Advanced Java 2D™ topics for Desktop Applications

Chet Haase, Sun Microsystems

Doug Felt, ICU

Phil Race, Sun Microsystems

TS-3214

JavaOne
Sun's 2005 Worldwide Java Developer Conference

Goal of this talk

Learn more about the Java 2D API so you can write better, more robust code and more visually compelling applications

Agenda

- Graphics effects
  - adding some wow
- Text Rendering
  - how it works and how to do it right
- Text measurement, Layout and Transforms
  - how complex is text?
- Printing
  - understand how it works to so you can avoid some common problems.

Agenda

- Graphics effects
  - adding some wow
- Text Rendering
- Text measurement, Layout and Transforms
- Printing

Effect^ive^s

- Effects are not just gratuitous...
  - ... or at least they do not have to be
- "Effectives" are about *useful* effects in the *appropriate* situations to enable more effective applications
- Many possibilities
  - Translucency: more data in less space
  - Animating components: subtle effects for more dynamic rich clients
  - Animated Transitions

Effectives: Animated Transitions

- Current GUIs (especially HTML) suffer from "How did I get here?" syndrome
- Wouldn't it be better to have a logical flow?
  - Keep the users linked to the application
  - Make them more productive
- Goal: Connect the user from where they *were* to where they *are*
- Idea: Smoothly move between GUI states
  - Remember: GUI widgets are not locked in place
  - ... or at least they don't have to be

Animated Transitions: Many Possibilities

- Translucent fades
- Translation (sliding widgets around on the screen)
- Rotation (easily overdone...)
- Scaling (growing/shrinking widgets)
- ... but wait, there's more!
- Combinations of effects

First Things First: 2D Rendering Basics

- Using `Graphics2D` to affect rendering results
- Translation
- Translucency

Translation

- Draw a red-filled rectangle in your component:

```
public void paintComponent(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(x, y, width, height);
}

public void paint(Graphics g) {
    super.paint(g);
}
```

Translation

- Now draw it somewhere else:

```
public void paintComponent(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(x, y, width, height);
}

public void paint(Graphics g) {
    Graphics2D g2d = (Graphics2D)g;
    g2d.translate(transX, transY);
    super.paint(g);
}
```

Translucency

- Now draw the rectangle with translucency:

```
public void paintComponent(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(x, y, width, height);
}
public void paint(Graphics g) {
    Graphics2D g2d = (Graphics2D)g;
    g2d.translate(transX, transY);
    AlphaComposite newComposite =
        AlphaComposite.getInstance
        (AlphaComposite.SRC_OVER, opacity);
    g2d.setComposite(newComposite);
    super.paint(g);
}
```

What about Components?

- Same approach exactly; works on all children of this component

```
public void paint(Graphics g) {
    Graphics2D g2d = (Graphics2D)g;
    g2d.translate(transX, transY);
    AlphaComposite newComposite =
        AlphaComposite.getInstance
        (AlphaComposite.SRC_OVER, opacity);
    g2d.setComposite(newComposite);
    super.paint(g);
}
```

- Call to **`super.paint(g)`** will paint all children with same Graphics settings

DEMO

Translation and Translucency

Animated Transitions: Timing is Everything

- A "timer" is used to animate your painting
- Timers fire events at periodic intervals
- During each timing event, modify the look and placement of components
- Various Timers available:
  - javax.swing.Timer / java.util.Timer: basic functionality
  - TimingFramework: project on java.net that builds on Timer classes
  - Other timing frameworks

Timing Details

- Basic timers all have:
  - resolution: how often you want to be called
  - delay: how long to wait before starting
- For these demos, I used "Timing Framework", a project on java.net
- Builds on core Timer utilities, adds extra features to simplify things
  - sets duration; automatically stops at end
  - calls `timingEvent()` with fraction elapsed of total duration
  - can set repeating and ending behavior (not used here)

Animated Rendering

- Set up your Timer and start it
- For each timing event
  - Modify Graphics object (or other attributes related to rendering)
  - Force repaint

