# Stylesheet Translations of SVG to VML

**A Master's Project**

**presented to**

**The Faculty of the Department of Computer Science**

**San Jose State University**

**In Partial Fulfillment**

**of the Requirements for the Degree of**

**Master of Science**

**Julie Nabong**

**Advisor: Dr. Chris Pollett**

**May 2004**

## Abstract

The most common graphics formats on the Web today are JPEG and GIF. In addition to these formats, two XML-based graphic types are available as open standards: SVG and VML. SVG and VML are vector graphic formats. These formats offer benefits such as fast Web download time, zoomable images, and searchable texts. Because these vector graphics are scalable, these images can be viewed in different screen sizes, such as PC displays and handheld devices. SVG and VML implementations are gaining popularity in Internet cartography and zoomable charts. SVG images can be viewed by downloading a plug-in; whereas, VML images are rendered in Microsoft's Internet Explorer browser versions 5.0 and higher. Although SVG may be considered a more mature format than VML, it is unlikely it will be supported natively by Microsoft anytime soon. In this master's project, SVG images will be transformed into VML images contained in an HTML document that can be viewed without a plug-in. SVG images will be manipulated through the Document Object Model API and transformed into VML images using JavaScript, XSLT, and XPath. JavaScript will play an important role in handling functionalities not present in XSLT. This project will address the issue of gradient discrepancies between the two formats, and try to get the speed of the translation as close to that of the plug-in based solution as possible.

# Table of Contents

# 1. Introduction

When browsing the web, an Internet user might observe that the popular image formats displayed are JPEG, GIF, or animated images displayed by proprietary multimedia programs. In addition to those formats, web developers can also create images by using XML-based languages called SVG and VML.

Compared to bitmap and JPEG files, files containing either SVG or VML download faster because of their smaller file sizes. When it comes to editing, SVG or VML can be edited without using a special image tool. Simply open the file in a text editor and change the values of the coordinates or other properties of the image. Both SVG and VML are open standards, and are World Wide Web Consortium (W3C) Recommendations. One difference between these two formats is that SVG can be viewed in browsers using a plug-in, while VML can be viewed in Microsoft's Internet Explorer version 5.0 and higher without a plug-in.

A stylesheet is a document that takes an XML (Extensible Markup Language) document as an input, processes this source document, and then outputs another document. A popular use of stylesheet is to accept raw XML data, process this data according to the contents of the stylesheet, and then output this data into a more readable format, such as an attractive web page.

A stylesheet can be used to receive a database containing customer information, use this data to perform calculations based on the goal of the stylesheet, then output a readable document. A stylesheet can take more than one source data, and can output more than one kind of document, such as an XML document and an HTML document. A stylesheet is reusable. It can be easily modified and extended by embedding script functions.

The process of converting raw data into a more readable document is referred to as translation. The purpose of this master's project is to develop a stylesheet that will translate images from one format to another. Specifically, the stylesheet will translate SVG images to VML images so they can be displayed within Internet Explorer without the required plug-in.

There have been efforts to convert images from one format to another. One example is a tool to convert SVG to PDF using proprietary software offered by Adobe as found in [AGS]. Another example is a tool to convert MathML to SVG using Java and XSLT as published in [S04]. There is also an effort to convert Internet maps to SVG known as Map2SVG as found in [M04]. One can download a commercial software that receives data and outputs the result as either SVG or VML offered by [G04]. As of this writing, we have not found a tool that would convert SVG images to VML images. This is probably due to the complexities of writing a program that would address issues of browser compatibility with W3C's specifications, and efficient translation in terms of

download time. We consider this project an opportunity to use an XML-based mechanism together with JavaScript to convert SVG images and display these images in a transparent manner.

Because the SVG document to be translated and the stylesheet itself are XML-based documents, we will rely on another W3C Recommendation, XMLDOM, to load and manipulate these files. This report is organized according to the following topics:

- A Conceptual Description of our Project
- An SVG Example
- SVG Tags Supported by the Project
- A VML Example
- XSLT and XPath
- Some of the XSLT Functions Used by the Translator
- XSLT, JavaScript, and the Translator
- XMLDOM
- The Translator Implementation
- The Translator Limitations
- Project Conclusion

# 2. Conceptual Description of our Project

In this section, we will give a broad description of how we developed our project. In addition, we will also provide an overview of the documents and software we used, and a general description of the transformation process.

## *SVG Documents*

Some of the reasons we create SVG documents are:

- to draw two-dimensional shapes
- to add texts to the shapes

Using the combination of the items above, using SVG, we can display a variety of information such as graphs, quantitative statistics, maps, and other technical diagrams. Figure 1 below is an example SVG document depicting the amount of computer science classes offered for two semesters at San Jose State University:

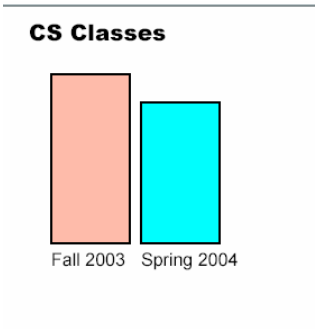



**Figure 1. SVG Bar Chart**

SVG documents are based on XML specification. Thus, they follow the syntax of XML. An XML document can be depicted in a tree-like structure. Figure 2 below shows an illustration of the SVG document in Figure 1 in a tree form.

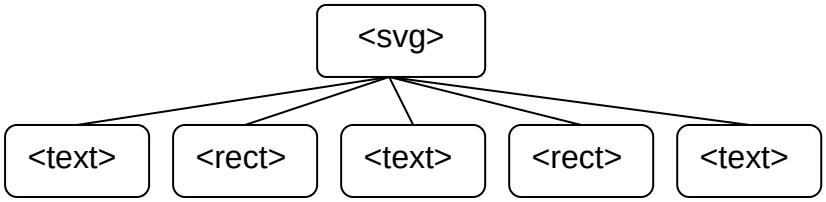



**Figure 2. Tree Structure of an SVG Document**

Using an XML editor, below is another way to look at the same SVG document.


**Figure 3. Viewing the SVG Document Using an XML Editor**

The two illustrations shown above reveal the structure of the SVG document having a parent node and five children nodes. To transform the SVG document into a document containing VML images that can be viewed without a plug-in, we need another XML-based document. This document is written using the language XSLT, and just like SVG documents, it follows the syntax of XML. This document is called "stylesheet".

### *How Do We Write a Stylesheet?*

As mentioned above, a stylesheet follows the syntax of XML. Its main purpose is to create a document based on an XML document. For example, the stylesheet writer could produce a document whose data could come from an XML database. Our project is unique in that we wrote a stylesheet to produce a document that renders images.

A stylesheet does not parse the source document line by line. Instead, it looks at the source document as a tree with nodes. The stylesheet writer could add some commands instructing the stylesheet to go to specific nodes and perform some actions. These commands are called templates. We can view a stylesheet as a document having many template rules. Figure 4 is an illustration.

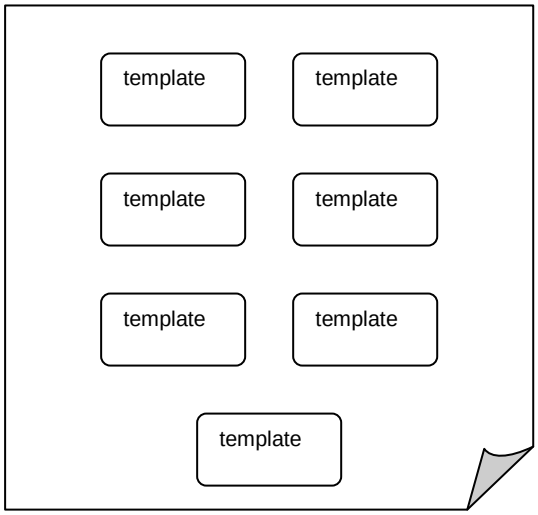



**Figure 4. A Stylesheet Containing Templates**

To write a stylesheet we can use a favorite text editor. In our project, we used an XML editor so we could always check if the stylesheet follows XML syntax. The XML editor we used could only check XML syntax, but not the syntax of XSLT.

Not all SVG documents are the same. An SVG document could have scripting code before, inside, or after an element. It could also group images so they would have the same look. An SVG document could have a collection of fill colors to be used by some images. That was the basis of our stylesheet design. We wrote a stylesheet that would handle different kinds of SVG documents.

The first template we wrote is for the root of the source document. Whenever the stylesheet encounters the root, it prepares a blank HTML document with all the requirements to be able to render VML code later. For the remaining templates, each of them would process a certain node encountered in the SVG document. For example, if there is a node equal to "defs", we instructed that template to process its child nodes because we know that this node contains one or many gradient fill information. Another example is, if the stylesheet encounters scripting code, we have a template that copies the script so we could use it later.

Developing templates is not trouble-free. Even writing one template could be cumbersome. As mentioned earlier, the XML editor does not check XSLT syntax. Any of the four things below could happen after writing a template and testing it:

1. it renders the output correctly, which occurred infrequently in our project, or
2. the browser outputs nothing, which happened on some occasions, or
3. the browser renders a different image, this gave us some hope, or
4. the browser displays a message, "method not supported", the most common output and one which gave us the most trouble, because it does not specify where and what the nature of the problem was

There are editors that would check the syntax, allow you to add attributes and comments easily, but that would not help us accomplish our goal of rendering

images and adding scripts on a document. Our stylesheet does not use one language alone. It has an embedded JavaScript functions, VML elements, and scripting code as part of the output. Overall, our stylesheet contains 23 templates. It has 1,430 lines, out of which 500 contain JavaScript code.

To accomplish such a stylesheet takes a lot of time, patience and hard work. In addition to XSLT, the writer also needs to learn another language that works side by side with XSLT. This language is called XPath, which stands for XML Path Language. This language is used to select nodes from the input tree. In our project, the stylesheet writer also needs to be familiar with Web Recommendations. These Web recommendations are used outside of the stylesheet such as in loading and parsing the documents.

One challenging aspect of this project is the support for user interaction such as a mouse click. If the mouse click renders another image, we had to develop a way to handle this situation. We came up with the idea of "double transformation". We stored the original source document in an XML object, and then applied the event to that object. More details are available in the *Implementation* section of this report. In the next section, we discuss an overview of the transformation process.

### *The Transformation Process*

Our XSLT file contains these things:

- what nodes to choose
- what to do with the chosen nodes

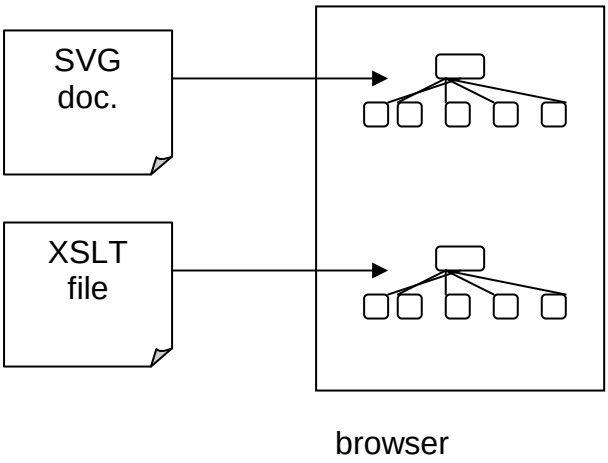
In our project, we only have two files, the SVG document and the XSLT document. We need a third document that is an HTML document. This HTML file will contain the following:

- methods to load the SVG document and XSLT file
- a method to perform the transformation
- a JavaScript function telling the browser where to display the result

In addition, we use Internet Explorer 6.0 for the following reasons:

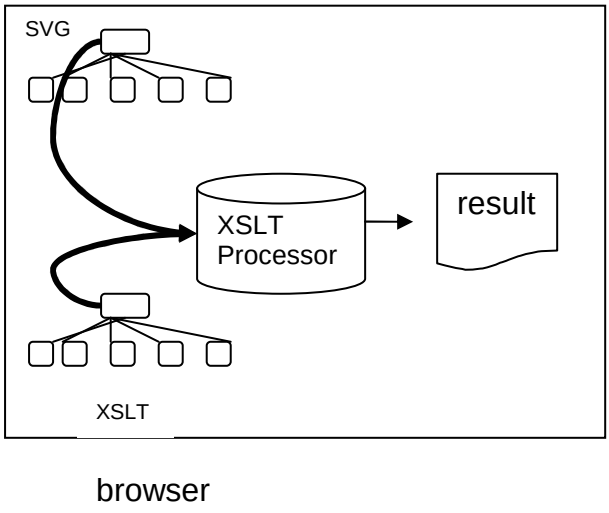
- it supports VML without additional software
- it has a built-in XML parser
- it has a built-in tool for processing XSLT documents (XSLT processor)

Now that we have the three documents, the transformation can begin. Figure 5 through 8 illustrate the transformation process.




**Step 1.** The HTML file through the browser loads and parses the two documents.

**Figure 5. Browser loads the SVG and XSLT documents.**




**Step 2.** The browser transforms the SVG document based on the contents of XSLT file.

**Figure 6. Browser Transforms SVG Document**

We want to note that the parsed SVG and XSLT documents, and the result of the transformation are on computer's memory. The last step is the command in the

HTML file telling the browser where to display the result. We want to display the result in a new browser document. The browser writes the result in a new document window. Figure 7 illustrates the result that is an HTML document containing VML images.

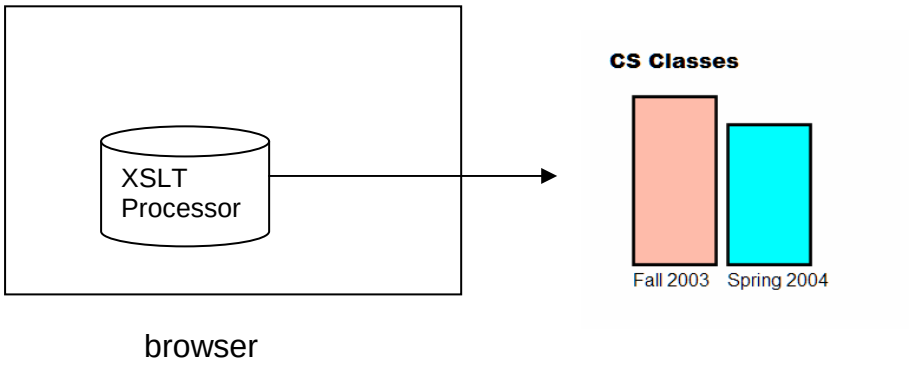



**Figure 7. The Result: HTML Document with VML Images**

Here is another illustration of a stylesheet transformation of SVG to VML. Let us look at figure 8.

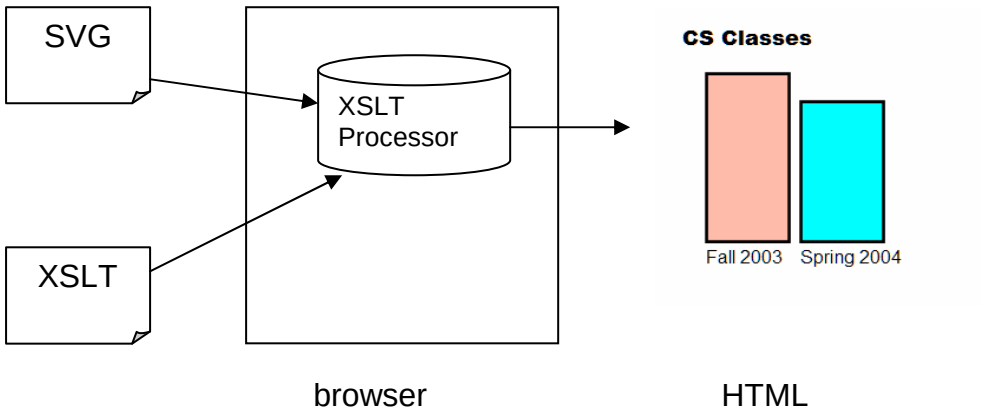



**Figure 8. Transformation of SVG to VML**

Although the result is an HTML document, and not an XML document, we can still view the result in a tree-like structure. Using the same XML editor we used earlier, Figure 9 shows the result tree.

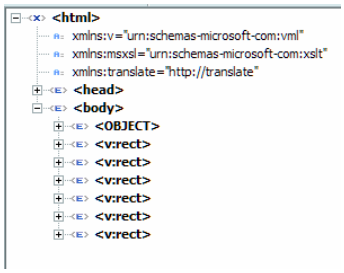



**Figure 9. The Result of the Transformation in Tree-Like Structure**

XSLT stylesheets (or documents) can create any of the following documents:

- HTML document
- XML document
- Text document
- Wireless Markup Language (WML) document

In this project, because we want to display the resulting VML images as part of an HTML document, we will tell the stylesheet to output the result as an HTML document.

To review, we started by having:

- an SVG document containing images
- an XSLT stylesheet containing commands, and
- an HTML document that loads, parses, and processes the XSLT functions

The three documents resulted in a new HTML document containing the VML images. This "newly created" document has more nodes in terms of tree. It has `<html>`, `<head>`, `<body>`, and `<object>` as additional nodes. Therefore, if we compare the SVG source tree and the result tree, the latter will always be bigger in terms of nodes.

## *Viewing Images*

Traditionally, if we want to display data with bar charts, graphs, and other quantitative statistics, we would use proprietary programs such as PowerPoint and Excel. We, the user of these programs, would be subject to the rules and limitations of these applications. If we want to send data over the Internet, we would make sure that the end viewer has the same program we used to create these images. With the advent of XML, we can now store textual and graphical data to be accessed, transformed, and viewed in Internet browsers. We can describe our project to be centered around XML technologies.

