TinyLine SVG Programming Guide

Version 2.6

Latest version: http://www.tinyline.com/

Change history

November, 2006	Version 1.11	Initial document release
November, 2007	Version 2.0	Updated to cover 2.0 API
November, 2008	Version 2.1	Updated to cover 2.1 API
April, 2012	Version 2.5	Updated to cover 2.5 API
July , 2015	Version 2.6	Updated to cover 2.6 API

Table of Content

1. TinyLine SVG Overview

1.1 What is Mobile SVG

SVG 1.1, SVG 1.0 and SVG Mobile Profiles are Web standards (W3C Recommendations). Work continues on SVG 1.2 and future profiles for Mobile and Printing.

In a relaxed way, Mobile SVG describes images in XML as shapes with attributes like colors, sizes, etc. In compare with other vector graphics formats, the advantages are that Mobile SVG is XML based, open and designed for wireless transmission and display. When a bitmap format, like JPEG or PNG, is perfect for photography or icons, Mobile SVG is more suitable for dynamic and interactive graphics.

Location-based and field services are well-suited to Mobile SVG because of the ability to zoom in on images without loss of quality. SVG maps with animated objects and hyperlinks provide views of different areas of maps or topographical layers. Field services also benefit from Mobile SVG by using technical drawings that can be viewed in full or in detail. Games, cartoon animations could be developed using Mobile SVG.

1.2 What is TinyLine SVG Toolkit

TinyLine SDK is the mobile graphics engines and framework for Java platform. Using TinyLine SDK developers are easily able to incorporate high quality, scalable and platform-independent graphics into their Java applications.

TinyLine SDK includes TinyLine SVG Tiny 1.1+ engine for Java platform.

1.3 Key features

- SVG Tiny 1.1+ engine
- Supports SVG fonts, raster image and text elements, paths.
- Supports SMIL animations and events.
- Allows both textual and gzipped SVG streams.
- Compact code (around 100K in jar file).
- Easy to use API.

TinyLine SVG API consists of the three packages

- com.tinyline.svg
- com.tinyline.tiny2d
- com.tinyline.util

All packages are extremely portable across Java flavors.

Here the set of Java classes TinyLine API uses:

- java.lang.Exception
- java.io.InputStream
- java.lang.Object
- java.lang.System
- java.lang.Throwable

2. TinyLine SVG API

Package com.tinyline.tiny2d

The TinyLine 2D implements an advanced 2D graphics and imaging.

Tiny2D	The Tiny2D defines a graphics context that allows an application to draw shapes, images and texts onto a TinyBuffer object.
TinyBuffer	The TinyBuffer class represents a rectangular array of pixels.
TinyColor	The TinyColor class defines colors in the ARGB color space.
TinyFont	The TinyFont class defines a collection of glyphs together with the information necessary to use those glyphs.
TinyGlyph	The TinyGlyph class specifies a glyph representing a unit of rendered content within a font.
TinyHash	The TinyHash class implements a hashtable, which maps keys to values.
TinyMatrix	The TinyMatrix class represents a 2D affine transformation matrix.
TinyNumber	The TinyNumber class wraps a value of the (double) fixed point type in an object.
TinyPaint	The TinyPaint class defines paint servers in the ARGB color space.
TinyPath	The TinyPath class represents a geometric path constructed from straight lines, and quadratic and cubic (Bézier) curves.
TinyPoint	The TinyPoint class specifies a point representing a location in (x, y) coordinate space specified in fixed point precision.
TinyProducer	The TinyProducer interface provides an interface for objects which can produce the image data from the TinyBuffer object (pixels buffer).

TinyRect	The TinyRect class specifies an area in a coordinate space that is enclosed by the TinyRect object's top-left point (xmin, ymin) and down-right point (xmax, ymax) in the coordinate space.
TinyState	A TinyState object encapsulates state information needed for the basic rendering operations.
TinyStop	The TinyStop class implements a gradient stop.
TinyString	The TinyString class represents character strings.
TinyVector	The TinyVector class implements a growable array of objects.

Package com.tinyline.util

This package contains miscellaneous utility classes.