Animated Rendering

- Set up your timer and start it

```
Envelope envelope =
    new Envelope(repeatCount, startDelay,
                 Envelope.RepeatBehavior.FORWARD,
                 Envelope.EndBehavior.HOLD);
Cycle cycle = new Cycle(duration, resolution);
TimingController timingController = new
    TimingController(cycle, envelope, target);
timingController.start();
```

Animated Rendering

- For each timing event
  - Modify Graphics object (or other attributes related to rendering)
  - force a repaint

```
public void timingEvent(float elapsedFraction) {
    transX = elapsedFraction * deltaX;
    transY = elapsedFraction * deltaY;
    opacity = elapsedFraction;
    repaint();
}
// paint() method from before
```

Animated Rendering

- **paint()** method same as before:

```
public void paint(Graphics g) {
    Graphics2D g2d = (Graphics2D)g;
    g2d.translate(transX, transY);
    AlphaComposite newComposite =
        AlphaComposite.getInstance
        (AlphaComposite.SRC_OVER, opacity);
    g2d.setComposite(newComposite);
    super.paint(g);
}
```

DEMO

Animated Rendering

Animated Transitions: Screen Transitions

- Now that we know:
    - How to use 2D APIs to modify Swing rendering
    - Timing/Animation techniques
- ... why not generalize the solution for applications?
- Transition between different “screens” of an application
    - move/scale components shared between screens
    - fade components that are on one screen but not another
    - custom effects, or composites of multiple effects
- And generalize the framework to avoid per-component coding of animations

DEMO

Screen Transitions

Screen Transitions: Transition Effects

- Hint: Many Effects faster or easier with images
  - With snapshot image of component, simply call **`drawImage()`** instead of drawing component
- Inverse hint: Image tricks will only work on components whose internal layout does not change between the screens
  - For example:
    - a scroll view whose scroll bar changes size between the two states
    - a label whose text changes size
  - Might need full-on component rendering to get these right

Effect[ive]s: For More Information

- TimingFramework: Simplified animation & timing
  - http://timingframework.dev.java.net
- Screen Transitions:
  - hopefully an article soon; check my blog at http://weblogs.java.net/blog/chet
- Swing + 2D: Article by Shannon Hickey:
  - http://java.sun.com/products/jfc/tsc/articles/swing2d/
- JGoodies: Provides some GUI animation capabilities, worth looking into
  - http://jgoodies.com

Agenda

- Graphics effects
- Text Rendering
  - how it works and how to do it right
- Text measurement, Layout and Transforms
- Printing

Java 2D Text Rendering APIs

- `Graphics.drawString()`
- `Graphics.drawChars()`
- `TextLayout.draw(Graphics2D, ...)`
- `Graphics2D.drawGlyphVector()`
  - not normally used directly by apps; used by `TextLayout`
- These APIs are used by the Swing toolkit for all text rendering
  - Text on labels, buttons, text fields, HTML, etc.

Java 2D Text Rendering Process

- `java.awt.Font` mapped to physical font(s)
  - cached on `java.awt.Font`and as `Graphics` state
- Unicode characters in String mapped to glyph codes
  - complex text (e.g., Arabic) auto-detected and laid out
- Transforms and text rendering hints applied and glyph images retrieved from cache
- Rendering pipe selected according to Graphics state and destination surface type
- Glyphs blitted to destination surface

