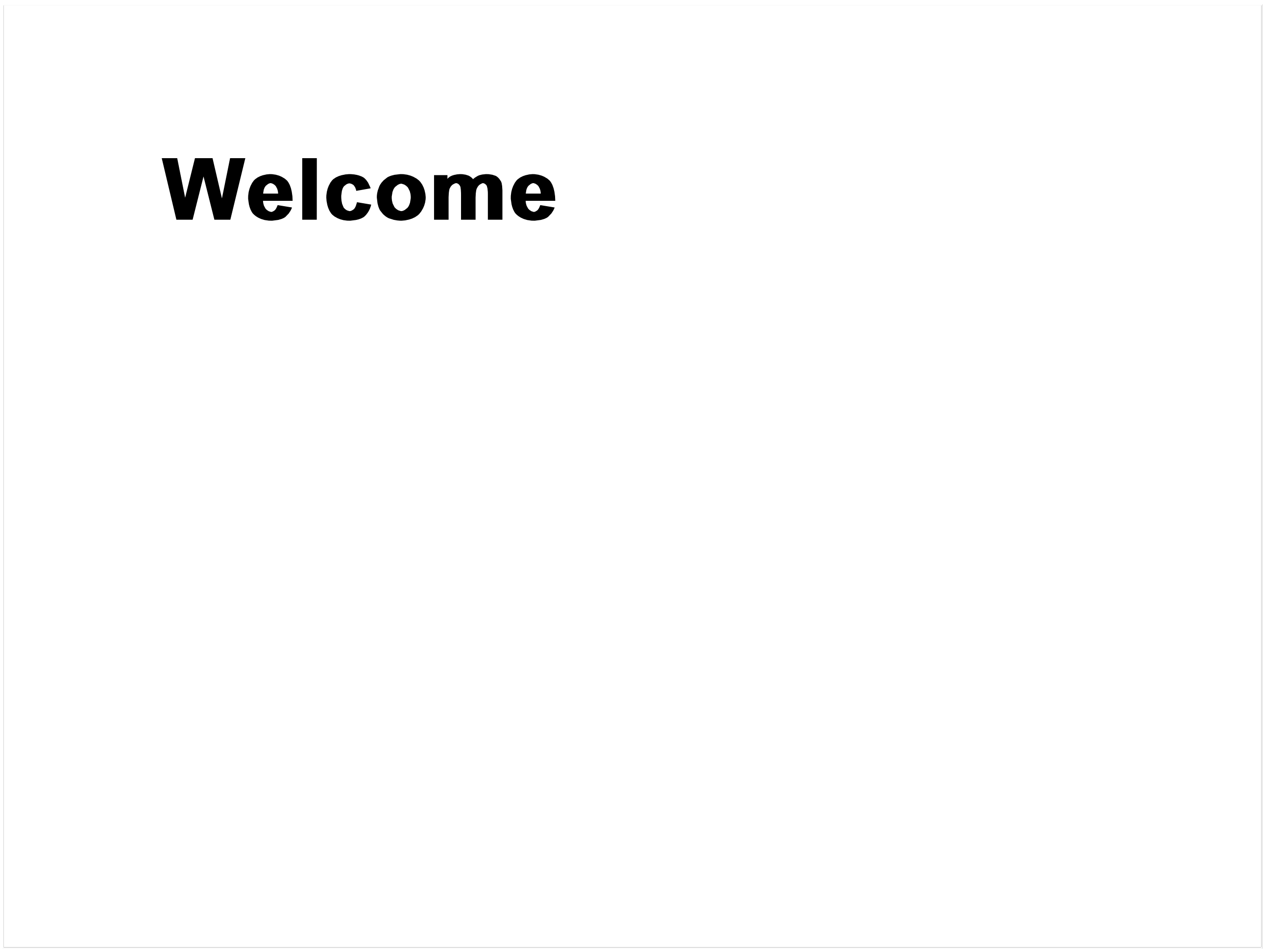
Java Applications with Apache Batik

Thomas DeWeese

ApacheCon US 2003



Welcome

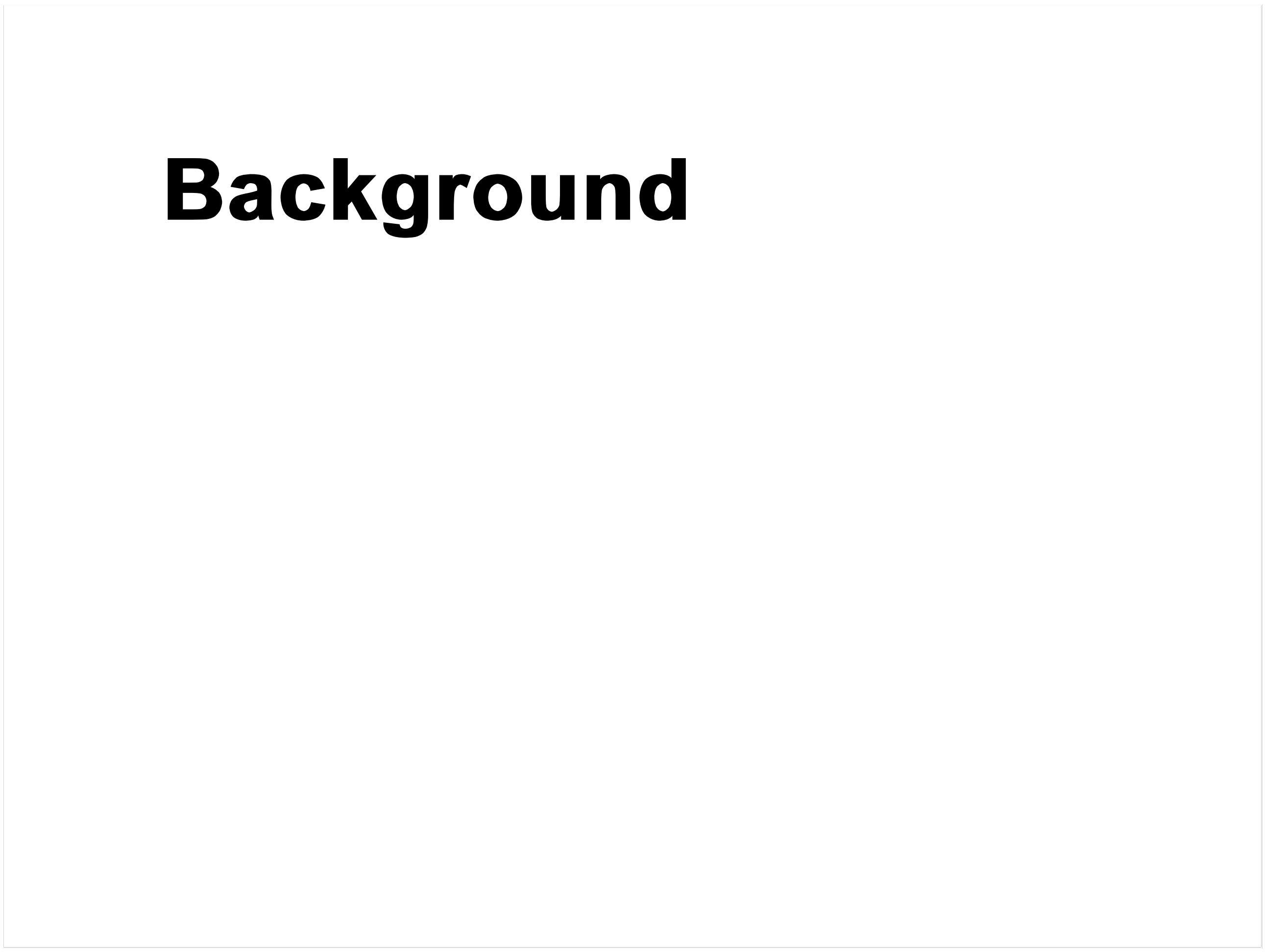
The goal of this presentation is to provide people with enough information to do "interesting" things with SVG and Batik. This means going beyond just showing SVG but really interacting with it and hopefully enabling surprising things.