SVG documents can be viewed in browsers, but it would present an inconvenience to the user because he or she needs to:

- search where the SVG viewer can be downloaded
- download the viewer which is about 2MB in size
- install the viewer

With VML images, the user could:

- instantly view the images
- avoid the trouble of downloading and installing
- save at least 2MB of disk space
- confidently share the document (no additional software needed)

## *This Project and XSLT*

XSLT has many benefits. It can also be used to transform non-XML data to an XML document as shown in [R04]. In addition, XSLT can be used to display data on wireless devices using WML (Wireless Markup Language) as explained in [L04]. XSLT is an XML-based document, which means the size of the file is generally small.

It is not surprising to see Internet documents explaining how to learn XSLT in different approaches. XSLT is a unique language that offers many challenges to the writer especially if the writer is accustomed to traditional programming languages. XSLT and XPath are always together. Thus, the writer needs to learn at least two languages. As Michael Kay, the editor of XSLT 2.0 working draft put it, "there is a steep learning curve and often a lot of frustration," regarding XSLT.

In this project, we have developed a means to render not textual data, but useful graphics images using XSLT that can be viewed without additional software. In addition, we have implemented user interaction or mouse events that would simulate events in the resulting document. We accomplished that by creating synthetic mouse events. If the event draws an additional image, we have included scripting code that would perform a second transformation. In the following sections, we will present more details about SVG, XSLT and implementation techniques we used to develop our project. Let us now look at an SVG example.

# 2. An SVG Example

Scalable Vector Graphics (SVG) is a specification based on XML. It is used to create two-dimensional images. As of this writing, SVG images can be viewed using a plug-in, which can be downloaded for free.

SVG has several advantages over existing graphics standards. In addition to smaller file sizes, once an SVG image is displayed in a browser, the user can zoom the image in or out by simply clicking the mouse. Using SVG we can create two-dimensional images such as rectangles, ovals, and lines. In addition, an SVG writer can also include text, colors, and animation effects. An SVG document can be displayed independently in a browser, or it can be displayed as an object with other text and images. Some of the predefined shapes of SVG are:

- Rectangle <rect>
- Line <line>
- Polygon <polygon>
- Circle <circle>

Figure 10 below shows an example of an SVG document that renders a rectangle.

```
<?xml version="1.0" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000303 Stylable//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width=”100%” height=”100%”>
<rect x=”30” y=”30” width="120" height="100"
  style="fill:red;stroke-width:2;stroke:blue"/>
</svg>
```

**Figure 10 – An SVG Document**

In this project, we have observed that an XML document can be displayed in a browser with or without a Document Type Declaration. Figure 10 shows this declaration in the second line. Internet Explorer can recognize SVG documents without this declaration; thus, we have elected not to include one in our SVG documents. Figure 11 illustrates the SVG image from Figure 10 using Internet Explorer 6.0.

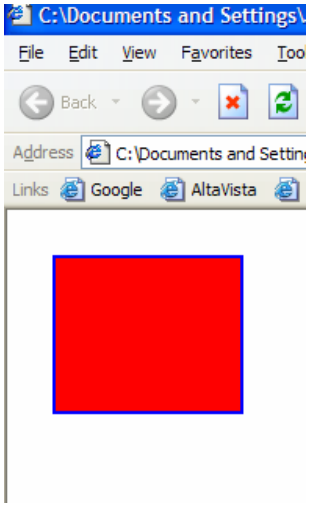



**Figure 11. An SVG Image**

Many organizations contributed to the development of SVG. The reader can acquire more details about SVG by visiting [SVG03].

# 3. SVG Elements Supported by the Project

In this project the SVG tags supported are:

- <svg> - the start and end element of an SVG document
- <g> - used to group shapes

- <polygon> - used to create closed shapes consisting of straight lines
- <polyline> - used to create shapes consisting of straight lines
- <circle> - used to produce circles
- <ellipse> - used to produce ellipse
- <text> - used to add texts
- <line> - describes a straight line
- <linearGradient> - used to create color changes either vertically or horizontally
- <path> - used to define two-dimensional shapes
- <radialGradient> - used to create color changes
- <rect> - used to create rectangles
- <script> - used to add scripting code such as JavaScript code

# 4. A VML Example

Vector Markup Language (VML) is another XML-based specification. Just like SVG, the purpose of VML is to create two-dimensional images that can be embedded within an HTML file. The HTML author can create shapes such as rectangles, circles, polygons, lines, and other predefined shapes using VML. Internet Explorer 5.0 and higher support VML without a required plug-in. Figure 12 shows a simple VML code embedded in an HTML file, and Figure 13

illustrates the resulting web page. The reader can obtain more information about VML by visiting [VML98].

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head>
<style type="text/css">
  v\:*{behavior:url(#default#VML);}
</style>
</head>
<body>

<v:polyline style="position:absolute;"
  points="50,50 180,138, 50,225 50,50"
  strokecolor="black"
  strokeweight="2"
  fillcolor="red">

</v:polyline>

</body>
</html>
```

**Figure 12. A Simple VML Example**

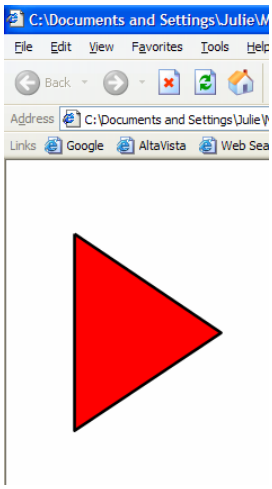



**Figure 13. A Web Page with VML**

# 5. XSLT and XPath

## *XSLT*

Extensible Stylesheet Language Transformations (XSLT) is a W3C Recommendation that can be used to accept XML data and convert this data into an HTML document or another XML document. The process of converting is referred to as translation.

XML data can be viewed as data organized in a tree-like structure. The XSLT transformation process converts the source XML data to another data that may not look the same as the original XML data. Figure 14 shows an example of a source tree and a result tree.

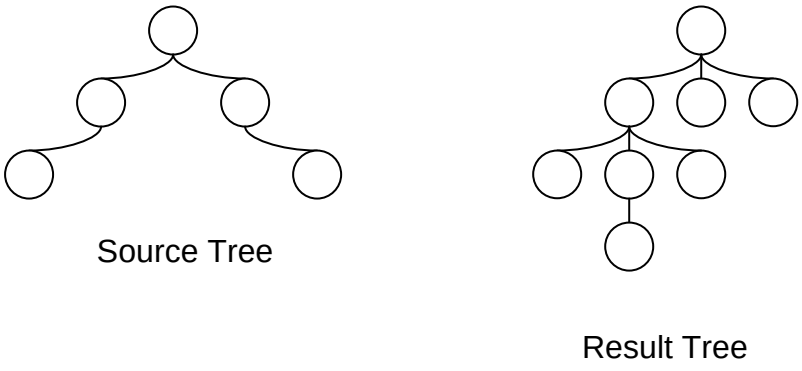



**Figure 14. A Source Tree and a Result Tree Example**

The file containing XSLT is referred to as a stylesheet. A stylesheet has template rules the XSLT writer can use to transform data to a different structure. In this project, our stylesheet contains:

- nodes or elements from the source tree
- rules, known as "template rules", on how to process the elements to form the result tree

Figure 15 shows an example of an XML file called `greeting.xml`, Figure 16 shows the stylesheet, `greeting.xsl`, transforming the XML file into an HTML file, and Figure 17 shows the result of the transformation.

```
<?xml version=”1.0” ?>
<?xml-stylesheet type=”text/xsl”
href=”greeting.xsl” ?>
<greeting>Hello, World!</greeting>
```

**Figure 15. XML File – greeting.xml**

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:template match="/">
            <html>
            <head><title>XSLT Example</title></head>
            <body>
            <h1>
                  <xsl:value-of select="greeting" />
            </h1>
            </body>
            </html>
      </xsl:template>
</xsl:stylesheet>
```

**Figure 16. XSLT File – greeting.xsl**

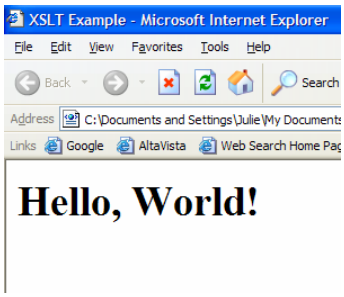



**Figure 17. Transformation Result**

The stylesheet in Figure 16 contains the following components:

- XML Declaration – an XML requirement

  <?xml version=“1.0“ ?>

- XSLT Document Element – root of XSLT document

  <xsl:stylesheet

- Version of XSLT Stylesheet

  version=“1.0“

- XSLT Namespace Declaration – to be able to use `<xsl:`

  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

- Template Rule – an instruction to find and process a node

  <xsl:template match="/"> - find and process the root, " / " , node

- Result Tree Elements – to be added to the output

  <html>

  <head><title>XSLT Example</title></head>

  <body>

  <h1>

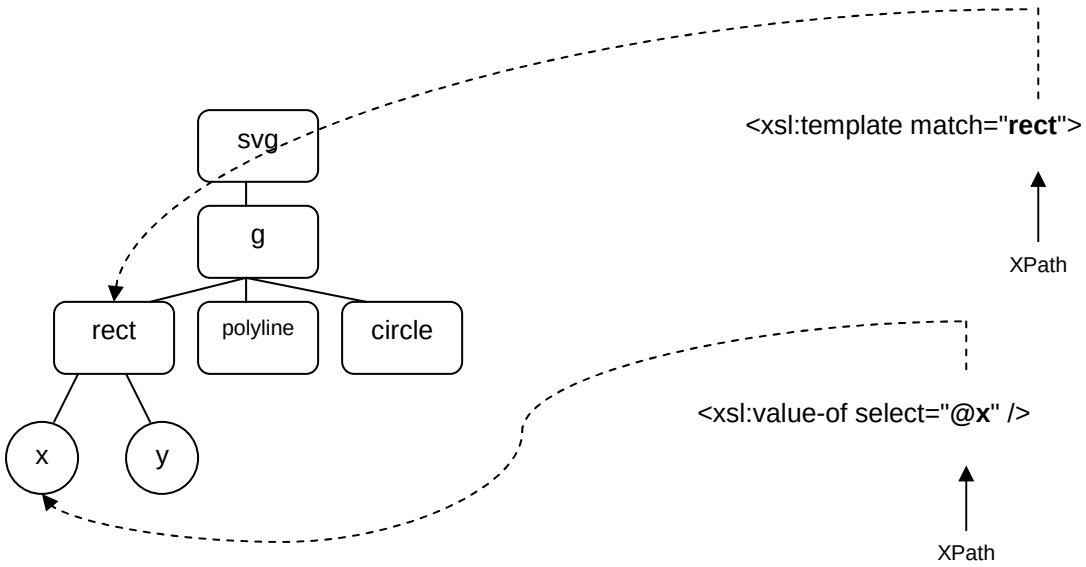
- XSLT Instruction – get the value of node equal to 'greeting'

  <xsl:value-of select="greeting" />

To develop the stylesheet, we also need to use XPath, another W3C Recommendation, to select which parts of the source document the stylesheet will process.

## *XPath*

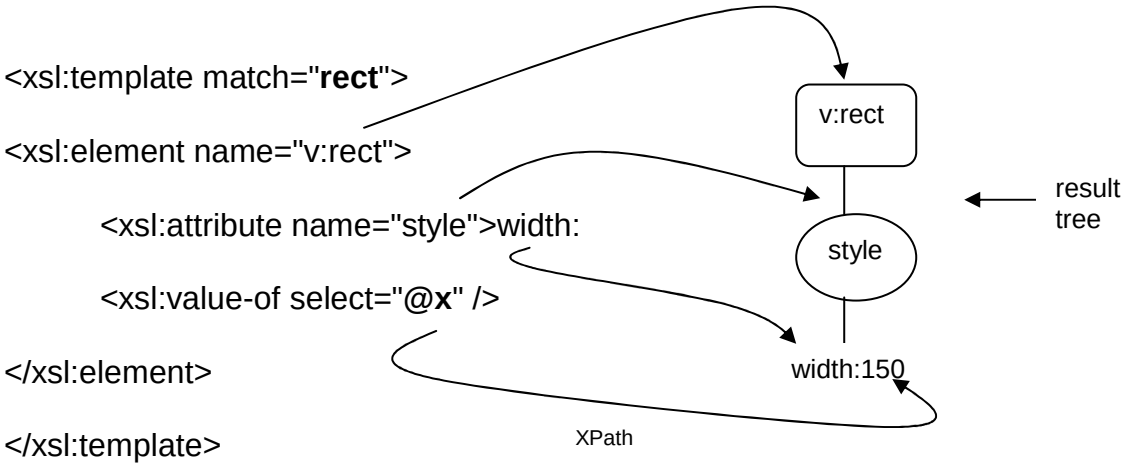
XPath stands for XML Path Language. It is a language we use with XSLT to select nodes from the source tree. Let us look at an example. From the source tree shown below, we will write an XPath expression to select the <rect> node, and then we will use XPath again to add a value to the result tree.




From the illustration above, the XPath expressions are:

- rect – choose the <rect> node
- @x – choose the "x" attribute of the node selected at the moment

Let us add more elements in the template shown above:




The first line of the template above tells the stylesheet, "whenever you come across a 'rect' node, select it." Then the remaining lines in the template add nodes to the result tree. Let us note that the only XPath expression above that adds to the result tree is "@x". The other elements in the result tree are elements added by the stylesheet writer.

In the next section, we will list some of the XSLT components we used in developing our stylesheet.

# 6. Some of the XSLT Elements Used in the Stylesheet

The stylesheet contains instructions on how to process the selected source nodes into another document. XSLT provides many functions on how to match a node, get the value of a node, continue processing other nodes, and many more. Table 1 lists some of the elements used by the stylesheet. The definition in Table 1 were taken from [H02].

| **XSLT Element** | **Definition** |
|---|---|
| <xsl:template> | a new template (or rule) |
| <xsl:apply-templates> | Selects nodes to be processed |
| <xsl:value-of> | Adds to the result tree the value of an expression |
| <xsl:if> | For conditional processing |
| <xsl:variable> | Stores value into a variable |
| <xsl:attribute> | Adds an attribute node in the result tree |
| <xsl:call-template> | Calls another template |
| <xsl:with-param> | Parameter used when calling another template |

**Table 1. Some of the XSLT Elements Used in the Stylesheet**

In traditional programming languages such as Java or C++, we can always replace the value of a variable. If we have assigned the value `20` to variable `x`, we can always write a line of code afterward similar to: `x = 30` (provided this line is within the scope of the variable). XSLT's variables are different. The way the language was designed was, once you have assigned a value to the variable, that value stays there and cannot be modified.

In this project, we find XSLT variables to be of great use. Below is an excerpt taken from the translator showing an XSLT variable called `style`.

```
<xsl:variable name="style">
```

# 7. XML DOM

XML Document Object Model (XML DOM) is another W3C specification to provide a way to access parts of an XML document. An XML parser can be used to load an XML document. After the XML document has been loaded, the stylesheet writer can then access the elements of this document.

We use Microsoft's XML DOM parser, which comes with Internet Explorer 6.0, to load both the SVG document and the stylesheet. Figure 18 demonstrates how an SVG document is loaded using JavaScript.

```
<script>...
var xml = new ActiveXObject("Microsoft.XMLDOM");
xml.async = false;
xml.load("mypicture.svg");
...
</script>
```

**Figure 18. XML DOM Example**

Below is an excerpt on how we load the stylesheet using XML DOM.

```
var xsl = new ActiveXObject("Microsoft.XMLDOM");
xsl.async = false;
xsl.load("translator.xsl");
```

The XML DOM method to do the actual translation is shown below:

```
document.write(xml.transformNode(xsl));
```

Microsoft extended the W3C methods by adding the `transformNode` method to provide a way to do stylesheet translation.

# 8. XSLT, JavaScript, and the Translator

To store some information more efficiently, we embedded JavaScript code inside the XSLT file. For example, an SVG document could contain several shapes with different radial gradients. JavaScript is used to store those radial gradients, which are later accessed as the VML shapes are drawn. Details of how this is done will be discussed in the next section. Figure 19 shows an example code from the stylesheet on how JavaScript is inserted at the beginning of the XSLT file.

```
<?xml version="1.0"?>
xmlns:msxsl="urn:schemas-microsoft-com:xslt"
…
<msxsl:script language="JavaScript" implements-
prefix="translate">
<![CDATA[

…

function getVMLcurvefrom(points)
{
      var arr = points.split(" ");
      var from = arr[0] + " " + arr[1];
      return from;
}

…

]]>
```

**Figure 19. JavaScript at the Beginning of XSLT File**

As shown in Figure 19, the JavaScript code is inside the CDATA section of the <msxsl:script> tag. A namespace declaration must also be included at the top of the XSLT file to be able to use the <msxsl>element within the stylesheet.

To access a JavaScript function, the prefix "translate" is used within the XSLT function. Figure 20 shows an example.

```
<xsl:attribute name="from">
      <xsl:value-of
      select="translate:getVMLcurvefrom(string(normalize-
      space($points)))" />
</xsl:attribute>
```

**Figure 20. JavaScript Function Call Inside an XSLT Function**

There are many JavaScript functions handling data storage and text parsing in our stylesheet. In the next section, we will continue with how we implemented the stylesheet.

# 9. The Stylesheet Implementation

## *Requirements*

To perform the translation we need the following requirements:

- translator.xsl – the XSLT stylesheet containing the template rules and processes to be followed
- SVG document – an SVG document containing any of the supported tags
- diplay.html – this HTML document contains the methods to load the SVG and XSLT files, transform the SVG document, and display the result
- browser – IE6.0 has full support for W3C's XSLT Recommendation

### *Stylesheet Design*

The main engine that does the transformation from SVG to VML is an XSLT stylesheet document called "translator.xsl". As mentioned earlier in this report, transformation is achieved through the use of stylesheet rules on how to create the result tree. There are several approaches to designing a stylesheet.