GZIPInputStream	This class implements a stream filter for reading compressed data in the GZIP format for Java platform.
TinyInputStream	A data input stream lets an application read primitive TinyLine 2D data types from an input stream.
TinyOutputStream	A data output stream lets an application write primitive TinyLine 2D data types to an output stream.

Package com.tinyline.svg

This package implements a SVG Tiny 1.1+ language.

Interfaces:

AnimationCallback	The AnimationCallback interface.
ImageLoader	The ImageLoader interface
XMLHandler	The XML callbacks interface.
XMLParser	The interface for parsing XML documents using callbacks.

Classes:

SMILTime	The SMILTime class is a datatype that represents times within the timegraph.
SVG	The SVG class defines SVG Tiny constants such as elements, attributes and values as for SVG Tiny specification.
SVGAnimationElem	The SVGAnimationElem class implements animation elements

	which are defined in SMIL Animation and 3GPP SMIL Language Profile specifications.
SVGAttr	The SVGAttr class implements the SVGT attributes parser.
SVGDocument	The SVGDocument class implements the header object in the SVG object hierarchy.
SVGEllipseElem	The SVGEllipseElem class implements the 'circle' and 'ellipse' elements.
SVGFontElem	The SVGFontElem class implements the 'font' element.
SVGFontFaceElem	The SVGFontFaceElem class implements the 'font-face' element.
SVGGlyphElem	The SVGGlyphElem class implements the 'glyph' element.
SVGGradientElem	The SVGGradientElem class implements the 'linearGradient' or the 'radialGradient' element.
SVGGroupElem	The SVGGroupElem class implements the container element.
SVGImageElem	The SVGImageElem class implements the 'image' element.
SVGLineElem	The SVGLineElem class implements the 'line' element.
SVGMPathElem	The SVGMPathElem class implements the 'mpath' sub-element.
SVGNode	The SVGNode class implements the base class for all elements in the SVG language.
SVGParser	The SVGParser class implements XMLHandler callbacks API for parsing the SVGT input stream.
SVGPathElem	The SVGPathElem class implements the 'path' element.
SVGPolygonElem	The SVGPolygonElem class implements the 'polyline' and 'polygon' elements.
SVGRaster	The SVGRaster class rasterizes the tree of SVGNodes onto the pixel buffer.
SVGRect	The SVGRect class implements the basic type 'rectangle'.
SVGRectElem	The SVGRectElem class implements the 'rect' element.
SVGStopElem	The SVGStopElem class implements the 'stop' element.
SVGSVGElem	The SVGSVGElem class implements the 'svg' element.
SVGTextElem	The SVGTextElem class implements the 'text' element.
SVGUnknownElem	The SVGUnknownElem class implements the 'unknown' element.
SVGUseElem	The SVGUseElem class implements the 'use' element.
TinyUtils	The TinyUtils class implements parsers with support for basic data types and objects.

The best way to learn and understand TinyLine API is to look at examples!

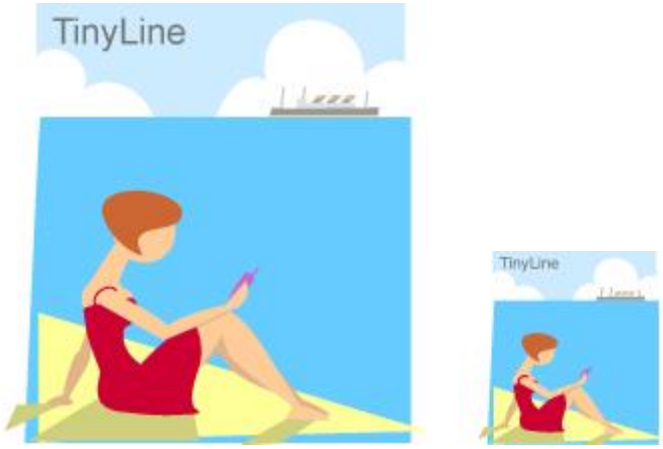
3 Viewer Example

3.1 Overview

The Viewer example is a static SVG Tiny viewer that:

- Parses and processes the static language features of SVG Tiny
- Supports zooming and panning of SVG Tiny documents

In other words, there are no animations and links. You can read, parse and display SVG Tiny document and also zoom and pan it.



