O padrão SVG (Scalable Vector Graphics) da W3C (World Wide Web Consortium) define uma linguagem de descrição de gráficos vetoriais em duas dimensões que usa XML como base. Além de ser um formato padrão e portátil, ele permite também interação e animação, inclusive em dispositivos móveis. Neste artigo, mostraremos como usar o toolkit Batik para a criação e manipulação de gráficos no formato SVG em aplicações em Java, com exemplos simples e claros.

Usando a API Batik

Criando gráficos vetoriais em SVG de forma programática

Rafael Santos
(rafael.santos@lac.inpe.br):
é doutor em Inteligência Artificial pelo Instituto Tecnológico de Kyushu, Japão. É tecnologista do Instituto Nacional de Pesquisas Espaciais, atuando em pesquisa e desenvolvimento de aplicações e técnicas de mineração de dados e processamento de imagens. É autor do livro "Introdução à Programação Orientada a Objetos usando Java e de várias palestras e tutoriais sobre Java.

Existem duas categorias básicas de gráficos: rasterizados (ou bitmaps) e vetoriais. Gráficos rasterizados são representados por matrizes retangulares de pixels, sendo o tipo mais comum, já que são usados pelos mecanismos mais populares de aquisição de imagens (scanners e câmeras digitais) e apresentação (monitores). Gráficos vetoriais são representados de forma diferente, por meio da descrição de seus elementos gráficos. Como um exemplo rápido, podemos imaginar um gráfico de um quadrado: em sua forma rasterizada teremos uma matriz retangular de pixels, com um tamanho predefinido e com alguns pixels pintados para representar o quadrado. Em sua forma vetorial, precisamos somente da descrição (tamanho, cor) da área onde o quadrado será pintado e da descrição do quadrado em si (coordenadas, cor). Para visualizar o gráfico vetorial, teremos que rasterizá-lo de alguma forma para compatibilizá-lo com os dispositivos de saída. Na rasterização de um gráfico vetorial, podemos usar parâmetros para melhorar a resolução ou adequar a saída para determinado tipo de dispositivo.

Para alguns tipos de gráficos, o modo vetorial é o mais adequado: plantas, gráficos, plotagens científicas, diagramas, dados de sistemas CAD e de sistemas de informação geográficas etc. podem ser representados pela descrição de seus elementos gráficos, e renderizados quando necessário em uma resolução qualquer. Por outro lado, fotografias digitais e outros tipos de imagens são melhor representadas como matrizes de pixels: sua representação como elementos seria complexa demais. Os dois tipos de gráficos têm aplicações distintas e são complementares. Neste artigo, veremos como criar gráficos vetoriais de forma programática usando Java, a API Java2D e o toolkit Batik. Este toolkit permite a criação, visualização e manipulação de gráficos vetoriais no formato SVG (Scalable Vector Graphics), formato definido pelo W3C (World Wide Web Consortium) e baseado em XML; e que podem ser inseridos em páginas na Web, editados em diversas aplicações e até mesmo convertidos em gráficos rasterizados. Além de parsers e utilitários, o toolkit Batik contém o framework de renderização GVT (Graphic Vector Toolkit), que é de grande ajuda para criação de aplicações com interfaces gráficas que usam SVG.

Este artigo não tem como um de seus objetivos ser uma referência à XML ou às estruturas internas de SVG, sendo mais uma introdução prática ao toolkit Batik.

Para visualizar os gráficos no formato SVG, devemos usar um navegador compatível ou uma aplicação que reconheça e mostre os gráficos neste formato. Algumas são citadas a seguir. Existem várias ferramentas gratuitas para visualização e edição de gráficos no formato SVG. Uma bastante recomendada é o software inkscape, que é multiplataforma e permite a visualização, edição e conversão de arquivos SVG. Outro editor gratuito e multiplataforma é o sodipodi. Várias ferramentas permitem a criação de arquivos vetoriais e exportação para o formato SVG, entre elas Adobe Illustrator CS2, Corel Draw (versões 10 e posteriores) e OpenOffice Draw. Possivelmente outras ferramentas de desenho vetorial permitem a importação e exportação de arquivos SVG.

