
JavaOne℠
Sun's Worldwide Java Developer Conference

JavaOne℠

Sun's Worldwide Java Developer Conference

Rendering with Java™

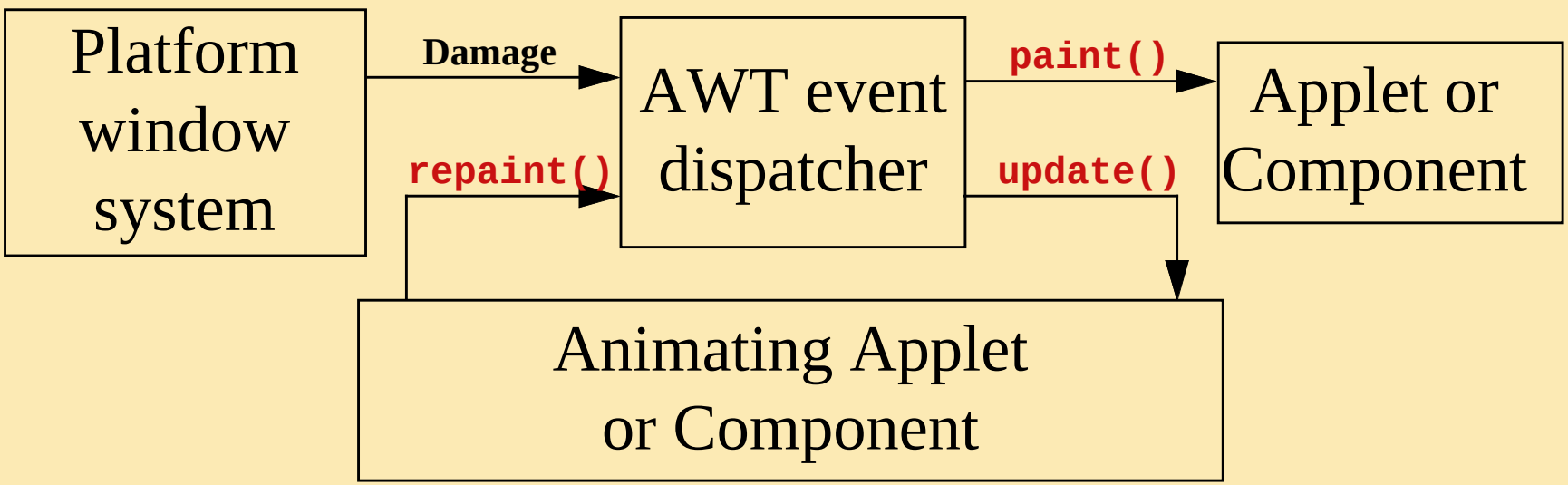
Jim Graham
Staff Engineer
JavaSoft

Overview

- AWT paint/update callback model
- Graphics rendering
- Image rendering and manipulation
  - Basic image fetching and drawing
  - Off-screen images for double buffering
  - Advanced image filtering and manipulation
- Futures: Java Media 2D API

AWT Callback Overview

- AWT calls `paint()` asynchronously
- call `repaint()` to redraw a Component
- AWT calls `update()` on demand



Component callback: paint()

```
public void paint(Graphics g);
```

- Asynchronously called when:
  - Component first becomes visible
  - Display damage occurs (visibility changes)
- Background will already be cleared
- Clip area is set to damaged area

Component Method: repaint()

```
public void repaint();
public void repaint(x,y,w,h);
public void repaint(t,x,y,w,h);
```

- Schedules a call to `update()` within `t` milliseconds
- Multiple repaint areas are collapsed

Component callback: update()

```
public void update(Graphics g);
```

- Called in response to calls to `repaint()`
- Background will ***not*** be cleared
- Clip area is set to requested repaint area
- Default implementation clears the background and calls `paint()`

Callback Thread Model

- `paint()` and `update()` methods are called from the AWT Callback Thread
- Perform rendering and return quickly
- No unnecessary calculations
- No embedded animation iterations:

```
for (y = 0; y < size().height; y += 10) {
    Thread.sleep(100);
    g.drawLine(x, y, x+10, y);
}
```

Why use repaint()?

