Java 2D

(Theorie und Beispiele)

1. Übersicht Java2D

- Allgemeiner Überblick Java2D
- Definieren des Graphics-Kontext + Rendering (Grafikeditor)
- Geometrische Operationen
- Einführung Java2D - Text
- Einführung Java2D - Imaging
- Einführung Java2D - Color
- Einführung Java2D - Print

1.1 Java 2D - News

- mit der Graphics-Klasse früherer AWT-Versionen konnte man „nur“ einfache Geometrische Formen darstellen (java.awt-package) wie Rechtecke,Ovale, Polygone
- Mit Graphics2D kann man buchstäblich alle geometrische Formen erzeugen

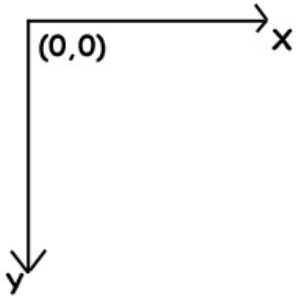
1.2 Graphics2D

- Geometrische Formen: Implementationen des Shape-Interfaces z.B. Rectangle2D,CubicCurve,Arc2D
- Fill- und Penstyles werden durch Implementationen des Paint- bzw. Stroke-Interfaces zur Verfügung gestellt
- Affine Transformationen → Lineare Transformation von 2D-Koordinaten
- Variable Clipping-Regions: Regionen, die gerendert werden sollen
- Color Composition: Implementation des Composite-Interfaces z.B. AlphaComposite
- Fonts: Definition eigener Fonts (Glyphen) möglich

1.3 Rendering

Bed.: Umsetzung von Daten im User-Space auf den Device-Space

KoordinatenSystem:



User-Space: Ein device-unabhängiges Koordinatensystem, auf dem die Java-Applikationen ausschließlich beruhen.

Device-Space: Darstellung auf einem bestimmte Drucker oder Bildschirm → meist anders als UserSpace

1.3.1 Klassen GraphicsEnvironment, GraphicsDevice, GraphicsConfiguration

GE: Enthält eine Beschreibung einer Ansammlung der auf einer Plattform bekannten rendering-devices. Enthält außerdem Infos über alle auf der spez. Plattform bekannten Fonts.

GD: Beschreibt einen bestimmten rendering-device (best. Bildschirm)

GC:Beschreibt alle möglichen Betriebsmodi eines rendering-device (z.B.: 640x480x256,800x600x256)

Ein GE kann mehrere GDs enthalten und ein GD mehrere GCs.

1.3.2 Rendering-Model

Altes Rendering-Model existiert weiterhin (rendering mit einfachem Graphics-Object in der Paint()-Methode weiterhin möglich)

- Setup des „Graphics-Kontext“ + Aufruf einer Rendering-Methode

Graphics2D erbt nur von Graphics und stellt weiter Funktionalitäten zur Verfügung

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

1.4 Abwärtskompatibilität

Um die Abwärtskompatibilität zu gewährleisten wurden bereits existierende Klassen und Interfaces nicht verändert.

Methoden:

1. Neue Klassen erweitern alte Klassen (Vererbung)

2. Generalisieren der alten Klasse

1.4.1 Generalisieren von Klassen

1) Eine oder mehrere „Parent-Classes“ werden in die Klassenhierarchie eingefügt und die jeweiligen Klassen upgedatet.

Zweck:

- Hinzufügen von allgemein zu implementierenden Methoden

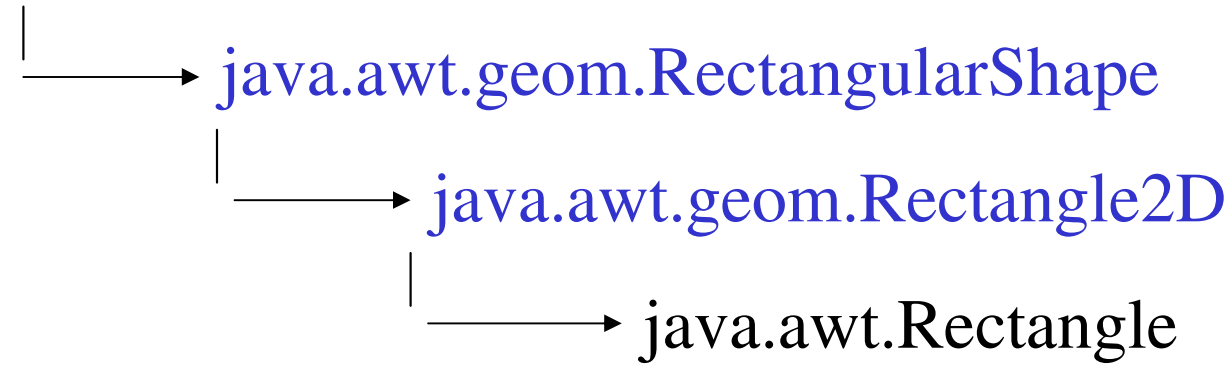
- Hinzufügen von InstanzDaten (neue Attribute)