In order to accomplish this, the presentation will attempt to give you a glimpse into some of the inner structure of the Batik components.

A few conventions, I commonly abbreviate the full Batik package: "`org.apache.batik`" as just "`batik`", to save space. I will occationally use "`[...]`" to indicate that something was removed/abbreviated.

Major Topics

- Background on Batik, SVG
- Presenting SVG
- Static vs. Dynamic Documents
- DOM Event Handlers
- The UpdateManager
- Overlays/Interactors
- Conclusion



Background

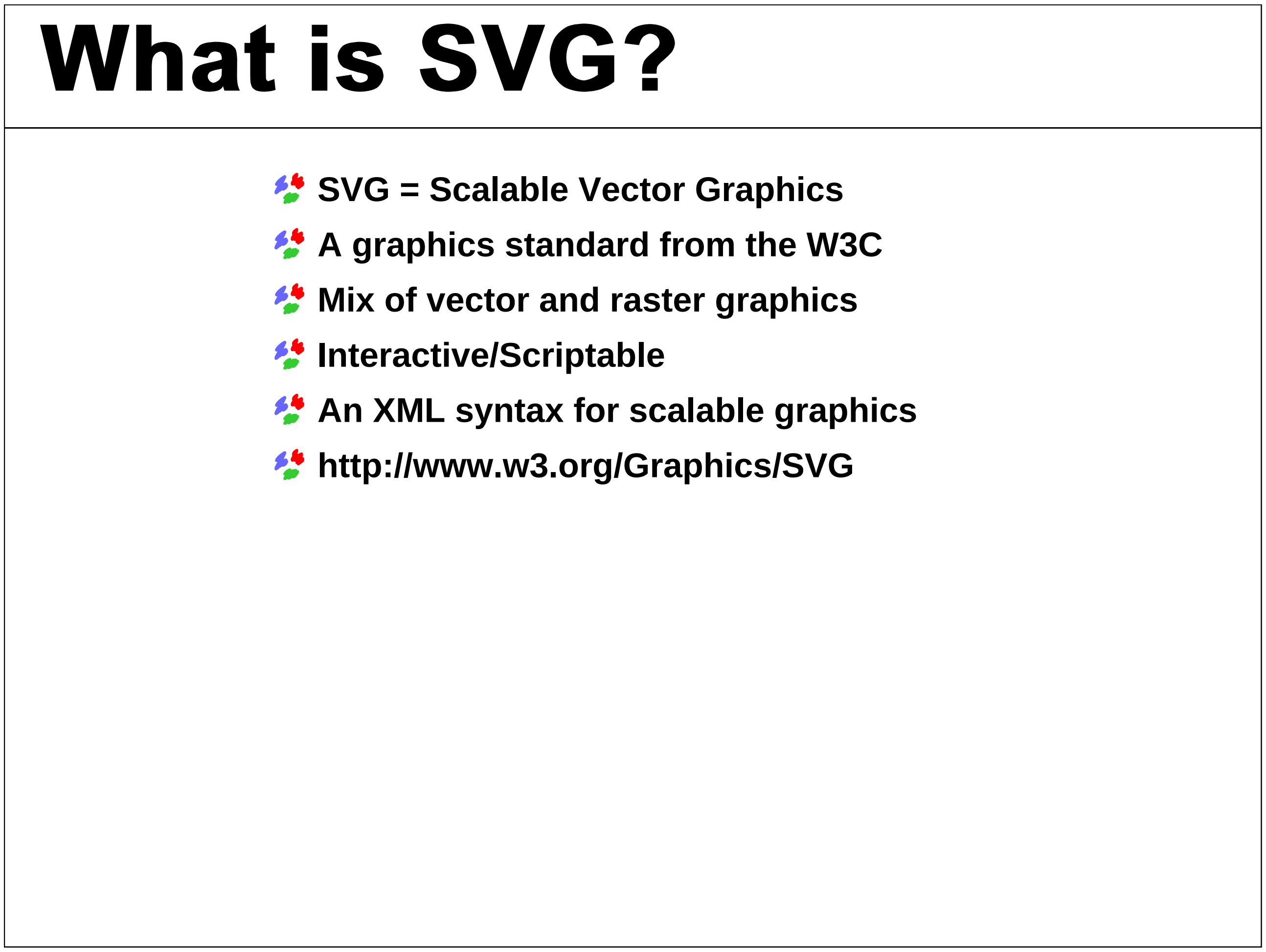
What is Apache Batik?

- **A Java language toolkit for generating, displaying, rasterizing and manipulating SVG.**
- **Batik has a number of reusable components that are designed to be incorporated into host applications.**
- **Batik also provides a number of "how-to" applications that may be useful.**
- **A strong reference point for SVG compliance.**

It is important to emphasize the "toolkit" nature of Batik. "Squiggle" the SVG browser gets a lot of attention, and it is a very cool piece of the the toolkit, but it is only one piece. The intent of Batik is not to be the ultimate SVG browser (replacing Adobe, or Corel) but to provide the tools people need to deploy SVG in "the real world".

Things like the SVG transcoders and generators, font converters are important pieces in the whole SVG puzzle. They provide support for SVG workflows.

Batik has been built to be as strictly conformant to the SVG specification as possible. It is very important for users to have multiple reference points on content compliance. As a toolkit, some of the browser pressure to "do what I mean" is gone.



What is SVG?

- SVG = Scalable Vector Graphics
- A graphics standard from the W3C
- Mix of vector and raster graphics
- Interactive/Scriptable
- An XML syntax for scalable graphics
- http://www.w3.org/Graphics/SVG

SVG 1.0 became a W3C Recommendation (Standard) in September 2001. SVG 1.1 (profiles 1.0) was released in January 2003. SVG 1.2 is currently in Working Draft.