These approaches were observed by [H02] and later coined as "pulling data" or "pushing data". In this project, our stylesheet can be considered "push-oriented" because the structure of the source data is not known in advance. For example a source data, or an SVG document as in the case of this project, can contain several rectangles and some lines, while another can contain polylines and Bezier curves. Thus, our stylesheet should work with either one of those documents. We do not know what is inside the SVG document, so we just let the translator go through the source elements, then "push" these nodes to the stylesheet for further processing. The "push" approach is more suitable for our project in the sense that we can process the source document without having to consider its structure.

Our stylesheet contains many templates. The template that processes the root of the SVG document has the following syntax:

```
<xsl:template match="/">
...
<xsl:apply-templates />
...
</xsl:template>
```

In XSLT, the function "apply-templates" tells the root template to continue processing the remaining nodes using the other template rules in the stylesheet. By using "apply-templates" function, the stylesheet visits each node and allows the template rules to specify how to process this template. If the SVG document contains a rectangle and a circle, the stylesheet will go through each node and will let the templates process these nodes. Because the goal of this project is to transform SVG images to VML within an HTML file, the result tree will also contain a rectangle and a circle.

### *Stylesheet Structure*

Figure 21 shows the diagram of the templates used in our stylesheet. The lowest level templates can be considered "subroutines" called by the other templates.

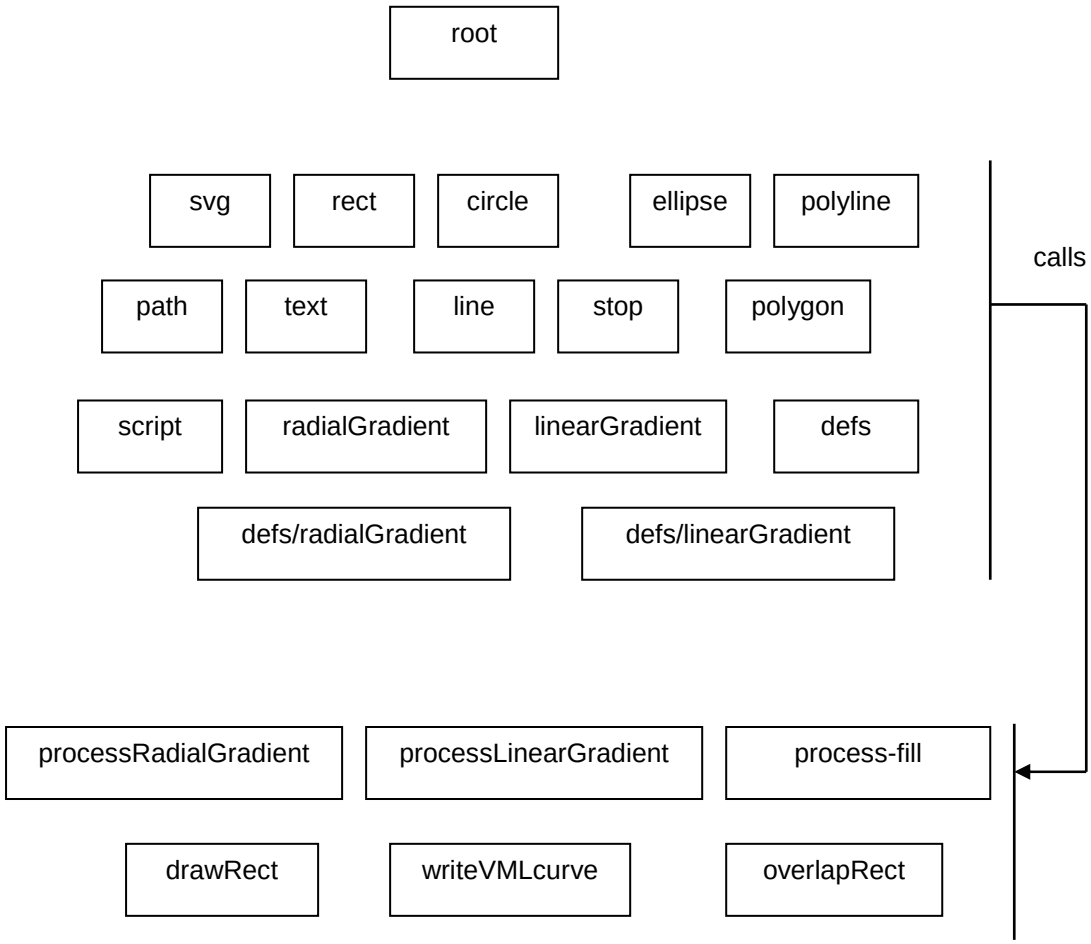



**Figure 21. Stylesheet Structure**

## *Stylesheet Design Techniques*

Design patterns are usually introduced in courses such as software engineering, object-oriented programming, as well as in game programming courses. Patterns are considered not only at the planning phase of software design, but also throughout the software development process, where we repeatedly test and modify our work. The goal of patterns is to help the software designers solve the programming problem by identifying patterns and reapplying them in current and future software projects.

A large number of materials exist with reference to design patterns in software. Most of the literature refers to design patterns with emphasis to objects interacting within a program. Are there design patterns that exist for stylesheet design? The answer is yes.

Patterns in object-oriented software denote how objects work together; however, in stylesheet design, patterns are more of "discovered" techniques that solve interesting problems. One example is the application of the mathematical "set" operation. In XSLT, the only set operation provided is the "union" operation. One design pattern acknowledged in the XSLT community is known as the "Kaysian" method, techniques that solve set intersection and difference discovered by Michael Kay. Other advanced techniques exist such as

"Muenchian" method for efficient grouping of nodes, and many more. The reader will find more discovered stylesheet design techniques in [B03].

Regarding techniques, we wish to share a technique, which probably has already been implemented by stylesheet writers in their own projects. We wish to share it because it also helped us in debugging our stylesheet. This technique concerns JavaScript function and XSLT variable.

To call a JavaScript function, we use the XSLT expression function `<xsl:value-of>`, then the XPath attribute `select` with the value equal to the function name. Below is an example taken from Section 8:

```
<xsl:attribute name="from">
     <xsl:value-of
     select="translate:getVMLcurvefrom(string(normalize-
     space($points)))" />
</xsl:attribute>
```

The example inserts the attribute `from` with a value returned by the function call. However, what if the function call is a JavaScript method that does not return anything, sometimes referred to as a "setter" function? We can still use the XSLT attribute `select`, but that attribute always returns something. Our solution is to create a "setter" function and make it return something, then we store that return value into an XSLT variable, whose value can be used for debugging

purposes. Below is a JavaScript excerpt from our stylesheet that stores a value in one of the global variables:

```
function setTextboxfill(fill)
{
     textboxfill = fill;
     return textboxfill;
}
```

Notice that although the function sets a value, it also returns a value that goes to the XSLT variable. The result is a function that is both a "getter" and a "setter". The XSLT function call is shown below:

```
<xsl:variable name="textfill"
     select="translate:setTextboxfill(string($textboxfill))" />
```

Now the value of the variable `textfill` can be used for debugging. After using the technique mentioned above, we also learned that we have to pass an XSLT value as a "string" if we want to parse it in JavaScript. We used the method mentioned above in almost every JavaScript function call to check if values were stored properly. In the next sections, we will discuss how we implemented the SVG elements the project supports. We start with the <svg> element.

### *The <svg> Element*

Below is an example of an empty SVG document.

```
<?xml version="1.0" ?>
<svg>
</svg>
```

The content of the SVG document is enclosed between the <svg> and </svg> elements. The <svg> tag can contain `width` and `height` attributes where the images will be contained. If there are no given `width` and `height`, the SVG document uses the default size: `height` of 100% and `width` of 100% of the browser window. Other units are also allowed in SVG such as `pixels, points, centimeters, inches`, and many more.

Internet Explorer's background is white by default, but if it is changed to gray, the resulting skeleton SVG document displays the file with a default off-white background. Thus, the <svg> tag is mapped to a VML rectangle with a white background. This rectangle will serve as the main background of all the remaining images to be rendered. The left frame on Figure 22 shows the skeleton SVG document in a browser with gray background.

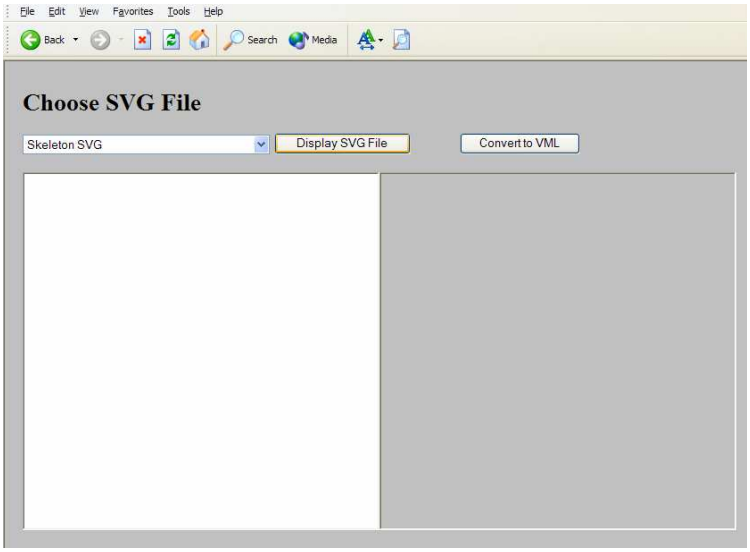



**Figure 22. A Skeleton SVG Document**

The nodes in an SVG document are contained within the <svg> tag, so to catch the remaining nodes of the source tree (the SVG document), the template rule for the <svg> tag contains the function “apply-templates” telling the stylesheet to continue processing the other nodes. Below is an excerpt from “translator.xsl”.

```
<xsl:template match="svg">
…
<xsl:apply-templates />
</xsl:template>
```

### *SVG Predefined Shapes*

The SVG predefined shapes our project supports are mapped to VML tags that draw the shapes. Table 3 lists the SVG and the corresponding VML tags used by the stylesheet.

| SVG | VML |
|---|---|
| <circle> | <v:oval> |
| <ellipse> | <v:oval> |
| <rect> | <v:rect> |
| <polyline> | <v:polygon> |
| <polygon> | <v:polygon> |
| <line> | <v:line> |
| <path> … (Bezier curve) | <v:curve> |
| <text> | <v:textbox> |

**Table 3. SVG Tags Mapped to VML Tags**

The algorithm for the template rule for each SVG predefined shape is as follows:

1. match the node
2. get node's attributes
3. store attributes in XSLT variables
4. check if parent is <g> element
    a. if yes, get parent's style attributes
    b. if no, get node's style attributes
5. check if next sibling is <text> attribute
    a. if yes, set textbox's fillcolor
    b. if no, do nothing
6. compute the equivalent VML geometric attributes using variables in Step 3
7. draw the corresponding VML shape
8. call the 'process-fill' template giving it the style attribute from Step 4

The SVG <g> tag is used to group shapes with similar properties to avoid redundant declaration of the same properties. It does not map to a VML tag and its implementation will be discussed in the latter part of this section.

### *The SVG <circle> and <ellipse> Elements*

The SVG <circle> and <ellipse> elements map directly to VML's <v:oval> element, but with some attribute differences. In the <circle> element, the `x` coordinate of the center of the circle is represented by the `cx` attribute, the `y` coordinate of the center is indicated by the `cy` attribute, and `r` refers to the radius attribute.

In VML, the <v:oval> element uses `top, left, width`, and `height` attributes to define the geometric information of an oval. Figure 23 compares the attributes of an SVG <circle> element and a VML <v:oval> element.

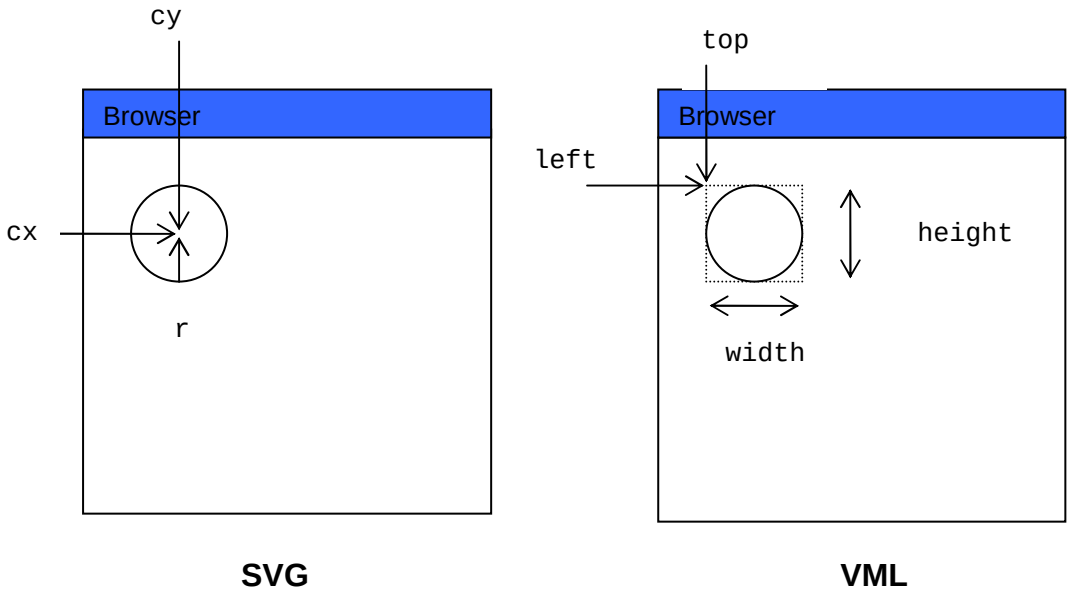



**Figure 23. SVG Circle and VML oval**

The calculation is straightforward. An SVG circle defined by radius 50, centered at coordinates 200, 200 can be drawn in VML using the computation shown in Table 4.

| SVG | VML |
|---|---|
| r = 50 | width = r * 2 |
| cy = 200 | height = r * 2 |
| cx = 200 | top = cy – r |
| | left = cx – r |

**Table 4. SVG Circle to VML Oval**

The following is an excerpt from the stylesheet's template rule for the <circle> node:

```
<xsl:template match="circle">
...
<xsl:element name="v:oval">
<xsl:attribute name="style">position:absolute;
    left:<xsl:value-of select="@cx - @r" />;
    top:<xsl:value-of select="@cy - @r" />;
    width:<xsl:value-of select="@r * 2" />;
    height:<xsl:value-of select="@r * 2" />
</xsl:attribute>
...
</xsl:element>
...
</xsl:template>
```

From the excerpt above, the <xsl:element name="v:oval"> writes <v:oval> to the result tree. Then <xsl:attribute name="style"...> adds the style attribute to the <v:oval> element. Thus, this element would become <v:oval style="..."> The value of the style attribute comes from the <xsl:value-of> element.

The only difference between the <ellipse> and <circle> elements are the `rx` and `ry` attributes that the <ellipse> element has. The calculations are similar to the <circle> element. The <ellipse> element is also mapped to the VML <v:oval> element.

### *The SVG <rect> Element*

The SVG <rect> element uses the attributes `x, y, width,` and `height`, which map directly to VML `left, top, width,` and `height` respectively for the <v:rect> element.

### *The SVG <polyline> and <polygon> Elements*

The <polyline> and <polygon> elements both contain the attribute `points` representing the `x` and `y` coordinates where the straight lines are drawn. They both map to VML's <v:polyline> element except for some adjustments to the

coordinates. The adjustments are calculated using a JavaScript function that subtracts 10 units from the x coordinate and 14 units from the y coordinate. Without this adjustment, theVML shape is rendered at a slightly lower location and slightly to the right of the browser. Figure 24, with the help of the dashed line, shows the position of the two images.

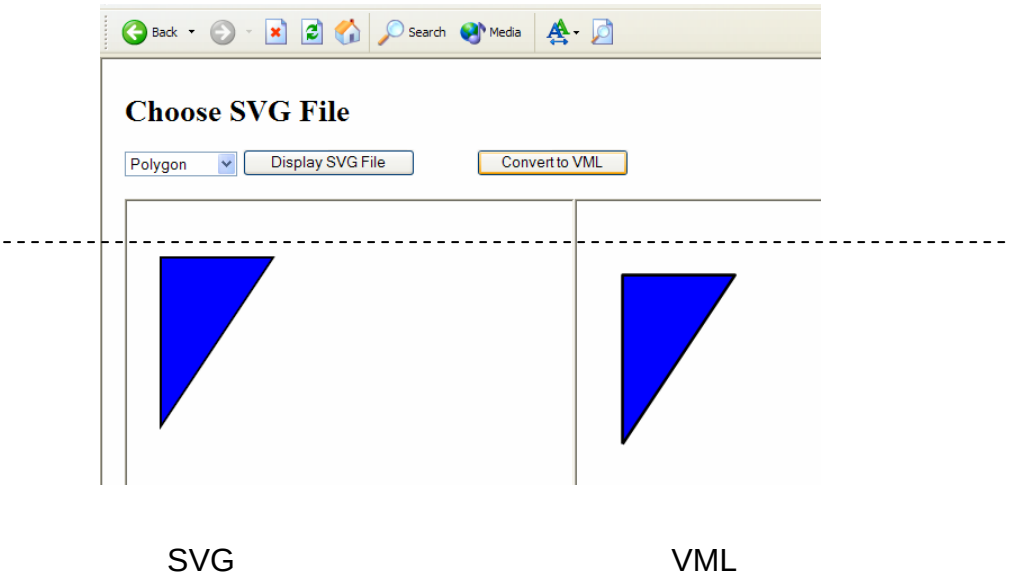



**Figure 24. Polygon in SVG and VML**

Below is the calling function excerpted from the polygon template.

```
<xsl:element name="v:polyline">
<xsl:attribute name="style">position:absolute;</xsl:attribute>
<xsl:attribute name="points">
    <xsl:value-of select="translate:getPoints(string(normalize-
            space(@points)))" />
</xsl:attribute>
...
</xsl:element>
```

The following is the JavaScript function that calculates the SVG points and returns the VML points:

```
function getPoints(svgpoints)
{
      var arr = svgpoints.split(" ");
      var points = "";

      //SEPARATE X AND Y POINTS
      for(var i=0; i < arr.length; i++)
      {
            var arrxy = arr[i].split(",");
            var x = arrxy[0] - 10;
            var y = arrxy[1] - 14;
            points = points + x + "," + y + " ";
      }
      return points;
}
```

### *The SVG <line> Element*

Just like <polyline> and <polygon> tags, the <line> element when mapped to VML <v:line> tag, appears slightly lower and to the right of the browser. Thus, the same calculation is applied to this element. Figure 25, with the aid of the horizontal dashed line, shows some lines without the adjustments.