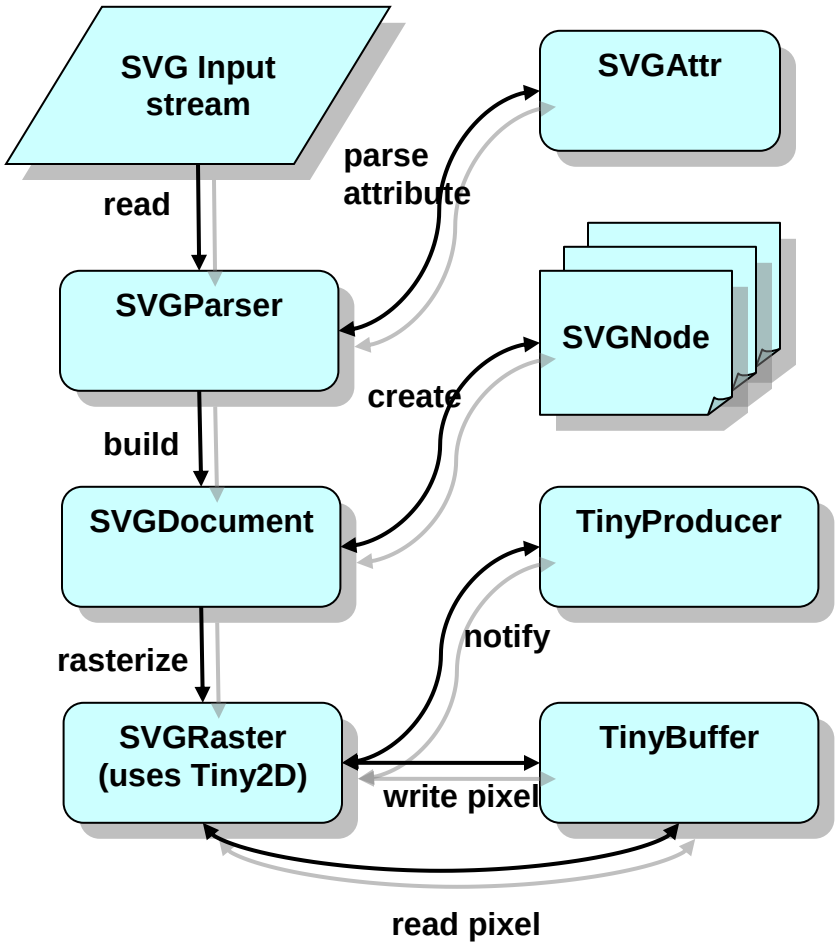
The Viewer Example includes the following classes:
Java:

```
tinyapp.Viewer
tinyapp.ViewerCanvas
com.tinyline.app.ImageConsumer
com.tinyline.app.PPSVGImageProducer
```

Please note, that we use Java source as a reference. But it's easy to see that there is not a big difference in code for other platforms as well.

Let's see how the Viewer application draws SVG Tiny documents. Here we describe how SVG Tiny input stream turns into `SVGNode` tree and then displays onto the Java device screen.

3.2 SVG Pipeline



During the parsing process `SVGParser` callbacks build internal OM (`SVGNode` tree) from the input SVG stream. The `SVGDocument` has the `SVGNode` tree root as its member.

The `SVGRaster` translates `SVGNode` objects into graphics primitives and draws them (rasterization process) using `Tiny2D` onto the `TinyBuffer` object.

The `TinyProducer` interface gets a notification that new pixels of the `TinyBuffer` object are ready to be send onto physical screen.

