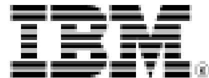


developerWorks®

Interactive, dynamic Scalable Vector Graphics

Generate graphics from XML and XSLT

Skill Level: Intermediate

Andrew Watt (SVGDeveloper@aol.com)
Independent Conultant

27 Jun 2003

This tutorial is for developers who want to use Scalable Vector Graphics (SVG) to create interactive SVG graphics and to produce such SVG images from an XML file using XSLT. It discusses the use of ECMAScript (JavaScript) to enable users to interact with and change an existing SVG image in real time without reloading the browser page. It also demonstrates linking from an SVG element.

Section 1. Introduction

Should I take this tutorial?

This tutorial is for developers who want to use Scalable Vector Graphics (SVG) to create interactive SVG graphics and to produce such SVG images from an XML file using XSLT. It discusses the use of ECMAScript (JavaScript) to enable users to interact with and change an existing SVG image in real time without reloading the browser page. It also demonstrates linking from an SVG element.

Readers should have read the Introduction to SVG tutorial here on *developerWorks*, or have some similar understanding of SVG fundamentals. Some previous knowledge of JavaScript and XSLT is also advantageous.

What is this tutorial about?

© Copyright IBM Corporation 2003. All rights reserved.

SVG is a powerful visual tool ideally suited to display data held as XML. This tutorial demonstrates the use of SVG to display a simplified interactive floor plan. JavaScript provides interactivity in SVG images in this tutorial, enabling visitors to the Web page to zoom in and out of the image, scroll between rooms, and perform additional functions. The tutorial covers:

- Creating the initial SVG image
- Zooming in and out of the image
- Using JavaScript to scroll the image from room to room
- Deleting SVG elements
- Adding new SVG elements
- Linking from an SVG element to another document
- Generating the SVG document dynamically using XSLT

Tools

The tutorial shows you how to build HTML, SVG, and XSLT documents. To follow along you need a text editor, an SVG viewer, and an XSLT engine:

- **A plain text editor** is all you need to create SVG files. XML-enabled text editors which can check for well-formedness and/or validity are very useful for detecting coding errors but are not necessary.
- **An SVG viewer:** The most widely used SVG viewer at the time of this writing is Adobe's SVG Viewer, version 3.0, which is available for free at http://www.adobe.com/svg/viewer/install/main.html. Other SVG viewers are listed in the Resources.
- **A browser:** The Adobe viewer works best with Microsoft Internet Explorer 5.x and above, and also with Netscape Navigator 4.x and above. The Adobe SVG Viewer version 3 does not work reliably with Mozilla 1.0 and above, particularly when scripting code is present.
- **An XSLT processor:** The Saxon XSLT processor is used for this example. It can be downloaded from http://saxon.sourceforge.net/.

Section 2. Overview

© Copyright IBM Corporation 2003. All rights reserved.

The goal

This tutorial centers on the creation of a simple floor plan that a user can visit in order to determine the type and size of rooms for a proposed office building. It starts with an HTML document that includes an SVG image representing a simplified floor plan of one floor of the building. The SVG image includes three rooms -- Rooms 101, 102, and 103 -- of which only Room 102 is visible when the document first loads. To see the entire floor plan, the user can zoom in and out to determine the overall scale. The HTML document also provides controls that enable the user to scroll between the three rooms on the first floor. Additional controls allow the user to change the character of a selected room -- in this example from a Standard Office to an Executive Office and back again. Finally, the user has the ability to enlarge or shrink the size of a particular room.

Each room on the first floor also contains a link to a plan for the second floor of the same building, which has the same functionality and links back to the first floor.

The functionality of the floor plans uses ECMAScript (JavaScript) code to provide interactivity. The code demonstrates several aspects of using ECMAScript from an HTML page to control the elements of an SVG image in that page.

The final part of the tutorial looks at using XSLT code to transform a source XML document into the SVG image of the first floor floor plan.

Next I'll show you the component parts of this simple application.

The component parts of the application

The overall application involves the following pieces:

- **The XML source document:** This document defines the initial characteristics of the SVG image. It can be a static XML file (as it will be here), or it could just as easily be XML generated out of a database.
- **The XSLT stylesheet:** Dynamically turning an XML document of, say, potential floor plans into SVG images of those plans can be easily accomplished using XSLT transformations. The application can also use different stylesheets to generate different presentations.
- **The SVG image:** This is the image that will ultimately be generated, and provides the display the user sees. An XML document itself, it can be manipulated using XML programming techniques such a the Document Object Model (DOM).

© Copyright IBM Corporation 2003. All rights reserved.

- **The HTML container document:** To display an SVG image directly in a browser, you generally need an HTML page that embeds that image within it. In this case, the HTML document also contains the Javascript code that manipulates that SVG image.

I'll start by looking at the SVG document itself.

Section 3. Creating the SVG Document

The SVG document structure

This tutorial uses two SVG images -- one to represent part of the first floor of an office building and the second SVG image to represent the corresponding rooms on the second floor. Each SVG image is nested inside an HTML page. I'll start by manually creating the first floor image so I have something to work with.

An SVG image starts with an `svg` root element:

```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
 "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg
 xmlns="http://www.w3.org/2000/svg"
 width="1200px" height="250px" viewBox="0 0 400 250"
 id="RoomsSVG">

</svg>
```

Notice that the namespace declaration specifies that SVG elements are in the SVG namespace and are to be written with no namespace prefix.

The overall SVG image is 1200 pixels wide and 250 pixels high, with a viewport of 400 pixels wide by 250 pixels high, as shown by the `viewBox` attribute. The first and second values of the `viewBox` attribute indicate that the top left of the viewport is at (0,0). The third and fourth values in the `viewBox` attribute indicate that the bottom right of the viewport is located at (400,250). Any part of the image not within this viewport is invisible to the user. Each room is 400 pixels wide by 250 pixels high, so only one room fits into the viewport at the outset.

Again, the viewport is simply the part of the SVG image that is visible. Later in the tutorial, I'll use ECMAScript to manipulate the contents of the `viewBox` attribute causing the SVG image to scroll to the left or right.

© Copyright IBM Corporation 2003. All rights reserved.

Each of three rooms starts at 400 pixels wide and 250 pixels high, representing a room 40 feet wide and 25 feet deep. Next, I'll show you how to add those rooms.

The rooms

Because this is a tutorial on SVG and not interior design, I'll just represent each room as an appropriately sized rectangle.

The SVG code for each of the three rooms on the first floor -- rooms 101, 102, and 103 -- follows the same basic pattern:

```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
              "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg
 xmlns="http://www.w3.org/2000/svg"
 width="1200px" height="250px" viewBox="0 0 400 250" id="RoomsSVG">

  <svg id="Room101" width="400px" height="250px" x="0px" y="0px">
    <rect id="Room101Rect" width="100%" height="100%" fill="#CCCCFF"
           stroke="black" stroke-width="5"/>
    <text id="Room101Label" font-size="24pt" x="55px" y="100px"
           fill="black">Room 101</text>
    <text id="Room101Type" font-size="24pt" x="55px" y="150px"
           fill="black">Standard office</text>
  </svg>

  <svg id="Room102" width="400px" height="250px" x="400px" y="0px">
     ...
  </svg>

  <svg id="Room103" width="400px" height="250px" x="800px" y="0px">
     ...
  </svg>

</svg>
```

The code for each room is contained in a nested `svg` element, with the x and y attributes determining the position of that nested `svg` element.

Now remember, the image is 1200 pixels wide but the viewport is only 400 pixels wide. Next I'll center the image in the viewport.