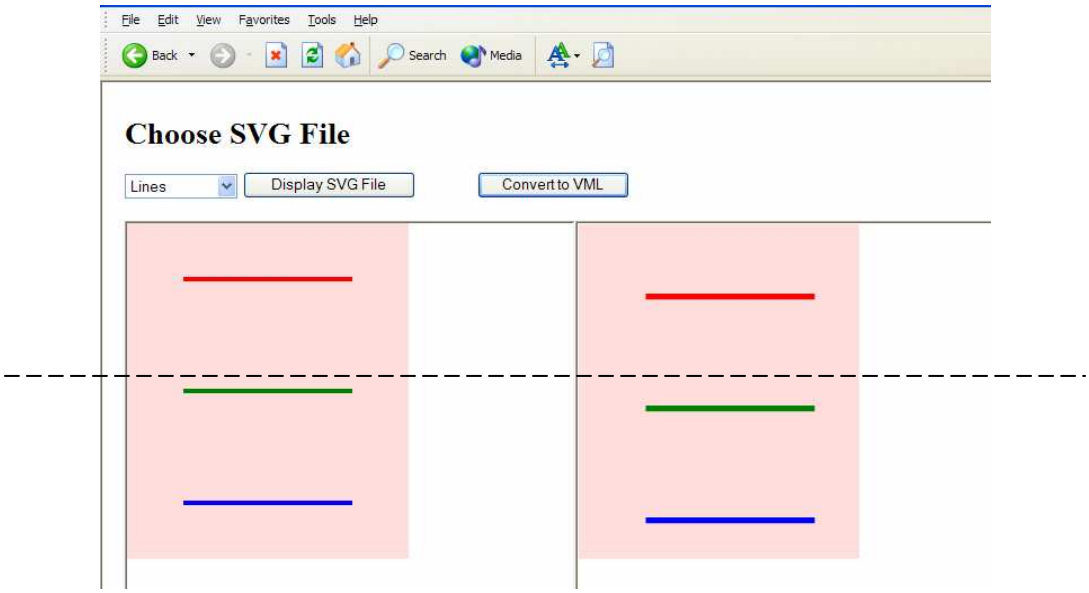



**Figure 25. SVG Line Translation**

Below is an excerpt from the stylesheet's template rule for the <line> node:

```
<xsl:element name="v:line">
<xsl:attribute name="from"><xsl:value-of select="$x1 - 10" />
<xsl:text>,</xsl:text>
<xsl:value-of select="$y1 - 14" />
</xsl:attribute>
<xsl:attribute name="to"><xsl:value-of select="$x2 - 10" />
<xsl:text>,</xsl:text>
<xsl:value-of select="$y2 - 14" />
</xsl:attribute>
```

### *The SVG <path> Element*

Our project supports cubic Bezier curve, which is defined under the <path> element. The cubic Bezier curve maps directly to VML's <v:curve> tag. The x and y coordinates are extracted through JavaScript functions. Table 5 shows the path's d attribute and the resulting VML curve attributes.

| SVG | VML |
|---|---|
| <path d="M20 20,<br>C300 50,<br>300 300,<br>20 20 z" /> | <v:curve from="20 20"<br>control1="300 50"<br>control2="300 300"<br>to="20 20" /> |

**Table 5. Cubic Bezier Curve**

Below is a JavaScript function excerpt from the stylesheet that calculates the points VML will use:

```
function getPathpoints(points)
{
	var startpt = "";
	var nextpt = "";
	var midpts = "";
	var endpt = "";

	var arr = points.split(" ");

	//REMOVE M
	if(arr[0].match("M"))
	{
	 var i = arr[0].indexOf("M");
	 startpt = arr[0].substring(i+1, arr[0].length+1) + " " + arr[1];
	}

	//REMOVE L
	if(arr[2].match("L"))
	{
	 var i = arr[2].indexOf("L");
	 nextpt = arr[2].substring(i+1, arr[2].length+1) + " " + arr[3];
	}

	//REMOVE Z
	if(arr[arr.length-1].match("z"))
		endpt = startpt;

	//CONCATENATE REMAINING POINTS
	for(i=4; i<arr.length-1; i++)
		midpts = " " + midpts + " " + arr[i];

	var newpoints = startpt + " " + " " + nextpt + " " + " " + midpts
		+ " " + endpt;
	return newpoints;
}
```

Figure 26 shows the result of translating a Bezier curve.

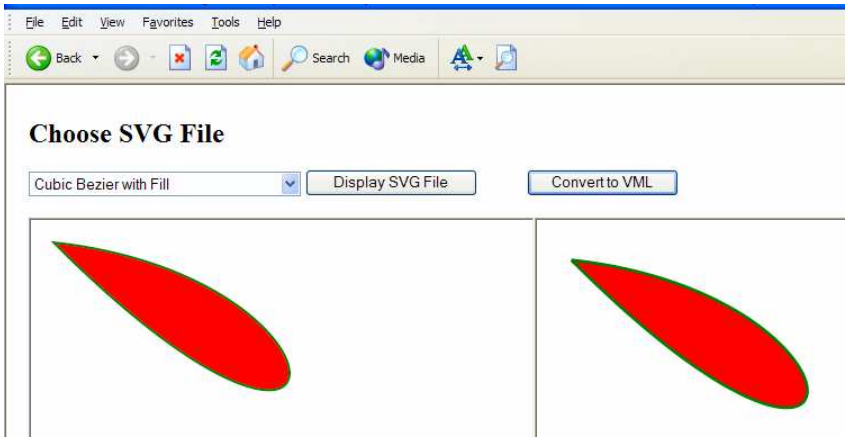



***Figure 26. Bezier Curve Translation***

## *Gradients*

Gradient is a smooth transition of one color to another. There are two types of gradient in SVG: linear and radial. Predefined shapes can use gradients as fill colors. Linear gradients can be horizontal, vertical, or diagonal. This project supports radial, horizontal, and vertical linear gradients of up to two colors with offsets of 0% and 100% to indicate smooth transition.

An SVG document can contain several gradients that are accessed by the shapes using the gradient `id` attribute. As mention earlier, once a value is bound to an XSLT variable, that value stays there and cannot be changed. For this reason, all gradient information is stored as hashtable objects in JavaScript.

Table 6 summarizes the hashtable objects and their purposes.

| | |
|---|---|
| **colorsTable** | stores id-colors / colors |
| **parentsTable** | stores id / 'radialgradient'<br>OR<br>id / 'lineargradient' |
| **radialGradientTable** | stores id / colors |
| **linearGradientTable** | stores id / y2, colors |

**Table 6. Hashtable Objects Used by the Stylesheet**

Gradients in SVG are nested inside the <defs> element, so when the stylesheet encounters this tag, it calls <xsl:apply-templates /> function to process the next element of either <radialGradient> or <linearGradient>. As shown in Table 6, the `linearGradientTable` stores the `y2` attribute to indicate a vertical gradient, otherwise it stores horizontal, which is the default in SVG if `y2` is omitted. In addition, both tables store the colors and their offset information. This is accomplished by getting the hashtable values from the colorsTable, which returns an array, then appending '`-colors`' to the `id` attribute to make it

different from the 'id'. The linearGradientTable would typically store data as shown in Table 7.

| id | value |
|---|---|
| red_gradient | 100% |
| red_gradient-colors | 0% red<br>100% yellow |
| blue_gradient | 100% |
| blue_gradient-colors | 0% green<br>100% blue |

**Table 7. An Example Hashtable Object for Linear Gradient**

Radial gradients appear differently in VML. Figure 27 and 28 illustrate this difference.

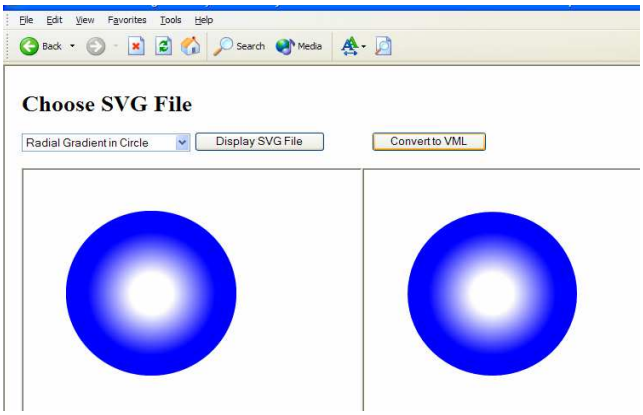



**Figure 27. Radial Gradient in Circle**

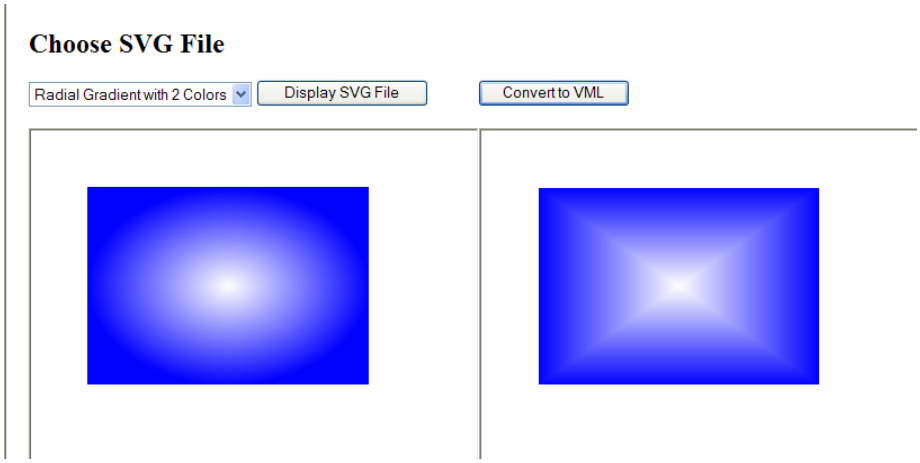



**Figure 28. Radial Gradient in Rectangle**

As illustrated by Figures 27 and 28, radial gradient appears differently in VML if applied to a rectangle. In VML, the gradient shape follows the shape of the rectangle. So how do we approach this discrepancy? Our solution is, if the shape maps to a VML <v:oval> element, the current template simply calls the 'processRadialGradient' template. However, if the shape is a rectangle, a slightly different algorithm is applied to this template. The steps are as follows:

1. Get the rectangle variables (`x, y, width, height`)
2. Get the gradient `id` attribute
3. Create an oval based on the variables on Step 1
4. Apply the gradient on Step 2 to the oval created in Step 3
5. Draw the rectangle
6. Draw the oval in Step 3 at the absolute position of the rectangle

Figure 29 shows a snapshot after the steps above have been implemented.

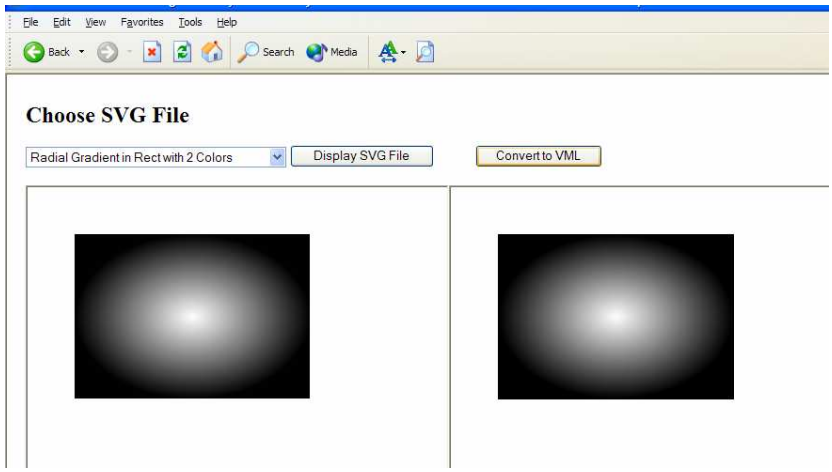



**Figure 29. Radial Gradient Implementation**

Both linear and radial gradients map to VML's <v:fill> element with the `type` attribute set to either `gradient` for linear or `gradientradial` for radial. Appendix C shows the complete stylesheet implementation.

### *The SVG <text> Element*

SVG texts are written within the SVG canvas, that is inside the enclosing <svg> and </svg> tags, while VML texts are shown within a browser between <html> and </html> tags. The project supports SVG text by using the x and y coordinates of the SVG <text> element and mapping them to a VML rectangle or oval element on which the text is drawn. The VML shape on which the text is drawn has a 'transparent' property as defined by its `opacity` attribute. This project supports text drawn on an SVG circle, rectangle, ellipse, or directly on the SVG canvas. The resulting VML text is best displayed when the browser is set to its default text size. Figure 30 shows a snapshot.

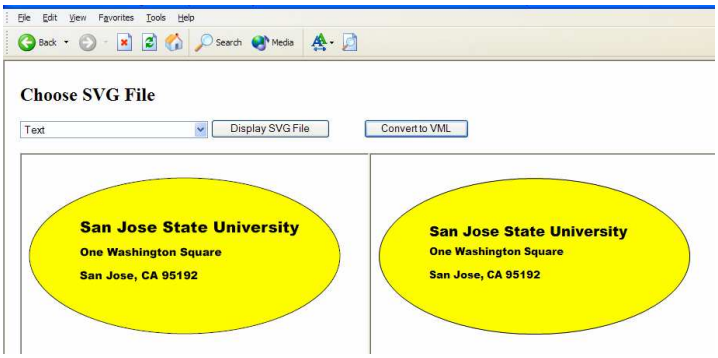



**Figure 30. Text Translation**

### *The SVG <g> Element*

The <g> element is used to group elements. An SVG writer can use the <g> element to avoid repeating the same attributes many times within the document. For example, if an SVG document contains three circles with the same properties, without using the <g> element, the code would appear:

```
<circle rx="50" ...
     style="fill:blue; stroke:black; stroke-width:4 />
<circle rx="150" ...
     style="fill:blue; stroke:black; stroke-width:4 />
<circle rx="250" ...
     style="fill:blue; stroke:black; stroke-width:4 />
```

The <circle> elements can be contained within the <g> and </g> container element to avoid repeating the style attribute three times. Below is a shorter code:

```
<g style="fill:blue; stroke:black; stroke-width:4">
     <circle ... />
     <circle ... />
     <circle ... />
</g>
```

There is no equivalent VML element for the SVG <g> element, as mentioned earlier in *Predefined Shapes* section. To take advantage of the <g> element, we implement it in the stylesheet by using the following lines of code:

```
<!--CHECK IF PARENT TAG IS 'G'-->
<xsl:variable name="style">
<xsl:if test="name(parent::*) = 'g' " />
      <xsl:value-of select="string((parent::*)/@style)" />
<xsl:if test="name(parent::*) != 'g' "/>
      <xsl:value-of select="string(@style)" />
</xsl:variable>
```

Once the stylesheet matches a predefined shape node (e.g. circle, or rect, ...) it checks whether the previous node or its parent node is a <g> element. To check this, we use the <xsl:if> function. For code readability, we use the unabbreviated XPath expression `name(parent::*)` in this template. If we would use the abbreviated syntax, the XPath expression would be `name(..)` . That expression is part of the conditional expression that checks if the parent node is equal to 'g'. As opposed to what is found in traditional programming language, there is no matching `'else'` element in XSLT, so we repeat the test with another <xsl:if> function. This time we check if the parent node is not equal to 'g'. Notice the use of `!=` syntax to indicate a `'not'` in the expression. In SVG the <g> element can be nested in many levels, but this project supports only one level. Figure 31 shows a snapshot of translating an SVG document containing the <g> element.

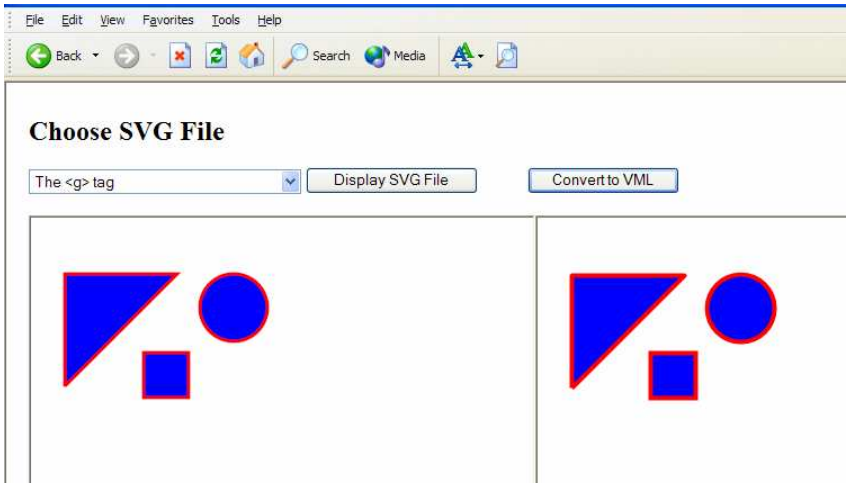



**Figure 31. Translation of <g> Tag**

### *The SVG style Attribute*

XSLT can parse strings using `substring`, `substring-before`, `string-length`, and other built-in string functions. An SVG writer can type the same style attribute for an element in many ways, some of which are:

```
<... style="fill:red; stroke-width:4; stroke:black">
     OR
<... style="stroke-width:4; stroke:black; fill:red">
     OR
<... style="fill:red; stroke:black; stroke-width:4">
```

In short, the order in which the style properties appear cannot be predicted. Because of this, the style attribute is parsed using JavaScript functions. Below is an excerpt of the JavaScript function that returns the `fill` property.

```
var arr = style.split(" ");
var fill = "";
for(var i=0; i < arr.length; i++)
{
	if(arr[i].match('fill'))
	{	//CHECK IF FILL IS GRADIENT
		if(arr[i].match('url'))
		{
			fill = arr[i].substring(10, arr[i].length-1);
			return fill;
		}
		else
		{
		fill = arr[i].substring(5, arr[i].length);
		return fill;
		}
	}
}
```

As mentioned earlier, SVG attributes can use units in `pixels`, `centimeters`, `points`, or `inches`. Figure 32 illustrates an SVG document that has a rectangle using `inches` as units, then translated to VML.