- Allows collapsing of multiple updates
  - Better mouse tracking
  - Multiple asynchronous calculation threads
- Enables model/view controller paradigm
  - Independent tracking of calculations and screen updating
  - One repaint may generate multiple updates

Graphics Overview

- Graphics object and attributes
- Coordinate System
- Rendering functions
- XOR mode behavior

Graphics Object

- Obtained for particular Component:

```
public Graphics getGraphics();
```

- Constructed automatically
  - Passed to `update()` and `paint()` as an argument
  - Already initialized with Component's color, font, and clip region for area to be drawn

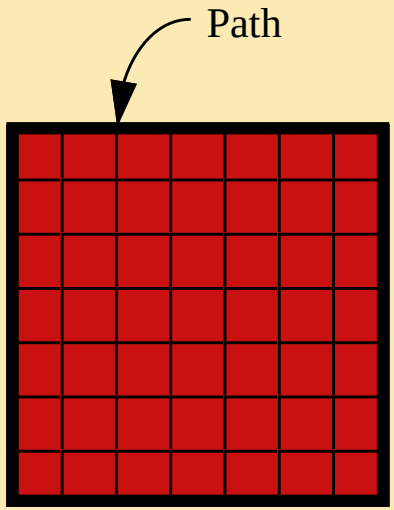
Graphics Attributes

- Rectangular clip area
  - Set ***smaller*** with `clipRect()`
  - Get with `getClipRect()`
- Current Color
- Current Font
- XOR (alternation) color

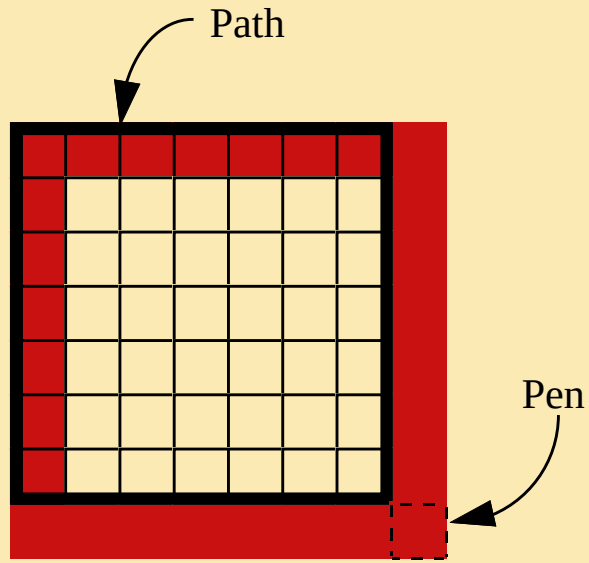
Coordinate System

- Integer coordinates fall between pixels
- Fill operations fill inside a path
  - `w`x`h` object covers `w`x`h` pixels
- Draw operations stroke along a path
  - Pen hangs down and to the right
  - `w`x`h` object covers `(w+1)`x`(h+1)` pixels

Coordinate System Diagram



7x7 filled rectangle



7x7 drawn rectangle

Basic Rendering Operations

```
drawLine(x1, y1, x2, y2);
draw/fillRect(x, y, w, h);
draw/fillRoundRect(x, y, w, h, arcw, arch);
draw/fill3DRect(x, y, w, h, raised);
draw/fillOval(x, y, w, h);
draw/fillArc(x, y, w, h, ang1, ang2);
draw/fillPolygon(Polygon);
draw/fillPolygon(xPts[], yPts[], numPts);
```

Other Rendering Operations

- Text:

```
drawString(String, x, y);
drawChars(chars[], start, len, x, y);
```

- Images:

```
drawImage(image, x, y, obs);
drawImage(image, x, y, bg, obs);
drawImage(image, x, y, w, h, obs);
drawImage(image, x, y, w, h, bg, obs);
```