SVG can represent complex graphics as a combination of Raster (images like PNG, JPEG) and vector graphics (lines arcs, polygons etc). It also allows the application of filters to these primitives to get sophisticated results.

SVG graphics can be rendered at multiple resolutions, so graphics can look good at high and low resolutions (no more blocky effects when printing Web graphics!).

SVG being based on XML allows for scripting using any language with a DOM binding. SVG also defines a set of events that allow for complex responses to user actions, such as changing the color of a graphic element (e.g., a rectangle) when the mouse moves over it.

SVG also supports SMILE animation. This allows one to declaratively define animations, based on time cues, or in response to user actions.

Other Batik Tools

- **ttf2svg**

  Converts True Type fonts to SVG Fonts
- **wmf2svg**

  Converts WMF files to SVG
- **CSS Engine**

  Implements CSS SAC/DOM Style
- **SVGGraphics2D**

  Graphics2D that outputs SVG

The "ttf2svg" application converters all or part of a TrueType font to an SVG Font. People do need to be careful and pay attention to Font Licenses.

The "wmf2svg" application converts Windows Meta Files to an SVG file. There are many variants of WMF, and the converter does not handle all of them but it does handle most of them. Additions and corrections are always welcome.

It seems that some people are making use of the Batik CSS Engine. Either just the SAC (CSS parser) or the complete "DOM Style" implementation.

The SVGGraphics2D is an implementation of Graphics2D that generates SVG. This makes it really easy to generate SVG content from existing Java applications.

SVGGraphics2D

Common Issues

- **`batik.svggen.SVGGraphics2D(Document factory)`**
  The Document is a factory, Elements are not added.
- **`getRoot()`**
  Returns the content tree.
- **`setSVGCanvasSize(Dimension)`**
  Set the width/height of the root SVG element.
- **`batik.svggen.SwingSVGPrettyPrint`**
  Renders a Swing component into an SVGGraphics2D

The SVGGraphics2D is a very commonly used component from Batik, but there are three really common mistakes people make.

First, people assume that because they provide a `Document` to the `SVGGraphics2D` constructor, it automatically appends elements to that document. This is not the case, the `Document` passed in is used as an element factory. To get the content tree, you need to call "`getRoot()`."

Second, unless you set the SVG canvas's size, it defaults width and height to 100%. This means that the document has no preferred size; this is generally not what you want.

Third, if you are going to use the SVGGraphics2D with a Swing component, you should use the "`batik.svggen.SwingSVGPrettyPrint`" class; it handles many tricky issues with rendering Swing components.

Presenting SVG

Presenting SVG

Transcoder

Convert SVG to PNG/JPEG/TIFF/PDF

JSVGCanvas

Swing Component for viewing SVG

Rasterizing

The Bridge & Static/Dynamic Renderers

In this section of the talk, I will cover how to load SVG and prepare it for display.

The first method I will cover is using the Transcoder APIs to generate traditional Raster formats (JPEG, PNG, TIFF). The second method is using the JSVGCanvas to view the SVG interactively in a Swing component. The third and last will be using the various loading/processing packages directly to get a BufferedImage.

Transcoder API

The Basic API is trivial

```
JPEGTranscoder t = new JPEGTranscoder();
// Set the transcoding hints
t.addTranscodingHint
     (JPEGTranscoder.KEY_QUALITY,
      new Float(.8));
// Create the transcoder input/output
input = new TranscoderInput(svgURI);
output = new TranscoderOutput(ostream);

// transcode the input to output
t.transcode(input, output);
```

This example is taken straight from the Batik Web site which offers a pretty good tutorial of these APIs, which it would be pointless to recite here.

One thing to notice, however, are the transcoder hints. These are how you control the rasterization process. This lets you adjust the size and region rendered, as well as provide stylesheets, etc. Each transcoder can also provide format specific hints, such as the JPEG "quality" setting above.

These APIs are designed to make it easy to rasterize web icons, buttons, menu's, etc. This service is, in fact, available as an Ant task, and at least one of the Apache Site Builders (Forrest) uses Batik to generate PNG images from SVG.

JSVGCanvas

The simplest code is

```
JSVGCanvas jsvg = new JSVGCanvas();
parent.add(jsvg, BorderLayout.CENTER);
jsvg.setURI("http://[...]");
```

The display classes

- **`batik.swing.gvt.JGVTComponent`**
  Swing component for viewing GVT Tree
- **`batik.swing.svg.JSVGComponent`**
  "Bare Bones" Swing component for viewing SVG
- **`batik.swing.JSVGCanvas`**
  Swing component for viewing SVG
- **`batik.apps.svgbrowser.JSVGViewerFrame`**
  JFrame used by Squiggle browser.

For simple viewing of SVG, the JSVGCanvas is the simplest solution. The Canvas has methods to accept DOM Documents in addition to URI Strings.

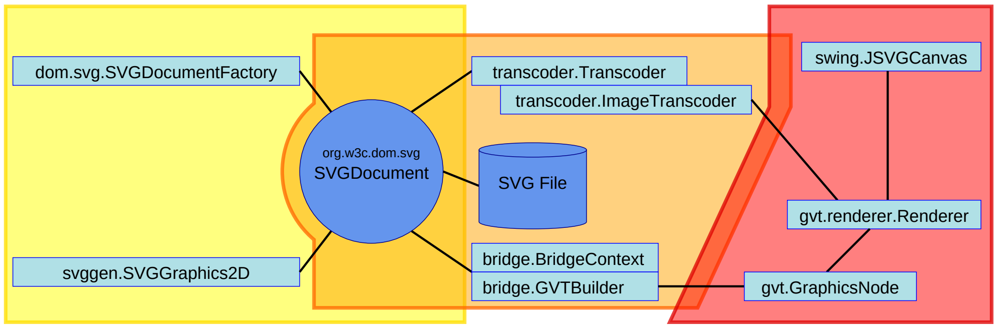
The JGVTComponent can be used to render a GVT tree. It may be helpful to know that "GVT" stands for "Graphics Vector Toolkit." The GVT is the package of classes that form the actual rendering tree for an SVG document. The JGVTComponent can only render static content. The GVT package will be discussed more a little later.

The JSVGComponent contains essentially all the display functionality for SVG, including support for dynamic updates. However, it has a minimal amount of "user interface."

The JSVGCanvas is essentially a drop in SVG Browser. This may offer more interaction capabilities than you really want/need the user to have (zoom, pan, link traversal, etc).

The JSVGViewerFrame is the frame used by the Squiggle browser. It has a few dependencies on other SVG browser classes and resources but may be a good starting point if you want to embed a complete SVG browser in your application.