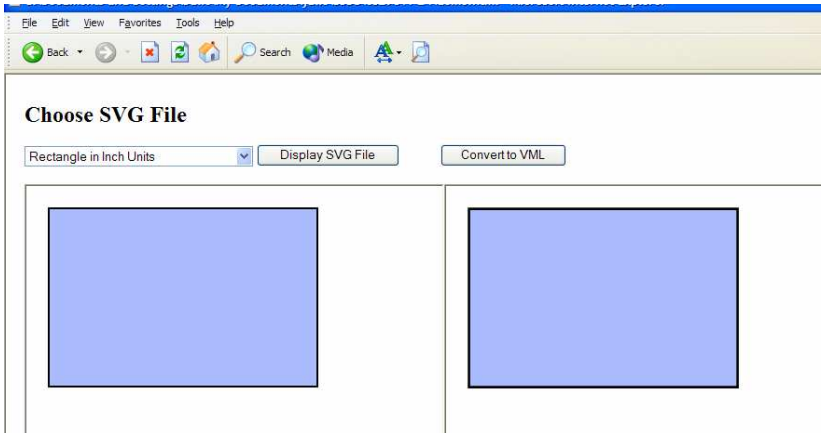



**Figure 32. An SVG Rectangle Using Inches as Units**

As illustrated in Figure 32, VML also supports the units mentioned above. Thus, these different units are mapped directly to the same units in VML.

### *The <script> Element*

The <script> element is one of the most interesting elements we integrated in this project because it does not map to a VML element, and it contains scripting code such as JavaScript to handle function calls. An SVG <script> element can contain functions to calculate coordinates, code to handle mouse events, or XML DOM commands to access nodes of the document tree.

We used an SVG document with a <script> element that would draw another element on the screen once the left mouse button is clicked. In SVG, this is accomplished by calling XML DOM commands that would obtain parts of the source document tree. Below is a fragment:

```
var SVGDoc = evt.getTarget().getOwnerDocument();
var SVGRoot = SVGDoc.getDocumentElement();
    ...
SVGRoot.appendChild(myShape);
```

From the code above, `evt` is a mouse event object, that when used, can convey which element of the document was clicked. The methods `getTarget()` and `getOwnerDocument()` are W3C's DOM methods used in conjunction with the event object to find out which document and document part was clicked by the user.

As mentioned above, the SVG document we tried contains scripting code that draws another image. The challenge we faced was how to draw the new image after the fact that the translation already took place. Before we explain the solution, let us first look at Figure 33.

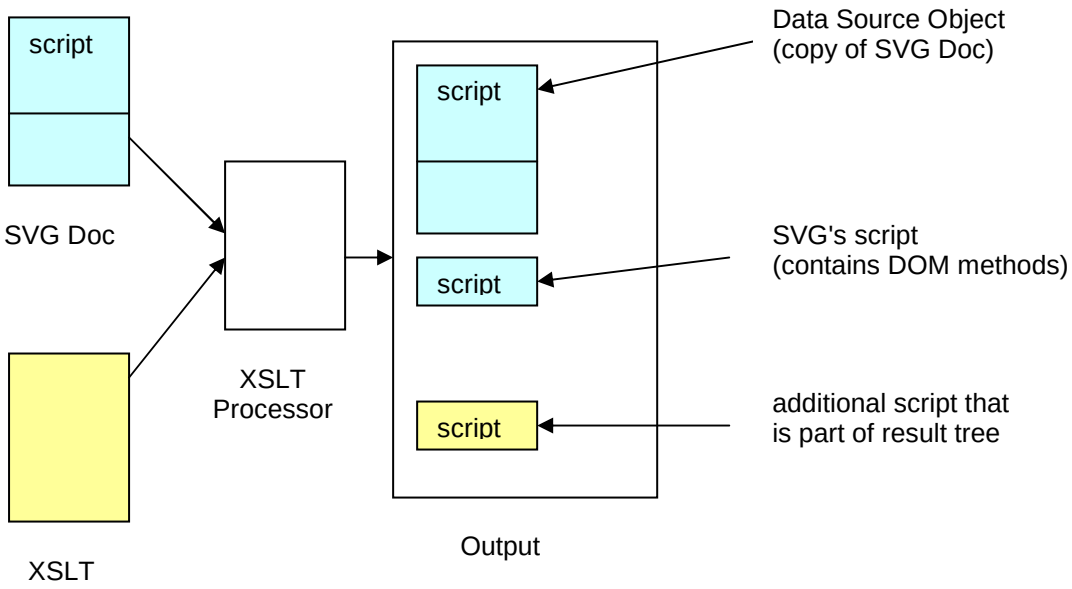



**Figure 33. Document Output with XML Data Source Object**

From the illustration above, we created an XML Data Source Object (DSO) to hold a copy of the original SVG document. We did that for two purposes: 1) to pass that object to the script as the source of the mouse event and, 2) to use it to

do another transformation. We also included a mouse event to the VML element so when the user clicks it, it will notify the event handler, which is JavaScript code that is part of XSLT's result tree. The sequence is:

1. the user clicks the VML element
2. the element calls the mouse event handler
3. the handler synthesizes a mouse event
4. the handler attaches the event to the Data Source Object
5. the handler calls the SVG script code
6. the SVG script code uses DOM methods to get the document and element that caused the mouse event
7. the SVG script code receives the Data Source Object
8. the SVG script code appends a node to this object
9. the handler calls a function that transforms the Data Source Object
10. the current window is cleared, the result tree is drawn

Before we continue, let us look again at item number 6 from the sequence above. Here the SVG script code receives a mouse event object that is used with DOM methods to get the document and element that triggered the mouse event. We had a problem using these DOM methods because Microsoft does not support the mouse event methods provided by W3C, also known as "DOM2 Events". We handled the incompatibility by creating our own object, `My_Event`, that loads the

Data Source Object as an XML document, and with a method that returns the XML document. We also replaced the method `getTarget().getOwnerDocument()` with our own `getXML()` method so we can pass the XML object (the original SVG document) to the caller. Below is an excerpt from the result tree:

```
function My_Event()
{
	xml = source.XMLDocument;
	xml.async = false;
	//LOAD DATA SOURCE OBJECT CONTAINING SVG
	xml.loadXML(source.altHtml);
	this.getXML = event_getXML;
}

function event_getXML()
{
	return xml;
}
```

Figure 34 shows a snapshot of an SVG document containing JavaScript code.

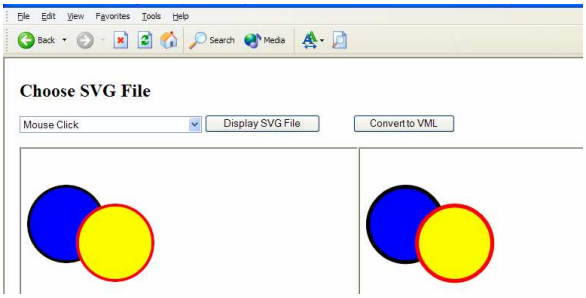



**Figure 34. Result of Double Transformation**

In addition to the script element, we also included "onclick" and "mouseover" events that would display alert windows. Figure 35 show a snapshot:

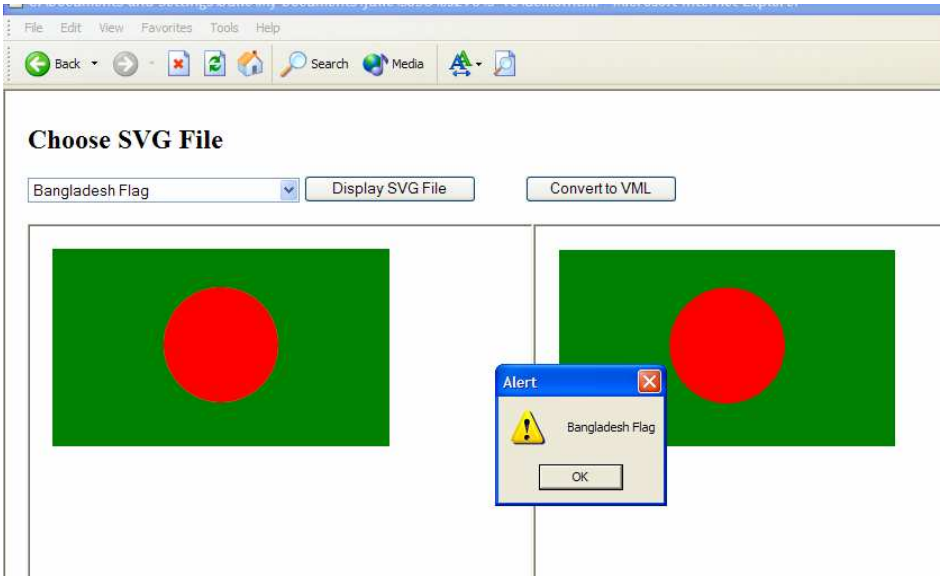



**Figure 35. Mouseover on Rectangle**

The reader will find more details regarding our stylesheet in Appendix C. In addition, more information about XML Data Source Object can be found by visiting [XDSO]. The next section shows an example of using our stylesheet to transform an SVG document containing an organizational chart.

## *Stylesheet Translation of SVG to VML Demonstration*

Figure 36 demonstrates the result of using our stylesheet to translate an SVG document containing a small organizational chart.

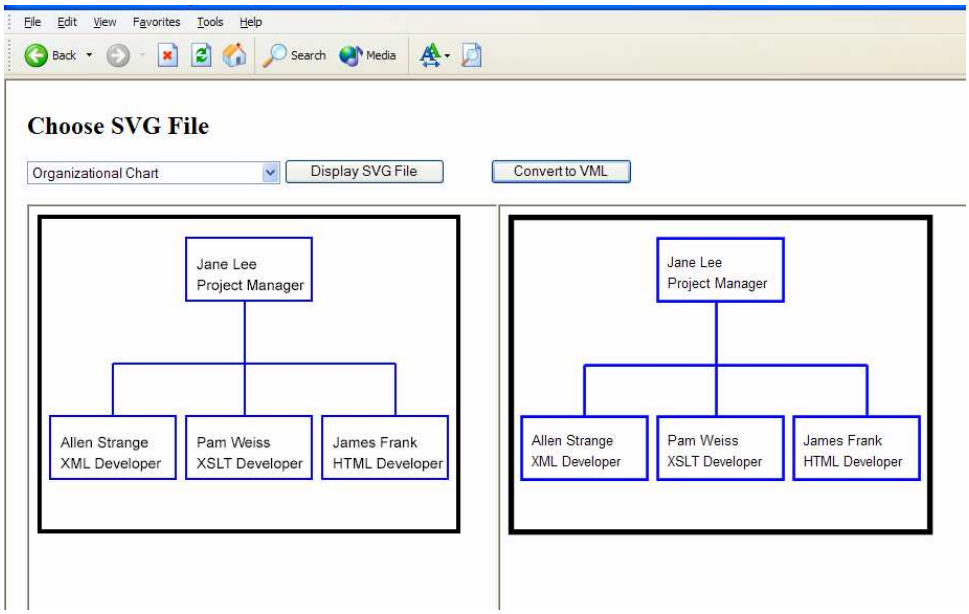



**Figure 36. SVG to VML Stylesheet Translation**

In the next section, we will discuss the limitations of our stylesheet.

# 11. The Stylesheet Limitations

SVG is a W3C specification that has a lot of powerful features. With SVG, an image can be transformed, moved, and animated. In addition, SVG can zoom in and out images just by clicking the right mouse button.

This project has many limitations compared to the full capability of SVG. Some of the limitations are the lack of support for many visual effects such as blending of images, drop shadows, diagonal gradients, and multi-level grouped elements. Gradients in SVG can contain more than two colors. That aspect is not implemented in our stylesheet. Regarding the line element, the translator supports only straight lines. In addition, the project lacks complete support for scripting codes that would handle different keyboard and mouse events.

This project can be viewed as a preliminary endeavor to convert selected SVG graphics images to VML images.

# 10. Project Conclusion

In this project, we have developed a stylesheet that translates SVG images to VML images displayed without the required plug-in. The stylesheet supports geometric shapes such as rectangle, circle, ellipse, line, polygon, and polyline. In addition, it also supports a special curve known as Bezier curve.

Our stylesheet also translates SVG texts displayed within the canvas or SVG's display area, rectangle, circle, and ellipse. The stylesheet can display combination of shapes and texts as demonstrated by the organizational chart.

Shapes can have color backgrounds or fills. Our translator supports shapes with basic colors as well as gradients. It supports radial and linear gradients of up to two colors. In addition, it can also support both horizontal and vertical linear gradients. Support for gradients with more than two colors can be left for future work. Different shapes can have the same attributes. The translator supports this feature known as grouped elements, but only in one level. Support for multi-level grouping can be left for future work.

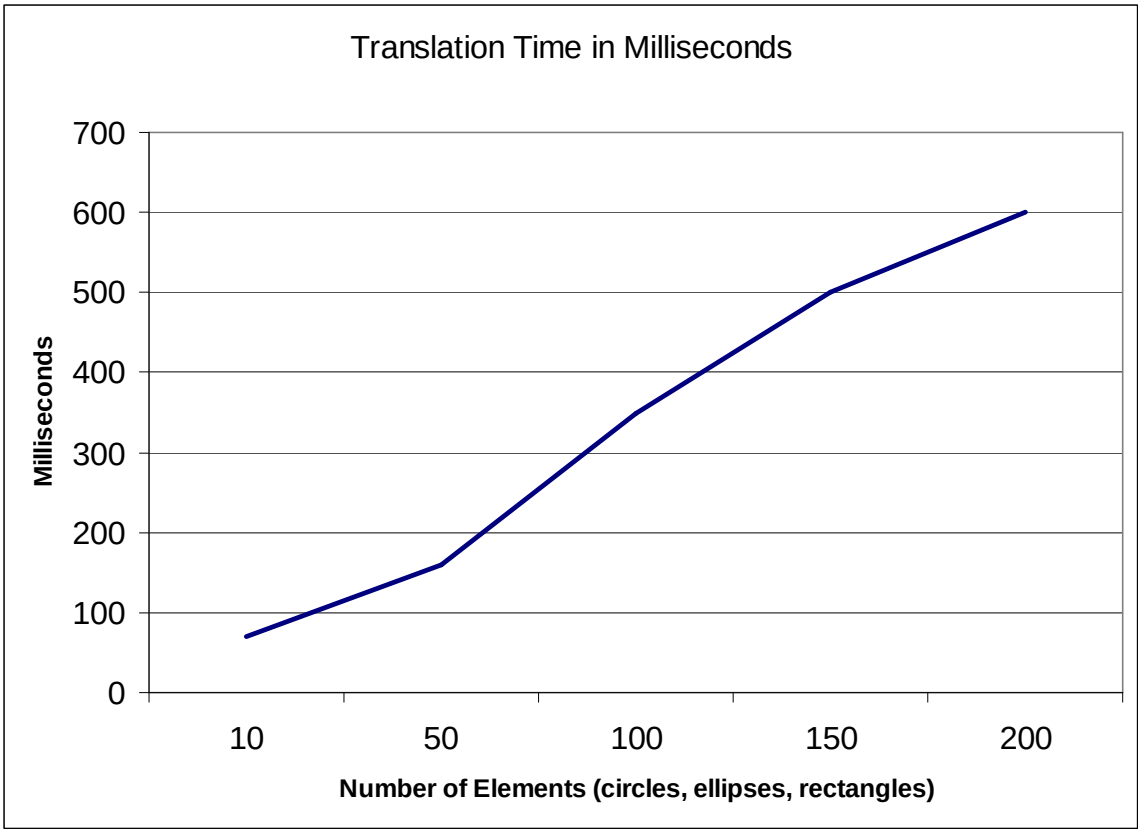
We supported SVG script code by creating an object that acts like a DOM mouse event to manage the incompatibility between W3C and Microsoft. We also used

an XML Data Source Object to store the original SVG document, which will be used for another transformation. Support for keyboard and other mouse events can be left for future work.

XSLT has limitations when compared to traditional programming. We have overcome those restrictions by using a scripting language, JavaScript. However, we found it very tough to rely on the browser to find and solve script and XSLT errors.

Although this project has limitations, the translator can be used as a tool to convert simple but interesting images to VML images displayed in a browser, as illustrated at the end of Section 9 without the required plug-in.

We also conducted an experiment, using a computer with 1.60 GHz CPU and 256MB of RAM, to compare the time it takes for our program to do its translation based on the number of elements from the source document. The chart below shows the result of our experiment.




The chart above shows that the more elements the source document has, the time to do the translation increases. With 200 elements from the source document, our stylesheet responded in less than a second. We can conclude that stylesheet translation could be as efficient as viewing a document with a required plug-in.

The table below shows the actual data from the above experiment:

| Num of Elements | Translation Time in Milliseconds |
|---|---|
| 10 | 70 |
| 50 | 160 |
| 100 | 350 |
| 150 | 501 |
| 200 | 601 |

Many software development lessons were learned during the formation of this project. Some of them are the challenges of learning a non-traditional programming language XSLT, understanding the syntax of XPath expressions, and how to use XSLT, XPath, JavaScript, and XML DOM to create an exciting software. Debugging techniques were also learned in this project. Because the stylesheet is structured, we hope that this project can be extended without difficulty.

