPaint
     (p1, Color.magenta, p2, Color.blue, true));
   g2.fill(new Rectangle2D.Double(0, 0, 500, 200));
   drawPoint(g2, p1);
   drawPoint(g2, p2);
  }
}
```

Texturen

- Eine *Textur* wird festgelegt durch
  - ein Rasterbild **`(BufferedImage)`**
  - eine Kachelgröße *(anchor rectangle)* **`(Rectangle2D)`**
- Die Füllung entsteht in folgenden Schritten:
  - Rasterbild wird auf Kachelgröße skaliert
  - Zu füllende Fläche wird mit Kacheln aufgefüllt
  - Jede Kachel enthält das gleiche Bild (aus dem Rasterbild)
- Java2D:

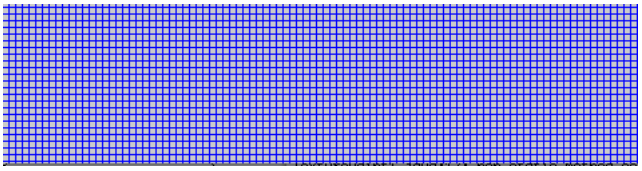
```
public TexturePaint
  (BufferedImage txtr, Rectangle2D anchor)
```

  - **`TexturePaint`** implementiert die **`Paint`**-Schnittstelle

Programmtechnische Erzeugung von Texturen

- Ein Rasterbild kann auch durch Programmierung erzeugt werden
- Java2D:
  - **`Graphics2D createGraphics()`** aus der Klasse **`BufferedImage`** erzeugt ein beschreibbares Grafikobjekt für ein (normalerweise vorher leeres) Rasterbild

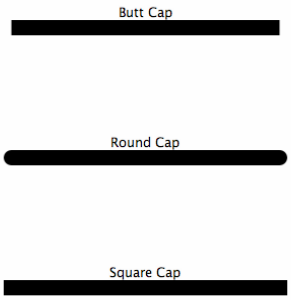
```
BufferedImage bi =
  new BufferedImage(5, 5, BufferedImage.TYPE_INT_RGB);
Graphics2D big = bi.createGraphics();
big.setColor(Color.blue);
big.fill(new Rectangle2D.Double(0, 0, 5, 5));
big.setColor(Color.lightGray);
big.fill(new Rectangle2D.Double(1, 1, 4, 4));
```



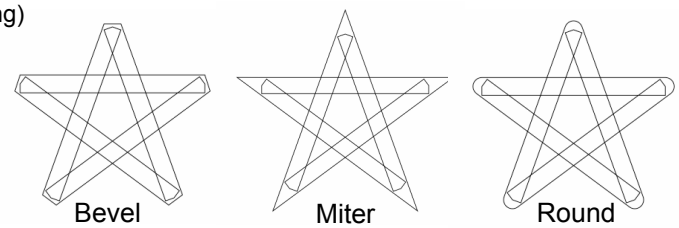
Linieneigenschaften

- Linien werden verwendet:
  - als eigenständige Elemente
  - um die Konturen von Formen zu zeichnen (mit **`draw()`**)
- Eine Linie stellt selbst wieder eine Form dar.
- Eigenschaften von Linien
  - Linienstärke
  - Füllung (durch **`setPaint()`** bestimmt)
  - Strichelung
  - Endstil
  - Verbindungsstil
- Strichelung *(dash)*:
  - *dash array*: Feld von Gleitkommazahlen für die Länge der Strichteile
    - » Gerade Position: sichtbar
    - » Ungerade Position: unsichtbar
  - *dash phase*: Versetzung des Beginns der Strichelung

Linienenden, Verbindungen

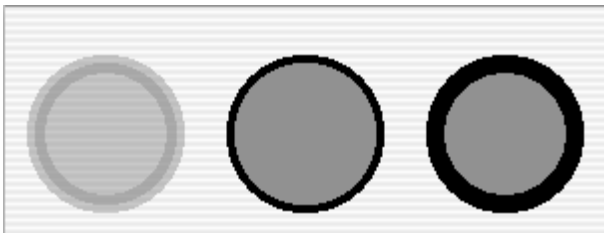


- Von Java2D unterstützte Stile für Linienenden *(line caps)*:
  - *BUTT*
  - *ROUND*
  - *SQUARE*
- Von Java2D unterstützte Stile für Linienverbindungen *(line joins)*:
  - *BEVEL* (Schrägschnitt)
  - *MITER* (Gehrung)
  - *ROUND*



Kontur und Füllung

- Die Linie der Kontur überlappt mit der Füllung der Figur.
- Werden verschiedene Füllungen bei **`draw()`** und **`fill()`** verwendet, dann hängt das Ergebnis von der Reihenfolge der beiden Schritte ab.



B2. 2D-Computergrafik mit Java

Kombination von Swing und Java2D

- Zeichenfläche innerhalb eines Swing-basierten Fensters (**`JFrame`**):
  - **`JPanel`**-Teilkomponente
- Zeichenvorgang im **`JPanel`**:
  - Überdefinieren von **`paintComponent()`**
    - » Schützt andere Bestandteile der Komponente
    - » **`paint()`** auf Swing-Komponenten ruft **`paintComponent()`** auf sowie weitere Methoden zum Zeichen des Rahmens und von Unterkomponenten
  - Tip Nr. 1: Die Grösse des Panels wird von Layoutmanagern mittels **`getPreferredSize()`** abgefragt, deshalb unbedingt diese Methode überdefinieren (**`setSize()`** ist wirkungslos!)
  - Tip Nr. 2: Bei Look-and-feel-abhängigen Komponenten wie **`JPanel`** sollte man beim Neuzeichnen zuerst **`super.paintComponent()`** aufrufen. Nur dann wird auf dem richtigen Hintergrund gezeichnet.
- Beispiel:
  - Programm zur Änderung der Farbe mittels Standard-Farbwahldialog

Beispiel: Farbwahldialog (1)