Batik Architecture



This diagram shows the basic structure of Batik. On the left are the various XML handling classes. In the middle are the bridge/transcoding classes. On the right are the rendering and display classes.

It is worth noting that the XML/DOM classes are independent of the Rendering & Display classes and vise versa. This is accomplished by having the Batik DOM implementation define interfaces for sources of geometric information. The bridge package then implements these interfaces by using the GVT classes.

Rasterizing SVG (1/2)

```
ua     = new UserAgentAdapter();
loader = new DocumentLoader(ua);
ctx    = new BridgeContext (ua, loader);

svgDoc  = loader.loadDocument(url)
svgRoot = svgDoc.getRootElement();
builder = new GVTBuilder();
gvtRoot = builder.build(ctx, svgDoc);
```

Useful methods from BridgeContext

```
gvtNode = ctx.getGraphicsNode(domElem);
domElem = ctx.getElement(gvtNode);
```

Useful methods from GraphicsNode

```
Rectangle2D r = gvtRoot.getBounds();
gvtRoot.paint(Graphics2D);
```

```
import org.apache.batik.bridge.DocumentLoader; import org.apache.batik.bridge.BridgeContext;
import org.apache.batik.bridge.GVTBuilder; import org.apache.batik.bridge.UserAgentAdapter;
import org.apache.batik.gvt.GraphicsNode;
```

This example shows how to load an SVG document from a URI and then how to build the rendering (GVT) tree.

The User Agent class handles error display and lots of other rendering info. More on this class later.

The DocumentLoader loads SVG files from URIs. It also caches documents. Within the Transcoder and JSVGCanvas, the DocumentLoader is normally newly created for each top level document loaded. However, at least for JSVGCanvas, you can provide the DocumentLoader it uses - so referenced SVG files can be cached in memory across multiple top level documents.

The BridgeContext manages the processing of the SVG Document. For dynamic documents, it stores lots of useful information, such as the mapping between GVT nodes and DOM nodes.

The GVT's GraphicsNode interface can be used directly. In particular, it has a "`paint(Graphics2D)`", which will paint that node and its children to the given Graphics2D, as well as a "`getBounds()`" method - which returns the "painted" bounds of the node.

Rasterizing SVG (2/2)

```
DynamicRenderer renderer =
     new DynamicRenderer();
renderer.setTree(gvtRoot);
renderer.setTransform
    (ViewBox.getViewTransform
     (null, svgRoot, imgW, imgH));

renderer.updateOffScreen(imgW, imgH);
r = new Rectangle(0, 0, imgW, imgH);
renderer.repaint(r);
BufferedImage image =
     renderer.getOffScreen();
```

```
 import org.apache.batik.bridge.ViewBox;
import org.apache.batik.gvt.renderer.DynamicRenderer;
```

The ImageRenderers are designed to handle the rendering of a GVT tree to an offscreen image. For simple use cases, you could just create your BufferedImage and call "`createGraphics()`", and pass that to the Paint method on the GVT tree (with an appropriate transform).

The DynamicRenderer is actually faster for "one off" rendering scenarios. The StaticRenderer caches image data in tiles for future rendering requests. This can be a huge help when panning around a large static document but is useless if you are only going to render the document once or if the document will be changing.

The ViewBox class handles the "`viewBox`" and "`preserveAspectRatio`" attributes on the SVG element, and it constructs an Affine Transform to map the viewBox (if there is one) to the provided image W/H.

The "`repaint(Rectangle)`" method is what actually does the rendering of the SVG document. Calling "`getOffScreen()`" returns the offscreen image; the renderer is using for rendering. It is worth noting that the renderer can operate in single and double buffer modes.