2) Eine oder mehrere Interface Implementationen werden hinzugefügt.

1.4.2 Beispiel für Generalisierung

In JDK 1.1 erbt AWT Rectangle einfach von Object.

In der Java 2D API:



Außerdem implementiert Rectangle nun das Serializable- und Shape-Interface.

1.5 Plattformunabhängigkeit

Um PU zu gewährleisten trifft Java 2D API keine Annahmen über:

- Auflösung,
- ColorSpace,
- und ColorModel des Rendering Devices

- Format von Images

1.6 Übersicht Java 2D API Packages

- java.awt
- java.awt.geom
- java.awt.font
- java.awt.color
- java.awt.image
- java.awt.image.renderable
- java.awt.print

2. Rendern mit Graphics2D

Setup des Graphics-Kontext→Aufruf einer Render-Methode (draw,fill)→Rendering

Graphics2D-Kontext:

Umfaßt alle Attribute die im Rendering-Prozeß eines graphischen Objektes vom Graphics2D-Object herangezogen werden. (z.B. StrokeWidth)

Setzen der Attribute mit „set-Methoden“

```
g2.setPaint(new GradientPaint(...));
```

2.1 Rendering Prozeß

Rendern einer einfachen geometrischen Form:

- Für die Umrisse (Path) wird das Stroke-Attribut heran-gezogen.
- Die Koordinaten des User-Spaces werden mit Hilfe des Transform-Attributs in die Koordinaten des spez. Device-Spaces umgewandelt
- Das Clip-Attribut wird herangezogen um den Bereich der „gerendert“ werden soll zu bestimmen
- Wenn nötig (extra Aufruf der fill-Methode) wir das innere des Shapes anhand der Infos im Paint- und Composite-Attribut gefüllt.

2.2 Kontrolle der Rendering-Qualität

Klasse RenderingHints

- Alpha Interpolation (default,quality,speed)
- Antialiasing (default,on,off)
- Color Rendering (default,quality,speed)
- Dithering (default,enable,disable)
- Interpolation (nearest-neighbour,bilinear,bicubic)
- Rendering (default,quality,speed)
- Text Antialiasing (default,on,off)
- Fractional Metrics (default,on,off)

2.3 Die Stroke-Attribute

Konstruktor:

public **BasicStroke**(float width, int cap,int join,
float miterlimit,float[] dash,
float dash_phase)

Methode im Graphics2D-Context:

public abstract void **setStroke** (Stroke s)

2.3.1 Basic-Stroke Parameter

width:	float-Wert spezifiziert die Breite des „Strichs“
cap:	Konstante, die den Stil am Ende einer Linie spezifiziert. (CAP_BUTT,CAP_ROUND,CAP_SQUARED)
join:	Konstante, die den Stil beim Aufeinandertreffen von Linien spezifiziert. (JOIN_BEVEL, JOIN_ROUND, JOIN_MITER)
miterlimit:	genau Spezifikation falls JOIN_MITER gewählt.
dash:	Array, das das Strichmuster spezifiziert
dash_phase:	Offset bei dem das Strichmuster beginnen soll

2.4 Die Fill-Attribute

Das Fill_Attribut wird im Graphics2D-Context von einem Paint-Objekt repräsentiert.

Wenn ein Shape oder Glyph gezeichnet wir (Graphics2D.draw), wird die im Paint-Objekt spezifizierte Farbe auf die Umrandung angewandt.

Wenn die Methode Graphics2d.fill() aufgerufen wird, dann wird die spezifizierte Farbe auf alle Pixel innerhalb des Shapes angewandt.

Hierbei gibt es in Java2D mehrere Möglichkeiten.