Before the pipeline can work we need to set up and create its elements
(See ViewerCanvas for example)

The init process:
Here we need to create all objects in the pipeline and define needed hooks:

```
// 1. Creates the pixels buffer (width X height)
TinyBuffer buffer = new TinyBuffer();
buffer.width = width;
buffer.height = height;
buffer.pixels32 = new int[width * height];

// 2. Creates the SVGRaster graphics
raster = new SVGRaster(buffer);
t2d = raster.getTiny2D();

// 3. Creates the TinyProducer and links it to the Tiny2D
imageProducer = new PPSVGImageProducer(t2d);
imageProducer.setConsumer(this);
t2d.setProducer(imageProducer);

// 4. Sets the ImageLoader implementation needed for
// loading bitmaps
SVGImageElem.setImageLoader(this);
```

The parsing process:

```
// 1. Create an empty SVGT document
SVGDocument doc = raster.createSVGDocument();

. . .

// 2. Read and parse the SVGT stream
TinyBuffer pixbuf = raster.getTiny2D().getTarget();
// 3. Create the SVGT attributes parser
SVGAttr attrParser = new SVGAttr(pixbuf.width, pixbuf.height);
// 4. Create the SVGT stream parser
SVGParser parser = new SVGParser(attrParser);
// 5. Parse the input SVGT stream parser into the document
parser.load(doc,is);
```

The rasterizing process:

```
// 1. Set the SVGT document to be drawn
raster.setSVGDocument(document);
// 2. Invalidate the current clip (i.e. set
// the clip to the pixel buffer bounds)
t2d.invalidate();
// 3. Update pixels under the current clip.
//    At this step all SVGNode objects are to be
//    drawn onto the pixel buffer.
raster.update();
// 4. Call TinyProducer interface to notify
//    consumers that there are new pixels
```



```
        t2d.sendPixels();
```

The `SVGRaster` class rasterizes the tree of `SVGNodes` onto the `TinyBuffer`. This process is called a “rendering process”.

3.3 Rendering process

Each `SVGNode` "paint" operation draws over some area of the pixel buffer. When the area overlaps a previously painted area the new paint partially or completely obscures the old. When the paint is not completely opaque, the result is defined by the simple alpha blending rules.
Nodes in the SVG tree have an implicit z drawing order. Subsequent nodes are painted on top of previously painted nodes.
When a graphic object is rendered, the geometry, image, and attribute information are combined to calculate which pixel values must be changed

paint() vs. update()

What is the purpose of having separate paint() and update() methods? A call to update() implies that the area defined by the device clip rectangle is "damaged" and must be completely repainted or updated, however a call to paint() does not imply this, which enables to do incremental painting.

3.4 Zoom and Pan

The `SVGSVGElem` class has ‘viewXform’ member that defines the SVG document transform from viewBox (SVG/USER COORDINATES) to viewPort (DEVICE COORDINATES).

In order to shift or ‘pan’ the SVG content onto (x,y) distance we call the following function:

```
/**
* Pans the current SVGT document.
* @param x The distance on X coordinate.
* @param y The distance on Y coordinate.
*/
 public void pan(int x, int y)
 {
 //    System.out.println("pan x=" + x + " y=" +y);
     SVGSVGElem root = (SVGSVGElem) this.raster.root;
     TinyMatrix xform = (TinyMatrix)root.cameraXform.data[1];
     int scale = (xform.a);
     // Scale pan distances according to the current
```



```
        // scale factor. Change the current viewport.
        xform = (TinyMatrix)root.cameraXform.data[0];
        xform.tx -= x << Tiny2D.FIX_BITS;
        xform.ty -= y << Tiny2D.FIX_BITS;
        root.createOutline();

        // Fire the update event
         // Fire the update event
         Tiny2D t2d = this.raster.getTiny2D();
         root.createOutline();
         t2d.invalidate();
        // 7. Update pixels under the current clip.
        raster.update();
        // 8. Notify TinyProducer about new pixels.
        t2d.sendPixels();
    }
```

The similar flow you can find for the zoom operation.

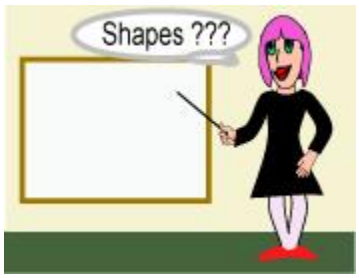
```
/**
 *  Zooms in and out the current SVGT document.
 *  @return true if the zoom is allowed; otherwise return false.
 */
   public boolean zoom(int direction)
   {
       SVGSVGElem root = (SVGSVGElem) this.raster.root;
       TinyMatrix xform = (TinyMatrix) root.cameraXform.data[1];
       int scale = (xform.a);

       // zoom in '0' size / 2
       if (direction == 0)
       {
           scale >>= 1;
           if (scale < 1 << 8)
           {
               scale = (1 << 8);
           }
       }
       else //zoom out size * 2
       {
           scale <<= 1;
           if (scale > 1 << 24)
           {
               scale = (1 << 24);
           }
       }
       xform.scale(scale, scale);

       // Fire the update event
       Tiny2D t2d = this.raster.getTiny2D();
       root.createOutline();
       t2d.invalidate();
       raster.update();
       t2d.sendPixels();
       return true;
    }
```