The UserAgent Classes

There are two main UserAgent classes

- **`batik.bridge.UserAgent`**
- **`batik.swing.svg.SVGUserAgent`**

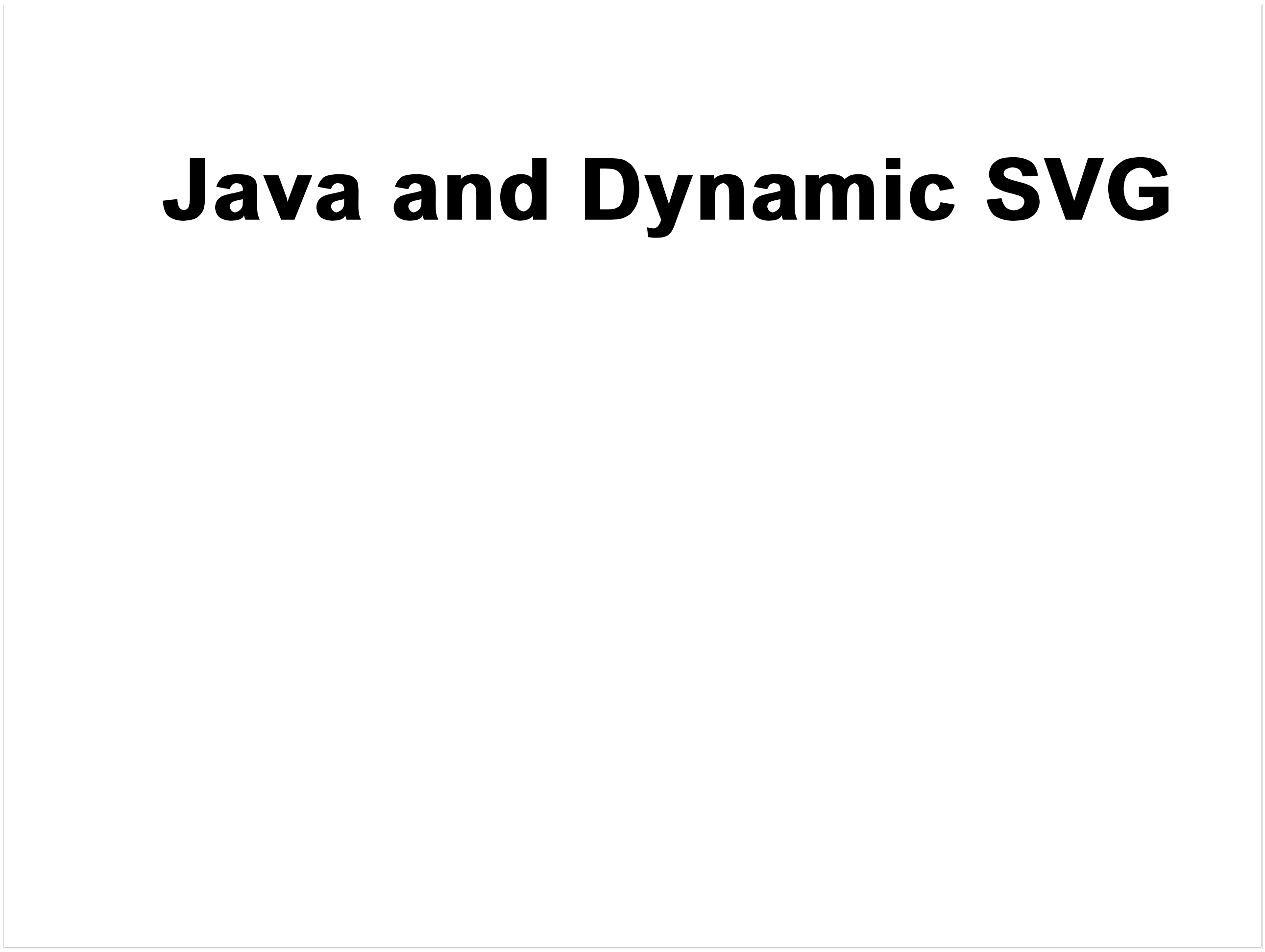
They have many common methods

- **`getXMLParserClassName()`**
- **`displayError/Message`**
- **`getPixelUnitToMillimeter()`**
- **`getDefaultFontFamily()`**
- **`getLanguages()`**
- **`getAlternate/UserStyleSheetURI()`**
- **`openLink`**

If you are planning to use Batik in your application, you should look at the UserAgent classes and customize them to your needs. The JSVGCanvas forwards most of the bridge UserAgent class calls to the SVGUserAgent.

Batik provides Adapter classes for both the Bridge UserAgent and the swing SVGUserAgent interfaces. These provide reasonable implementations for most of the methods. For the SVGUserAgent, two adapters are provided: one that sends error messages to System.err/out and one that outputs to dialogs.

The most common reasons to implement/override these interfaces is probably to control which XML parser is used.



Java and Dynamic SVG

Static vs. Dynamic

- **Static - `ALWAYS_STATIC`**
  No user interaction at all
  DOM & GVT essentially independent

- **Interactive - `ALWAYS_INTERACTIVE`**
  Links and cursors work
  DOM & GVT not fully linked

- **Dynamic - `ALWAYS_DYNAMIC`**
  DOM modification works
  DOM & GVT fully linked

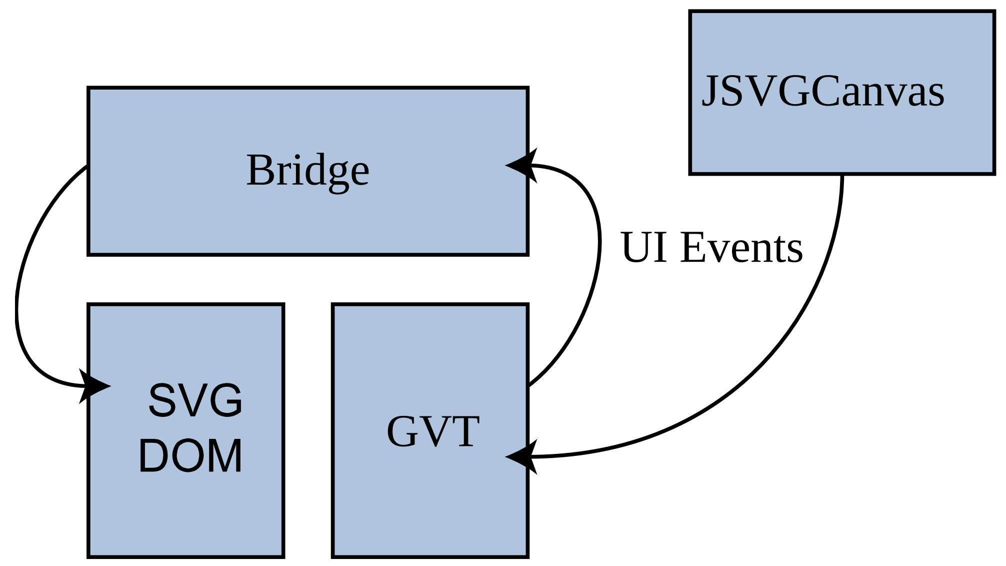
The JSVGCanvas and the BridgeContext support three classes of documents; static, interactive and dynamic. As you might expect, the more dynamic the document type the more overhead involved in constructing and maintaining the document.

It is at least theoretically possible to dispose of the DOM tree for static documents (all rendering state is held in the GVT tree). This never actually happens in normal usage, as the JSVGCanvas interface offers access to the document being displayed, and the transcoders store a reference as well. However, building a static document is still lighter weight than an interactive document (no listeners are registered).

For both Interactive and Dynamic, the DOM tree must stay around, because they use the DOM tree to handle event propagation - for links and cursors. However, an Interactive tree build is only slightly more expensive than a static one.

For a Dynamic build, there is quite a bit of additional overhead, as an additional object is created for each rendered Element in the DOM tree to track and update the GVT tree as changes happen. These objects are available from a Batik proprietary method called "`getSVGContext()`" on all SVG Elements. The `SVGContext` interface provides information to support the `SVGLocatable` interface. SVG nodes that have more sophisticated interfaces (text for example) define subclasses that provide these additional methods. The Bridge package then implements these interfaces using the GVT package.

UI Events



JSVGCanvas receives events from AWT. It forwards the events to the GVT tree, which calculates which graphic element is the target of the event and then dispatches the requisite event or events (mouseover/out when the graphic element changes). The event model here is very simple.

The Bridge listens for these events and creates similar events in the the associated DOM tree. For dynamic documents, these events trigger script or Java code to run.

DOM events are dispatched in two passes: capture and bubble (in that order). During the capture phase, the event trickles down the DOM tree from the root node to the target of the event.

DOM Events

Event listeners on any element in the DOM

```
document = canvas.getSVGDocument();
elem = document.getElementById("foo");
elem.addEventListener
   ([event-type], [listener], [use-capture])
```

- **`event-type` - The event to listen for**
  - **SVGScroll/Zoom/Resize - Canvas transform change**
  - **DOM UI - key/mouse/focus events**
  - **DOM Mutation - DOM tree change**
- **`listener` - Object to notify, interface**
  `org.w3c.dom.events.EventListener`
- **`use-capture` - The event phase to call listener**

DOM defines a fairly rich event structure. SVG extended this to include a bunch of additional event types, specific to SVG. The list above includes the most commonly used event types but one should read the SVG/DOM specifications to get a complete understanding of what events are available.