```
class ColorChooserFrame extends JFrame {

    Color c  = Color.red;
    DrawPanel drawPanel = new DrawPanel();

    class DrawPanel extends JPanel {

      public DrawPanel() {
        setBackground(Color.lightGray);
      }

      public Dimension getPreferredSize() {
        return new Dimension(200, 100);
      }

      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        g2.setPaint(c);
        g2.fill(new Ellipse2D.Double(50, 20, 100, 50));
      }
    }

  ...
```

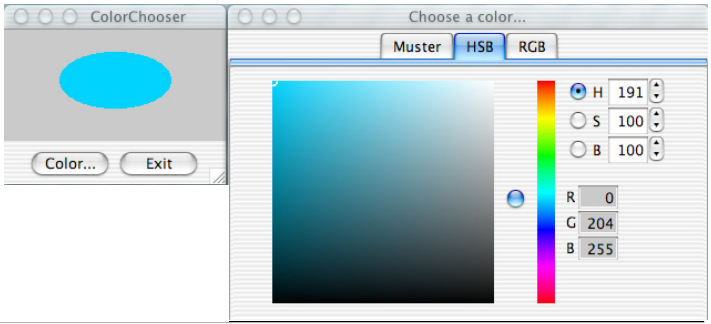
Beispiel: Farbwahldialog (2)

```
...
public ColorChooserFrame() {
    setTitle("ColorChooser Demo");
    getContentPane().add(drawPanel,BorderLayout.NORTH);
    JPanel buttonPanel = new JPanel();
    getContentPane().add(buttonPanel,BorderLayout.SOUTH);
    JButton chooseButton = new JButton("Color...");
    chooseButton.addActionListener(new ActionListener(){
      public void actionPerformed (ActionEvent event) {
          Color cnew = JColorChooser.showDialog
            (drawPanel, "Choose a color...", c);
          if (cnew != null) {
            c = cnew;
            drawPanel.repaint();
          }
       }
    });
    buttonPanel.add(chooseButton);

    JButton exitButton = new JButton("Exit");
    exitButton.addActionListener(new ActionListener(){
      public void actionPerformed (ActionEvent event) {
          System.exit(0);
       }
    });
    buttonPanel.add(exitButton);                    ...
```


Beispiel: Farbwahldialog (3)