As versões mais novas do navegador Firefox (1.5 em diante) permitem a renderização direta de arquivos SVG, porém sem suporte completo a interação e animação. O próprio toolkit Batik contém squiggle, um visualizador simples para arquivos SVG. Se você usa Linux, o software gThumb permite a visualização rápida de arquivos SVG, possibilitando também exportar estes arquivos para outros formatos, inclusive raster.

Você pode baixar um plug-in para visualização de arquivos SVG a partir do Internet Explorer (veja as referências), mas a Adobe já anunciou que não suportará este plug-in no futuro.

Exemplo de SVG

Gráficos em SVG são compostos de declarações dos elementos que compõem o gráfico, em um formato baseado em XML. Um exemplo simples de documento SVG é mostrado na Listagem 1. Este exemplo foi codificado manualmente, e é importante observar que existem várias maneiras de declarar os componentes XML de um arquivo SVG.

Listagem 1. Gráfico simples em SVG.

```
<?xml version= 1.0 encoding= UTF-8 ?>
<!DOCTYPE svg PUBLIC '-//W3C//DTD SVG 1.0//EN 'http://www.w3.org/
TR/2001/REC-SVG-20010904/DTD/svg10.dtd >
<svg width= 30cm height= 20cm xmlns:xlink= http://www.
w3.org/1999/xlink xmlns= http://www.w3.org/2000/svg >
  <!-- Gráfico bem simples em SVG -->
  <g>
    <rect fill= rgb(250,250,250) x= 0cm y= 0cm width= 30cm
    height= 20cm stroke= none />
    <rect fill= none x= 2cm y= 2cm width= 26cm height= 16cm
    stroke= rgb(80,80,80) stroke-width= 0.5mm />
    <rect fill= rgb(220,255,220) x= 4cm y= 13cm width= 5cm
    height= 5cm stroke= rgb(0,255,0) stroke-width= 1mm />
    <rect fill= rgb(220,220,255) x= 12.5cm y= 10cm width= 5cm
    height= 8cm stroke= rgb(0,0,255) stroke-width= 2mm />
    <rect fill= rgb(255,220,220) x= 21cm y= 6cm width= 5cm
    height= 12cm stroke= rgb(255,0,0) stroke-width= 2.5mm />
  </g>
</svg>
```

Descartando as declarações iniciais, podemos observar três estruturas importantes neste documento: a declaração do arquivo em si (<svg..>), que contém o tamanho da área usada para desenho; a declaração de um bloco gráfico (<g>) e a declaração de elementos gráficos neste bloco (<rect />). Comentários são declarados da mesma forma que em XML.

Vários tipos de elementos gráficos ou primitivas (retângulos, círculos, elipses, linhas, linhas múltiplas e polígonos) podem ser declarados. Neste exemplo, somente retângulos foram usados (alguns preenchidos, alguns somente desenhados, alguns preenchidos e desenhados, com diferentes cores de preenchimento e cores, espessuras e estilos de linhas). A renderização deste arquivo é mostrada na figura 1 (apesar do arquivo SVG declarar suas dimensões como 30x20cm, na renderização podemos usar outros tamanhos).

Figura 1. Renderização de um gráfico SVG simples.

Um ponto interessante é que as coordenadas dos retângulos na Listagem 1 são dadas com unidades em centímetros e milímetros. Unidades podem ser dadas em pixels, centímetros, milímetros, polegadas e outras unidades tipográficas.

Apesar do arquivo SVG ser simples e pequeno, podemos claramente observar que a criação de gráficos complexos de forma manual é impraticável: devemos usar ou um editor gráfico para isto ou criar os gráficos por meio de programas, o que é o foco deste artigo. O toolkit Batik da fundação Apache permite a criação de arquivos SVG de forma programática de duas formas: por meio da manipulação de um documento XML (com a criação do documento e sua estrutura completa de forma programática) ou por meio do reaproveitamento de código que usa a API Java2D. Veremos as duas formas nas seções seguintes.

Instalando as bibliotecas do Batik

O primeiro passo é baixar o toolkit Batik de http://xmlgraphics.apache.org/batik/download.cgi (para este artigo, foi usada a versão 1.7). Para desenvolvimento de aplicações, utilizamos a distribuição binária.