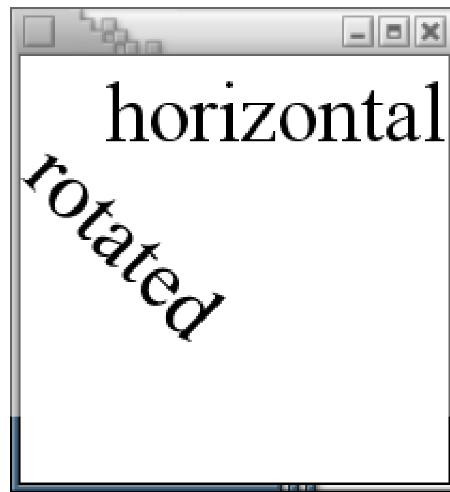
Java 2D Text Rendering Example

- Use graphics transform to draw rotated text

```
Graphics2D g2d = (Graphics2D)graphics;
Font f = new Font(Font.SERIF, Font.PLAIN, 40);
g2d.setFont(f);
g2d.drawString("horizontal", 40, 40);
g2d.rotate(Math.PI/4);  // do not use setTransform()
g2d.drawString("rotated", 40, 40);
g2d.rotate(-Math.PI/4); // restore the transform
```

Java 2D Text Rendering Example

- Using a graphics transform to draw rotated text



Rotated Text Rendering Notes

- Glyph images, advances are different when rotated
  - is another "strike" in the glyph cache – using different rotations will need more memory in the cache
  - rendering performance very similar to unrotated text
- Do not use `setTransform()` - instead concatenate
  - `setTransform()` can cause Swing repaint errors
  - use only to restore a saved transform, to eliminate accumulated error

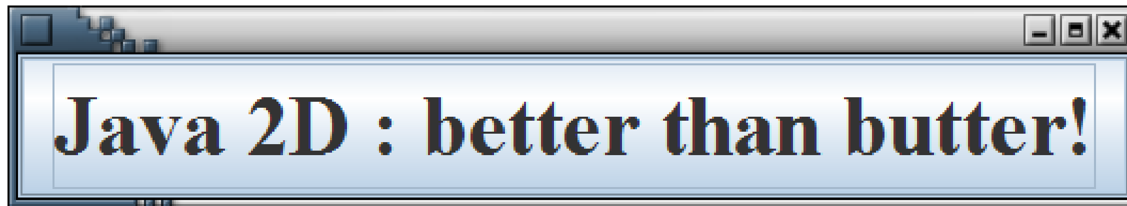
```
savedTx = g2d.getTransform();
g2d.rotate(Math.PI/4); // concatententates to existing TX
g2d.drawString("rotated", 40, 40);
g2d.setTransform(savedTX); // this is OK
```

New Font/Text APIs in JDK 6.0

- Constants for logical fonts
  - DIALOG, SERIF, SANS_SERIF, MONOSPACED
- RenderingHints for LCD text
  - e.g., VALUE_TEXT_ANTIALIAS_HRGB
- Kerning, optional ligatures, tracking
  - add as new TextAttributes
- All TextAttributes now supported on Font
  - no need to use TextLayout, just set the attributes on the font and call **`drawString()`**