```
...     setDefaultExitOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
}
class ColorChooser {
  public static void main (String[] argv) {
    ColorChooserFrame cf = new ColorChooserFrame();
  }
}
```



B2. 2D-Computergrafik mit Java

Affine Transformationen

- Eine *affine Transformation* ist eine Abbildung eines Koordinatenraums in einen Koordinatenraum, bei der Parallele erhalten bleiben.
- Mathematisch gesehen, ist eine affine Transformation eine lineare Transformation, d.h. Matrixmultiplikation (und Addition eines Vektors):

$x' = a\,x + c\,y + t_x$
$y' = b\,x + d\,y + t_y$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & c & t_x \\ b & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

  - Affine Transformationen können zu komplexeren affinen Transformationen verkettet werden.

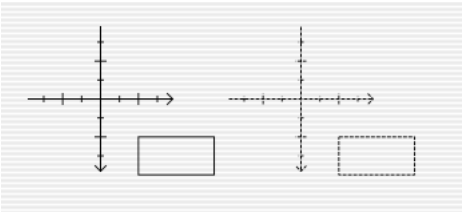
- Java 2D:
  - Jedes **`Graphics2D`**-Objekt erfährt beim Rendering eine affine Transformation aus den Benutzer- in die Gerätekoordinaten.
  - Die Standard-Transformation des **`Graphics2D`**-Objekts kann per Programm modifiziert werden (Methode **`transform()`**).
  - Man kann temporär (ohne die Standard-Transformation zu verändern) eine Transformation beim Anzeigen eines Bildes (**`Image`**) anwenden.

Verschiebung *(translation)*

- Eine *Verschiebung (translation)* bewegt den Ursprungspunkt des Koordinatensystems an eine neue Stelle.
  Alle Grafikelemente werden entsprechend der neuen Lage angezeigt.
- Java 2D:
  ```
  public static AffineTransform
    getTranslateInstance(double tx, double ty)
  ```
  In `AffineTransform: translate(double tx, double ty)`
- Beispiel:
  - Anzeige vor der Transformation und (gestrichelt) nach der Transformation

```
AffineTransform at = AffineTransform.getTranslateInstance(150, 0);
g2.transform(at);
```

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

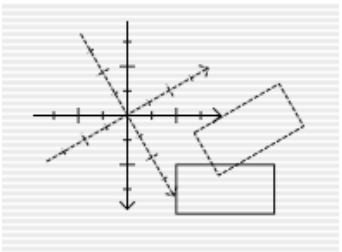


Drehung *(rotation)*

- Eine *Drehung (rotation)* dreht alle Punkte des Zeichenbereichs um den Ursprung entsprechend einem gegebenen Winkel.
  - Der Drehwinkel wird im Bogenmass *(radian)* angegeben.
- Java 2D:
  ```
  public static AffineTransform
    getRotateInstance(double theta)
  ```
  In `AffineTransform: rotate(double theta)`
- Beispiel:

```
AffineTransform at = AffineTransform.getRotateInstance(-Math.PI/6);
g2.transform(at);
```

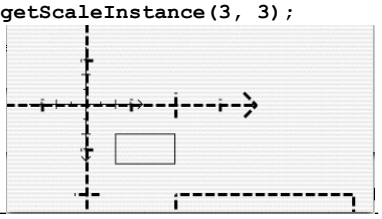
$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\vartheta) & -\sin(\vartheta) & 0 \\ \sin(\vartheta) & \cos(\vartheta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$



Skalierung *(scaling)*

- Eine *Skalierung (scaling)* bewirkt eine maßstäbliche Vergrösserung oder Verkleinerung der Darstellung.
- Skalierung verändet die Lage aller Punkte ausser dem Ursprung des Koordinatensystems.
- Auch Linienstärken werden mitskaliert.
- Java 2D:
  ```
  public static AffineTransform
    getScaleInstance(double sx, double sy)
  ```
  In `AffineTransform: scale(double sx, double sy)`
- Beispiel:
  ```
  AffineTransform at = AffineTransform.getScaleInstance(3, 3);
  g2.transform(at);
  ```

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

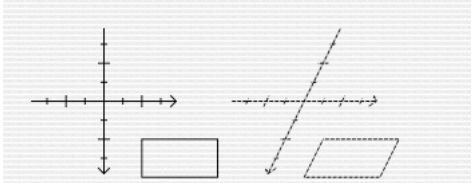


Verzerrung *(shearing)*

- Eine *Verzerrung oder Scherung (shearing)* dehnt oder staucht den Koordinatenraum entlang einer bestimmten Richtung.
- Java 2D:
  ```
  public static AffineTransform
    getShearInstance(double shx, double shy)
  ```
  In `AffineTransform: shear(double shx, double shy)`
- Beispiel:
  - Anzeige vor der Transformation und (gestrichelt) nach der Transformation