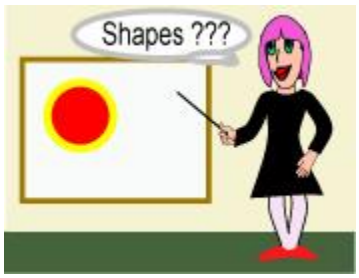

The next example application is more interesting.

4 Shapes Example

4.1 Overview

In this example we want to load an SVG Tiny stream and then change it! We are going to add a couple of new `SVGNode` objects to the current SVG tree.





The Shapes Example includes the following classes:

Java:
`tinyapp.Shapes`
`tinyapp.ViewerCanvas`
`com.tinyline.app.ImageConsumer`
`com.tinyline.app.PPSVGImageProducer`

In this example we want to load an SVG Tiny stream and then change it! We are going to

The `Shapes()` and `startApp()` have a standard TinyLine stuff in order to create pipeline objects, hook them together, and load a static SVG Tiny stream.

The `Circle()` and `Rect()` do all the business. Before we take a closer look at them, we need to say a couple of words about `SVGNode` class and its derived classes.

4.2 Change SVGNode properties

The `SVGNode` class implements the base class for all elements in the SVG language. What is important here for our example, its how you can change `SVGNode` object properties. (The same apply for all `SVGNode` based classes).

There are two different ways to change `SVGNode` object properties:
1. The Direct TinyLine API.

```
rect.y =  80 << Tiny2D.FIX_BITS;
```


2. The OM (Object Model) TinyLine API

```
rect.setAttribute(SVG.ATT_X, 80 << Tiny2D.FIX_BITS);
```

You can use both of them.
Here is the sample code using direct API:

```
// Use the Direct TinyLine API to change properties
rect.id = new TinyString("myrect".toCharArray());
rect.x = 120 << Tiny2D.FIX_BITS;
rect.y = 80 << Tiny2D.FIX_BITS;
rect.width = 40 << Tiny2D.FIX_BITS;
rect.height = 40 << Tiny2D.FIX_BITS;
rect.fill   = yellowColor;
rect.stroke = navyColor;
rect.strokeWidth = 4 << Tiny2D.FIX_BITS;
```

Before your changes become valid you need to create the element outline:

```
// Create the outline.
rect.createOutline();
// Repaint the damaged area.
t2d.setClip(rect.getDevBounds(raster));
raster.update();
t2d.sendPixels();
```

4.3 Search, add, remove SVGNodes

The following code searches the SVG document for the rect element with “myrect” id. If such element is found we remove it, otherwise we create a new SVG rect element with “myrect” id and add it to the SVG document after the element which id is “whiteboad”.

```
// Get the raster
SVGRaster   raster   = canvas.raster;
// Get the SVGT document
SVGDocument document = raster.getSVGDocument();
// Get the graphics
Tiny2D t2d = raster.getTiny2D();
// Get the root of the SVGT document
SVGSVGElem root = (SVGSVGElem)document.root;

// If there is a node with id 'myrect' then remove it.
// Otherwise create and add it.
SVGNode node =
    SVGNode.getNodeById(document,
               new  TinyString("myrect".toCharArray()));
if(node != null)
{
```



```
    // Get the parent
    SVGNode parent = node.parent;
    int index = parent.children.indexOf(node,0);
    // Remove the child node and update the ID cache table
    parent.removeChild(index);
    document.idTable.clear();
    document.addIds(document.root);
    // Repaint the damaged area.
    t2d.setClip(node.getDevBounds(raster));
    raster.update();
    t2d.sendPixels();
    return;
}

// Create a new rect element
SVGRectElem rect =
  (SVGRectElem)document.createElement(SVG.ELEM_RECT);

// Add the rect element AFTER the whiteboard element
node = SVGNode.getNodeById(document,
          new TinyString("whiteboard".toCharArray()));
if(node != null)
{
    // Get the parent
    SVGNode parent = node.parent;
    int index = parent.children.indexOf(node,0);
    // Add the child node and update the ID cache table
    parent.addChild(rect, index + 1);
    document.idTable.clear();
    document.addIds(document.root);
}
```

Now lets talk about dynamic SVG content.