# 11. Bibliography


| | |
|---|---|
| [AGS] | Adobe Graphics Server. http://www.adobe.com/products/server/graphics/main.html |
| [B03] | Bondre, P. XSLT – Efficient Programming Techniques. http://www.xml.org/xml/xslt_efficient_programming_techniques.pdf |
| [DOM04] | W3C Document Object Model. http://www.w3.org/DOM/ |
| [G04] | GraPL. http://www.grapl.com/desktop/index.html |
| [H02] | Holman, G. Definitive XSLT and XPath. Prentice Hall. Upper Saddle River, NJ. 2002 |
| [L04] | Lee. W. M. "Transforming XML into WML." http://www.wirelessdevnet.com/channels/wap/training/xslt_wml.html |
| [M04] | Map2SVG. http://www.gis-news.de/svg/map2svg.htm |
| [MSDN98] | Introduction to Vector Markup Language. http://msdn.microsoft.com/library/default.asp?url=/workshop/author/vml/. MSDN 1998. |
| [R04] | Ryan, J. "Transforming Flat Files to XML Using SAX and XSLT." http://www.developer.com/xml/article.php/2108031 |
| [S04] | Schemasoft. http://www.schemasoft.com/mathml/index.htm |
| [SVG03] | Scalable Vector Graphics (SVG) 1.1 Specification. http://www.w3.org/TR/SVG/ |
| [VML98] | Vector Markup Language (VML). http://www.w3.org/TR/NOTE-VML |
| [W03] | Watt A, Lilley C., Ayers, D., George, R., Wenz, C., Hauser T., Lindsey K., Gustavsson, N. SVG Unleashed. Sams Publishing. Indianapolis, Indiana. 2003 |
| [W04] | W3Schools. http://www.w3schools.com |
| [X03] | XML DOM. http://www.devguru.com/Technologies/xmldom/quickref/node_transformNode.html |
| [XDSO] | XML Data Source Object. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xmlsdk30/htm/ontransformnode_event.asp |
| [XML99] | XML Path Language. http://www.w3.org/TR/xpath |
| [XSLT99] | XSL Transformations (XSLT). http://www.w3.org/TR/xslt |

# Appendix A – Glossary

**HTML** Hyper Text Markup Language – a language to create documents to be viewed on the Web. An HTML document contains markup up tags describing the structure of the web page.

**JavaScript** A scripting language that can be embedded on Web documents to add interactivity. It is supported by Netscape and Internet Explorer.

**Stylesheet** A way of defining how documents and their elements should be rendered

**SVG** Scalable Vector Graphics – an XML-based application to create two-dimensional images useful for charts, graphs, and statistical data

**VML** Vector Markup Language – an XML-based application to create two-dimensional images which can be viewed in Internet Explorer 5.0 and higher

**W3C** World Wide Web Consortium – an organization created in 1994 that develops interoperable software, guidelines, tools, and specifications to make the Web available to all users.

**XML** Extensible Markup Language – a language to structure and store data. It describes data using markup tags such as <customer>, <product>, etc.

**XMLDOM** XML Document Object Model – a programming interface to provide the programmer a way to create, access, and modify an XML document

**XPath** XML Path Language – a language that contains expressions on how we can locate information in an XML document. It is an part of an XSLT document.

**XSLT** Extensible Stylesheet Language Transformation – a language used to change the look of an XML document into another XML document, HTML document, or text document.

# Appendix B – Some of the SVG Files Used by the Project

## *SVG File in Figure 30 - Text Translation*

```
<?xml version="1.0" standalone="yes" ?>
<svg width="100%" height="100%">

 <ellipse cx="210" cy="130" rx="200" ry="100"
        style="stroke:black; stroke-width:1; fill:yellow" />

<text x="75" y="100" style="font-family:'Arial Black'; font-size:20;
fill:black; stroke:none">
        San Jose State University
</text>

<text x="75" y="130" style="font-family:'Arial Black'; font-size:14;
fill:black; stroke:none">
      One Washington Square
</text>

<text x="75" y="160" style="font-family:'Arial Black'; font-size:14;
fill:black; stroke:none">
      San Jose, CA 95192
</text>

</svg>
```

## *SVG File in Figure 36 – Organizational Chart*

```
<?xml version="1.0" ?>
<svg width="100%" height="100%">
      <rect x="10" y="10" width="400" height="300"
            style="stroke:black; stroke-width:4; fill:white" />

      <!--FIRST LEVEL BOX-->
      <rect x="150" y="30" width="120" height="60"
            style="stroke:blue; stroke-width:2; fill:white" />

      <text x="160" y="60"
            style="font-family:'Arial'; font-size:14; fill:black;
            stroke:none">
        Jane Lee
      </text>

      <text x="160" y="80"
            style="font-family:'Arial'; font-size:14; fill:black;
            stroke:none">
        Project Manager
       </text>

      <g style="stroke:blue; stroke-width:2">
            <!--LINE CONNECTING TWO LEVELS-->
            <line x1="206" y1="90" x2="206" y2="150"/>

            <!--HORIZONTAL LINE-->
            <line x1="80" y1="150" x2="350" y2="150"/>

            <!--THE THREE VERTICAL LINES-->
            <line x1="80" y1="150" x2="80" y2="200"/>
            <line x1="206" y1="150" x2="206" y2="200"/>
            <line x1="350" y1="150" x2="350" y2="200"/>
      </g>

      <!--FIRST BOX, SECOND LEVEL-->
      <rect x="20" y="200" width="120" height="60"
            style="stroke:blue; stroke-width:2; fill:white" />

      <g style="font-family:'Arial'; font-size:14; fill:black;
      stroke:none">
            <text x="30" y="230"> Allen Strange </text>
            <text x="30" y="250"> XML Developer </text>
      </g>
```

```
      <!--SECOND BOX, SECOND LEVEL-->
      <rect x="150" y="200" width="120" height="60"
            style="stroke:blue; stroke-width:2; fill:white" />

      <g style="font-family:'Arial'; font-size:14; fill:black;
      stroke:none">
            <text x="160" y="230"> Pam Weiss </text>
            <text x="160" y="250"> XSLT Developer </text>
      </g>
      <!--THIRD BOX, SECOND LEVEL-->
      <rect x="280" y="200" width="120" height="60"
            style="stroke:blue; stroke-width:2; fill:white" />
      <g style="font-family:'Arial'; font-size:14; fill:black;
      stroke:none">
            <text x="290" y="230"> James Frank </text>
            <text x="290" y="250"> HTML Developer </text>
      </g>
</svg>
```