Color, GradientPaint, TexturePaint

2.4.1 Farbarten (1)

1) normales Color-Objekt

Methode für den Graphics2D-Kontext:

public void setColor(Color c)

2) GradientPaint

Konstruktor:

public **GradientPaint** (float x1, float y1,
Color color1, float x2,
float y2, Color color2)

2.4.1 Farbarten (2)

3) TexturePaint

Konstruktor:

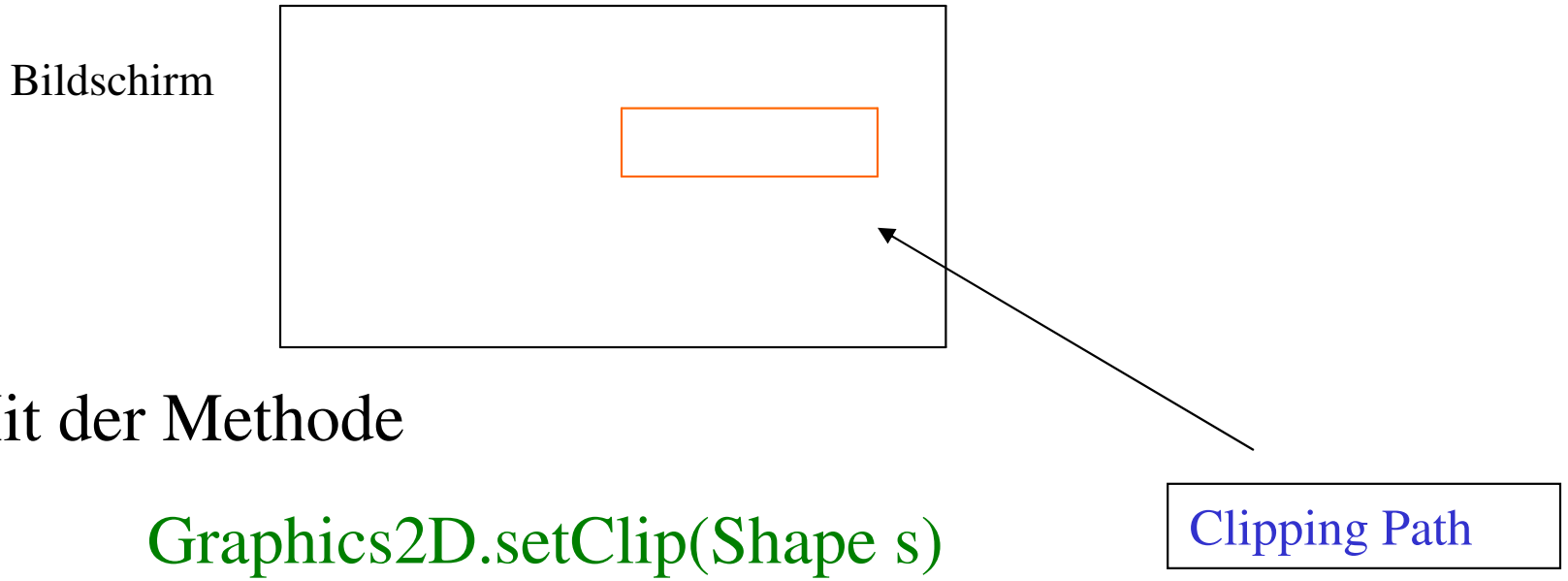
public **TexturePaint**(BufferedImage txtr,

Rectangle anchor)

GradientPaint und TexturePaint werden mit der Methode Graphics2D.setPaint() dem Graphics2D-Kontext hinzugefügt.

2.5 Clipping Path

Definieren eines Bereiches der „gerendert“ werden soll.



Mit der Methode

Graphics2D.setClip(Shape s)

wird der Clipping-Path festgelegt.

2.6 Affine Transformationen

Unter einer Affinen Transformation versteht man die Manipulation von 2D-Koordinaten (lineare Transformation), durch Matrizen-Multiplikation.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m00x + m01y + m02 \\ m10x + m11y + m12 \\ 1 \end{bmatrix}$$

Die neuen Koordinaten x‘ und y‘ entstehen durch Multiplikation von x und y mit einer 3*3 Matrize.