Centering the image

Notice that the initial creation of Room 101 shows that it should appear at the top left corner of the viewport:

```
<svg id="Room101" width="400px" height="250px" x="0px" y="0px">
```

Left as-is, this would mean Room 101 fills the viewport when the page is loaded, but instead I want Room 102. To do this, I need to create a container and move all of the

© Copyright IBM Corporation 2003. All rights reserved.

rooms 400 pixels to the left:

```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
              "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg
 xmlns="http://www.w3.org/2000/svg"
 width="1200px" height="250px" viewBox="0 0 400 250" id="RoomsSVG">

<g id="mover" transform="translate(-400,0)">
  <svg id="Room101" width="400px" height="250px" x="0px" y="0px">
    <rect id="Room101Rect" width="100%" height="100%" fill="#CCCCFF"
           stroke="black" stroke-width="5"/>
    <text id="Room101Label" font-size="24pt" x="55px" y="100px"
           fill="black">Room 101</text>
    <text id="Room101Type" font-size="24pt" x="55px" y="150px"
           fill="black">Standard office</text>
  </svg>

  <svg id="Room102" width="400px" height="250px" x="400px" y="0px">
    <rect id="Room102Rect" x="0" y="0" width="100%" height="100%"
           fill="#CCFFCC" stroke="black" stroke-width="5"/>
    <text id="Room102Label" font-size="24pt" x="55px" y="100px"
           fill="black">Room 102</text>
    <text id="Room102Type" font-size="24pt" x="55px" y="150px"
           fill="black">Standard office</text>
  </svg>

  <svg id="Room103" width="400px" height="250px" x="800px" y="0px">
    <rect id="Room103Rect" width="100%" height="100%" fill="#FFCC00"
           stroke="black" stroke-width="5"/>
    <text id="Room103Label" font-size="24pt" x="55px" y="100px"
           fill="black">Room 103</text>
    <text id="Room103Type" font-size="24pt" x="55px" y="150px"
           fill="black">Standard office</text>
  </svg>
</g>

</svg>
```

The `g` element acts as a container, so any instructions, such as the `translate()` transformation, apply to any elements within it. The end result is that the transformation moves all three rooms 400 pixels to the left, leaving the middle room showing in the viewport.

Adding linking

The only thing missing from the image now is the links from each room on the first floor to the floor plan for the second floor:

```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns:xlink="http://www.w3.org/1999/xlink"
 xmlns="http://www.w3.org/2000/svg"
 width="1200px" height="250px" viewBox="0 0 400 250" id="RoomsSVG">

<g id="mover" transform="translate(-400,0)">
  <svg id="Room101" width="400px" height="250px" x="0px" y="0px">
    <rect id="Room101Rect" width="100%" height="100%" fill="#CCCCFF"
```


© Copyright IBM Corporation 2003. All rights reserved.

```
          stroke="black" stroke-width="5"/>
    <text id="Room101Label" font-size="24pt" x="55px" y="100px"
          fill="black">Room 101</text>
    <text id="Room101Type" font-size="24pt" x="55px" y="150px"
          fill="black">Standard office</text>
    <a xlink:href="Rooms2ndFloor.html"><text id="Room101Link"
          font-size="15pt" x="55px" y="200px"
          fill="black">Click to view 2nd Floor</text></a></svg>
...
</g>

</svg>
```

Linking in SVG is similar to linking in HTML, in that it uses an `a` tag and an `href` attribute. But SVG actually refers back to the XLink recommendation, so the document must declare the `xlink` namespace and use it for the `href` attribute. Otherwise, the link works just like a normal HTML link, as you'll see when the SVG image is embedded in the HTML page.

The HTML container document

In this application, the SVG file is embedded in an HTML document, **Rooms.html**, using the `embed` element. Much of the HTML container document consists of ECMAScript which is described in detail later in the tutorial, but for now here's a look at the basic framework:

```
<html>
<head>
<title>XMML Room Design</title>

<style type="text/css">
input {width: 150px;}
</style>

</head>

<body onload="Initialize()" bgcolor="#EEEEFF">

<form action="">

<table width="600">
<tr>
   <td align="center">

        <h3>XMML Room Design</h3>
        <h2>Scarman Building</h2>

        <embed id="RoomsHere" width="400px" height="250px"
               type="image/svg+xml" src="Rooms.svg"></td>
</tr>
</table>

<table>
<tr>
   <td colspan="4" height="40" align="center"><b>Current room to edit is: </b>
               <span id="currentRoom"></span></td>
</tr>
<tr>
   <td>
       <input type="button" onclick="goLeft=true; panDoc()"
```


© Copyright IBM Corporation 2003. All rights reserved.

```
                value="&lt;&lt;   Scroll Left" />
    </td>
    <td>
        <input type="button" onclick="toExec()"
                value="Make Executive Office" />
    </td>
    <td>
        <input type="button" onclick="toStandard()"
                value="Make Standard Office" />
    </td>
    <td>
        <input type="button" onclick="goRight=true; panDoc()"
                value="Scroll Right   &gt;&gt;" />
    </td>
</tr>
<tr>
    <td>
       <input type="button" onclick="zoomIn()"
                value="Zoom In" />
    </td>
    <td>
       <input type="button" onclick="increaseHeight()"
                value="Increase Height" />
    </td>
    <td>
       <input type="button" onclick="decreaseHeight()"
                value="Decrease Height" />
    </td>
    <td>
       <input type="button" onclick="zoomOut()"
                value="Zoom Out" />
    </td>
</tr>
</table>

</form>

</body>
</html>
```

The SVG image is embedded using the non-official, but very useful, `embed` element. The `embed` element is, at present, the most commonly used element for embedding SVG images in HTML Web pages because of limitations in the browser implementations of the official `object` element. Notice that the `type` attribute of the `embed` element specifies that the image to be embedded is of type `image/svg+xml`, so the browser knows what to do with it.

A simple table contains controls for the user to scroll left and right, to change the character of the room from a "standard" room to an "executive" one and back to standard, to change the height (or really, the depth) of the active room, and to zoom in and out. Notice that each `button` has an `onclick` attribute that specifies what happens when the user clicks the respective control. You'll see the actual scripts behind these handlers in Viewing floor plans .

The document also has a `span` element to tell the user which room is currently active. Scripting determines the contents of the element.

Viewing the document

© Copyright IBM Corporation 2003. All rights reserved.

Opening the HTML document in an SVG-enabled browser initially shows the middle room:

Figure 1. Middle room of the floor plan

At this point, the browser signals an error because the `Initialize()` function isn't in place yet, but that's not a problem. If you don't see the room, however, you need to go back and check the installation of your SVG viewer.

Section 4. Viewing floor plans

The viewport

© Copyright IBM Corporation 2003. All rights reserved.

So far, you've created an image that is bigger than the *window* onto it. The user can clearly see Room 102, but Room 101 is off to the left where it can't be seen and Room 103 is off to the right where it can't be seen. You can make them visible in one of three ways:

1. **Moving the image:** To see the other rooms, you can move the image using the `transform` attribute, as in the original SVG image. Doing this moves the image under the *open window* that represents the viewport. This is not, however, the most efficient way of doing things.

2. **Moving the viewport:** Rather than moving the image, you can simply move the viewport so that it is over the part of the image you wish to view, much like moving a magnifying glass over a page in the phone book. Because the actual boundaries of the viewport are fixed in place by the browser, it appears to the user as though the image itself is moving, but it's actually the viewport moving with respect to the image.

3. **Scaling the image:** In addition to moving the image, you can also make it larger or smaller to determine the area of the picture that is still visible through the viewport, which remains the same size.

In this section, I'll show you how to move the viewport in order to see different parts of the image, and how to scale the image to see more or less of it.