A distribuição contém vários arquivos .jar, que possuem classes para várias funcionalidades do Batik, sendo que nem todos são necessários para as diferentes aplicações. Estes arquivos .jar devem ser referenciados pelo compilador e pela máquina virtual para manipulação dos arquivos SVG, ou seja, devem estar ou no classpath ou associados a um projeto em uma IDE.

Instalando os arquivos .jar do toolkit Batik no Eclipse

Para facilitar o desenvolvimento usando o Eclipse, é aconselhável criar uma biblioteca de jars. Os passos para fazer isso (no Eclipse 3.2) são:

1. faça o download do Batik e descompacte o arquivo que foi copiado;
2. no Eclipse, crie um projeto (ou use um já criado). No Package Explorer, clique com o botão direito do mouse no nome do projeto e selecione Properties no menu;
3. selecione a entrada Java Build Path no menu à esquerda e a aba Libraries no diálogo;
4. clique em Add Library e selecione User Library. Clique no botão Next;
5. clique no botão User Libraries para criar a biblioteca. Clique em New para criar uma nova biblioteca. Use o nome Batik, por exemplo, para a nova biblioteca. Clique em OK para fechar este diálogo;
6. clique em Add Jars para adicionar jars à nova biblioteca. Procure o diretório lib, localizado sob o diretório onde o download do batik foi descompactado. Selecione todos os arquivos .jar neste diretório e clique em OK;
7. os arquivos .jar aparecerão como parte da biblioteca batik. Clique em OK para fechar o diálogo de seleção da biblioteca;
8. no diálogo Add Library, selecione a biblioteca recém-criada e clique em Finish. Clique em OK para fechar o diálogo Java Build Path.

Outras IDEs podem usar outros mecanismos para importar os arquivos .jar, e os mesmos sempre podem ser incluídos no classpath para compilação e execução das aplicações.

Com os arquivos .jar do Batik instalados, podemos iniciar o desenvolvimento de aplicações que usam SVG de forma programática. É importante lembrar que estes arquivos .jar devem ser distribuídos juntamente com as aplicações que os usam.

Criando arquivos SVG como arquivos de texto

Existem várias maneiras de criar documentos SVG de forma programática. Como gráficos SVG são baseados em XML (que por sua vez é um formato de texto puro) podemos criá-los programaticamente de forma simples por meio de métodos que criem as declarações dentro de um "esqueleto de arquivo SVG (imagine as linhas da Listagem 1 envoltas em uma série de chamadas a System.out.println). Apesar da simplicidade desta abordagem, ela pode causar código de difícil legibilidade (com excesso de comandos de saída em arquivos), portanto propenso a erros e de difícil manutenção.

Outra alternativa é criar uma instância de classe que implementa a interface Document (que representa um documento em XML) e adicionar a esta, programaticamente, os componentes de um documento SVG. Os passos básicos para esta abordagem seriam:

1. criar um documento (instância de classe que implementa Document);
2. criar um elemento raiz no documento e modificar seus atributos básicos (para todo o documento SVG);
3. criar nós abaixo deste elemento raiz, onde cada nó é uma instância de classe que implementa a interface Element;
4. armazenar o documento em um arquivo.

Estes passos são demonstrados na Listagem 2, que gera um arquivo XML semelhante ao mostrado na Listagem 1.

O documento SVG criado pela classe CriaSVGDoc (Listagem 2) é estruturalmente idêntico ao mostrado na Listagem 1, embora apresente algumas diferenças de formatação do texto (que não influem no resultado da visualização ou renderização do SVG). A classe CriaSVGDoc também mostra como podemos criar objetos gráficos/elementos em métodos auxiliares, simplificando um pouco a tarefa de criação do Document. Apesar desta simplificação, a criação programática de SVGs complexos com este método pode ser muito trabalhosa e sujeita a erros.

Criando arquivos SVG a partir de componentes Swing (JComponent)

Uma maneira mais interessante de criar documentos SVG é usando componentes Swing já existentes (instâncias de classes que herdam de JComponent). A idéia é que se o componente já existir ele só precisa ser renderizado de forma diferente para criação do documento SVG, o que pode ser feito por uma instância da classe SVGGraphics2D o código do componente em si não precisa ser modificado.