Using Text Attributes on a Font

GradientPaint for text color



Paint Text Attribute Code Sample

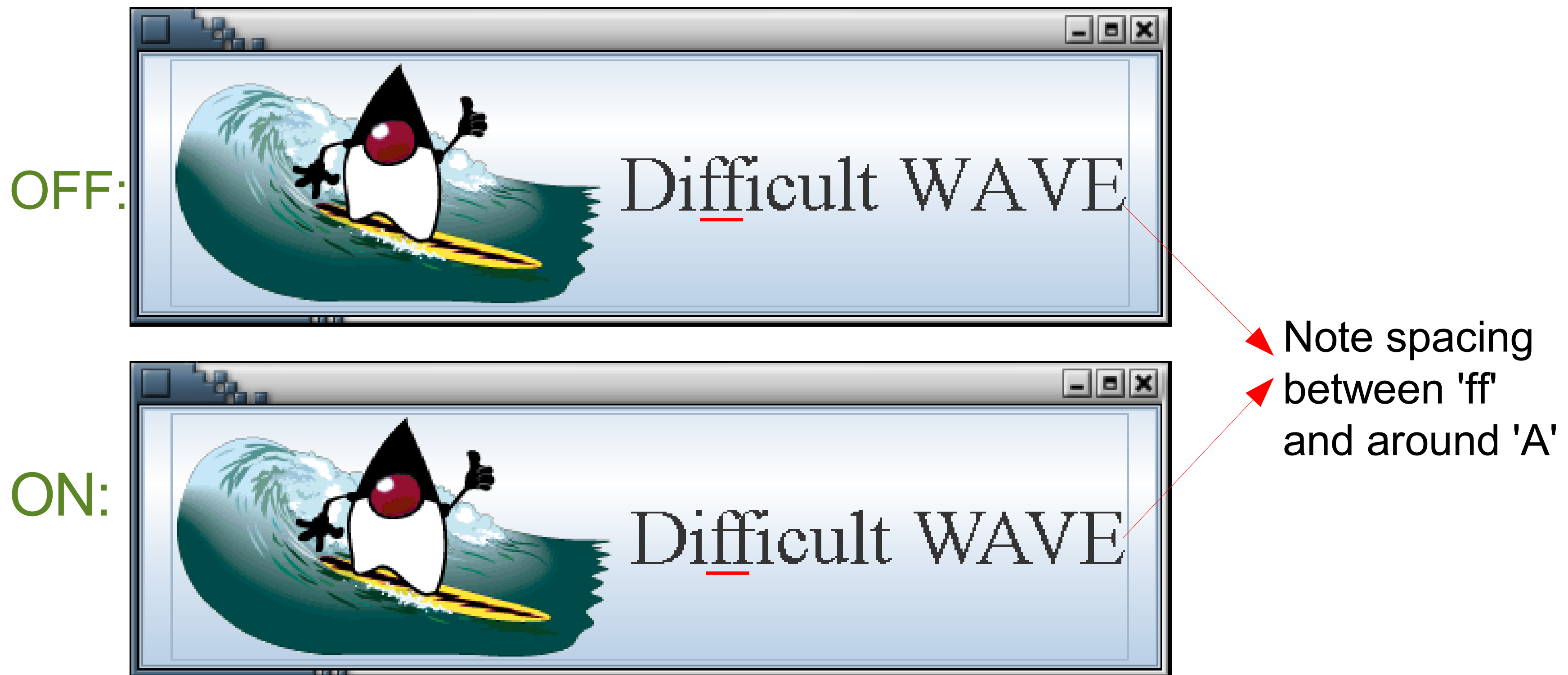
```
JFrame f = new JFrame();
JButton b =
    new JButton("Java 2D: better than butter!);
Font font = new Font(Font.SERIF, Font.BOLD, 40);

attrs = new HashMap<TextAttribute,Object>();
Paint p = new GradientPaint(0,0, Color.red, 25, 25,
                            Color.yellow, true);
attrs.put(TextAttribute.FOREGROUND, p);
font = font.deriveFont(attrs);

b.setFont(font);
f.add(b);
```

Using Text Attributes on a Font

Kerning and optional ligatures



Kerning & Ligatures Code Sample

```
import static java.awt.font.TextAttribute.*;

JFrame f = new JFrame();
Icon i = new ImageIcon("surfing.gif");
JButton b = new JButton("Difficult WAVE", i);
Font font = new Font(Font.SERIF, Font.PLAIN, 40);

attrs = new HashMap<TextAttribute,Object>();
attrs.put(KERNING, KERNING_ON);
attrs.put(LIGATURES, LIGATURES_ON);
font.deriveFont(attrs);

b.setFont(font);
f.add(b);
...
```

Agenda

- Graphics effects
- Text Rendering
- Text measurement, Layout and Transforms
    - how complex is text?
- Printing