2.6.1 Arten von Transformationen (1)

(1) Translation (Verschiebung in X- und/oder Y-Richtung)

Befehl:

```
public static AffineTransform getTranslateInstance (double tx,double ty)
```

Multiplikations-Matrix:

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

tx= delta x

ty= delta y

2.6.1 Arten von Transformationen (2)

(2) Rotation (Rotieren des Shapes um eine bestimmten Winkel)

Befehl:

public static AffineTransform **getRotateInstance** (double theta, double x, double y)

M-M:

```
[cos(theta)  -sin(theta)  x-x*cos(theta)+y*sin(theta)]
[sin(theta)   cos(theta)  y-x*sin(theta)-y*cos(theta)]
[ 0              0              1                     ]
```

theta = Drehwinkel

x,y = Koordinaten des Ankerpunktes

2.6.1 Arten von Transformationen (3)

(3) Skalierung (Multiplikation aller X bzw. Y-Werte mit einem bestimmten Faktor)

Befehl:

public static AffineTransform **getScaleInstance** (double sx,double sy)

M-M:

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

sx= SkalierungsFaktor X-Richtung

sy= Skalierungsfaktor Y-Richtung

2.6.1 Arten von Transformationen (4)

(3) Verzerrung (Verschiebung der X- und Y-Werte um einen bestimmten Wert, der sich aus einem Faktor und dem jeweilig anderen Wert ergibt)

Befehl:

public static AffineTransform **getShearInstance** (double shx,double shy)

M-M:

[1	shx	0]
[shy	1	0]
[0	0	1]

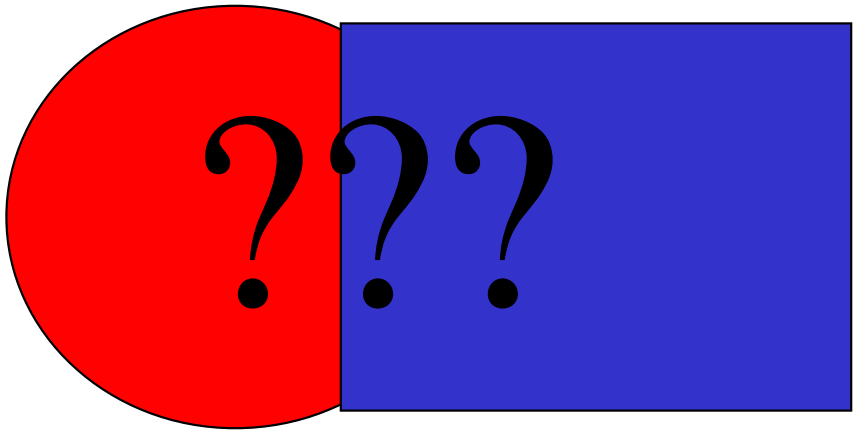
shx: Faktor, für shx *y (X-Verschiebung)

shy: Faktor, für shy*x (Y-Verschiebung)

2.7 Composite-Attribute

Problemstellung:

Zwei überlappende Graphik-Objekte, wie soll die Schnitt-fläche „gerendert“ werden?



2.7.1 Alpha-Composite

Die Festlegung, welche Farbe für sich überlappende Pixel gerendert werden soll nennt man „Compositing“. Dies wird in dem Composite-Attribut des Graphics2D-Kontexts festgelegt.

Methode:

```
AlphaComposite.getInstance(int rule,float alpha)
```

Mit Graphics2D.setComposite(AlphaComposite ac)

Wird das Objekt dem Graphics2D-Kontext hinzugefügt.

2.7.2 Parameter AlphaComposite

int rule: Spezifiziert den Compositing-Style

CLEAR,

DEST_IN, DEST_OVER, DEST_OUT

SRC, SRC_IN, SRC_OUT, SRC_OVER

float alpha: Spezifiziert den Grad der Transparenz

1.0 → Opaque

0.0 → Transparent

2.8 Geometrische Formen (1)

I) Rectangle2D

public **Rectangle2D.Double**(double x,double y, double w, double h)

II) RoundRectangle2D

public **RoundRectangle2D.Double** (double x, double y, double w, double h, double arcw, double arch)

III) Line2D

public **Line2D.Double**(double X1, double Y1,double X2,double Y2)

IV) GeneralPath

public **GeneralPath**(int rule)

2.8. Geometrische Formen (2)

V) Ellipse2D

public **Ellipse2D.Double** (double x,double y,double w,double h)

VI) CubicCurve2D

public **CubicCurve2D.Double** (double x1,double y1,double ctrlx1,
double ctrly1,double ctrlx2,double ctrly2,
double x2,double y2)

VII) Arc2D

public **Arc2D.Double** (double x,double y,double w,double h,
double start,double extent,int type)

VIII) QuadCurve2D

public **QuadCurve2D.Double** (double x1, double y1, double ctrlx,
double ctrly, double x2, double y2)

2.8 Konstruieren komplexere Shapes aus geometrischen Grundformen

Constructive area geometry (CAG):

Zusammensetzen komplexerer geometrischer Formen aus einfacheren Formen mit Hilfe bool‘scher Operationen.