Para exemplificar como converter o conteúdo de um componente para SVG, usaremos um exemplo mais completo, composto de três classes. A primeira classe, mostrada na Listagem 3, é um pouco extensa, pois representa completamente um componente gráfico (classe que herda de JComponent) e que representa um gráfico de torta.

A classe MeuComponente, mostrada na Listagem 3, serve para criar um gráfico simples de torta. O número de fatias e o valor absoluto de cada fatia são passados como parâmetro para seu construtor na forma de um array de valores de ponto flutuante.

O componente tem um tamanho fixo de 500 por 500 pixels, e usa um array de 11 cores para pintar as fatias do gráfico. As cores são reusadas

Listagem 2. A classe CriaSVGDoc.

```
package batik;

import java.io.IOException;

import org.apache.batik.dom.svg.SVGDOMImplementation;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;

import com.sun.org.apache.xml.internal.serialize.XMLSerializer;

public class CriaSVGDoc {
   public static void main(String[] args) throws IOException {
      // Criamos uma instância correspondente à implementação do
      // Document Object Model para nosso documento.
      DOMImplementation dom =
         SVGDOMImplementation.getDOMImplementation();
      // Criamos uma instância de Documento.
      String svgNS = SVGDOMImplementation.SVG_NAMESPACE_URI;
      Document documento = dom.createDocument(svgNS, svg ,null);
      // Recuperamos o elemento raiz do documento (o "svg ).
      Element raiz = documento.getDocumentElement();
      // Mudamos alguns de seus atributos.
      raiz.setAttribute("width , 30cm );
      raiz.setAttribute("height , 20cm );
      // Criamos alguns retângulos e os adicionamos à raiz do documento.
      raiz.appendChild(criaRect(documento, 0cm , 0cm , 30cm , 20cm ,
          rgb(250,250,250) , none , ));
      raiz.appendChild(criaRect(documento, 2cm , 2cm , 26cm , 16cm ,
          none , rgb(80,80,80) , 0.5mm ));
      raiz.appendChild(criaRect(documento, 4cm , 13cm , 5cm , 5cm ,
          rgb(220,255,220) , rgb(0,255,0) , 1mm ));
      raiz.appendChild(criaRect(documento, 12.5cm , 10cm , 5cm , 8cm ,
          rgb(220,220,255) , rgb(0,0,255) , 2mm ));
      raiz.appendChild(criaRect(documento, 21cm , 6cm , 5cm , 12cm ,
          rgb(255,220,220) , rgb(255,0,0) , 2.5mm ));
      // Concatenamos o conteúdo do Document em uma saída.
      // Dica em http://weblogs.java.net/blog/iasandcb/archive/2006/06/
      // serializing_xml.html
      XMLSerializer serializer = new XMLSerializer();
      serializer.setOutputByteStream(System.out);
      serializer.serialize(documento);
   }

   // Método auxiliar para criar retângulos como elementos em SVG.
   private static Node criaRect(Document doc,String x,String y,String w,
      String h,String f,String s,String sw) {
      Element rect = doc.createElement("rect );
      rect.setAttribute(«x»,x);
      rect.setAttribute(«y»,y);
      rect.setAttribute(«width»,w);
      rect.setAttribute("height ,h);
      rect.setAttribute("fill",f);
      rect.setAttribute("stroke ,s);
      rect.setAttribute("stroke-width ,sw);
      return rect;
   }

}
```