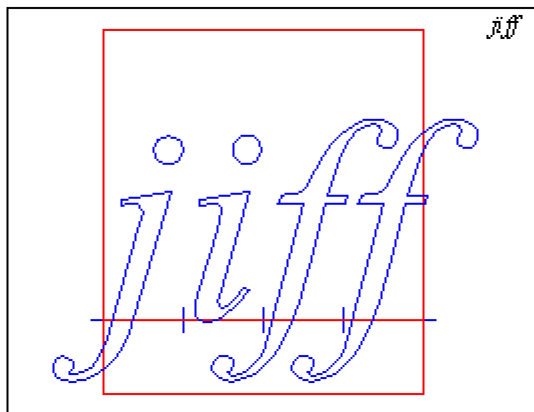
Text Metrics

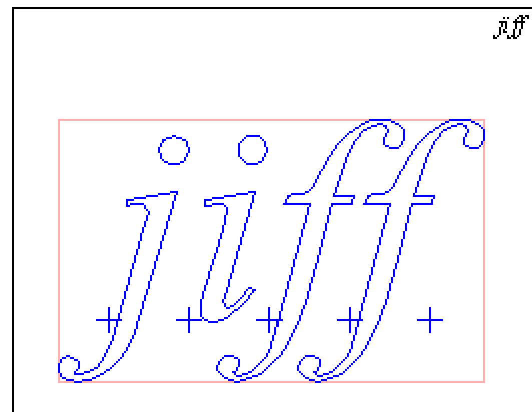
A primer on Java text metrics

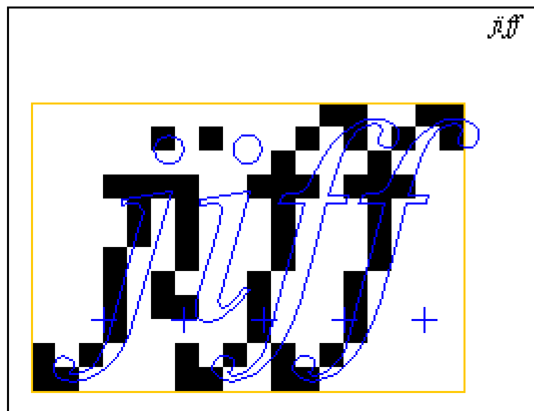
- Types of text bounds
  - Logical (ascent, descent, leading, advance)
  - Visual (glyph bounding box)
  - Pixel (pixel bounding box)
- Only pixel bounds are guaranteed to enclose all the text
- TextLayout bounds is a union of visual and logical bounds
- Pixel bounds only available through **`GlyphVector`**

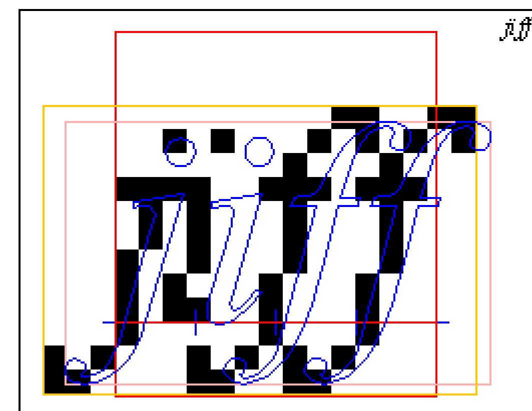
Text Metrics

Logical, Visual, and Pixel bounds illustrated









Assumptions that may break down

- Monospaced fonts
  - not all scripts have this concept
- Text is left to right
- Text scales linearly
- Characters map to glyphs 1:1
- Unicode characters can be represented in a 16 bit char
- Ascent of a font is at least as high as the tallest char/glyph

Layout

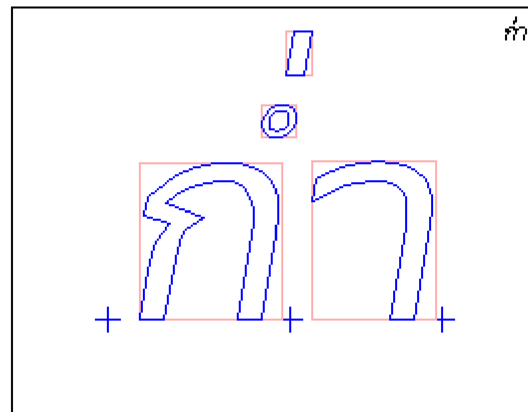
A primer on layout

- Arabic, Hebrew, Indic, Thai, Supplementary
- All rendered text examined for possible layout
  - If you know it needs layout, construct and reuse a TextLayout
- Bidi can reorder segments of text
  - Works on entire paragraphs
  - Don't segment text yourself
  - use TextMeasurer or LineBreakMeasuer for multiple lines
- Layout can add, replace, reorder glyphs
  - Don't assume you know the width of a character
  - Don't assume you know where the character went