5. TinyLine Example

5.1 Overview

The TinyLine Example is a SVG Dynamic Player. Here “dynamic” means that it supports user interaction, SMIL animations and hyper linking.

(c) Edik Mitgartz

The TinyLine Example consist of the `com.tinyline.app` classes.
Some of the classes are more platform oriented, so we leave them alone and discuss the core classes that are more interesting:

Java
`com.tinyline.app.PPSVGCanvas`
`com.tinyline.app.PlayerListener`
`com.tinyline.app.SVGEvent`
`com.tinyline.app.SVGEventQueue`
`com.tinyline.app.TinyLine`

In the Viewer Example we have a very simple flow. All actions are initiated by the user. In the case of dynamic SVG content we have a difficult situation. The user can pan and zoom SVG content, he can also press links or wants to stop or resume animations. Animations can trigger other animations or start or stop themselves depending on time.

5.2 Reactor framework

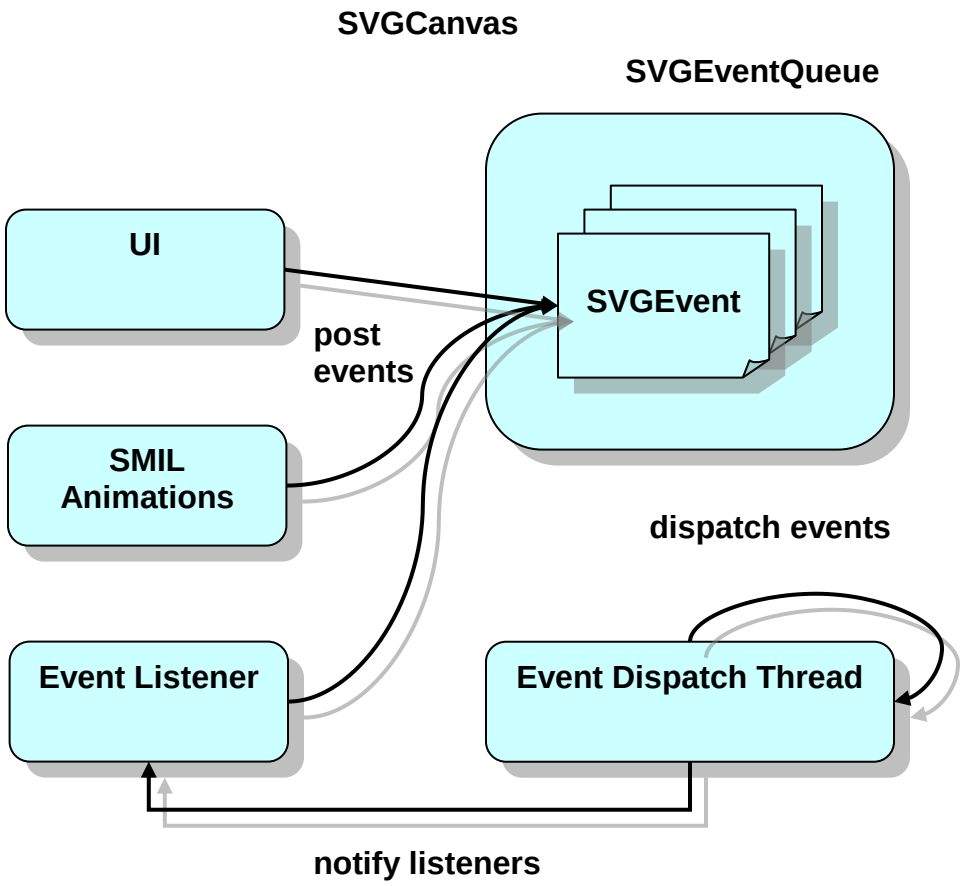
To solve these difficulties we use the Reactor framework. In our context, the Reactor framework is responsible for