Listagem 3. A classe MeuComponente.

```
package batik;

import java.awt.*;
import java.awt.geom.Arc2D;
import javax.swing.JComponent;

public class MeuComponente extends JComponent {
   private float[] valores;
   private final int largura = 500;
   private final int altura = 500;
   private final Color[] cores =
   {
      Color.BLACK,Color.RED,Color.GREEN,Color.BLUE,
      Color.CYAN,Color.MAGENTA,Color.YELLOW,
      Color.DARK_GRAY,Color.LIGHT_GRAY,Color.ORANGE,Color.PINK
   };

   public MeuComponente(float[] param) {
      // Normalizamos os valores para que a soma seja 360 graus.
      float soma = 0f;
      for(float val:param) soma += val;
      valores = new float[param.length];
      for(int v=0;v<param.length;v++) valores[v] = 360f*param[v]/soma;
   }

   public Dimension getMaximumSize() {
      return getPreferredSize();
   }

   public Dimension getMinimumSize() {
      return getPreferredSize();
   }

   public Dimension getPreferredSize() {
      return new Dimension(largura,altura);
   }

   protected void paintComponent(Graphics g) {
      // Usamos um contexto gráfico melhor.
      Graphics2D g2d = (Graphics2D)g;
      // Pintamos o fundo de azul bem claro.
      g2d.setColor(new Color(245,245,255));
      g2d.fillRect(0,0,largura,altura);
      // Pintamos uma borda azul escura.
      g2d.setColor(new Color(0,0,80));
      g2d.drawRect(20,20,largura-40,altura-40);
      // Pintamos as fatias do gráfico
      float ângulo = 0;
      for(int v=0;v<valores.length;v++) {
         Arc2D arco = new Arc2D.Float(40,40,largura-80,
         altura-80,ângulo,valores[v],Arc2D.PIE);
         g2d.setColor(cores[v % cores.length]);
         g2d.fill(arco);
         ângulo += valores[v];
      }
      // Pintamos uma borda em volta.
      g2d.setColor(Color.BLACK);
      g2d.setStroke(new BasicStroke(3f));
      g2d.drawArc(40,40,largura-80,altura-80,0,360);
   }

}
```

Listagem 4. Uma aplicação que usa uma instância de MeuComponente para exibição na tela.

```
package batik;

import javax.swing.JFrame;

public class DemoMeuComponente extends JFrame {
   public DemoMeuComponente(float[] valores) {
      setTitle("Meu Componente  );
      getContentPane().add(new MeuComponente(valores));
      pack();
      setVisible(true);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }

   public static void main(String[] args) {
      float[] valores = {4.9f,12.1f,12.8f,6.8f,143f,125f,51.1f,49.5f,2.3f};
      new DemoMeuComponente(valores);
   }
}
```

caso sejam utilizadas mais do que 11 fatias. O método mais importante da classe é o paintComponent, que recebe como argumento uma instância de classe que representa um contexto gráfico (Graphics) e que pinta todos os elementos do gráfico.

Para demonstrar de forma simples este componente, usaremos uma classe que herda de JFrame para criar uma aplicação com interface gráfica bem simples. Esta aplicação é mostrada na Listagem 4.

A aplicação DemoMeuComponente, mostrada na Listagem 4, simplesmente cria uma instância de si mesma, usando um Frame que contém uma instância de MeuComponente com o array de valores declarados em seu método main. O resultado gráfico da execução da aplicação é mostrado na figura 2.

Figura 2. Resultado da execução da classe DemoMeuComponente.

As classes nas Listagens 3 e 4 usam classes das APIs Swing e Java2D para mostrar um gráfico na tela de um desktop. É possível salvar o contexto gráfico do componente (incluindo o que for desenhado no contexto) usando as classes BufferedImage para replicar o contexto gráfico e ImageIO para armazenar em um arquivo, mas o arquivo gerado será um bitmap, com possibilidades limitadas de edição e escala.

O toolkit Batik provê uma classe, SVGGraphics2D, que pode ser usada como contexto gráfico virtual (a classe herda de Graphics2D). Instâncias desta classe podem ser usadas como argumentos para o método paintComponent de um componente qualquer, permitindo que o mesmo código em uma classe que herda de JComponent possa ser usado para renderização do componente em uma interface gráfica e em uma instância de Document, possibilitando a criação de SVGs sem esforço adicional. A mesma classe permite o armazenamento do documento SVG em disco ou sua serialização. A classe mostrada na Listagem 5 demonstra o uso de uma instância de SVGGraphics2D juntamente com a classe MeuComponente para renderizar o componente usando SVG. O arquivo SVG resultante é armazenado em um arquivo local.

O arquivo grafico.svg, criado como resultado da aplicação mostrada na Listagem 5, pode ser visto no lado esquerdo da figura 3. Como o gráfico foi gravado em formato vetorial, podemos ver no lado direito da mesma figura que a ampliação da mesma não produz efeitos de pixelação (ou "escada") que ocorrem em imagens bitmap. Ambas as figuras foram renderizadas pelo software inkscape, com diferentes ampliações.