XOR Rendering Mode

- Specify a color to alternate the foreground with:

```
// Alternate foreground with Black
setXORMode(Color.black);
// Set foreground color to Green
setColor(Color.green);
// Rendering on Green produces Black
// Rendering on Black produces Green
```

XOR Example

```
g.setColor(Color.black);
g.fillRect(0, 0, 400, 80);
g.setColor(Color.green);
g.fillRect(0, 80, 400, 80);
g.setXORMode(Color.black);
g.setColor(Color.green);
g.fillRect(40, 40, 80, 80);
g.setColor(Color.red);
g.fillRect(160, 40, 80, 80);
g.drawImage(img, 280, 40, this);
```

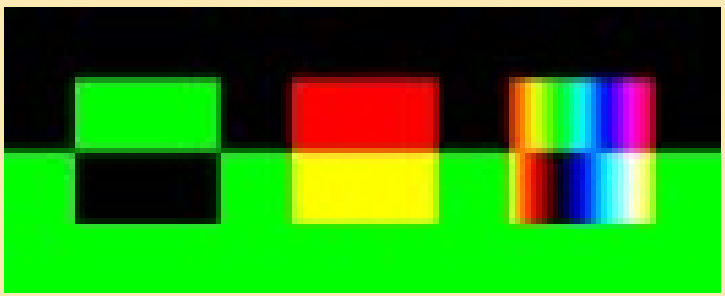


Image Overview

- Image fetching and drawing
- Off-screen images for double buffering
- Image filtering and manipulation
  - Creating your own images
  - Filtering existing images
- http://java.sun.com/people/flar/images.html
- http://www.gamelan.com/

Drawing Images

- Get an image handle with `getImage()`
- Images are loaded asynchronously
- Data loaded on demand when calling:
  - `getWidth()` or `getHeight()`
  - `drawImage()`
  - `getProperty()`

ImageObserver Interface

- Observer is notified via callback as data becomes available:
  - Size information
  - Properties
  - More pixels converted for drawing
- Component implements ImageObserver

Double Buffering

- Component method:

```
public Image createImage(int w, int h);
```

- Returns an image you can draw into

```
Image img = createImage(w, h);
Graphics g2 = img.getGraphics();
```

- Copy results to screen like any image

```
g.drawImage(img, x, y, this);
```

ImageProducer Interface

- Create new images with:

```
public Image createImage(ImageProducer p);
```

- All images have an ImageProducer

```
ImageProducer p = img.getSource();
```

- Recreate image data on demand
- Consumers contact the producer to retrieve the information

Standard ImageProducers

- Built-in classes handle most common image needs:
  - `MemoryImageSource`
  - `FilteredImageSource`
  - `URLImageSource` (internal)
  - `OffscreenImageSource` (internal)

ImageConsumer Interface

- ImageConsumer interface defines how producers deliver image data
- Consumers register with a producer using the ImageProducer interface
- Data is delivered asynchronously after a consumer is registered

Standard ImageConsumers

- The image rendering system utilizes ImageConsumer classes to store pixels:
  - `ImageRepresentation` (internal)
    - One object per rendered size
  - `ImageInfoGrabber` (internal)
    - Intercepts width, height, properties

ImageConsumer Utilities

- The AWT provides utility classes that implement ImageConsumer
  - `PixelGrabber`
    - Used to retrieve pixels asynchronously from any image
  - `ImageFilter`
    - Acts as ImageConsumer for one image and ImageProducer for another

Producer/Consumer Interactions

- Consumer registers with producer:

```
producer.startProduction(consumer);
```

- Then producer delivers the data:

```
consumer.setDimensions(w, h);
consumer.setHints(…);
consumer.setColorModel(cm);
for (int y = 0; y < h; y++) {
   // … Calculate row y …
   consumer.setPixels(0, y, w, 1,
    cm, pix, 0, w);
}
```

ColorModel Class

- Each setPixels() call has a ColorModel
- Defines mapping from pixels to colors

```
public int getRGB(int pixel);
```