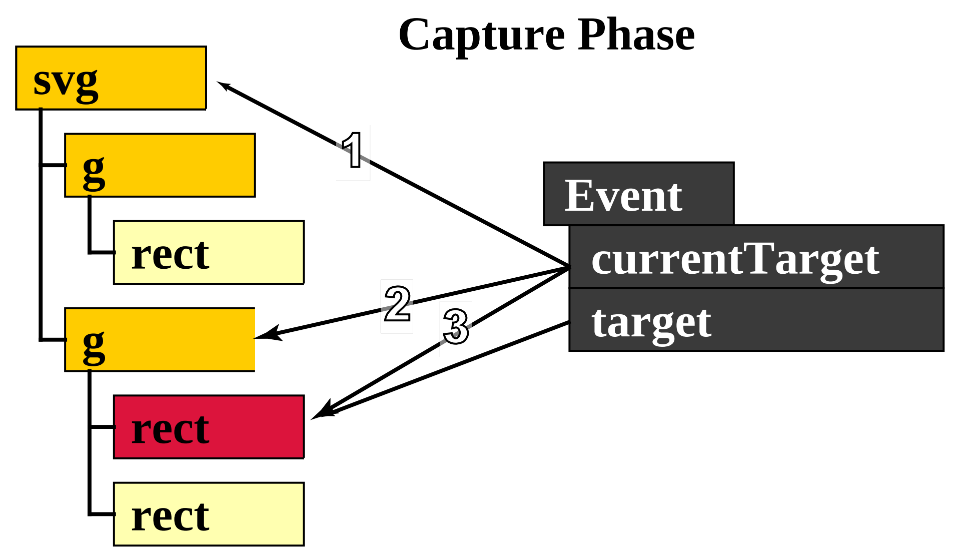
To register event listeners, first you need to get an element. The easiest way to get an element is with "`getElementById(String)`". This can be expensive, as it walks the tree looking for a matching element, so for "fixed" elements it is best to do the lookup once and cache the result.

Then you need to call "`addEventListener`" (you many need to cast the element to an `EventTarget`). The first parameter is a string indicating what event type to register for, the second is the listener object, and the third is when to call the listener - during capture or bubble.

DOM Event Capture

Event dispatched from root to target

```
elem.addEventListener
   ([event type], [listener], [use capture])
```

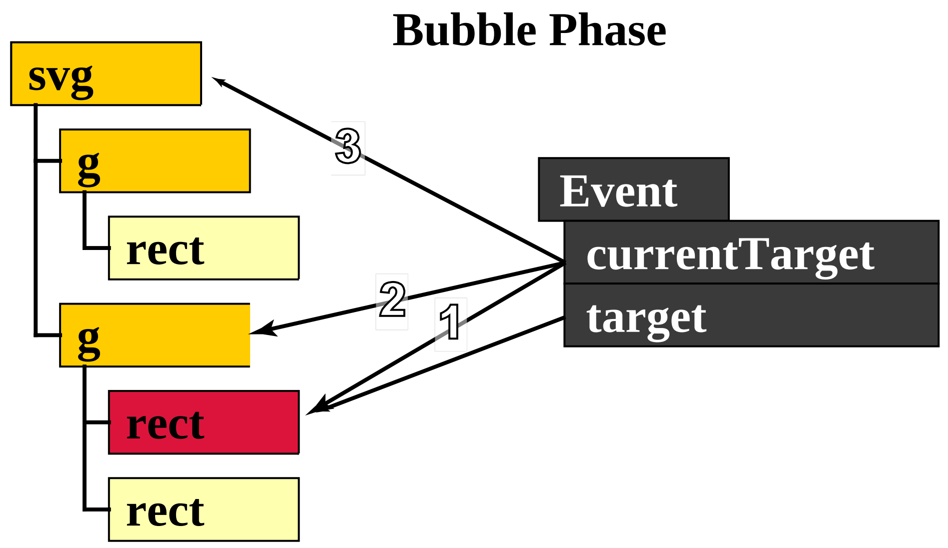


The third argument to addEventListener controls if the listener is registered for capture or bubble. All of the event attributes for SVG are registered on bubble. For Java programmers, this doesn't mean much as the event attributes can only contain "script" content, not Java code.

For those unfamiliar with DOM events, they have two similar attributes: "`target`" and "`currentTarget`." The "`target`" attribute always points at the element that originated the event. The "`currentTarget`" is updated as the event is dispatched to the various elements in the document tree.

DOM Event Bubble

This is when event "attributes" are triggered
`event.stopPropagation()`
`event.preventDefault()`



This just shows the order of notifying listeners is just reversed for the Bubbling phase.

The DOM event object has two useful methods called "`stopPropagation()`" and "`preventDefault()`". "`stopPropagation()`", prevents the event from being dispatched to any other elements in the tree although, dispatch completes for the current element. "`preventDefault`" for cancelable events prevents the "default action" from occuring (the most common example of this is link activation for click events) the event does continue to propagate.

At first blush, this can seem a little excessive or an odd system for event dispatch. However, it in fact mirrors many of the standard procedures for event dispatch in most GUI systems, where the frame and other container objects get first chance at an event as it travels down the component tree. If the leaf component does not want the event, it moves up the component hierarchy back to the window frame.

The UpdateManager

- **Needed for Java to modify the DOM**
- **Serializes access to DOM in dynamic documents**
- **Triggers repaints when rendering tree modified**
- **Needed to ensure "coherent" view of Document**
  Creating a Rectangle takes 6 or more DOM calls!
- **Becomes available after the first GVT**
  rendering completes.
- **`invokeAndWait` can deadlock from Swing thread!**

```
UpdateManager um;
um = canvas.getUpdateManager();
um.getUpdateRunnableQueue().invokeLater
      (new Runnable() {...});
```

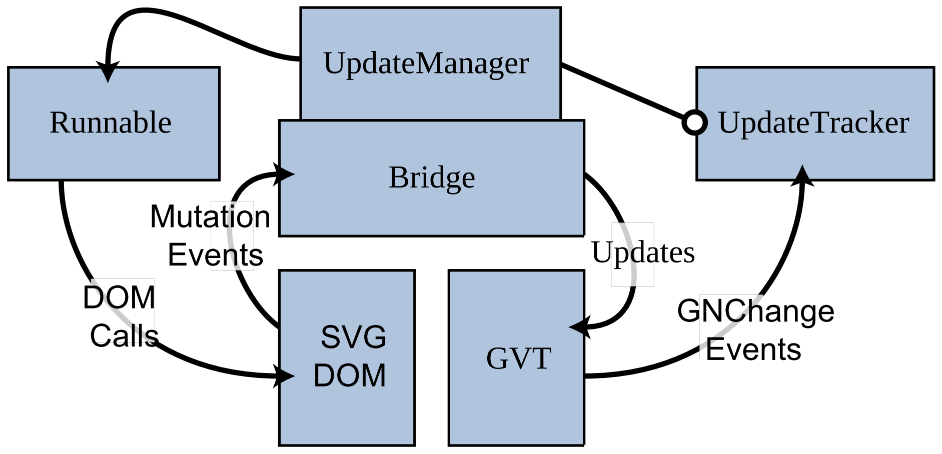
If you want to use Java code to interact with the SVG DOM, you **must** go through the UpdateManager. This is the most common mistake people make. The most common symptom of this mistake is that the canvas does not update unless you move the cursor over the canvas.