- Detecting the occurrence of events from various event sources
- Demultiplexing the events to their preregistered event listeners
- Dispatching events to the listeners to process the events in an application-defined manner

The event objects (`SVGEvent`) are placed onto a single event queue (`SVGEventQueue`) ordered by their entry time. While that happens, a separate thread, called the event-dispatch thread (thread is a member of `SVGCanvas`), regularly checks the event queue's state. As long as the event queue is not empty, the event-dispatch thread takes event objects from the queue one by one and sends them to the interested parties

(`PlayerListener`). Finally, the interested parties react to the event notification by processing logic such as event handling.

Since the event-dispatch thread executes all event-processing logic sequentially, it avoids undesirable situations such as simultaneously changing the same `SVGNode` property.

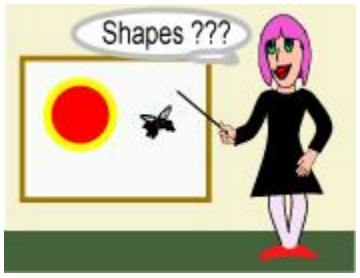


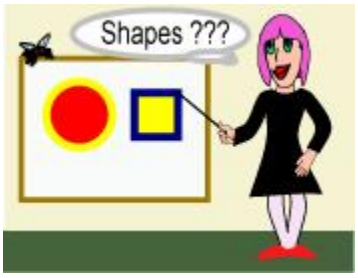
6. Shapes2 Example

6.1 Overview

The event model, which we saw at its simplest in the preceding example, is quite powerful and flexible. Any number of event listener objects can listen for all kinds of events.

The Shapes2 Example shows how to write your own event listener and how to manipulate `SVGNode` tree when you have SMIL animations running.





6.2 Custom SVGEvent

First, we define our own SVGEvent code equals 100.

```
public static final int USER_EVENT     = 100;
```

Then, we need to make standard arrangements with the default SVG font and also we need to register the default event listener, which allows running SMIL animations.

```
// Load the default SVG font
SVGDocument doc =  canvas.loadSVG("/tinyline/helvetica.svg");
SVGFontElem font = SVGDocument.getFont(doc,
                   SVG.VAL_DEFAULT_FONTFAMILY);
SVGDocument.defaultFont = font;

// Add the default events listener
PlayerListener defaultListener = new PlayerListener(canvas);
canvas.addEventListener("default", defaultListener, false);
```

Next, we register our own ShapesListener and start the events dispatching thread.

```
// Add the user defined (custom) events listener
ShapesListener shapesListener = new ShapesListener(canvas);
canvas.addEventListener("shapes", shapesListener, false);

// Start the event queue
canvas.start();
// Load the SVGT image
canvas.goURL("/svg/shapes.svg");
```

UI commands post our custom events into the event queue

```
public void commandAction(Command c, Displayable s)
{
    ///System.out.println("Command " +c);
    SVGEvent event;
    if(c == menuCircle)
    {
       event = new SVGEvent(USER_EVENT,
                   new TinyNumber(SHAPE_CIRCLE ));
```



```
            canvas.postEvent(event);
        }
        else if(c == menuRect)
        {
            event = new SVGEvent(USER_EVENT,
                        new TinyNumber(SHAPE_RECT ));
            canvas.postEvent(event);
        }
```

And, all the logic is done in the `ShapesListener`.

7. References

1. Mobile SVG Profiles: SVG Tiny and SVG Basic, Tolga Capin (Nokia), editor, W3C, 14 January 2003. Available at (http://www.w3.org/TR/SVGMobile/).

2. Scalable Vector Graphics (SVG) Version 1.1 Specification , Dean Jackson, editor, W3C, 15 February 2002. Available at (http://www.w3.org/TR/SVG11/

3. Computer Graphics : Principles and Practice Second Edition, James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, Richard L. Phillips, Addison-Wesley, pp. 488-491

4. TinyLine SVG http://www.tinyline.com/svgt/

5. TinyLine 2D http://www.tinyline.com/2d/