Layout

Inserting new glyphs illustrated

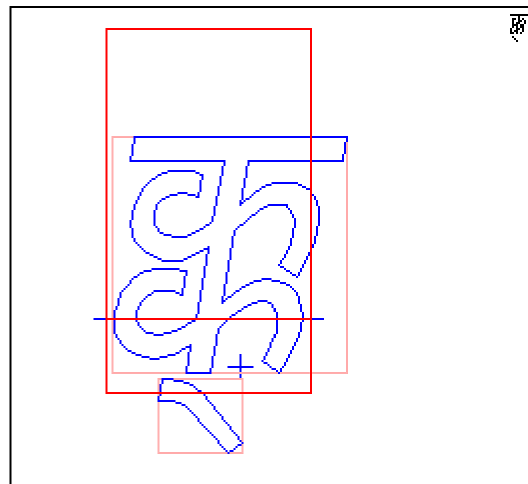
- Thai (\u0e01\u0e48\u0e33)
- Three characters, four glyphs



Layout

Ligatures and positioning illustrated

- Hindi (\u0915\u094d\u0915\u094d)
- One ligature, two invisible glyphs, positioned mark glyph (a8f ffff ffff 937)

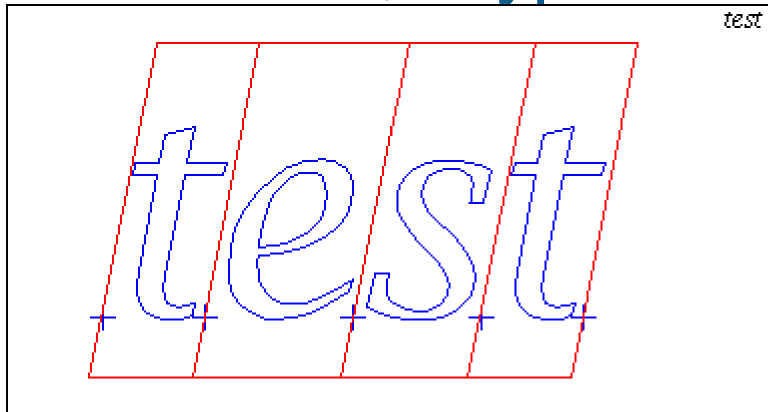


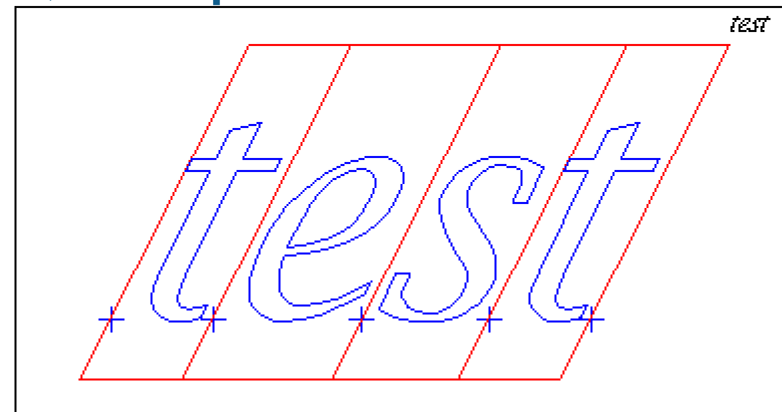
Transforms

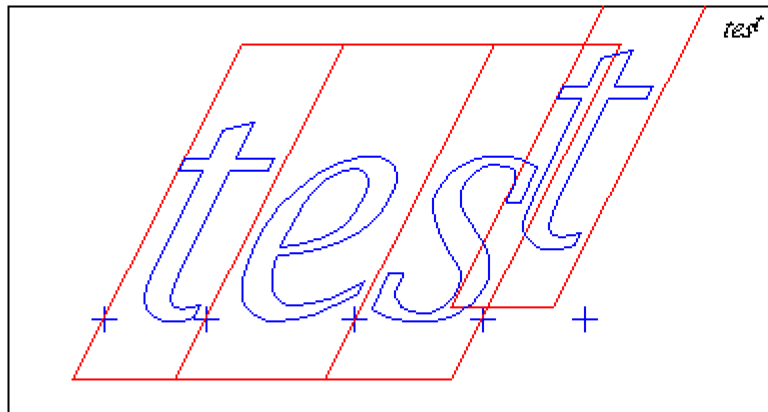
- Can be applied to `Graphics`, `Font`, glyph,
  - Concatenated* to generate final glyph outline
  - Effect on logical bounds is more complicated
- `Graphics` transforms are clean
- Simple `Font` transforms are ok, but
  - rotation gives layout headaches, rotate graphics instead
  - effect on metrics can be problematic
- Per-glyph transforms for special effects
  - * Translation not concatenated with font transform