# Appendix C – Stylesheet Code

filename: translator.xsl

```
<?xml version="1.0"?>
<!--AUTHOR: JULIE NABONG-->
<!--STYLESHEET THAT TRANSLATES SVG TO VML-->
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:v="urn:schemas-microsoft-com:vml"
 xmlns:msxsl="urn:schemas-microsoft-com:xslt"
 xmlns:translate="http://translate">
<xsl:output method="html" indent="no"/>

<msxsl:script language="JavaScript" implements-prefix="translate">
<![CDATA[

        var linearGradientTable = new Hashtable();
        var radialGradientTable = new Hashtable();
        var colorsTable = new Hashtable();
        var parentsTable = new Hashtable();
        var textboxfill = "";

function getFunctionName(scriptText)
{
        var arr = scriptText.split(" ");
        //ARR[2] CONTAINS FUNCTION NAME

        var closeParenIndex = arr[2].indexOf(')');
        var name = arr[2].substring(0,closeParenIndex + 1);
        return name;

}

function getNewScript(script)
{
        //REPLACE DOM2 EVENTS WITH BROWSER EVENTS
        var text = script.replace('getTarget().getOwnerDocument()','getXML()');
        var newText = text.replace('getDocumentElement()', 'documentElement');
        return newText;
}
```

```
function setTextboxfill(fill)
{
	textboxfill = fill;
	return textboxfill;
}

function getTextboxfill()
{
	return textboxfill;
}

function storeLinearGradientInfo(id, y2)
{
	linearGradientTable.put(id, y2);
	return y2;
}

function getY(id)
{
	var y2 = linearGradientTable.get(id);
	return y2;
}

function storeParent(id, parentname)
{
	var parent = parentname;
	parentsTable.put(id, parent);
	return parent;
}

function getParent(id)
{
	var parent = parentsTable.get(id);
	return parent;
}

//THIS FUNCTION IS CALLED MANY TIMES
function storeColors(id, color, info)
{
	var colorID = id + '-' + color;
	colorsTable.put(colorID, info);
	return colorID;
}
```

```
function storeColorsInTable(id)
{
	//RETURNS AN ARRAY
	var colorsArray = colorsTable.values();
	var newColorID = id + '-colors';

	//CHECK PARENT
	var parent = getParent(id);

	if(parent.match('radial'))
		radialGradientTable.put(newColorID, colorsArray);
	else
		linearGradientTable.put(newColorID, colorsArray);

	//CLEAR COLORSTABLE FOR NEXT SET OF COLORS
	colorsTable.clear();
	return newColorID;
}

function getColors(id)
{
	//CHECK PARENT
	var parent = getParent(id);

	var newid = id + '-colors';

	//RETURNS ARRAY
	if(parent.match('radial'))
	{
		var colorsArray = radialGradientTable.get(newid);
		var colors = colorsArray.join(',');
		return colors;
	}
	if(parent.match('linear'))
	{
		var array = linearGradientTable.get(newid);
		var newcolors = array.join(',');
		return newcolors;
	}
}
```

```
function getColor2(id)
{
	var colorsArray;

	//CHECK PARENT
	var parent = getParent(id);

	var newid = id + '-' + 'colors';

	if(parent.match('radialgradient'))
	{
		//RETRIEVE COLORS ARRAY
		colorsArray = radialGradientTable.get(newid);
	}
	else
		colorsArray = linearGradientTable.get(newid);

	//RETRIEVE THE LAST ELEMENT IN ARRAY
	var largestColor = colorsArray[colorsArray.length-1];

	//RETURNS ARRAY WITH TWO ELEMENTS
	var largestColorContent = largestColor.split(" ");
	return largestColorContent[1];
}

function getRadialfillcolor(id)
{
	var info = radialGradientTable.get(id);
	var arr = info.split(" ");
	var color = arr[8].substring(11, arr[8].length-1);
	return color;
}

//START HASHTABLE OBJECT
//AUTHOR: MICHAEL SYNOVIC
	/**
	    Contructor(s):
	     Hashtable()
	             Creates a new, empty hashtable

	    Method(s):
	     void clear()
	             Clears this hashtable so that it contains no keys.
	     boolean containsKey(String key)
	             Tests if the specified object is a key in this hashtable.
	     boolean containsValue(Object value)
	             Returns true if this Hashtable maps one or more keys to this value.
```

```
            Object get(String key)
                    Returns the value to which the specified key is mapped in this
hashtable.
            boolean isEmpty()
                    Tests if this hashtable maps no keys to values.
            Array keys()
                    Returns an array of the keys in this hashtable.
            void put(String key, Object value)
                    Maps the specified key to the specified value in this hashtable. A
NullPointerException is thrown is the key or value is null.
            Object remove(String key)
                    Removes the key (and its corresponding value) from this hashtable.
Returns the value of the key that was removed
            int size()
                    Returns the number of keys in this hashtable.
            String toString()
                    Returns a string representation of this Hashtable object in the form
of a set of entries, enclosed in braces and separated by the ASCII characters ", " (comma
and space).
            Array values()
                    Returns an array view of the values contained in this Hashtable.

        */
        function Hashtable(){
            this.clear = hashtable_clear;
            this.containsKey = hashtable_containsKey;
            this.containsValue = hashtable_containsValue;
            this.get = hashtable_get;
            this.isEmpty = hashtable_isEmpty;
            this.keys = hashtable_keys;
            this.put = hashtable_put;
            this.remove = hashtable_remove;
            this.size = hashtable_size;
            this.toString = hashtable_toString;
            this.values = hashtable_values;
            this.hashtable = new Array();
        }
        /*=======Private methods for internal use only========*/

        function hashtable_clear(){
            this.hashtable = new Array();
        }

        function hashtable_containsKey(key){
            var exists = false;
            for (var i in this.hashtable) {
                if (i == key && this.hashtable[i] != null) {
                    exists = true;
```

```
            break;
        }
    }
    return exists;
}

function hashtable_containsValue(value){
    var contains = false;
    if (value != null) {
        for (var i in this.hashtable) {
            if (this.hashtable[i] == value) {
                contains = true;
                break;
            }
        }
    }
    return contains;
}

function hashtable_get(key){
    return this.hashtable[key];
}

function hashtable_isEmpty(){
    return (this.size == 0) ? true : false;
}

function hashtable_keys(){
    var keys = new Array();
    for (var i in this.hashtable) {
        if (this.hashtable[i] != null)
            keys.push(i);
    }
    return keys;
}

function hashtable_put(key, value){
    if (key == null || value == null) {
        throw "NullPointerException {" + key + "},{" + value + "}";
    }else{
        this.hashtable[key] = value;
    }
}

function hashtable_remove(key){
    var rtn = this.hashtable[key];
    this.hashtable[key] = null;
    return rtn;
```

```
        }

        function hashtable_size(){
            var size = 0;
            for (var i in this.hashtable) {
                if (this.hashtable[i] != null)
                    size ++;
            }
            return size;
        }

        function hashtable_toString(){
            var result = "";
            for (var i in this.hashtable)
            {
                if (this.hashtable[i] != null)
                    result += "{" + i + "},{" + this.hashtable[i] + "}\n";
            }
            return result;
        }

        function hashtable_values(){
            var values = new Array();
            for (var i in this.hashtable) {
                if (this.hashtable[i] != null)
                    values.push(this.hashtable[i]);
            }
            return values;
        }

//END HASHTABLE OBJECT

function getPoints(svgpoints)
{
        var arr = svgpoints.split(" ");
        var points = "";

        //SEPARATE X AND Y POINTS
        for(var i=0; i < arr.length; i++)
        {
                var arrxy = arr[i].split(",");
                var x = arrxy[0] - 10;
                var y = arrxy[1] - 14;
                points = points + x + "," + y + " ";
        }
        return points;
}
```

```
function getFill(style)
{
	var arr = style.split(" ");
	var fill = "";
	for(var i=0; i < arr.length; i++)
	{
		if(arr[i].match('fill'))
		{
			//CHECK IF FILL IS GRADIENT
			if(arr[i].match('url'))
			{
				fill = arr[i].substring(10, arr[i].length-1);
				return fill;
			}
			else
			{
				fill = arr[i].substring(5, arr[i].length);
				return fill;
			}
		}
	}
}

function getStrokewidth(style)
{
	var arr = style.split(" ");
	var strokewidth = 0;
	for(var i=0; i < arr.length; i++)
	{
		if(arr[i].match('stroke-width'))
		{
			var temp = arr[i].substring(13, arr[i].length);
			if (temp.match(';'))
			{
				var index = temp.indexOf(';');
				strokewidth = temp.substring(0,index);
			}
			else
				strokewidth = temp;
		}
	}
	return strokewidth * 1;
}
```

```
function getStroke(style)
{
	var arr = style.split(" ");
	var stroke = "";
	for(var i=0; i < arr.length; i++)
	{
		if(arr[i].match('stroke:'))
			stroke = arr[i].substring(7, arr[i].length);
	}
	if(stroke == "")
		return 'NONE';
	else
		return stroke;
}

function getVMLcurvefrom(points)
{
	var arr = points.split(" ");
	var from = arr[0] + " " + arr[1];
	return from;
}

function getVMLcurvecontrol1(points)
{
	var arr = points.split(" ");
	var control1 = arr[2]  + " " + arr[3];
	return control1;
}

function getVMLcurvecontrol2(points)
{
	var arr = points.split(" ");
	var control2 = arr[4] + " " + arr[5];
	return control2;
}

function getVMLcurveto(points)
{
	var arr = points.split(" ");
	var to = arr[6] + " " + arr[7];
	return to;
}

function getPathpoints(points)
{
	var startpt = "";
	var nextpt = "";
```

```
	var midpts = "";
	var endpt = "";

	var arr = points.split(" ");

	//REMOVE M
	if(arr[0].match("M"))
	{
		var i = arr[0].indexOf("M");
		startpt = arr[0].substring(i+1, arr[0].length+1) + " " + arr[1];
	}

	//REMOVE L
	if(arr[2].match("L"))
	{
		var i = arr[2].indexOf("L");
		nextpt = arr[2].substring(i+1, arr[2].length+1) + " " + arr[3];
	}

	//REMOVE Z
	if(arr[arr.length-1].match("z"))
		endpt = startpt;

	//CONCATENATE REMAINING POINTS
	for(i=4; i<arr.length-1; i++)
		midpts = " " + midpts + " " + arr[i];

	var newpoints = startpt + " " + " " + nextpt + " " + " " + midpts
		+ " " + endpt;
	return newpoints;
}

function getTypeface(style)
{
	var arr = style.split(";");
	var typeface = "";
	for(var i=0; i < arr.length; i++)
	{
		if(arr[i].match('font-family'))
		{
			typeface = arr[i].substring(13, arr[i].length-1);
			return typeface;
		}
	}
}
```

```
function getFontcolor(style)
{
        var arr = style.split(" ");
        var fontcolor = "";
        for(var i=0; i < arr.length; i++)
        {
                if(arr[i].match('fill'))
                {
                        fontcolor = arr[i].substring(5,arr[i].length-1);
                        return fontcolor;
                }
        }
}


function getFontsize(style)
{
        var arr = style.split(" ");
        var fontsize = 0;
        for(var i=0; i < arr.length; i++)
        {
                if(arr[i].match('font-size'))
                {
                        fontsize = arr[i].substring(10, arr[i].length-1);
                        return fontsize * 1;
                }
        }
}


]]>
</msxsl:script>

<xsl:template match="/">
  <html xmlns:v="urn:schemas-microsoft-com:vml">
    <head>
        <style type="text/css">
        v\:*{behavior:url(#default#VML);}
        </style>
    </head>
    <body>
<!--CREATE AN XML DATA SOURCE OBJECT FOR THE SVG DOCUMENT-->
<OBJECT width="0" height="0"
    classid="clsid:550dda30-0541-11d2-9ca9-0060b0ec3d39"
    id="source">
 <xsl:copy-of select="/" />
 </OBJECT>
```

```
        <xsl:apply-templates />
        </body>
  </html>
</xsl:template>

<!--TO SUPPRESS CDATA TEXT-->
<xsl:template match="*" />

<xsl:template match="g">
        <xsl:apply-templates />
</xsl:template>

<!--PROCESS SVG TAG-->
<xsl:template match="svg">

    <xsl:element name="v:rect">
      <xsl:attribute name="style">
            position:absolute;
            top:0;left:0
      </xsl:attribute>

          <xsl:if test="@width = '' ">
                <xsl:attribute name="width">100%</xsl:attribute>
                <xsl:attribute name="height">100%</xsl:attribute>
          </xsl:if>

          <xsl:if test="not(@width = '')" >
                <xsl:attribute name="width">
                        <xsl:value-of select="@width" /></xsl:attribute>
                <xsl:attribute name="height">
                        <xsl:value-of select="@height" /></xsl:attribute>
          </xsl:if>

      <xsl:attribute name="strokecolor">white</xsl:attribute>
      <xsl:attribute name="fillcolor">white</xsl:attribute>
    </xsl:element>

    <xsl:apply-templates />
</xsl:template>

<!--PROCESS RECT TAG-->
<xsl:template match="rect">
 <xsl:variable name="left" select="@x" />
 <xsl:variable name="top" select="@y" />
 <xsl:variable name="rectWidth" select="@width" />
 <xsl:variable name="rectHeight" select="@height" />
```

```
<!--CHECK IF PARENT TAG IS 'G'-->
<xsl:variable name="style">
<xsl:if test="name(parent::*) = 'g' " />
	<xsl:value-of select="string((parent::*)/@style)" />
<xsl:if test="name(parent::*) != 'g' "/>
	<xsl:value-of select="string(@style)" />
</xsl:variable>

<!--CHECK IF NEXT NODE IS TEXT-->
<xsl:if test="name(following-sibling::*) = 'text' ">
	<xsl:variable name="textboxfill"
		select="translate:getFill(string($style))" />
	<xsl:variable name="textfill"
select="translate:setTextboxfill(string($textboxfill))" />
</xsl:if>

<xsl:variable name="gradientID" select="translate:getFill(string($style))" />

<!--CHECK IF RADIALGRADIENT-->
<xsl:if test="contains($style, 'fill:url')" >
	<xsl:variable name="parent" select="translate:getParent(string($gradientID))" />
	<xsl:choose>
		<xsl:when test="contains($parent, 'radial')" >
			<xsl:call-template name="overlapRect">
				<xsl:with-param name="new-style" select="$style" />
				<xsl:with-param name="new-id" select="$gradientID" />
				<xsl:with-param name="new-left" select="$left" />
				<xsl:with-param name="new-top" select="$top" />
				<xsl:with-param name="new-width" select="$rectWidth" />
				<xsl:with-param name="new-height" select="$rectHeight" />
			</xsl:call-template>
		</xsl:when>

		<!--GRADIENT IS LINEAR-->
		<xsl:otherwise>
			<xsl:call-template name="drawRect">
				<xsl:with-param name="rect-style" select="$style" />
				<xsl:with-param name="rect-id" select="$gradientID" />
				<xsl:with-param name="rect-left" select="$left" />
				<xsl:with-param name="rect-top" select="$top" />
				<xsl:with-param name="rect-width" select="$rectWidth" />
				<xsl:with-param name="rect-height" select="$rectHeight" />
			</xsl:call-template>
		</xsl:otherwise>
	</xsl:choose>
</xsl:if>
```

```
<!--THERE IS NO GRADIENT-->
<xsl:if test="not(contains($style, 'fill:url'))" >
               <xsl:call-template name="drawRect">
                      <xsl:with-param name="rect-style" select="$style" />
                      <xsl:with-param name="rect-id" select="$gradientID" />
                      <xsl:with-param name="rect-left" select="$left" />
                      <xsl:with-param name="rect-top" select="$top" />
                      <xsl:with-param name="rect-width" select="$rectWidth" />
                      <xsl:with-param name="rect-height" select="$rectHeight" />
               </xsl:call-template>

</xsl:if>
<!--END RECT TAG-->
</xsl:template>

<!--TEMPLATE TO DRAW OVAL ON TOP OF RECTANGLE-->
<!--OVERLAPRECT-->
<xsl:template name="overlapRect">
       <xsl:param name="new-style" />
       <xsl:param name="new-id" />
       <xsl:param name="new-left" />
       <xsl:param name="new-top" />
       <xsl:param name="new-width" />
       <xsl:param name="new-height" />
       <xsl:variable name="maincolor" select="translate:getColor2(string($new-id))" />

 <xsl:element name="v:rect">
       <xsl:attribute name="style">
              position:absolute;
       width:<xsl:value-of select="$new-width" />;
       height:<xsl:value-of select="$new-height" />;
       top:<xsl:value-of select="$new-top" />;
       left:<xsl:value-of select="$new-left" />
       </xsl:attribute>

       <xsl:attribute name="fillcolor">
              <xsl:value-of select="$maincolor" />
       </xsl:attribute>

       <xsl:attribute name="strokecolor">
              <xsl:value-of select="$maincolor" />
       </xsl:attribute>
</xsl:element>

<xsl:element name="v:oval">
       <xsl:attribute name="style">
              position:absolute;
       width:<xsl:value-of select="$new-width" />;
```

```
        height:<xsl:value-of select="$new-height" />;
        top:<xsl:value-of select="$new-top" />;
        left:<xsl:value-of select="$new-left" />
        </xsl:attribute>

        <xsl:attribute name="strokecolor">
               <xsl:value-of select="$maincolor" />
        </xsl:attribute>

        <xsl:attribute name="strokeweight">
               <xsl:value-of select="translate:getStrokewidth(string(normalize-
space($new-style)))" />
        </xsl:attribute>

        <xsl:call-template name="processRadialGradient">
               <xsl:with-param name="id" select="string($new-id)" />
        </xsl:call-template>

</xsl:element>
</xsl:template>

<!--DRAWRECT-->
<xsl:template name="drawRect">
        <xsl:param name="rect-style" />
        <xsl:param name="rect-id" />
        <xsl:param name="rect-left" />
        <xsl:param name="rect-top" />
        <xsl:param name="rect-width" />
        <xsl:param name="rect-height" />

 <!--DRAW VML RECTANGLE-->
 <xsl:element name="v:rect">
         <xsl:attribute name="style">
               position:absolute;
        width:<xsl:value-of select="$rect-width" />;
        height:<xsl:value-of select="$rect-height" />;
        top:<xsl:value-of select="$rect-top" />;
        left:<xsl:value-of select="$rect-left" />
         </xsl:attribute>

<xsl:attribute name="id">
        <xsl:value-of select="@id" />
</xsl:attribute>
```

```
<!--CHECK FOR ONCLICK EVENT-->
<xsl:attribute name="onclick">
	<xsl:if test="contains(@onclick, 'alert') ">
		<xsl:value-of select="@onclick" />
	</xsl:if>
</xsl:attribute>

<!--CHECK FOR ONMOUSEOVER EVENT-->
<xsl:attribute name="onmouseover">
	<xsl:if test="contains(@onmouseover, 'alert') ">
		<xsl:value-of select="@onmouseover" />
	</xsl:if>
</xsl:attribute>

<!--CHECK FOR FUNCTION NOT EQUAL TO ALERT-->
<xsl:if test="not(contains(@onclick, 'alert')) ">
<xsl:variable name="onclickname" select="string(@onclick)" />
<xsl:variable name="tempName" select="substring-before($onclickname, '&#40;')" />
<xsl:variable name="event" select="'(event)'" />
<xsl:variable name="onclickEventName">
	<xsl:value-of select="concat('pre', $tempName, $event)" />
</xsl:variable>
<xsl:attribute name="onclick">
	<xsl:value-of select="$onclickEventName" />
</xsl:attribute>
<xsl:attribute name="id">
	<xsl:value-of select="string(@id)" />
</xsl:attribute>
</xsl:if>

	<xsl:call-template name="process-fill">
		<xsl:with-param name="svgstyle" select="$rect-style" />
		<xsl:with-param name="id" select="string($rect-id)" />
	</xsl:call-template>

 </xsl:element>
<!--END DRAWRECT-->
</xsl:template>
```

```
<!--PROCESSFILL-->
<xsl:template name="process-fill">
	<xsl:param name="svgstyle" />
	<xsl:param name="id" />
	<xsl:attribute name="strokeweight">
		<xsl:value-of select="translate:getStrokewidth(string(normalize-
space($svgstyle)))" />
	</xsl:attribute>
	<xsl:attribute name="strokecolor">
		<xsl:value-of select="translate:getStroke(string(normalize-
space($svgstyle)))" />
	</xsl:attribute>

	<!--CHECK FILLCOLOR-->
	<xsl:if test="not(contains($svgstyle, 'url'))">
		<xsl:attribute name="fillcolor">
			<xsl:value-of select="translate:getFill(string($svgstyle))" />
		</xsl:attribute>
	</xsl:if>

<!--CHECK GRADIENT IF LINEAR OR RADIAL-->
<xsl:if test="contains($svgstyle, 'fill:url')" >
	<xsl:variable name="parent" select="translate:getParent(string($id))" />
	<xsl:choose>
		<xsl:when test="contains($parent, 'radial')" >
			<xsl:call-template name="processRadialGradient">
				<xsl:with-param name="id" select="string($id)" />
			</xsl:call-template>
		</xsl:when>

		<xsl:otherwise>
			<xsl:call-template name="processLinearGradient">
				<xsl:with-param name="id" select="string($id)" />
			</xsl:call-template>
		</xsl:otherwise>
	</xsl:choose>
</xsl:if>

<!--END PROCESS-FILL TEMPLATE-->
</xsl:template>
```

```
<!--PROCESS CIRCLE TAG-->
<xsl:template match="circle">

<!--CHECK IF PARENT TAG IS 'G'-->
<xsl:variable name="style">
<xsl:if test="name(parent::*) = 'g' " />
       <xsl:value-of select="string((parent::*)/@style)" />
<xsl:if test="name(parent::*) != 'g' "/>
       <xsl:value-of select="string(@style)" />
</xsl:variable>

<!--CHECK IF NEXT NODE IS TEXT-->
<xsl:if test="name(following-sibling::*) = 'text' ">
       <xsl:variable name="textboxfill"
              select="translate:getFill(string($style))" />
       <xsl:variable name="ok" select="translate:setTextboxfill(string($textboxfill))" />
</xsl:if>

<xsl:variable name="gradientID" select="translate:getFill(string($style))" />