  ```
  AffineTransform at = AffineTransform.getTranslateInstance(150, 0);
  at.shear(-0.5, 0);
  g2.transform(at);
  ```

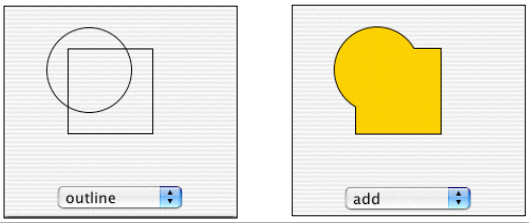
$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & shx & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$



(Hier mit zusätzlicher Translation gezeigt)

Formen kombinieren (Constructive Area Geometry)

- Ein *Gebiet (area)* der Zeichenfläche kann aus Formen und anderen Gebieten berechnet werden.
- Die wichtigsten Operationen auf Gebieten (gemäss Java 2D):
  - Klasse **`java.awt.geom.Area (implements Shape)`**
  - Konstruktor **`Area(Shape s)`**
  - Hinzufügen (Vereinigung): **`void add(Area rhs)`**
  - Wegnehmen (Differenz): **`void subtract(Area rhs)`**
  - Schneiden: **`void intersect(Area rhs)`**
  - Exklusives Oder: **`void exclusiveOr(Area rhs)`**



Zusammensetzung von Bildern *(Compositing)*

- Unter *Compositing* versteht man die Zusammensetzung (Überlagerung) von mehreren Bildern zu einem Gesamtbild
- Beim Zeichnen grafischer Objekte:
  - Wie wirkt sich ein evtl. vorhandener Hintergrund aus?
  - Wenn zwei Objekte überlappen, was wird angezeigt?
    - » Das zuletzt gezeichnete?
    - » Eine Mischung der beiden Objekte?
- Wichtige Unterscheidung:
  - Undurchsichtige *(opaque)* Grafikelemente
  - Durchsichtige *(transparent)* Grafikelemente
    - » Verwendung eines *Alpha-Kanals* (nicht auf allen Ausgabemedien unterstützt)
- Java 2D:
  - Klasse **`java.awt.AlphaComposite`**
  - Konstante für Kompositionsregeln, z.B. **`AlphaComposite.SrcOver`**
  - Methode **`setComposite()`** in Klasse **`Graphics2D`**

Porter-Duff Regeln

- Java 2D orientiert sich an:
  - T. Porter and T. Duff, "Compositing Digital Images", SIGGRAPH 84
- Definition der Pixel (Java SDK documentation):
  - Cs = a color component of the source pixel.
  - Cd = a color component of the destination pixel.
  - As = alpha component of the source pixel.
  - Ad = alpha component of the destination pixel.
  - Fs = fraction of the source pixel that contributes to the output.
  - Fd = fraction of the input destination pixel that contributes to the output.
  - Cr = a color component of the result
  - Ar = alpha component of the result

`Cr = Cs*Fs + Cd*Fd` `Ar = As*Fs + Ad*Fd`

- Fs und Fd sind in verschiedenen Regeln spezifiziert (sh. nächste Folie)
- Source und destination Pixel: vormultipliziert mit Alphawert
- Siehe auch: http://www.ibm.com/developerworks/java/library/j-mer0918/

Wichtige Compositing-Rules

- SrcOver:
  - Neu gezeichnete Inhalte *(source)* überlagern existierende *(destination)*
- DstOver:
  - *Source* scheint unter *destination* zu liegen
- SrcIn:
  - *Source* wird mit der Transparenz der *destination* gezeichnet
  - Wo keine *destination* vorhanden, wird *source* nicht dargestellt
- SrcOut:
  - *Source* wird mit der inversen Transparenz der *destination* gezeichnet
  - Wo *destination* vorhanden, wird *source* nicht dargestellt

SRC_OVER

`public static final int SRC_OVER`

The source is composited over the destination (Porter-Duff Source Over Destination rule).

$F_s = 1$ and $F_d = (1\text{-}A_s)$, thus:

$$A_r = A_s + A_d*(1-A_s)$$
$$C_r = C_s + C_d*(1-A_s)$$

See Also:
Constant Field Values

DST_OVER

`public static final int DST_OVER`

The destination is composited over the source and the result replaces the destination (Porter-Duff Destination Over Source rule).

$F_s = (1\text{-}A_d)$ and $F_d = 1$, thus:

$$A_r = A_s*(1-A_d) + A_d$$
$$C_r = C_s*(1-A_d) + C_d$$

See Also:
Constant Field Values

SRC_IN

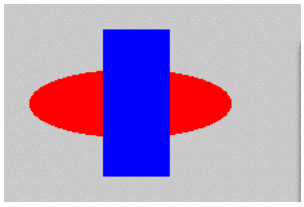
`public static final int SRC_IN`

The part of the source lying inside of the destination replaces the destination (Porter-Duff Source In Destination rule).

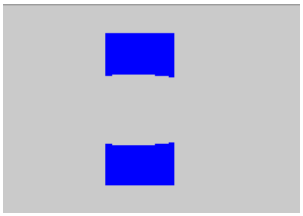
$F_s = A_d$ and $F_d = 0$, thus:

$$A_r = A_s*A_d$$
$$C_r = C_s*A_d$$

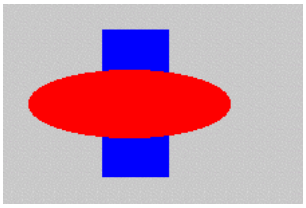
Beispiele: Compositing bei opaken Objekten



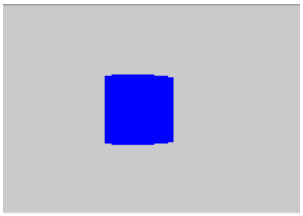
Src-Over



Src-Out

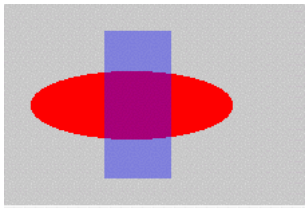


Dst-Over

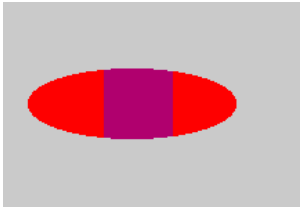


Src-In

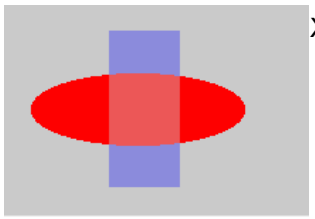
Beispiele: Alpha-Compositing



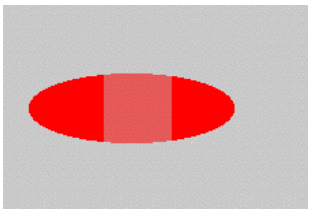
Src-Over



Src-Atop



Xor



Dst-Out

Programmiertechnik: Off-Screen Images