  - bulks up the `GlyphVector`(more expensive)

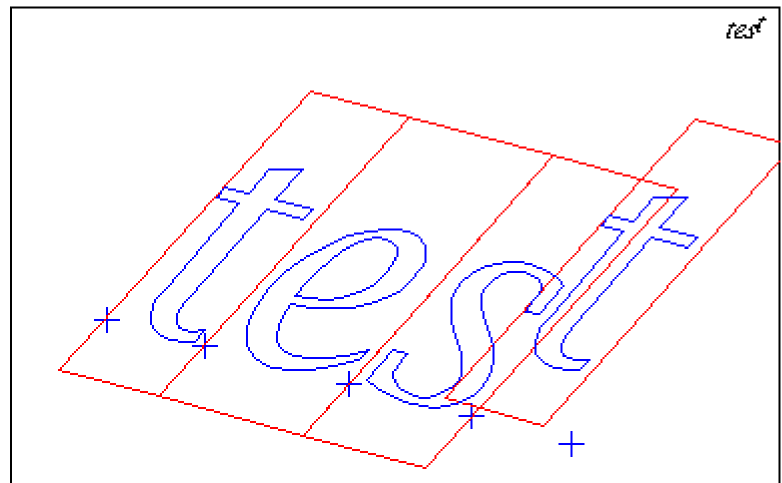
Transforms

Font skew, Glyph translate, Graphics rotation









Agenda

Graphics effects

Text Rendering

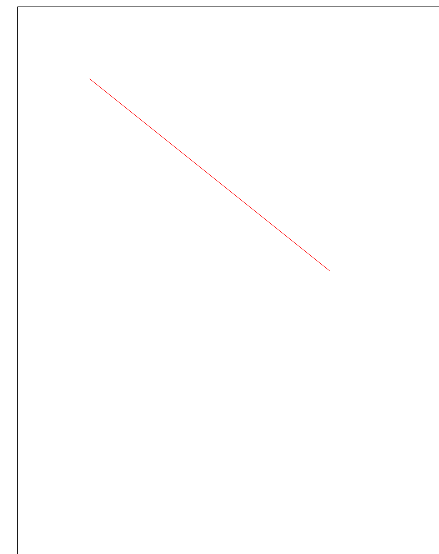
Text measurement, Layout and Transforms

Printing

- understand how it works to so you can avoid some common problems

Printing 2D graphics

- Minimal printing code:



```
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(new MyPrintable());
try {
    job.print();
} catch (PrinterException) {
}

class MyPrintable implements Printable {
   public int print(Graphics g, PageFormat pf, int pg) {
       if (pg > 0) return Printable.NO_SUCH_PAGE;
       g.setColor(new Color(255,0,0)); // Color.red
       g.drawLine(0,0,400,400);
       return Printable.PAGE_EXISTS;
   }
}
```

Printing 2D graphics

Where's the pain?