Global variables and initialization

You can start by declaring the global variables and calling the `Initialize()` function when the browser loads the page:

```
<html>
<head>
<title>XMML Room Design</title>

<style type="text/css">
input {width: 150px;}
</style>

<script type="text/ecmascript">

// set global variables
var htmlObj, SVGDoc, SVGRoot, viewBox, goLeft, goRight, innerSVG;
var currentSize, currentPosition, currentRoomId, currentRoomLabel;
var svgns = "http://www.w3.org/2000/svg";

function Initialize(){

   htmlObj = document.getElementById("RoomsHere");
   SVGDoc = htmlObj.getSVGDocument();
   SVGRoot = SVGDoc.documentElement;

} // end function Initialize()
```


© Copyright IBM Corporation 2003. All rights reserved.

```
</script>
</head>

<body onload="Initialize()" bgcolor="#EEEEFF">

<form action="">

<table width="600">
...
```

The `htmlObj` variable is used later as a reference to the `embed` element that accesses the SVG image. The `SVGDoc` variable represents the root node of the SVG image, the main `svg` element. The `SVGRoot` variable is used to represent the `document` element of the SVG image. The `viewBox` variable is used to hold the value of the `viewBox` attribute on the `document` element `svg` element.

The `svgns` variable represents the namespace URI of the SVG namespace, to be used later in the script when you create new elements and attributes.

The `htmlObj` variable uses the `getElementById()` method of the HTML document to identify the `embed` element --

```
<embed id="RoomsHere" width="400px"
        height="250px" type="image/svg+xml" src="Rooms.svg">
```

-- as the value assigned to it.

Next, the `getSVGDocument()` method associates the root node of the SVG document with the `SVGDoc` variable. References to the `SVGDoc` variable are references to the root node of the SVG image contained in the `embed` element. Thus you have created a way to associate events in the HTML document with variables that represent part of the SVG image displayed inside that HTML document.

Finally, the script associates the `SVGRoot` variable with the document element of the SVG image.

You'll need all of this information when you begin to scroll the image.

Obtaining the current position

The `panDoc()` function controls scrolling both to the left and to the right:

```
...
   SVGRoot = SVGDoc.documentElement;

} // end function Initialize()

function panDoc(evt){

   viewBox = SVGRoot.getAttribute('viewBox');
   var viewVals = viewBox.split(' ');
```


© Copyright IBM Corporation 2003. All rights reserved.

```
   currentPosition = parseFloat(viewVals[0]);

} // end function panDoc()

</script>
...
```

First the script assigns the `viewBox` variable the value of the `viewBox` attribute of the `svg` document element. Remember, the `SVGRoot` element represents the main `svg` element of the image, so the `getAttribute()` method works just as it would on any other XML document, retrieving the string value of the `viewBox` attribute.

The difficulty here is that `viewBox` is actually a set of space-delimited values, of which you only want the first, so the script creates an array variable, `viewVals`, by using the ECMAScript `split()` function to extract each component of the value `0 0 400 250`.

The `parseFloat()` method then converts the string value `0` so that it can be used as a numeric value `0`.

Once you have the current position, you can change it.

Moving the viewport

Next, determine whether the viewport should move left or right and adjust `currentPosition` accordingly:

```
...
function panDoc(evt){

   viewBox = SVGRoot.getAttribute('viewBox');
   var viewVals = viewBox.split(' ');
   currentPosition = parseFloat(viewVals[0]);

   if (goLeft == true){
      if (currentPosition > -400) {
          currentPosition = currentPosition - 200;
      }
      goLeft = false;
   }

   if (goRight == true){
     if (currentPosition < 400) {
          currentPosition = currentPosition + 200;
     }
     goRight = false;
   }

   viewVals[0] = currentPosition;
   SVGRoot.setAttribute('viewBox', viewVals.join(' '));

} // end function panDoc()
...
```

If `goLeft` is `true`, then `panDoc()` decreases `currentPosition`'s value by 200,

© Copyright IBM Corporation 2003. All rights reserved.

if the value is greater than -400. This prevents the SVG image from scrolling further to the left than the left edge of Room 101. The goLeft variable is then assigned the value false.

Similarly, if you are moving to the right, the script adds 200 pixels to the current position.

Now remember, it's the viewport you're moving, so the new value of the current position has to go back into the viewBox attribute. Once the new value has been added back into the array, the join() function puts the altered value of the viewBox variable back together again. The value is then assigned to the viewBox attribute of the svg element using the setAttribute() method.

As a consequence, the SVG image appears to be scrolled 200 pixels in the desired direction. For example, to fully reveal Room 101 the user needs to click twice on the **Scroll Left** text control.

Activating the scroll

When the user clicks a control, the browser executes any ECMAScript code contained in the onclick attribute. In the case of the **Scroll Left** and **Scroll Right** buttons, the onclick attribute specifies that a global variable should be assigned the value of true, and the panDoc() function should be called:

```
...
<tr>
   <td>
       <input type="button" onclick="goLeft=true; panDoc()"
           value="&lt;&lt;   Scroll Left" />
   </td>
   <td>
       <input type="button" onclick="toExec()"
           value="Make Executive Office" />
   </td>
   <td>
       <input type="button" onclick="toStandard()"
           value="Make Standard Office" />
   </td>
   <td>
       <input type="button" onclick="goRight=true; panDoc()"
           value="Scroll Right   &gt;&gt;" />
   </td>
</tr>
...
```

(It's also possible to pan an image in the Adobe SVG viewer by holding down the ALT key, pressing down the left mouse button, and moving the mouse, but for the purposes of this tutorial you need to directly control where and how much the viewport moves instead.)

You can also change the portions of the image seen through the viewport by making

© Copyright IBM Corporation 2003. All rights reserved.

the image itself larger or smaller.

Zooming in and out

The controls also provide a way to zoom in and out of the image. When the user clicks the **Zoom In** button, the browser executes the `zoomIn()` function. When the user clicks the **Zoom Out** button, the browser executes the `zoomOut()` function:

```
...
   SVGRoot.setAttribute('viewBox', viewVals.join(' '));

} // end function panDoc()

function zoomIn(){

   if (SVGRoot.currentScale < 5){
      SVGRoot.currentScale = SVGRoot.currentScale * 1.5;
   }

} // end function zoomIn

function zoomOut(){

   if (SVGRoot.currentScale > 0.3){
      SVGRoot.currentScale = SVGRoot.currentScale * 0.67;
   }

} // end function zoomOut

</script>
...
```

The `zoomIn()` function is straightforward. The `SVGRoot` variable has already been assigned the `svg` document element of the SVG image in the `Initialize()` function, and when an SVG image is first loaded the value of the `currentScale` property of the `SVGRoot` variable is 1.0.

Each time the user clicks the **Zoom In** button, the script multiplies the value of the `currentScale` property by 1.5, but only if the existing value of that property is less than 5. This has the effect of zooming into the SVG image or, if you prefer, increasing the size of all parts of the image.

The `zoomOut()` function works the same way as `zoomIn()`, but in the other direction.