  - No assumptions made about pixel format
- Abstract ColorModel class
  - Subclass defines methods to extract red, green, blue and alpha components

Standard ColorModels

- Two ColorModel classes handle most common pixel formats
  - `IndexColorModel`
    - 8-bit indexed colormap
  - `DirectColorModel`
    - Masks specify RGB and alpha components
- Pixel conversion code recognizes and optimizes conversion of these models

ImageProducer Example

```
int w = 80;
int h = 80;
int pix[] = new int[w * h];
for (int y = 0; y < h; y++) {
   for (int x = 0; x < w; x++) {
      pix[y * w + x] =
         Color.HSBtoRGB(x * 1.0f / w, 1.0f, 1.0f);
   }
}
ImageProducer prod =
   new MemoryImageSource(w, h, pix, 0, w);
Image img = createImage(prod);
```

ImageConsumer Example

```
int[] pixels = new int[w * h];
PixelGrabber pg =
   new PixelGrabber(img, x, y, w, h,
                    pixels, 0, w);
try {pg.grabPixels();}
   catch (InterruptedException e) {return;}
for (int j = 0; j < h; j++) {
   for (int i = 0; i < w; i++) {
      handlesinglepixel(x+i, y+j,
                        pixels[j * w + i]);
   }
}
```

IndexColorModel Example

```
byte red[] = { 0, 255,   0,   0, 255 };
byte grn[] = { 0,   0, 255,   0, 255 };
byte blu[] = { 0,   0,   0, 255, 255 };
cm = new IndexColorModel
  (8, 5, red, grn, blu);
                     //      AARRGGBB
cm.getRGB(0);        // => 0xff000000
cm.getRGB(2);        // => 0xff00ff00
cm.getRGB(10);       // => 0x00000000
```