The Update Manager is needed to serialize access to the DOM. It is essentially impossible to synchronize the DOM, as many atomic actions require numerous calls to complete. The example I always give is that creating an SVG rectangle requires around 7 calls, "`createElementNS`", set x, y, width, height, and fill/stroke colors, then finally adding the element to the DOM tree.

Rather than introduce something like "lock", "unlock" calls, which are really error prone and not part of DOM, we took an approach similar to Swing and restrict, by convention, all access to the DOM to one thread of execution.

The DOM event listeners should always be called from the Update Manager thread, so you don't have to do anything special from within a DOM event listener.

The UpdateManager



This diagram shows how a Runnable, given to the Update Manager typically interacts with Batik.

The UpdateManager runs the runnable which typically makes a bunch of DOM calls, usually causing mutation events, which the bridge is listening for. When the bridge receives mutation events, it updates the GVT tree (setting path, paint, or whatever).

When nodes in the GVT tree are modified, they generate GraphicsNodeChangeEvents, which the UpdateTracker listens for. The UpdateTracker accumulates a list of "dirty regions" that need repainting.

When the Runnable finishes, the UpdateManager queries the UpdateTracker to see if there are any dirty regions; if their are, it triggers a repaint of the affected areas in the JSVGCanvas.

This last part is why directly modifying the DOM does not cause the screen to be updated. There is no trigger for the repaint (no one checks the UpdateTracker) until a mouse event is sent to the DOM in an UpdateManager runnable.

Interactors/Overlays

Interactors/Overlays

Overlay

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

- **An item that gets drawn on top of SVG**
- **Example: text selection**

Interactor

```
boolean startInteraction(InputEvent ie);
boolean endInteraction();
```

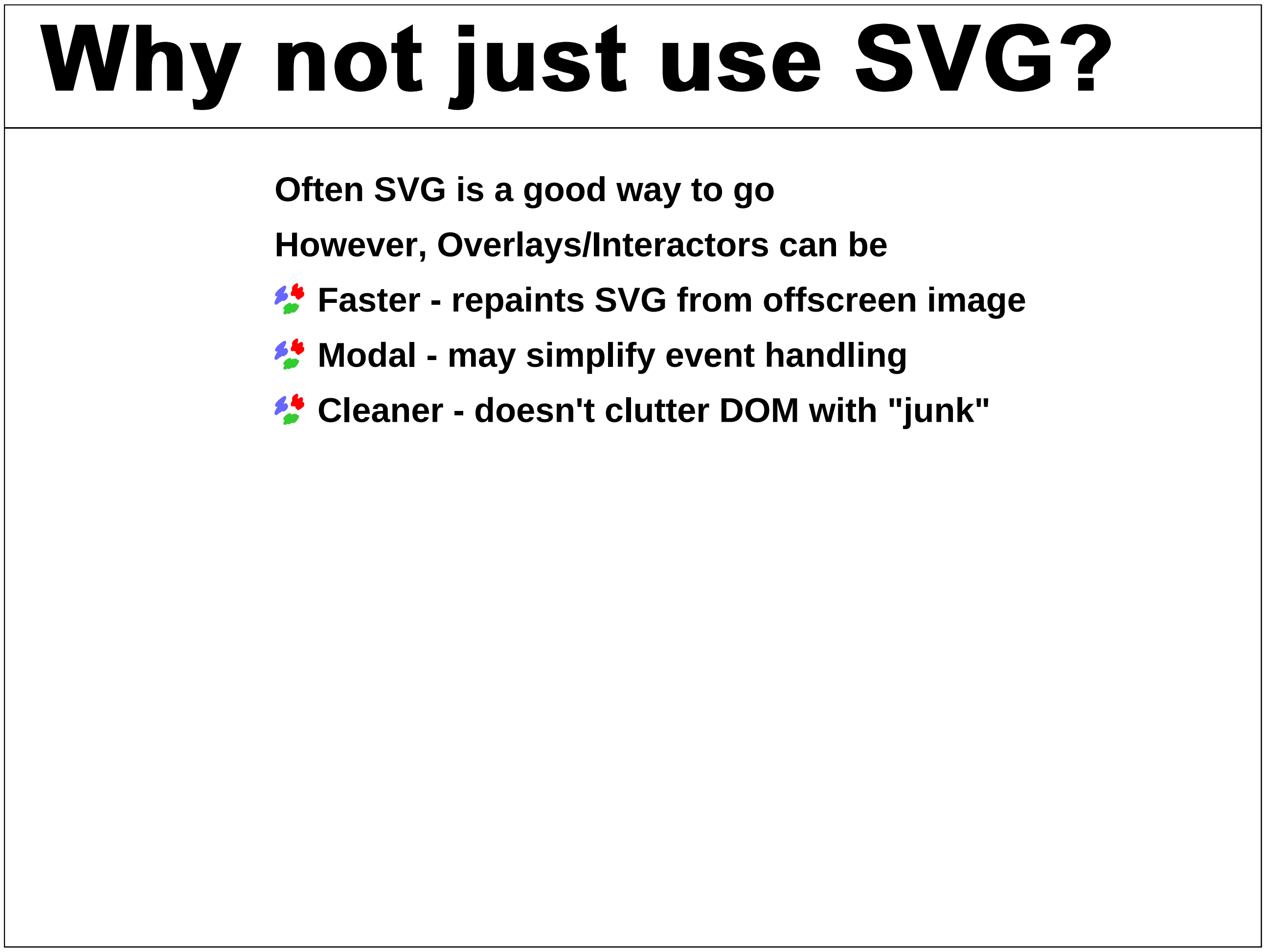
- **A mechanism for modal interaction with the user**
- **Event dispatch to SVG is suspended while active**

The JSVGCanvas has two mechanisms outside of standard SVG for user interaction and for decorating an SVG document: Interactors and Overlays.

The Overlay interface is really simple; it has one method "`paint(Graphics g)`" that gets called after any part of the SVGDocument is repainted. This is typically used to simply highlight a region on the canvas but can be used to draw almost anything.

The Interactor interface is also fairly simple; it has methods to indicate that it wants to start/end a modal interaction, and it extends AWT Key and Mouse Listeners. It is common for an Interactor to have an associated Overlay that it uses to display user feedback.

These interfaces are used for the built-in pan, zoom, rotate "tools", and text selection.



Why not just use SVG?

Often SVG is a good way to go

However, Overlays/Interactors can be

- **Faster - repaints SVG from offscreen image**
- **Modal - may simplify event handling**
- **Cleaner - doesn't clutter DOM with "junk"**

In general, using SVG directly is the most flexible and powerful solution and, if done with JavaScript, can be used with any SVG viewer. However, it has some drawbacks, perhaps the largest is speed.