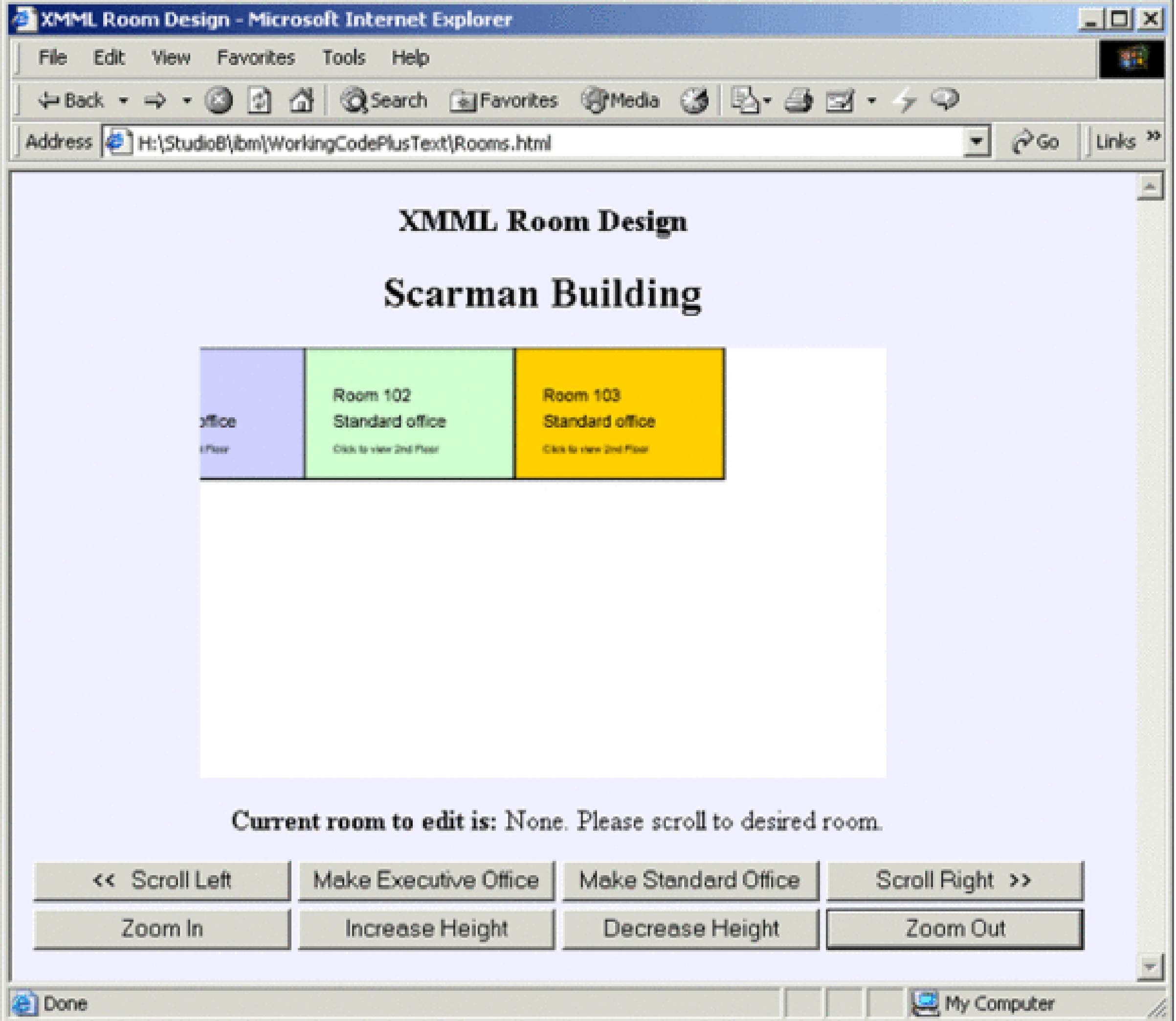
Zooming and panning in action

It's important to understand the effect that changes in one property have on the others. For example, each "scroll" action moves the image 200 pixels in one direction or the other, so it's natural to assume that if the image has been scaled to half its normal size, you can scroll past an entire 400-pixel-wide room in one click,

© Copyright IBM Corporation 2003. All rights reserved.

but that's not actually the case. When you change the scale, all measurements are affected, as you can see in this image:

Figure 2. Effect of zoom and scroll



Now that you can control what parts of the image you see, I'll show you how to change the image itself.

Section 5. Manipulating an SVG image

Changing the room

© Copyright IBM Corporation 2003. All rights reserved.

The purpose of this tutorial is to create an SVG floor plan that the user can manipulate to make changes to better suit his or her needs. For example, a home builder might enable users to substitute a den for a bedroom, or a utility room for a bathroom. Users might also be able to stretch or shrink rooms.

In this tutorial, you'll get to know all of those concepts by enabling users to change a room's type from Executive Office to Standard Office and back, and to make the room larger or smaller. To make these changes to the image, you'll alter, add, and remove objects that are part of the SVG image's Document Object Model (DOM).

First, you need to identify which of the three rooms in the SVG image is to be changed.

Identifying the current room

To unambiguously identify which room is to be altered, it is necessary to determine which room is visible to the user at the time the user clicks the **Make Executive Office** or **Make Standard Office** button. To do that, you'll add the `setCurrentRoom()` function to the page:

```
...
} // end function Initialize()

function setCurrentRoom(){

  viewBox = SVGRoot.getAttribute('viewBox');
  var viewVals = viewBox.split(' ');
  currentPosition = parseFloat(viewVals[0]);

  if (currentPosition % 400 != 0) {
    currentRoomId = null;
    currentRoomLabel =
            "None.  Please scroll to desired room.";
  } else if (currentPosition / 400 == 0) {
    currentRoomId = "Room102";
    currentRoomLabel = "Room 102";
  } else if (currentPosition / 400 == -1) {
    currentRoomId = "Room101";
    currentRoomLabel = "Room 101";
  } else if (currentPosition / 400 == 1) {
    currentRoomId = "Room103";
    currentRoomLabel = "Room 103";
  }

}

function panDoc(evt){
...
```

This is the same technique you used in Obtaining the current position, but now you're using that information to determine which of the rooms currently lines up with the left-hand edge of the viewport. If none of the boxes is lined up, as is the case if the viewport is not a multiple of 400 pixels from its original position, there is no

© Copyright IBM Corporation 2003. All rights reserved.

current room, and the script sets the value of `currentRoomId` accordingly. In the other cases, notice that the `currentRoomId` value matches the prefix of the relevant `id` values within the SVG document (as seen in The rooms). You'll use that information later, when you start manipulating the document.

But this function does you no good unless it gets executed at the proper time.

Setting the current room

Two situations warrant the explicit checking for the current room: the initialization of the page and panning of the document. Executing the `setCurrentRoom()` script at those moments is straightforward:

```
...
function Initialize(){

   htmlObj = document.getElementById("RoomsHere");
   SVGDoc = htmlObj.getSVGDocument();
   SVGRoot = SVGDoc.documentElement;

   setCurrentRoom();

} // end function Initialize()

function setCurrentRoom(){
...
}

function panDoc(evt){

   currentPosition = parseFloat(viewVals[0]);

   if (goLeft == true){
      if (currentPosition > -400) {
          currentPosition = currentPosition - 200;
      }
      goLeft = false;
   }

   if (goRight == true){
     if (currentPosition < 400) {
          currentPosition = currentPosition + 200;
     }
     goRight = false;
   }

   viewBox = SVGRoot.getAttribute('viewBox');
   var viewVals = viewBox.split(' ');
   viewVals[0] = currentPosition;
   SVGRoot.setAttribute('viewBox', viewVals.join(' '));

   setCurrentRoom();

} // end function panDoc()
...
```

When the page is loaded, the `Intialize()` function executes, so this is an easy way to start off by initializing the global `currentPosition` variable. Also, because

© Copyright IBM Corporation 2003. All rights reserved.

it is a global variable, and because you know it's been set, you don't have to retrieve the current `viewBox` attribute at the beginning of `panDoc()`. You will, however, need the array at the end of the script so you can save the new value.

Once the viewport has been moved, you need to re-set the current room to the new value.

Now you just have to tell the user what room he or she is about to change.