<!--DRAW VML OVAL-->
<xsl:element name="v:oval">
<xsl:attribute name="style">position:absolute;
    left:<xsl:value-of select="@cx - @r" />;
    top:<xsl:value-of select="@cy - @r" />;
    width:<xsl:value-of select="@r * 2" />;
    height:<xsl:value-of select="@r * 2" />
</xsl:attribute>

<!--CHECK IF THERE ONCLICK EVENT-->
<xsl:if test="contains(@onclick, 'alert') ">
       <xsl:attribute name="onclick">
              <xsl:value-of select="@onclick" />
       </xsl:attribute>
</xsl:if>

<!--CHECK FOR FUNCTION NOT EQUAL TO ALERT-->
<xsl:if test="not(contains(@onclick, 'alert')) ">
<xsl:variable name="onclickname" select="string(@onclick)" />
<xsl:variable name="tempName" select="substring-before($onclickname, '&#40;')" />
<xsl:variable name="event" select="'(event)'" />
<xsl:variable name="onclickEventName">
       <xsl:value-of select="concat('pre', $tempName, $event)" />
</xsl:variable>
<xsl:attribute name="onclick">
       <xsl:value-of select="$onclickEventName" />
</xsl:attribute>
```

```
<xsl:attribute name="id">
	<xsl:value-of select="string(@id)" />
</xsl:attribute>
</xsl:if>

<xsl:call-template name="process-fill">
	<xsl:with-param name="svgstyle" select="string($style)" />
	<xsl:with-param name="id" select="string($gradientID)" />
</xsl:call-template>

</xsl:element>

<!--END CIRCLE TAG-->
</xsl:template>


<!--PROCESS POLYLINE TAG-->
<xsl:template match="polyline">

<xsl:variable name="points"
	select="translate:getPoints(string(normalize-space(@points)))" />

<!--CHECK IF PARENT TAG IS 'G'-->
<xsl:variable name="style">
<xsl:if test="name(parent::*) = 'g' " />
	<xsl:value-of select="(parent::*)/@style" />
<xsl:if test="name(parent::*) != 'g' "/>
	<xsl:value-of select="@style" />
</xsl:variable>

<xsl:if test="contains($style, 'url')" />
<xsl:variable name="gradientID" select="translate:getFill(string($style))" />

  <!--DRAW VML POLYLINE-->

<xsl:element name="v:polyline">
<xsl:attribute name="style">position:absolute;</xsl:attribute>

<xsl:attribute name="id">
	<xsl:value-of select="@id" />
</xsl:attribute>

<xsl:attribute name="onclick">
	<xsl:if test="contains(@onclick, 'alert') ">
		<xsl:value-of select="@onclick" />
	</xsl:if>
</xsl:attribute>
```

```
<xsl:attribute name="onmouseover">
	<xsl:if test="contains(@onmouseover, 'alert') ">
		<xsl:value-of select="@onmouseover" />
	</xsl:if>
</xsl:attribute>

<!--CHECK FOR FUNCTION NOT EQUAL TO ALERT-->
<xsl:if test="not(contains(@onclick, 'alert')) ">
<xsl:variable name="onclickname" select="string(@onclick)" />
<xsl:variable name="tempName" select="substring-before($onclickname, '&#40;')" />
<xsl:variable name="event" select="'(event)'" />
<xsl:variable name="onclickEventName">
	<xsl:value-of select="concat('pre', $tempName, $event)" />
</xsl:variable>
<xsl:attribute name="onclick">
	<xsl:value-of select="$onclickEventName" />
</xsl:attribute>
<xsl:attribute name="id">
	<xsl:value-of select="string(@id)" />
</xsl:attribute>
</xsl:if>

<xsl:attribute name="points">
    <xsl:value-of select="$points" />
</xsl:attribute>

<xsl:call-template name="process-fill">
	<xsl:with-param name="svgstyle" select="string($style)" />
	<xsl:with-param name="id" select="string(gradientID)" />
</xsl:call-template>

</xsl:element>
</xsl:template>

<!--PROCESS LINE TAG-->
<xsl:template match="line">
	<xsl:variable name="x1" select="@x1" />
	<xsl:variable name="y1" select="@y1" />
	<xsl:variable name="x2" select="@x2" />
	<xsl:variable name="y2" select="@y2" />

<!--CHECK IF PARENT TAG IS 'G'-->
<xsl:variable name="style">
<xsl:if test="name(parent::*) = 'g' " />
	<xsl:value-of select="(parent::*)/@style" />
<xsl:if test="name(parent::*) != 'g' "/>
	<xsl:value-of select="@style" />
</xsl:variable>
```

```
<!--DRAW VML LINE-->

<xsl:element name="v:line">
<xsl:attribute name="from"><xsl:value-of select="$x1 - 10" />
<xsl:text>,</xsl:text>
<xsl:value-of select="$y1 - 14" />
</xsl:attribute>
<xsl:attribute name="to"><xsl:value-of select="$x2 - 10" />
<xsl:text>,</xsl:text>
<xsl:value-of select="$y2 - 14" />
</xsl:attribute>

<xsl:attribute name="id">
	<xsl:value-of select="@id" />
</xsl:attribute>

<xsl:attribute name="onclick">
	<xsl:if test="contains(@onclick, 'alert') ">
		<xsl:value-of select="@onclick" />
	</xsl:if>
</xsl:attribute>

<xsl:attribute name="onmouseover">
	<xsl:if test="contains(@onmouseover, 'alert') ">
		<xsl:value-of select="@onmouseover" />
	</xsl:if>
</xsl:attribute>

<!--CHECK FOR FUNCTION NOT EQUAL TO ALERT-->
<xsl:if test="not(contains(@onclick, 'alert')) ">
<xsl:variable name="onclickname" select="string(@onclick)" />
<xsl:variable name="tempName" select="substring-before($onclickname, '&#40;')" />
<xsl:variable name="event" select="'(event)'" />
<xsl:variable name="onclickEventName">
	<xsl:value-of select="concat('pre', $tempName, $event)" />
</xsl:variable>
<xsl:attribute name="onclick">
	<xsl:value-of select="$onclickEventName" />
</xsl:attribute>
<xsl:attribute name="id">
	<xsl:value-of select="string(@id)" />
</xsl:attribute>
</xsl:if>

<xsl:attribute name="strokeweight">
	<xsl:value-of select="translate:getStrokewidth(string(normalize-space($style)))"
/>
```

```
</xsl:attribute>
<xsl:attribute name="strokecolor">
       <xsl:value-of select="translate:getStroke(string(normalize-space($style)))" />
</xsl:attribute>
<!--NO FILLCOLOR SUPPORTED-->
</xsl:element>
<!--END LINE TAG-->
</xsl:template>

<!--PROCESS ELLIPSE TAG-->
<xsl:template match="ellipse">
<xsl:variable name="ellipseWidth" select="number(@rx * 2)" />
<xsl:variable name="ellipseHeight" select="number(@ry * 2)" />
<xsl:variable name="cx" select="@cx" />
<xsl:variable name="cy" select="@cy" />
<xsl:variable name="rx" select="@rx" />
<xsl:variable name="ry" select="@ry" />

<!--CHECK IF PARENT TAG IS 'G'-->
<xsl:variable name="style">
<xsl:if test="name(parent::*) = 'g' " />
       <xsl:value-of select="(parent::*)/@style" />
<xsl:if test="name(parent::*) != 'g' "/>
       <xsl:value-of select="@style" />
</xsl:variable>

<xsl:if test="name(following-sibling::*) = 'text' ">
       <xsl:variable name="textboxfill"
              select="translate:getFill(string($style))" />
       <xsl:variable name="setTextboxfill"
select="translate:setTextboxfill(string($textboxfill))" />
</xsl:if>


<!--DRAW VML OVAL-->
<xsl:element name="v:oval">
<xsl:attribute name="style">position:absolute;
  left:<xsl:value-of select="number($cx - $rx)" />;
    top:<xsl:value-of select="number($cy - $ry)" />;
    width:<xsl:value-of select="$ellipseWidth" />;
    height:<xsl:value-of select="$ellipseHeight" />
</xsl:attribute>
<xsl:if test="contains($style, 'url')" />
<xsl:variable name="gradientID" select="translate:getFill(string($style))" />

<xsl:attribute name="id">
       <xsl:value-of select="@id" />
</xsl:attribute>
```

```
<xsl:attribute name="onclick">
	<xsl:if test="contains(@onclick, 'alert') ">
		<xsl:value-of select="@onclick" />
	</xsl:if>
</xsl:attribute>

<xsl:attribute name="onmouseover">
	<xsl:if test="contains(@onmouseover, 'alert') ">
		<xsl:value-of select="@onmouseover" />
	</xsl:if>
</xsl:attribute>

<!--CHECK FOR FUNCTION NOT EQUAL TO ALERT-->
<xsl:if test="not(contains(@onclick, 'alert')) ">
<xsl:variable name="onclickname" select="string(@onclick)" />
<xsl:variable name="tempName" select="substring-before($onclickname, '&#40;')" />
<xsl:variable name="event" select="'(event)'" />
<xsl:variable name="onclickEventName">
	<xsl:value-of select="concat('pre', $tempName, $event)" />
</xsl:variable>
<xsl:attribute name="onclick">
	<xsl:value-of select="$onclickEventName" />
</xsl:attribute>
<xsl:attribute name="id">
	<xsl:value-of select="string(@id)" />
</xsl:attribute>
</xsl:if>

<xsl:call-template name="process-fill">
	<xsl:with-param name="svgstyle" select="string($style)" />
	<xsl:with-param name="id" select="string(gradientID)" />
</xsl:call-template>

</xsl:element>
</xsl:template>

<!--PROCESS POLYGON TAG-->
<xsl:template match="polygon">
	<xsl:variable name="points" select="@points" />

<!--CHECK IF PARENT TAG IS 'G'-->
<xsl:variable name="style">
<xsl:if test="name(parent::*) = 'g' " />
	<xsl:value-of select="(parent::*)/@style" />
<xsl:if test="name(parent::*) != 'g' "/>
	<xsl:value-of select="@style" />
</xsl:variable>
```

```
 <!--DRAW VML POLYLINE FOR POLYGON-->
  <xsl:element name="v:polyline">
<xsl:attribute name="style">position:absolute;</xsl:attribute>

<xsl:attribute name="id">
       <xsl:value-of select="@id" />
</xsl:attribute>

<xsl:attribute name="onclick">
       <xsl:if test="contains(@onclick, 'alert') ">
              <xsl:value-of select="@onclick" />
       </xsl:if>
</xsl:attribute>

<xsl:attribute name="onmouseover">
       <xsl:if test="contains(@onmouseover, 'alert') ">
              <xsl:value-of select="@onmouseover" />
       </xsl:if>
</xsl:attribute>

<!--CHECK FOR FUNCTION NOT EQUAL TO ALERT-->
<xsl:if test="not(contains(@onclick, 'alert')) ">
<xsl:variable name="onclickname" select="string(@onclick)" />
<xsl:variable name="tempName" select="substring-before($onclickname, '&#40;')" />
<xsl:variable name="event" select="'(event)'" />
<xsl:variable name="onclickEventName">
       <xsl:value-of select="concat('pre', $tempName, $event)" />
</xsl:variable>
<xsl:attribute name="onclick">
       <xsl:value-of select="$onclickEventName" />
</xsl:attribute>
<xsl:attribute name="id">
       <xsl:value-of select="string(@id)" />
</xsl:attribute>
</xsl:if>

<xsl:attribute name="points">
    <xsl:value-of select="translate:getPoints(string(normalize-space(@points)))" />
</xsl:attribute>

<xsl:if test="contains($style, 'url')" />
<xsl:variable name="gradientID" select="translate:getFill(string($style))" />

<xsl:call-template name="process-fill">
       <xsl:with-param name="svgstyle" select="$style" />
       <xsl:with-param name="id" select="$gradientID" />
</xsl:call-template>
```

```
</xsl:element>
<!--END POLYGON TAG-->
</xsl:template>

<!--PROCESS TEXT TAG-->
<xsl:template match="text">
<xsl:variable name="x" select="@x" />
<xsl:variable name="y" select="@y" />
<xsl:variable name="text" select="." />

<!--CHECK IF PARENT TAG IS 'G'-->
<xsl:variable name="style">
<xsl:if test="name(parent::*) = 'g' " />
       <xsl:value-of select="(parent::*)/@style" />
<xsl:if test="name(parent::*) != 'g' "/>
       <xsl:value-of select="string(@style)" />
</xsl:variable>

<xsl:variable name="fontsize"
       select="translate:getFontsize(string(normalize-space($style)))" />

<xsl:variable name="fontcolor"
       select="translate:getFontcolor(string(normalize-space($style)))" />

<xsl:variable name="typeface"
       select="translate:getTypeface(string(normalize-space($style)))" />

<!--DRAW VML RECTANGLE BEHIND TEXT-->
<xsl:element name="v:rect">
       <xsl:attribute name="style">
              position:absolute;
       width:<xsl:value-of select="string-length($text) * 20" />;
       height:<xsl:value-of select="$fontsize * 2" />;
       top:<xsl:value-of select="$y - 20"/>;
       left:<xsl:value-of select="$x - 10"/>
    </xsl:attribute>

    <!--RECT IS DRAWN WITHOUT STROKECOLOR-->
       <xsl:element name="v:stroke">
              <xsl:attribute name="on">false</xsl:attribute>
       </xsl:element>

       <xsl:variable name="fillcolor"
                     select="translate:getTextboxfill()" />

       <xsl:if test="$fillcolor = '' ">
              <xsl:attribute name="fillcolor">none</xsl:attribute>
```

```
                <xsl:attribute name="strokecolor">none</xsl:attribute>
        </xsl:if>

        <!--RECT IS DRAWN AS TRANSPARENT-->
        <xsl:if test="not($fillcolor = '' )">
                <xsl:element name="v:fill">
                        <xsl:attribute name="opacity">0.0</xsl:attribute>
                        <xsl:attribute name="color">
                                <xsl:value-of select="$fillcolor" /></xsl:attribute>
                </xsl:element>
        </xsl:if>


<!--DRAW VML TEXTBOX-->
 <xsl:element name="v:textbox">
  <xsl:element name="font">
  <xsl:attribute name="size">
        <xsl:value-of select="$fontsize div 5" />pt
  </xsl:attribute>

  <xsl:attribute name="id">
        <xsl:value-of select="@id" />
  </xsl:attribute>

  <xsl:attribute name="onclick">
        <xsl:if test="contains(@onclick, 'alert') ">
                <xsl:value-of select="@onclick" />
        </xsl:if>
  </xsl:attribute>

  <xsl:attribute name="onmouseover">
        <xsl:if test="contains(@onmouseover, 'alert') ">
                <xsl:value-of select="@onmouseover" />
        </xsl:if>
</xsl:attribute>

<!--CHECK FOR FUNCTION NOT EQUAL TO ALERT-->
<xsl:if test="not(contains(@onclick, 'alert')) ">
<xsl:variable name="onclickname" select="string(@onclick)" />
<xsl:variable name="tempName" select="substring-before($onclickname, '&#40;')" />
<xsl:variable name="event" select="'(event)'" />
<xsl:variable name="onclickEventName">
        <xsl:value-of select="concat('pre', $tempName, $event)" />
</xsl:variable>
<xsl:attribute name="onclick">
        <xsl:value-of select="$onclickEventName" />
</xsl:attribute>
<xsl:attribute name="id">
```

```
        <xsl:value-of select="string(@id)" />
</xsl:attribute>
</xsl:if>

  <xsl:attribute name="face"><xsl:value-of select="$typeface" /></xsl:attribute>
  <xsl:attribute name="color"><xsl:value-of select="$fontcolor" /></xsl:attribute>
  <xsl:value-of select="$text"/>
  </xsl:element>
 </xsl:element>
</xsl:element>
</xsl:template>


<!--PROCESS LINEAR GRADIENT TAG-->
<xsl:template name="processLinearGradient">
        <xsl:param name="id" />
                <!--GET COLORS-->
                <xsl:variable name="colors"
                        select="translate:getColors(string($id))" />

                <!--GET COLOR2-->
                <xsl:variable name="color2"
                        select="translate:getColor2(string($id))" />
                <!--WRITE VML FILL TAG-->
        <xsl:element name="v:fill">
                <xsl:attribute name="type">gradient</xsl:attribute>
                <xsl:attribute name="method">sigma</xsl:attribute>

                <!--CHECK IF HORIZONTAL OR VERTICAL-->
                <xsl:variable name="y2" select="translate:getY(string($id))" />

                <xsl:if test="contains($y2, '100')">
                        <xsl:attribute name="angle">-180</xsl:attribute>
                </xsl:if>

                <xsl:if test="not(contains($y2, '100'))">
                        <xsl:attribute name="angle">-90</xsl:attribute>
                </xsl:if>

                <xsl:attribute name="focussize">0,0</xsl:attribute>
                <xsl:attribute name="focus">100%</xsl:attribute>
                <xsl:attribute name="focusposition">50%,50%</xsl:attribute>
                <!--ADD COLORS-->
                <xsl:attribute name="color2">
                        <xsl:value-of select="$color2" /></xsl:attribute>
                <xsl:attribute name="colors">
                        <xsl:value-of select="$colors" /></xsl:attribute>
```

```
	</xsl:element>

<!--END LINEAR GRADIENT TAG-->
</xsl:template>

<!--PROCESS RADIAL GRADIENT TAG-->
<xsl:template name="processRadialGradient">
	<xsl:param name="id" />
		<!--GET COLORS-->
		<xsl:variable name="colors"
			select="translate:getColors(string($id))" />
		<!--GET COLOR2-->
		<xsl:variable name="color2"
			select="translate:getColor2(string($id))" />
		<!--WRITE VML FILL TAG-->
	<xsl:element name="v:fill">
		<xsl:attribute name="type">gradientradial</xsl:attribute>
		<xsl:attribute name="method">sigma</xsl:attribute>
		<xsl:attribute name="angle">-45</xsl:attribute>
		<xsl:attribute name="focussize">0,0</xsl:attribute>
		<xsl:attribute name="focus">100%</xsl:attribute>
		<xsl:attribute name="focusposition">50%,50%</xsl:attribute>
		<!--ADD COLORS-->
		<xsl:attribute name="color2">
			<xsl:value-of select="$color2" /></xsl:attribute>
		<xsl:attribute name="colors">
			<xsl:value-of select="$colors" /></xsl:attribute>

	</xsl:element>
<!--END PROCESSRADIALGRADIENT-->
</xsl:template>

<!--PROCESS DEFS/LINEARGRADIENT TAG-->
<xsl:template match="defs/linearGradient">

<xsl:variable name="id" select="string(@id)" />
<xsl:variable name="y2" select="string(@y2)" />

<xsl:variable name="storeParent" select="translate:storeParent($id, 'lineargradient')" />

<!--STORE Y2 IN JAVASCRIPT-->
<xsl:variable name="storeLinear" select="translate:storeLinearGradientInfo($id, $y2)" />

<xsl:apply-templates />
<!--STORE COLORS AFTER STOP TAGS HAVE BEEN PROCESSED-->
<xsl:variable name="storeColors" select="translate:storeColorsInTable($id)" />
<!--END DEFS/LINEARGRADIENT TAG-->
</xsl:template>
```

```
<!--PROCESS DEFS/RADIALGRADIENT TAG-->
<xsl:template match="defs">
	<xsl:apply-templates />
</xsl:template>

<!--RADIAL TEMPLATE-->
<xsl:template match="radialGradient">
	<xsl:variable name="id" select="string(@id)" />

	<xsl:variable name="storeParent" select="translate:storeParent($id,
'radialgradient')" />
	<xsl:apply-templates />

<!--STORE IN JAVASCRIPT-->
<xsl:variable name="storeColors" select="translate:storeColorsInTable($id)" />
<!--END RADIAL TEMPLATE-->
</xsl:template>

<!--PROCESS STOP TAGS-->
<xsl:template match="stop">
	<xsl:variable name="id" select="string(parent::*/@id)" />
	<xsl:variable name="offset" select="string(@offset)" />
	<xsl:variable name="style" select="string(@style)" />
	<xsl:variable name="color" select="substring-after($style,'stop-color:')" />
	<xsl:variable name="colorinfo" select="string(concat($offset, ' ', $color))" />
	<xsl:variable name="storeColors" select="translate:storeColors($id, $color,
$colorinfo)" />
<!--END STOP TAGS-->
</xsl:template>

<!--PROCESS PATH TAG-->
<xsl:template match="path">
<xsl:variable name="d" select="string(@d)" />

<!--CHECK IF PARENT TAG IS 'G'-->
<xsl:variable name="style">
<xsl:if test="name(parent::*) = 'g' " />
	<xsl:value-of select="string((parent::*)/@style)" />
<xsl:if test="name(parent::*) != 'g' "/>
	<xsl:value-of select="string(@style)" />
</xsl:variable>

<xsl:if test="contains($style, 'url')" />
<xsl:variable name="gradientID" select="translate:getFill(string($style))" />
```

```
<!--CHECK IF CUBIC BEZIER CURVE-->
<xsl:if test="contains($d, 'C')" >
	<xsl:call-template name="writeVMLCurveTag">
		<xsl:with-param name="points" select="$d" />
		<xsl:with-param name="curvestyle" select="$style" />
		<xsl:with-param name="curveid" select="$gradientID" />
	</xsl:call-template>
</xsl:if>
<!--END PATH TAG-->
</xsl:template>

<!--WRITE VML CURVE TAG-->
<xsl:template name="writeVMLCurveTag">
	<xsl:param name="points" />
	<xsl:param name="curvestyle" />
	<xsl:param name="curveid" />

<xsl:element name="v:curve">
	<xsl:attribute name="from">
		<xsl:value-of select="translate:getVMLcurvefrom(string(normalize-
space($points)))" />
	</xsl:attribute>

	<xsl:attribute name="control1">
		<xsl:value-of select="translate:getVMLcurvecontrol1(string(normalize-
space($points)))" />
	</xsl:attribute>

	<xsl:attribute name="control2">
		<xsl:value-of select="translate:getVMLcurvecontrol2(string(normalize-
space($points)))" />
	</xsl:attribute>

	<xsl:attribute name="to">
		<xsl:value-of select="translate:getVMLcurveto(string(normalize-
space($points)))" />
	</xsl:attribute>

	<xsl:call-template name="process-fill">
		<xsl:with-param name="svgstyle" select="$curvestyle" />
		<xsl:with-param name="id" select="$curveid" />
	</xsl:call-template>

</xsl:element>
<!--END WRITEVMLCURVETAG-->
</xsl:template>
```

```
<!--PROCESS SCRIPT TAG-->
<xsl:template match="script">
<xsl:variable name="scriptText" select="." />
<xsl:variable name="newScript" select="translate:getNewScript(string($scriptText))" />

<!--SCRIPT ELEMENT AND TYPE ATTRIBUTE OK-->
<xsl:element name="script">
<xsl:attribute name="type">text&#47;javascript</xsl:attribute>
<xsl:text>
 var xsl = new ActiveXObject("Microsoft.XMLDOM");
 xsl.async = false;
 xsl.load("translator.xsl");
 var xml;
</xsl:text>
<!--LET JAVASCRIPT PARSE THE FUNCTION NAME-->
<xsl:variable name="functionName"
       select="translate:getFunctionName(string($scriptText))" />

<xsl:variable name="preFunctionName"
       select="concat('pre', $functionName)" />
<xsl:variable name="postFunctionName"
       select="concat('post', $functionName)" />

<!--INSERT VML FUNCTIONS-->
<xsl:text>function </xsl:text>
<xsl:value-of select="$preFunctionName" />
<xsl:text>
{
       var my_evt = new My_Event();
       createShape(my_evt);
</xsl:text>
<xsl:value-of select="$postFunctionName" />;
<xsl:text>
}
</xsl:text>
<!--CREATE CUSTOM EVENT FUNCTION-->
<xsl:text>function </xsl:text>
<xsl:value-of select="$postFunctionName" />
{
       document.close();
       document.open();
       document.write(xml.transformNode(xsl));
       document.close();
}
<!--INSERT ORIGINAL SVG SCRIPT-->
<xsl:value-of select="$newScript" />
<xsl:text>function My_Event()</xsl:text>
{
```

```
        xml = source.XMLDocument;
        xml.async = false;
        //LOAD XML DATA SOURCE OBJECT CONTAINING SVG
        xml.loadXML(source.altHtml);
        this.getXML = event_getXML;
}

<xsl:text>function event_getXML()</xsl:text>
{
        return xml;
}
</xsl:element>
<!--END OF SCRIPT TAG-->
</xsl:template>
</xsl:stylesheet>
```