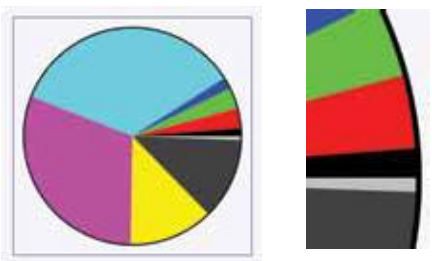

Figura 3. Resultado da renderização do componente MeuComponente em um arquivo SVG (direita: detalhe ampliado).

Listagem 5. Uma aplicação que usa uma instância de MeuComponente para criação como SVG e armazenamento em disco.

```
package batik;

import java.io.IOException;
import javax.swing.JFrame;
import org.apache.batik.dom.GenericDOMImplementation;
import org.apache.batik.svggen.SVGGraphics2D;
import org.w3c.dom.*;

public class DemoMeuComponenteSVG extends JFrame {

   public static void main(String[] args) throws IOException {
      // Obtemos uma implementação do DOM.
      DOMImplementation dom =
         GenericDOMImplementation.getDOMImplementation();
      // Criamos uma instância de org.w3c.dom.Document.
      Document doc = dom.createDocument("http://www.w3.org/2000/svg ,
         "svg , null);
      // Criamos uma instância do gerador de SVGs.
      SVGGraphics2D svg = new SVGGraphics2D(doc);
      // Criamos uma instância do componente que queremos armazenar.
      float[] valores = {4.9f,12.1f,12.8f,6.8f,143f,125f,51.1f,49.5f,2.3f};
      MeuComponente componente = new MeuComponente(valores);
      // "Pintamos  a instância como SVG.
      componente.paintComponent(svg);
      // Armazenamos em um arquivo.
      svg.stream("grafico.svg");
   }
}
```

Listagem 6. Uma aplicação que lê e modifica um SVG.

```
package batik;

import java.io.IOException;

import org.apache.batik.dom.svg.SAXSVGDocumentFactory;
import org.apache.batik.util.XMLResourceDescriptor;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

import com.sun.org.apache.xml.internal.serialize.XMLSerializer;

public class EditaSVG {
   public static void main(String[] args) throws IOException {
      // Criamos um parser e uma fábrica para o documento.
      String parser = XMLResourceDescriptor.getXMLParserClassName();
      SAXSVGDocumentFactory f = new SAXSVGDocumentFactory(parser);
      String uri = "file://crux.svg"; // Poderia ser uma URL!
      Document documento = f.createDocument(uri);
      // Recuperamos o elemento raiz.
      Element raiz = documento.getDocumentElement();
      // Recuperamos a área do SVG (não nos preocupamos com dimensões).
      String w = raiz.getAttribute("width  );
      String h = raiz.getAttribute("height  );
      // Criamos um retângulo envolvente para todo o SVG.
      Element borda = documento.createElement(«rect»);
      borda.setAttribute(«x», «0»);
      borda.setAttribute(«y», «0»);
      borda.setAttribute("width  , w);
      borda.setAttribute("height  , h);
      borda.setAttribute("fill", "none  );
      borda.setAttribute("stroke  , "green  );
      borda.setAttribute("stroke-width  , "1mm  );
      raiz.appendChild(borda);
      // Concatenamos o conteúdo do Document em uma saída.
      XMLSerializer serializer = new XMLSerializer();
      serializer.setOutputByteStream(System.out);
      serializer.serialize(documento);
   }
}
```

Lendo um documento SVG a partir de uma URI

Documentos SVG já existentes podem ser lidos e armazenados em uma instância de Document, para processamento posterior ou visualização. Para ler um SVG de um arquivo ou URL, precisamos criar um parser XML e com ele uma fábrica de documentos SVG. A partir da fábrica, podemos criar o documento em memória e manipulá-lo. Um exemplo de manipulação é dado na Listagem 6, que lê um arquivo SVG do disco local e cria uma borda verde nele (a borda é criada como um retângulo sem preenchimento, do tamanho exato da área do SVG).

A modificação do conteúdo do arquivo SVG e sua saída são feitas de forma análoga à mostrada na Listagem 2. O resultado da execução da aplicação usando como entrada um arquivo local é ilustrado na figura 4.