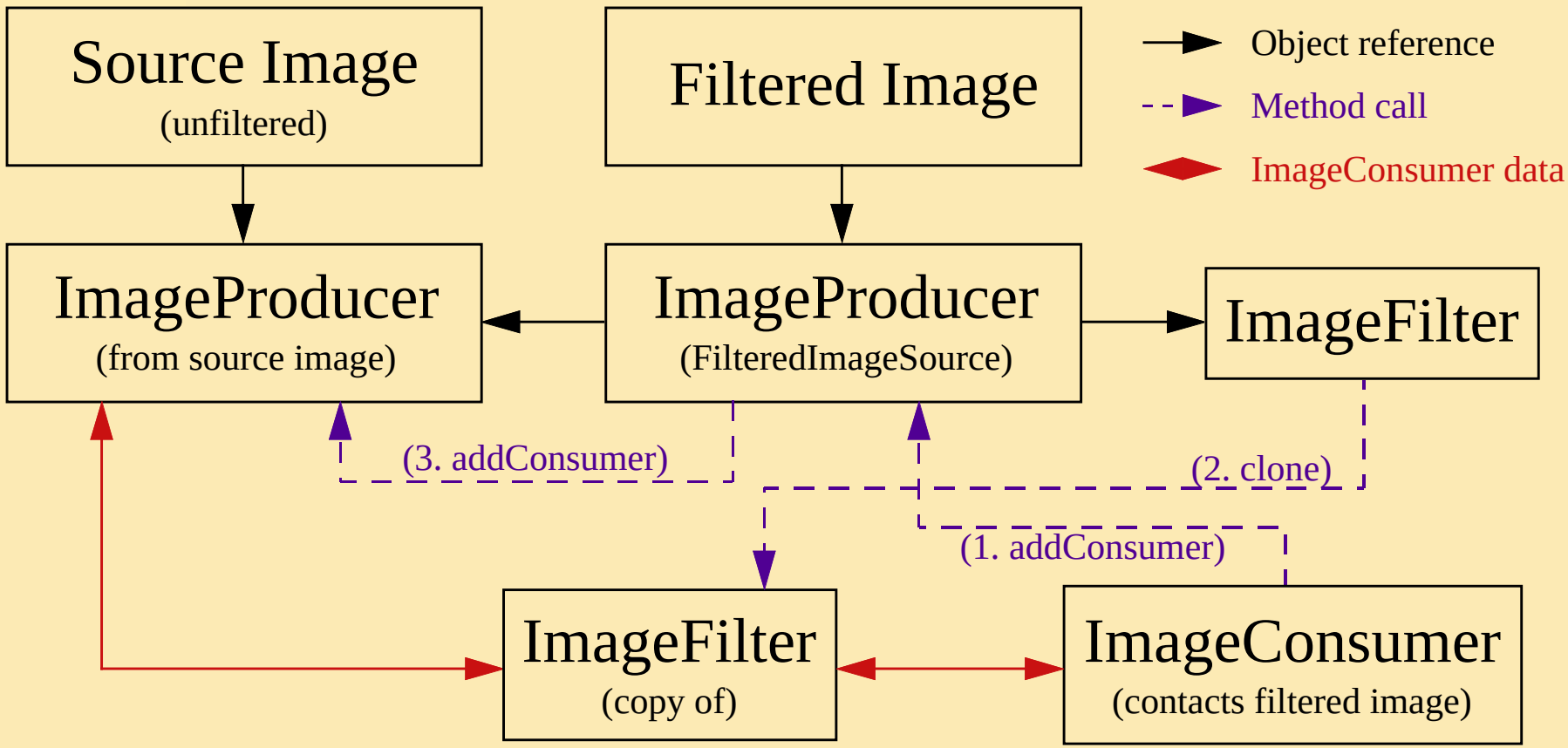
DirectColorModel Example

```
cm = new DirectColorModel
  (16, 0x001f, 0x03e0, 0x7c00, 0x8000);
                       //       AARRGGBB
cm.getRGB(0xffff); // => 0xffffffff
cm.getRGB(0x801f); // => 0xffff0000
cm.getRGB(0x83e0); // => 0xff00ff00
cm.getRGB(0xfc00); // => 0xff0000ff
cm.getRGB(0x001f); // => 0x00ff0000
cm.getRGB(0x0);    // => 0x00000000
```

ImageFilter Class

- ImageFilter implements ImageConsumer
  - ImageFilter performs “null filter”
  - Subclass to modify data as it is delivered
- Used with FilteredImageSource
  - Manages list of consumers
  - Associates copy of ImageFilter with consumer
  - Registers filter with original ImageProducer

ImageFilter Diagram



ImageFilter Example

```
class RedBlueSwapFilter extends RGBImageFilter {
   public RedBlueSwapFilter() {
      // filter colormaps faster than pixels
      canFilterIndexColorModel = true;
   }
   public int filterRGB(int x, int y, int rgb) {
      //                    AARRGGBB
      return (((rgb & 0xff00ff00)) |
              ((rgb & 0x00ff0000) >> 16) |
              ((rgb & 0x000000ff) << 16));
   }
}
```

Futures: JavaMedia 2D API

- Collaborative effort with key partners
  - Adobe, SunSoft, and JavaSoft
- Rich rendering model
- Extensible paths & transforms
- Exposed layout images

Futures:
Java Media Rendering

- 2D affine (and custom) transforms
- Line/Bezier (and custom) paths
- Solid, gradient (and custom) fills
- Wide strokes with join and cap attributes
- Alpha blend (and custom) compositors
- Advanced text layout

Futures: Java Media 2D Extensibility

- Base abstract transform and path classes
  - Implementations must satisfy certain basic functionality to work with the renderer
- Specific standard optimized classes
  - `AffineTransform` and `BezierPath` recognized and optimized by the renderer

Futures: Java Media images

- New image subclass with exposed layout
  - ImageDescriptor defines memory layout
  - Satisfy current ImageProducer interface
- Extended ColorModels
  - Non-RGB color spaces
  - ICC profiles (color matching)
  - Default color space is RGB709

Questions?