Indicating the current room

You may remember that within the HTML document, a `span` element has an `id` value of `currentRoom`:

```
...
<tr>
   <td colspan="4" height="40"
        align="center"><b>Current room to edit is: </b>
        <span id="currentRoom"></span></td>
</tr>
...
```

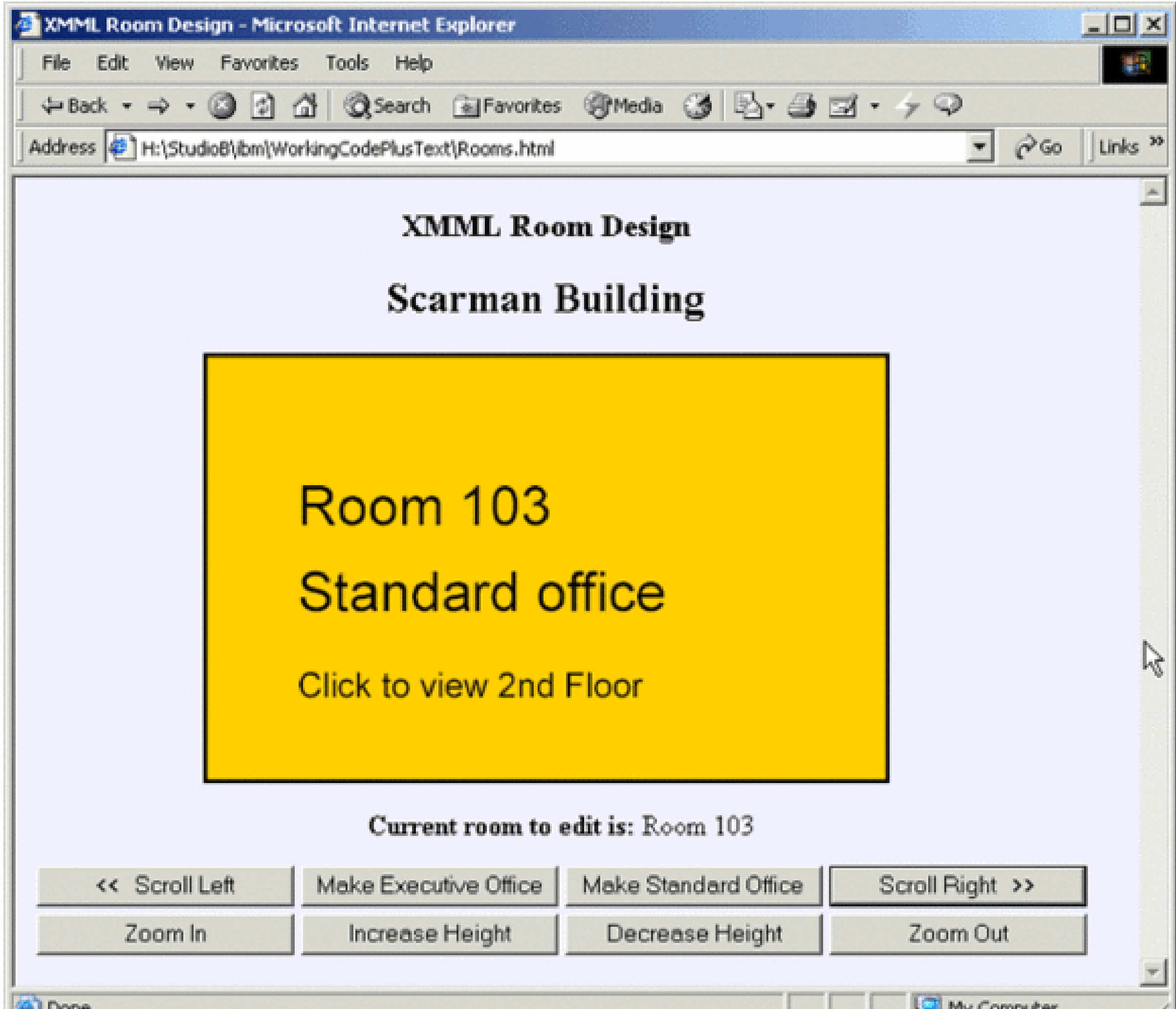
It's important to provide the user with this information because no changes can be made to any room if the image is *between rooms,* so to speak. So within the `setCurrentRoom()` function, you can set a value for this text:

```
...
function setCurrentRoom(){
   viewBox = SVGRoot.getAttribute('viewBox');
   var viewVals = viewBox.split(' ');
   currentPosition = parseFloat(viewVals[0]);

   if (currentPosition % 400 != 0) {
      currentRoomId = null;
      currentRoomLabel = "None.  Please scroll to desired room.";
   } else if (currentPosition / 400 == 0) {
      currentRoomId = "Room102";
      currentRoomLabel = "Room 102";
   } else if (currentPosition / 400 == -1) {
      currentRoomId = "Room101";
      currentRoomLabel = "Room 101";
   } else if (currentPosition / 400 == 1) {
      currentRoomId = "Room103";
      currentRoomLabel = "Room 103";
   }

   document.getElementById("currentRoom").innerHTML =
                                   currentRoomLabel;

}
...
```

After the calculations are done, the contents of the `span` element are set to the current label, so the user knows what room he or she is about to edit, or in the case of a split situation, what to do about it:

© Copyright IBM Corporation 2003. All rights reserved.

Figure 3. Displaying the current room number



Now that you know what room you're working with, you can start changing it.

Changing the background color

The easiest way to make a change to an SVG image is to change the value of an attribute. To signify the power of an Executive Office, that room's background color is to be changed to red.

Changing the room to an Executive Office involves changing the fill of the `rect` element. When the `currentRoomId` is `Room102`, the `setAttribute()` method changes the value of the `fill` attribute of the `rect` element, which has the `id` attribute with a value of `Room102Rect`.

```
...
} // end function panDoc()
```


© Copyright IBM Corporation 2003. All rights reserved.

```
function toExec(){

   if (currentRoomId == null) {

      alert("Ambiguous room position! \n Please align a room "+
              "precisely before attempting to change room type.");

   } else {

      SVGDoc.getElementById(currentRoomId+"Rect")
                                .setAttribute("fill", "#FF0000");

   }

} // end function toExec()

function zoomIn(){

   if (SVGRoot.currentScale < 5){
...
```

At this stage, the background color is red but everything else remains the same. Now you can start removing pieces.

Deleting parts of the SVG image

In changing a room to an Executive Office, you not only change the color, but also the text. The current room label and room number are removed, and a new label and number are added, though the number is essentially the same:

```
...
function toExec(){

   if (currentRoomId == null) {

      alert("Ambiguous room position! \n Please align a room "+
         "precisely before attempting to change room type.");

   } else {

      SVGDoc.getElementById(currentRoomId+"Rect")
                                .setAttribute("fill", "#FF0000");

      innerSVG = SVGDoc.getElementById(currentRoomId);

      oldChild = SVGDoc.getElementById(currentRoomId+"Label");
      innerSVG.removeChild(oldChild);

      oldChild = SVGDoc.getElementById(currentRoomId+"Type");
      innerSVG.removeChild(oldChild);

   }

} // end function toExec()
...
```

The `innerSVG` variable is assigned the object retrieved by the `getElementById()` method. For example, if the `currentRoomId` is `Room102`,

© Copyright IBM Corporation 2003. All rights reserved.

the script essentially executes:

```
innerSVG = SVGDoc.getElementById("Room102");
```

This is the nested `svg` element for Room 102.

Next the script retrieves a reference to the label ("Room102Label") text node and removes it from the `innerSVG` element. The script then does the same for the type.

At this stage, the first two `text` elements (but not the one contained in the hyperlink) have been removed.