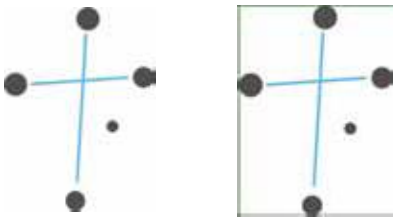

Figura 4. Um arquivo SVG (original: esquerda) modificado com a aplicação da Listagem 6 (direita).

Alguns pontos importantes da abordagem ilustrada pela Listagem 6 são:

- nenhum esforço é feito para reposicionar ou modificar os elementos gráficos já existentes, somente um novo elemento é adicionado ao elemento raiz do documento SVG. Se fosse necessário modificar os documentos um parsing mais completo (e complexo), deveria ser implementado, recuperando os elementos (Elements) do documento e processando um a um;
- o arquivo SVG usado no exemplo mostrado na figura 4 é consideravelmente mais complexo do que o mostrado na Listagem 1, contendo várias declarações inseridas pelo software de edição usado para criar a figura (inkscape). Mesmo assim o parser consegue ler e processar corretamente o conteúdo do gráfico, copiando as declarações para a saída do programa.

Outra tarefa comum envolvendo arquivos ou URLs SVG já existentes é a exibição ou renderização destes em uma aplicação com interface gráfica. O toolkit Batik provê uma classe pronta para isto: JSVGCanvas, que herda de JComponent e que pode ser usada para renderizar um documento SVG (usando como argumento a URI deste documento SVG ou mesmo uma instância de classe que implementa SVGDocument, que herda de Document). Um exemplo de aplicação com interface gráfica que exibe um arquivo ou URI SVG é mostrado na Listagem 7.

A classe JSVGCanvas facilita bastante a exibição de conteúdo de SVGs em aplicações gráficas, mas faz mais do que isso: ela já contém mecanismos prontos para movimentação (pan), escala (zoom) e rotação do conteúdo apresentado. Podemos mudar a área do documento que é mostrada no componente pressionando shift e o botão esquerdo do mouse enquanto movemos o mouse (pan). Podemos girar o documento SVG pressionando control com o botão direito do mouse enquanto movemos o mouse. Mudança de escala (zoom) pode ser feita de duas formas: pressionando control e o botão esquerdo do mouse para selecionar uma área retangular para zoom ou pressionando shift e o botão direito do mouse enquanto movemos o mouse para cima (diminuindo o desenho) ou para baixo (aumentando o desenho). A aplicação exibida na Listagem 7 carrega e mostra o documento SVG em

Listagem 7. Uma aplicação com interface gráfica que mostra o conteúdo de um SVG.

```
package batik;

import java.io.IOException;

import javax.swing.JFrame;

import org.apache.batik.swing.JSVGCanvas;

public class ExibeSVG extends JFrame {
    public ExibeSVG(String svg) {
        // Criamos uma instância de JSVGCanvas e associamos a string com a
        // URI do SVG.
        JSVGCanvas svgCanvas = new JSVGCanvas();
        svgCanvas.setURI(svg);
        // Adicionamos o JSVGCanvas ao JFrame.
        getContentPane().add(svgCanvas);
        // Especificamos o comportamento geral da aplicação.
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
    }

    public static void main(String[] args) throws IOException {
        // Criamos uma instância desta própria classe.
        String uri = "file:crux.svg";
        new ExibeSVG(uri);
    }
}
```