- Printable is a “callback”
  - Want to use PDL for optimal graphics rendering but printer languages cannot support translucency needed by 2D
  - Implementation calls at least twice per page
  - First call is a peek – used to determine if PDL can be used or whether to print as bands of device resolution off-screen images (also known as the raster path)
  - The banding raster path will be chosen if custom paints or translucent colors or translucent images are rendered

Printing 2D graphics without (much) pain

- Add tracing

```
class MyPrintable implements Printable {
   public int print(Graphics g, PageFormat pf, int pg) {
       System.out.println("page="+pg);
       if (pg > 0) return Printable.NO_SUCH_PAGE;
       g.setColor(new Color(255,0,0)); // Color.red
       g.drawLine(0,0,400,400);
       Rectangle2D r = g.getClip().getBounds2D();
       System.out.println(
        "ClipTop = " + (float)r.getY() +
        " ClipBottom="+(float)(r.getY()+r.getHeight()));
       return Printable.PAGE_EXISTS;
   }
}
```

Printing 2D graphics without (much) pain

- Trace with opaque color
  - Just one peek, then one call to generate PS for entire page

```
page=0
Graphics = sun.print.PeekGraphics[..]
ClipTop = 72.0 ClipBottom=720.0

page=0
Graphics = sun.print.PSPathGraphics[..]
ClipTop = 72.0 ClipBottom=720.0

page=1
```

Printing 2D graphics with translucency

- Trace with translucent color
  - `g.setColor(new Color(255,0,0,128));`

```
page=0
Graphics = sun.print.PeekGraphics[..]
ClipTop = 72.0 ClipBottom=720.0

page=0
Graphics = sun.print.ProxyGraphics2D[..]
ClipTop = 72.0 ClipBottom=243.84

page=0
Graphics = sun.print.ProxyGraphics2D[..]
ClipTop = 243.84 ClipBottom=415.68

page=1
```

Printing 2D graphics with translucency

Where's the pain?

- What the user/developer sees when printing translucent graphics
  - Large spool files – it has a huge image
    - Postscript spool file sizes for sample code:
      - Opaque red: 2,393 bytes
      - Translucent red: 216,024 bytes
  - Slower printing – multiple calls to print a page
  - Printer fonts not used

Printing 2D graphics with translucency

What are the workarounds?

- Don't try to trick the peek – it'll bite you
- Pay attention to the clip to minimize printing time
- If the translucency is known and limited to a defined area, render that area to an image at a suitable resolution, and print the image
- Maybe use opaque colors
- Long term, better PDLs and implementation improvements may help

Printing 2D graphics : Pagination pain

- Printable can be called multiple times per page
  - Must be prepared to render the page requested
  - Complicates page break calculations – need to flexibly store state between calls
- Calculating breaks for multi-page text requires using the correct **`FontRenderContext`**
  - Ideal layout for screen will not look ideal on printer; measure text using `FontRenderContext` from printer graphics
  - On-screen approximate printer metrics using fractional metrics:

```
Graphics2D.setRenderingHint(KEY_FRACTIONALMETRICS,
                            VALUE_FRACTIONALMETRICS_ON);
```

Summary

- The 2D API is the rendering engine for desktop applications
  - graphics rendering, text, printing
- Use Java 2D graphics for more effective GUI applications
- Use your understanding of the 2D API and how 2D works to write robust code that performs well

Q&A

Chet Haase

Doug Felt

Phil Race

For More Information

- 2D-related JavaOne sessions
  - TS-3605: GUI Makeover 1: today@1:30
  - TS-3902: GUI Makeover 2: today@2:45
  - Swing/AWT/2D BOFS: tonight@730-1030
  - TS-3843: GUI Puzzlers: Thursday@1045
- 2D-related websites:
  - http://javadesktop.org
    - articles, blogs, forums
  - http://javagaming.org
    - forums on various graphics and performance topics
  - http://java.sun.com/products/java-media/2D/index.jsp
    - Java 2D™ homepage