When an area of the SVG tree changes, everything that intersects with that area must be repainted from the original document. However, because the overlay is always drawn on top of the SVG, it can cheat, when the overlay changes it can just repaint the SVG from the offscreen buffer and then redraw the overlay. For complex SVG documents, this is **much** faster.

Interactors are inherently modal. By contrast, the DOM event model is normally nonmodal (you can make it modal but it takes at least a little work). So can be easier to use an Interactor than to try and deactivate the normal processing.

Finally, it can be cleaner; it does not involve adding event handlers and new elements to the DOM tree. Thus, your document can continue to be the graphics and not get overly mixed up with the UI.

Add Overlay/Interactor

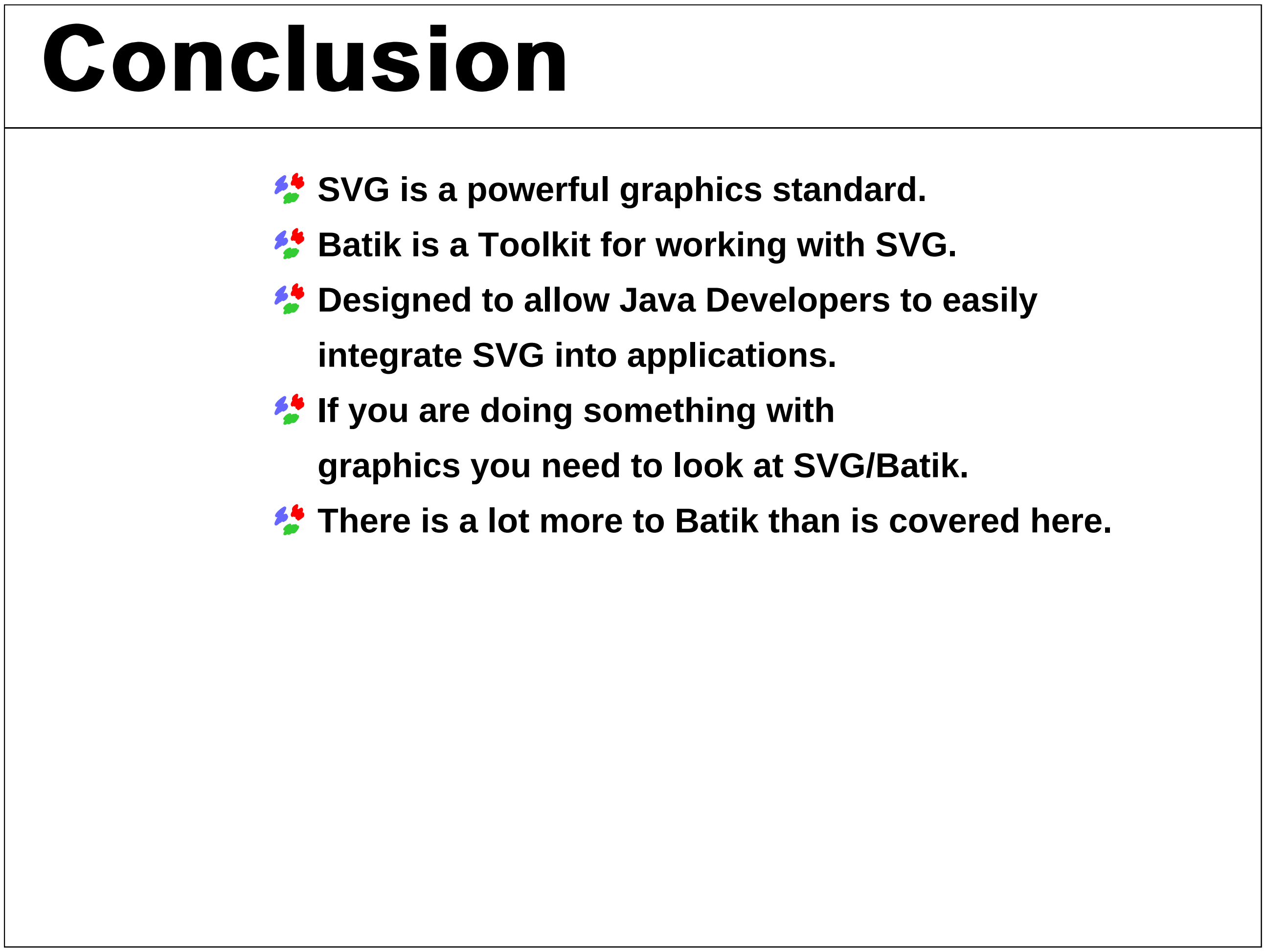
```
class JGVTComponent {
    List getOverlays()

    void setDisableInteractions(boolean b)
    List getInteractors()
}
```

"`getOverlays()`" returns a standard List containing all of the overlays. To add your overlay, just add it to the list. This also allows you to order the Overlays; they are drawn in the order of the list.

Similarly, "`getInteractors()`" returns a standard List containing all of the Interactors. To add your Interactor, just add it to the list. Your Interactor will then have it's "`startInteraction`" method called for each InputEvent. If it returns true, then all events will be routed to your Interactor. This continues until you return true from "`endInteraction()`," which is called after each event is dispatched.

When you disable interactions, you are shutting off all the Interactors.

When your Overlay changes, it is up to you to request a repaint of the affected region of the component. You can use the standard AWT calls to do this: "`canvas.repaint(Rectangle)`."



Conclusion

- SVG is a powerful graphics standard.
- Batik is a Toolkit for working with SVG.
- Designed to allow Java Developers to easily integrate SVG into applications.
- If you are doing something with graphics you need to look at SVG/Batik.
- There is a lot more to Batik than is covered here.

SVG is a really powerful standard. It does more than just about any previous graphics format, and it does it in a nice open way using existing technologies (XML, DOM, Web).

Batik makes it trivial to integrate this powerful standard into your applications. This can be for something as simple as Pie Chart generation or as complex as gene-sequence display and analysis.

Even if you don't think you need SVG, you may want to look at Batik as it embeds a really powerful graphics engine, doing many things that are needed for any application working with complex graphics (event handling, repaint management, etc).

Batik is a large toolkit, and there are many parts that are not really covered in this presentation (SVG Generator, font converter, etc). So, if you didn't see what you were interested in, please check out the Batik Web site and/or the mailing lists.

Additional Info

- Batik: http://xml.apache.org/batik
- batik-users@xml.apache.org
- batik-dev@xml.apache.org

- SVG: http://www.w3.org/Graphics/SVG
- XML, XSL, FOP, ...: http://www.w3.org
- XML and Java tools: http://www.xml.apache.org

- deweese@apache.org

The first three references on this slide are the Batik Web site URL and the two Batik mailing lists. There is an archive of email messages at: `http://nagoya.apache.org/eyebrowse`

The SVG standard and lots of other information are available from the W3C's Web site.