uma interface gráfica, mas com um problema: o tamanho inicial da janela (JFrame) pode não corresponder exatamente ao tamanho necessário para mostrar todo o conteúdo do documento SVG. Por exemplo, se usarmos como entrada o SVG no lado esquerdo da figura 4, somente parte do documento SVG será apresentado. Antes de resolver este problema, consideremos o processo de renderização de um documento SVG como um todo. O documento deve ser carregado, analisado (com um parser) e renderizado pelo componente JSVGCanvas e, como mostrado no exemplo, tudo isso pode ser feito sem intervenção do programador. Para possibilitar um controle mais fino do processo de renderização, podemos implementar interfaces que contêm assinaturas de métodos que podem ser executados durante este processo. As interfaces são SVGDocumentLoaderListener, GVTTreeBuilderListener e GVTTreeRendererListener. A interface SVGDocumentLoaderListener contém métodos que podem ser executados quando o documento SVG é carregado, quando sua carga é interrompida ou quando sua carga causa erros. A interface GVTTreeBuilderListener contém métodos que podem ser executados para indicar o andamento do parsing do documento para uma árvore interna do Graphics Vector Toolkit, e a interface GVTTreeRendererListener faz o mesmo para a renderização desta árvore. Uma forma simples de modificar a aparência da aplicação gráfica que contém a instância de JSVGCanvas é verificar quando o método gvtBuildCompleted da interface GVTTreeBuilderListener for executado, significando que a árvore foi totalmente criada e que a instância de JSVGCanvas tem acesso às dimensões do documento SVG. Neste ponto, basta pedir à aplicação gráfica que execute o comando pack(), ajustando as dimensões do componente. Esta técnica é mostrada na Listagem 8.

Considerações finais

O toolkit Batik permite a criação programática de gráficos em SVG de diversas formas. Além das capacidades exemplificadas neste artigo, é possível criar animações com scripts em JavaScript (embutidos no próprio SVG) além de ter interações mais complexas com documentos carregados em instâncias do componente JSVGCanvas. Por razões de espaço, alguns exemplos mais complexos não foram incluídos neste artigo, mas as referências indicadas podem mostrar como construí-los• MJ

Listagem 8. Melhorias na aplicação com interface gráfica que mostra o conteúdo de um SVG.

```
package batik;

import java.io.IOException;

import javax.swing.JFrame;

import org.apache.batik.swing.JSVGCanvas;
import org.apache.batik.swing.svg.GVTTreeBuilderEvent;
import org.apache.batik.swing.svg.GVTTreeBuilderListener;

public class ExibeSVG extends JFrame implements GVTTreeBuilderListener {
    public ExibeSVG(String svg) {
        // Criamos uma instância de JSVGCanvas e associamos a string com a
        // URI do SVG.
        JSVGCanvas svgCanvas = new JSVGCanvas();
        svgCanvas.setURI(svg);
        // Adicionamos o JSVGCanvas ao JFrame.
        getContentPane().add(svgCanvas);
        // Registramos esta classe para responder a eventos tipo
        // GVTTreeBuilderEvent.
        svgCanvas.addGVTTreeBuilderListener(this);
        // Especificamos o comportamento da aplicação.
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void gvtBuildFailed(GVTTreeBuilderEvent e) { }

    public void gvtBuildStarted(GVTTreeBuilderEvent e) { }

    public void gvtBuildCancelled(GVTTreeBuilderEvent e) { }

    public void gvtBuildCompleted(GVTTreeBuilderEvent e) {
        pack();
    }

    public static void main(String[] args) throws IOException {
        // Criamos uma instância desta própria classe.
        String uri = "file:crux.svg";
        new ExibeSVG(uri);
    }

}
```

Saber mais

Uma primeira referência é o próprio site do Batik: http://xmlgraphics.apache.org/batik/. A partir deste site, você pode baixar a API, sua documentação e ler alguns exemplos.

Estes livros sobre SVG podem ser consultados para mais detalhes sobre a estrutura de documentos SVG e como criá-los. Os dois primeiros cobrem a parte mais genérica sobre SVG e o terceiro comenta em detalhes a API Batik. Não são livros de design gráfico, são mais de interesse de desenvolvedores.

- Fundamentals of SVG Programming: Concepts to Source Code, Oswald Campesato, Charles River Media, 2004.
- Designing SVG Web Graphics, Andrew H. Watt, New Riders Publishing, 2001.
- Java Drawing with Apache Batik: A Tutorial, Alexander Kolesnikov, Brainy Software Corp., 2007.

Referências

- *Definição de SVG: http://www.w3.org/TR/SVG11/*
- *Batik SVG Toolkit: http://xmlgraphics.apache.org/batik/index.html*
- *Inkspace: http://www.inkscape.org/*
- *Sodipodi: http://www.sodipodi.com/index.php3*
- *Plug-in da Adobe: http://www.adobe.com/svg/viewer/install/*