```
// Create off-screen image for destination
 BufferedImage dest =
   new BufferedImage(200, 150, BufferedImage.TYPE_INT_ARGB);
 Graphics2D  destG = dest.createGraphics();
// Paint destination element (red ellipse)
 destG.setPaint(Color.red);
 destG.fill(new Ellipse2D.Double(20, 50, 150, 50));
// Create off-screen image for source
 BufferedImage source =
   new BufferedImage(200, 150, BufferedImage.TYPE_INT_ARGB);
 Graphics2D  sourceG = source.createGraphics();
// Paint source element (blue rectangle)
 sourceG.setPaint(Color.blue);
 sourceG.fill(new Rectangle2D.Double(75, 20, 50, 110));
// Obtain compositing rule from user
 String rule = (String)options.getSelectedItem();
 if (rule.equals("Src"))
  destG.setComposite(AlphaComposite.Src);…
//Compose source with destination
 destG.drawImage(source, 0, 0, null);
// Display on screen
 g2.drawImage(dest, 0, 0, null);
```

"Off-screen image" für Überlagerung verwenden: Kann Alphawerte darstellen (Bildschirmausgabe nicht!)

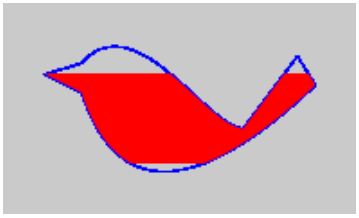
Clipping

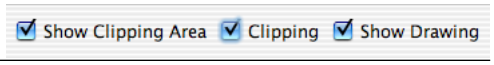
- *Beschneiden (clipping)* beschränkt die Darstellung des erzeugten Bildes auf einen Teilbereich der Zeichenfläche.
  - Clipping ist in allen Grafik-Bibliotheken implementiert, da für die Darstellung in Fenstern dringend benötigt.
  - Einfache Bibliotheken unterstützen nur Clipping mit einfachen Formen, z.B. Rechtecken; in Java2D: Beliebige Form (GeneralPath)
- Java2D:
  - *Current clipping shape* ist Bestandteil des Zustands jedes `Graphics2D`-Objektes
  - Methoden zum Verändern der *clipping shape:*