Klasse Area ist ein spezieller Typ von Shape, der bool‘sche Operationen zulässt.

Eine Area läßt sich aus jedem Shape (z.B. Rechteck) konstruieren.

→ Beispiel-Applet (face.java)

3.0 Java2D - Text

TextLayout mit Java2D:

• Für die meisten Anwendungen reicht eine JTextComponent (JTextArea, JTextField) aus.

→ „Using the JFC/Swing-Packages“ (Java-Tutorial)

•Wenn man eigene Editier-Routinen entwerfen will, steht einem die Klasse TextLayout zur Verfügung

→ Java2D-Text (auf www.sun.com)

•Zusätzlich kann man eigene Fonts entwerfen indem man mit Glyph-Vektoren arbeitet. (auf www.sun.com)

3.1 Fonts

Ein Font besteht aus vielen Glyphen (Implementationen das Shape-Interfaces).

Jede Version eines Fonts (bold,italic) wird „Face“ genannt.

Zusammenhang:

mehrere Glyphen bilden ein FontFace

mehrere FontFaces bilden eine Font-Familie

ein Set von Font-Familien wird in einem GraphicsEnvironment - Objekt beschrieben

3.2 Möglichkeiten mit TextLayout

- Defninieren der LineMetrics:
  Ascent, Descent, Leading
- „Shaping“ von Text:
  Definition eigener Fonts durch Benutzung von Glyphvektoren möglich
- Bearbeiten der Textanordnung:
  bidirektionaler Text
- Textmanipulation:
  Hit-Detection, Dual-Carets, Cursorbewegungen, Text-Highlighting

4.0 Imaging

Ein BufferedImage -Objekt besteht aus einem Raster-Objekt und einem ColorModel-Objekt.

Das Raster kümmert sich um die Bilddatenverwaltung, der DataBuffer enthält die Bilddaten, das SampleModel gibt an wie die Daten zu interpretieren sind.

Das ColorModel gibt an, wie die Pixeldaten in Bezug auf die Farben auszuwerten sind.

Das BufferedImage kann als OffscreenBuffer benutzt werden, d.h. man rendert das Bild im Speicher bevor man es auf den Bildschirm kopiert.

4.1 Imaging - Möglichkeiten der Bildmanipulation

Zur Bildmanipulation sind zwei Interfaces zur Verfügung gestellt. RasterOP und BufferedImageOP.

Implementationen zur Bildmanipulation stehen zur Verfügung:

- AffineTransformOp (Affine Transformation von Pixeldaten)
- ColorConvertOp (Farbumwandlung)
- ConvolveOp (Schärfen und Glätten von Kanten)
- LookupOp (Farbpaletten manipulieren)
- RescaleOp (Skalierung)

5.0 Java2D - Color

Color-Management-Klassen:

ColorSpace: System um Farben zu beschreiben, selbst definierte ColorSpaces müssen Methoden zur Umrechnung in Standard-ColorSpaces (RGB, CIEXYZ) enthalten.

Color: Spezieller FarbWert, der in bestimmter Weise (je nach ColorSpace) beschrieben wird.

Color-Model: Wird mit einem Image verbunden, dient zur Farblichen Interpretation von Pixeldaten.

6.0 Java2D - Print

Java Printing API basiert auf dem „callback printing“- Model.

Das Printing-System und nicht die Applikation kontrolliert den Druckvorgang. Die Applikation stellt nur die Informationen bez. des Dokumentes zur Verfügung.

Man unterscheidet zwischen Pageable- und Printable-Jobs.

(bei Pageable ist mehrfacher Druck einer best. Seite möglich, bei Printable werden alle gedruckt)

public print(Graphics g, PageFormat pf, int pageIndex)

THE END