Adding new elements to the SVG image

After the deletion of two `text` elements and the change of background color, the new Executive Office gets new `text` elements:

```
...
   } else {

      SVGDoc.getElementById(currentRoomId+"Rect")
                                    .setAttribute("fill", "#FF0000");

      innerSVG = SVGDoc.getElementById(currentRoomId);
      oldChild = SVGDoc.getElementById(currentRoomId+"Label");
      innerSVG.removeChild(oldChild);

      oldChild = SVGDoc.getElementById(currentRoomId+"Type");
      innerSVG.removeChild(oldChild);

      myData = SVGDoc.createTextNode("Executive Office");
      newText = SVGDoc.createElementNS(svgns, "text");
      newText.appendChild(myData);
      SVGDoc.getElementById(currentRoomId).appendChild(newText);

      myData = SVGDoc.createTextNode(currentRoomLabel);
      newText = SVGDoc.createElementNS(svgns, "text");
      newText.appendChild(myData);
      SVGDoc.getElementById(currentRoomId).appendChild(newText);
   }

} // end function toExec()
...
```

As with any DOM operation, you first create a text node using the `Document` object (in this case, `SVGDoc`). It should contain the text `"Executive Office"`. This text node needs to be added to an element named `"text"`, but that element must be aware that it is part of the SVG namespace, so use `createElementNS()` to create the new element. The global variable `svgns` specifies the namespace URI.

Next, add the text node to the newly created `text` element node using the `appendChild()` method. The `text` element node (and its text node child) need to be added to the correct `svg` element node. The `getElementById()` method

© Copyright IBM Corporation 2003. All rights reserved.

retrieves the `svg` element node for the correct room and appends the new `text` element node.

Carry out the same process to add the second new `text` element to the executive room.

Setting attributes

Once the script creates the text nodes and adds them to the `text` elements, their positioning and appearance depend on the attributes of the `text` element. Set those within the script as well:

```
...
      myData = SVGDoc.createTextNode("Executive Office");
      newText = SVGDoc.createElementNS(svgns, "text");
      newText.setAttributeNS(null, "id", currentRoomId+"Type");
      newText.setAttributeNS(null, "x", "55px");
      newText.setAttributeNS(null, "y", "100px");
      newText.setAttributeNS(null, "font-size", "24pt");
      newText.setAttributeNS(null, "stroke", "white");
      newText.appendChild(myData);
      SVGDoc.getElementById(currentRoomId).appendChild(newText);

      myData = SVGDoc.createTextNode(currentRoomLabel);
      newText = SVGDoc.createElementNS(svgns, "text");
      newText.setAttributeNS(null, "id", currentRoomId+"Label");
      newText.setAttributeNS(null, "x", "55px");
      newText.setAttributeNS(null, "y", "150px");
      newText.setAttributeNS(null, "font-size", "24pt");
      newText.setAttributeNS(null, "stroke", "white");
      newText.appendChild(myData);
      SVGDoc.getElementById(currentRoomId).appendChild(newText);
   }

} // end function toExec()
...
```

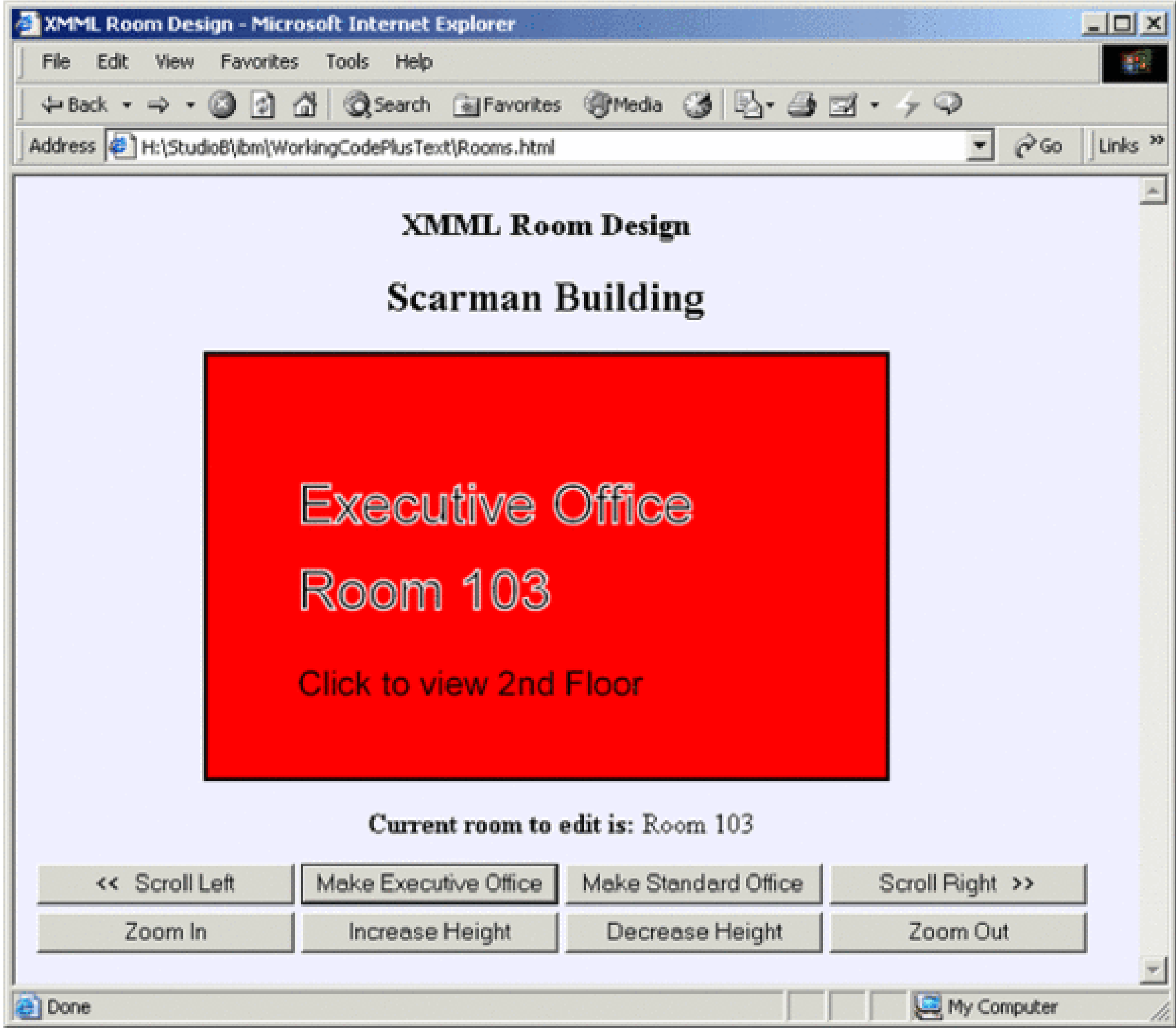
Remember, performing these operations is the same as creating a `text` element as:

```
<svg:text id="Room102Label" x="55px" y="150px" font-size="24pt"
        stroke="white">Room 102</svg:text>
```

The end result is a new Executive Office:

Figure 4. Viewing a newly created Executive Office

© Copyright IBM Corporation 2003. All rights reserved.



The same process can create a Standard Office out of an Executive Office.