```
public void clip(Shape s)
  // Durchschnitt mit vorhandener clipping shape
public Shape getClip()
public void setClip(Shape s)
  // Definiert clipping shape neu
```

Beispiel: Clipping

```
private JCheckBox optShowCA =
  new JCheckBox("Show Clipping Area", false);
private JCheckBox optClip = new JCheckBox("Clipping", false);
private JCheckBox optDraw = new JCheckBox("Show Drawing", true);
class DrawPanel extends JPanel {
  ...
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    Rectangle2D r = new Rectangle2D.Double(20, 50, 180, 50);
    GeneralPath p = new GeneralPath();
    ...
    if (optShowCA.isSelected()) {
      g2.setPaint(Color.blue);
      g2.draw(p);
    };
    if (optClip.isSelected()) {
      g2.clip(p);
    };
    if (optDraw.isSelected()) {
      g2.setPaint(Color.red);
      g2.fill(r);
    };
  }
}
```





Rendering Hints

- *Rendering Hints:* Globale Einstellungen, die das Verhalten der Rendering Engine bestimmen, z.B.:
  - Antialiasing
  - Text-Antialiasing
  - Dithering
  - Color Rendering
  - ...
- Der Programmierer kann festlegen, welche Einstellung er bevorzugt, die letztliche Auswahl bleibt dem Laufzeitsystem überlassen.
- Beispiel:

```
g2.setRenderingHint(
  RenderingHints.KEY_DITHERING,
  RenderingHints.VALUE_DITHER_DISABLE);
```

Text in Java2D

- Option 1:
  - Swing-Komponenten (JTextField, JTextArea, JEditorPane)
  - Fertige Bausteine z.B. für Benutzungsoberflächen
- Option 2:
  - Methode **`drawString()`** unter automatischer Berücksichtigung des Font-Attributs im **`Graphics2D`**-Objekt (Klasse **`java.awt.Font`**)
  - Interne Konversion in Glyphen **`(Shape)`** in der Rendering-Pipeline
  - Verwendung aller Spezialeffekte für Formen in Java2D,
    - » z.B. Füllungen (incl. Gradient, Textur)
    - » z.B. Affine Transformationen (incl. Drehung, Verzerrung)
- Option 3:
  - Detailliertes Layout mit **`java.awt.font.TextLayout`**
  - Sehr spezielle Features wie bidirektionaler Text, genaue Metriksteuerung, kontrollierte Zeichenselektion beim Klicken
- Option 4:
  - Direkte Manipulation von Glyphen als Java2D-**`Shape`**s