Creating the Standard Office

The process for creating a Standard Office is essentially the same as for creating an Executive Office:

```
...
} // end function toExec()

function toStandard(){

   if (currentRoomId == null){
      alert("Ambiguous room position! \n Please align a room "+
        "precisely before attempting to change room type.");
   } else {

      SVGDoc.getElementById(currentRoomId+"Rect")
                                .setAttribute("fill", "#CCFFCC");
```


© Copyright IBM Corporation 2003. All rights reserved.

```
      innerSVG = SVGDoc.getElementById(currentRoomId );

      oldChild = SVGDoc.getElementById(currentRoomId+"Label");
      innerSVG.removeChild(oldChild);

      oldChild = SVGDoc.getElementById(currentRoomId+"Type");
      innerSVG.removeChild(oldChild);

      myData = SVGDoc.createTextNode("Standard Office");
      newText = SVGDoc.createElementNS(svgns, "text");
      newText.setAttributeNS(null, "id", currentRoomId+"Type");
      newText.setAttributeNS(null, "x", "55px");
      newText.setAttributeNS(null, "y", "150px");
      newText.setAttributeNS(null, "font-size", "24pt");
      newText.setAttributeNS(null, "stroke", "black");
      newText.appendChild(myData);

      SVGDoc.getElementById(currentRoomId).appendChild(newText);
      myData = SVGDoc.createTextNode(currentRoomLabel);
      newText = SVGDoc.createElementNS(svgns, "text");
      newText.setAttributeNS(null, "id", currentRoomId+"Label");
      newText.setAttributeNS(null, "x", "55px");
      newText.setAttributeNS(null, "y", "100px");
      newText.setAttributeNS(null, "font-size", "24pt");
      newText.setAttributeNS(null, "stroke", "black");
      newText.appendChild(myData);
      SVGDoc.getElementById(currentRoomId).appendChild(newText);

   } // end if

} // end function toStandard()

function zoomIn(){

   if (SVGRoot.currentScale < 5){
...
```

Increasing room height

One of the goals of this project is to enable the user to change the size of a room in the layout at will. In this example, the user gets to change the height of the room as it's rendered in the floor plan:

```
...
   } // end if

} // end function toStandard()

function increaseHeight(){

   if (currentRoomId == null) {
      alert("Ambiguous room position! \n Please align a room "+
        "precisely before attempting to change room height.");
   } else {

      innerSVG = SVGDoc.getElementById(currentRoomId);
      currentSize = parseInt(innerSVG.getAttributeNS(null,
                               "height").substr( 0, 3 ));
      if (currentSize < 500){
         currentSize = currentSize + 50;
         currentSize = currentSize + "px";
         innerSVG.setAttributeNS(null, "height", currentSize);
```


© Copyright IBM Corporation 2003. All rights reserved.

```
      } else {
         alert("Size of "+currentRoomLabel+
            "is already at the maximum allowed.");
      }
   }

} // end function increaseHeight()

function zoomIn(){

   if (SVGRoot.currentScale < 5){
...
```

Once again, the `SVGDoc` object represents the root `svg` element, so `innerSVG` represents the appropriate room. The value of the `height` attribute when the document loads is `"300px"`, which is a string value, so the `substr()` method is used to extract the first three characters of the string and the `parseInt()` method is used to convert that to an integer.

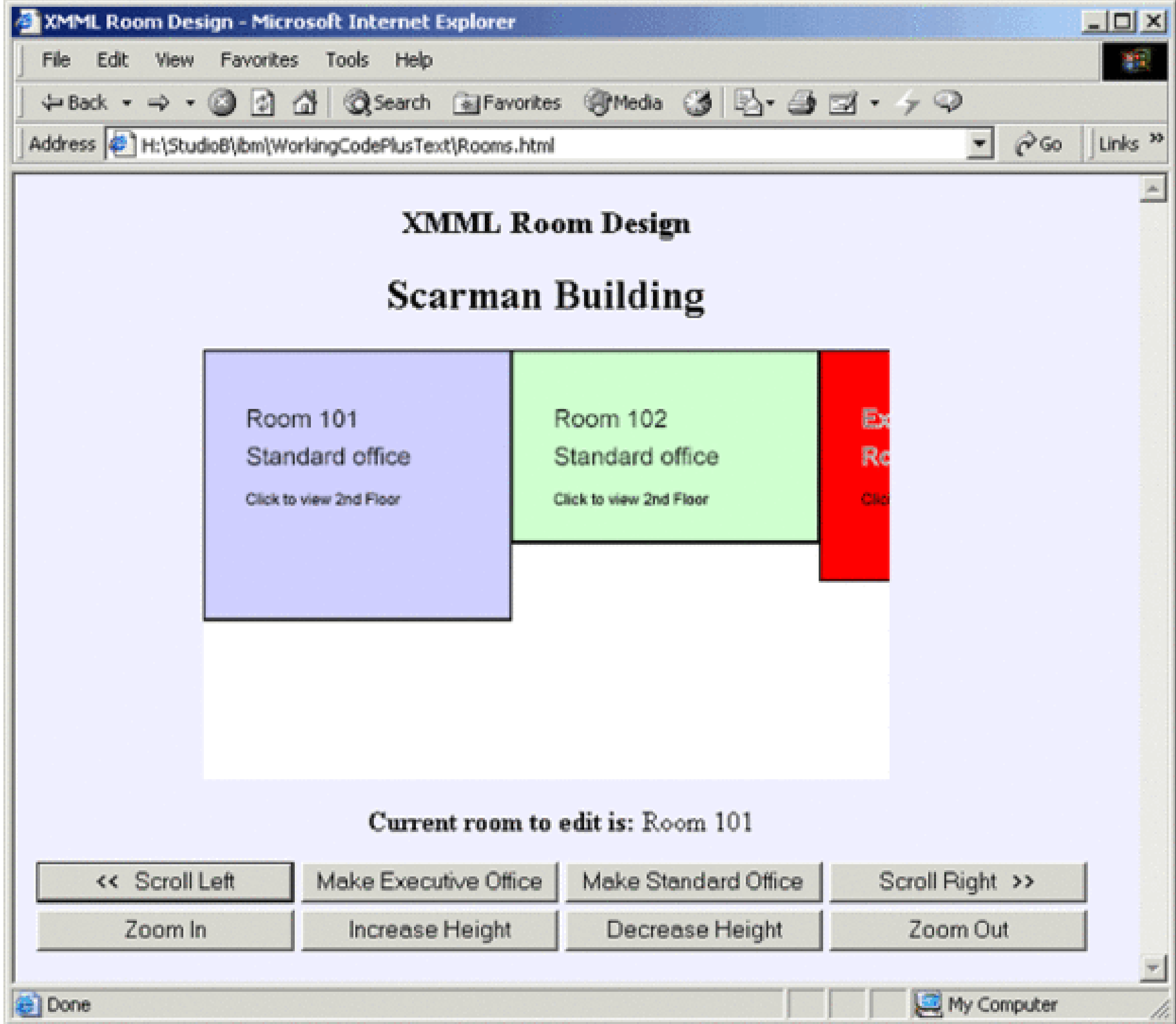
For the `substr()` function, the first argument is the starting character (numbering starts at zero) and the second argument indicates the number of characters in the substring.

The maximum allowed height dimension for the room is 500 pixels (or for the real room, a depth of 50 feet), so the script tests that before any change is made to the height. If the height is less than 500, increase the value of `currentSize` by 50. Finally, concatenate the string `"px"` to the value of `currentSize` and assign the new value of the `currentSize` variable to the height attribute of the `svg` element for the appropriate room.

Because this is a *live* object, the changes, like the changes in the color and text of the rooms, show immediately. The size of the `svg` element increases, and because the contained `rect` element has a `height` attribute of `100%`, it increases to the new size of the `svg` element:

Figure 5. Viewing the increased height of an office

© Copyright IBM Corporation 2003. All rights reserved.



Decreasing the height involves basically the same process.

Decreasing room height

The `decreaseHeight()` script is virtually the same as the `increaseHeight()` script:

```
...
      } else {
         alert("Size of "+currentRoomLabel+
            "is already at the maximum allowed.");
      }
   }

} // end function increaseHeight()

function decreaseHeight(){

   if (currentRoomId == null) {
```


© Copyright IBM Corporation 2003. All rights reserved.

```
      alert("Ambiguous room position! \n Please align a room "+
        "precisely before attempting to change room height.");
   } else {

      innerSVG = SVGDoc.getElementById(currentRoomId);
      currentSize = parseInt(innerSVG.getAttributeNS(null,
                                "height").substr( 0, 3 ));
      if (currentSize > 250){
         currentSize = currentSize - 50;
         currentSize = currentSize + "px";
         innerSVG.setAttributeNS(null, "height", currentSize);
      } else{
         alert("Size of "+currentRoomLabel+
            "is already at the minimum allowed.");
      }
   }

} // end function decreaseHeight()

function zoomIn(){

   if (SVGRoot.currentScale < 5){
...
```

Taking things one step further

In this section, you've seen how to add, remove, and alter elements in the portion of the XML document that is the SVG image. Although the tutorial shows a simple example, it discusses all of the concepts necessary for a sophisticated floor plan editing system.

For example, the tutorial explains how to remove elements from an SVG image. You can take this one step further and enable the user to remove a room from the floor plan altogether. Similarly, the tutorial demonstrates the addition of new objects to the SVG image. You can use this same concept to enable the user to add an entirely new room to the floor plan.

The tutorial shows a simple increase and decrease in the height of a bounding box in response to the user's clicking of a button. You can expand this to enable the user to drag walls, or even rooms, to new locations. Just remember, though, that if a user wants to move a wall that's adjacent to another room, that room has to move as well.

All of these capabilities are available to you because of the flexibility provided by SVG in particular, and XML in general. In addition, you can dynamically generate SVG using XSL Transformations.

Section 6. XSLT and SVG

© Copyright IBM Corporation 2003. All rights reserved.

Creating SVG dynamically

SVG images can be created by hand, either by coding directly in a text editor or by drawing in a graphics, CAD, or similar package that has SVG export capabilities. An alternate approach is to create SVG dynamically by transforming source XML using XSLT or a procedural programming language. Typically this is done server-side to allow SVG images to be created to reflect the current state of, for example, a data store. Just as dynamic creation of HTML Web pages is the norm for many purposes, dynamic creation of SVG images is also likely to become the normal method of creating them. Hand coding or drawing will produce the skeleton of the SVG document and dynamic content will be added using XSLT and various other alternative server-side languages.

Dynamic SVG generation also enables you to quickly make changes. For example, just as an SVG chart can react to new data in real time (or as frequently as its updated, at any rate) SVG images such as floor plans can react to situations such as product line changes. If all of the initial floor plans are stored as XML documents that describe each room, changes to that information in the database will be reflected in the document. For example, if rooms were to start out at 20 feet instead of 25, this would show as soon as the change was made.

In this section I'll show you a simple example of dynamic creation of SVG by transforming the source XML document, `RoomData.xml`, into an SVG image that can be inserted into the HTML container document, `Rooms.html`.

The XML source document

The data held in XML about each room on the first floor is very straightforward. The room number, dimensions, and the type of room are stored in the XML document, `RoomData.xml`:

```
<?xml version='1.0'?>
<RoomData>
   <Room position="1" type="Standard" width="40" depth="25" />
   <Room position="2" type="Standard" width="40" depth="22" />
   <Room position="3" type="Standard" width="40" depth="25" />
</RoomData>
```

In this case, the source XML is very simple. It describes the initial state of each room. Notice that it shows the position of each room, and not the room number. This way, the data can be used to generate a floor plan for different floors, or with a different numbering scheme. Notice also that the middle room is slightly less deep than the others, unlike the static examples so far.

© Copyright IBM Corporation 2003. All rights reserved.

Choosing an XSLT processor

At the time of this writing, several XSLT processors are available. Two widely used open-source XSLT processors are Saxon and Xalan (see Resources).

Currently I am exploring the additional functionality of XSLT 2.0 and so am using Saxon 7.5, which implements more of XSLT 2.0 than any other XSLT processor meantime. Fortunately, Saxon is also a powerful and well-tested XSLT 1.0 processor so it meets our needs for this tutorial too.

The skeleton of the SVG image

Here's the basic skeleton of the SVG image:

```
                      <?xml version='1.0'?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns="http://www.w3.org/2000/svg">
<xsl:output method="xml" indent="yes" encoding="UTF-8" />

<xsl:template match="/">

  <svg xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns="http://www.w3.org/2000/svg"
        width="1200px" height="250px" viewBox="0 0 400 250"
        id="RoomsSVG">

    <g id="mover" transform="translate(-400,0)">
       <xsl:apply-templates select="RoomData/Room" />
    </g>
  </svg>
</xsl:template>

</xsl:stylesheet>
```

Notice that the stylesheet has namespace declarations for the XSLT namespace and the SVG namespace on the document element, and that the SVG namespace is the default namespace for the document.

The stylesheet specifies the output method as XML. The `xsl:output` element can, if preferred, be omitted because the default output method is `xml`. However, if you want to indent the output you need to use the `xsl:output` element and indicate it using the `indent` attribute.

The main template creates the root `svg` element and the `g` container element, and then calls additional templates for each room.

© Copyright IBM Corporation 2003. All rights reserved.

Creating each room

Now take a look at the room template.

```
...
<g id="mover" transform="translate(-400,0)">
<xsl:apply-templates select="RoomData/Room" />
</g>
</svg>
</xsl:template>

<xsl:template match="Room">
  <xsl:element name="svg">
    <xsl:attribute name="id"><xsl:value-of
        select="concat('Room10', @position)" /></xsl:attribute>
    <xsl:attribute name="width"><xsl:value-of
        select="(10 * @width)" />px</xsl:attribute>
    <xsl:attribute name="height"><xsl:value-of
        select="(10 * @depth)" />px</xsl:attribute>
    <xsl:attribute name="x"><xsl:value-of
        select="concat(((@position - 1) * 400), 'px')" />
    </xsl:attribute>
    <xsl:attribute name="y">0px</xsl:attribute>

  </xsl:element> <!-- End of nested svg Element -->
</xsl:template>

</xsl:stylesheet>
```

The `xsl:element` element creates the nested `svg` element that is the container element for each room. The `xsl:attribute` element is used to specify the attributes of the `svg` element. An alternate approach would be to specify the `svg` element and its attributes as literal result elements, but that's not an option here because some of the attributes are dynamically generated based on the data.

The value of the `id` attribute is specified using the XPath `concat()` function to concatenate the literal string `Room 10` with the value of the `position` attribute of the `Room` element node, which is derived from the source document.

The values of the `width` and `height` attributes are determined by assuming that the scale is 10 pixels per foot, so again the `xsl:value-of` element supplies the calculated value. This makes it simple to change the dimensions of the room; as long as the XML source is correct, the room will be the proper size.

The value of the `x` attribute is different for each room, and depends on the position. In this project, you assume that each room is the same width, but in reality you would either need to build in logic to determine the appropriate location based on the previous rooms or include the position in the data. The `concat()` function then adds the text `px`, signifying pixels, to complete the value of the `x` attribute. The value of the `y` attribute doesn't change, and is supplied literally.

So far, the resulting document looks like this:

© Copyright IBM Corporation 2003. All rights reserved.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink"
      id="RoomsSVG" viewBox="0 0 400 250" height="250px" width="1200px">
<g transform="translate(-400,0)" id="mover">
<svg id="Room101" width="400px" height="250px" x="0px" y="0px"/>
<svg id="Room102" width="400px" height="220px" x="400px" y="0px"/>
<svg id="Room103" width="400px" height="250px" x="800px" y="0px"/>
</g>
</svg>
```

Now I'll show you the `rect` element child of the `svg` element.

Adding the background rectangle

To add the colored background rectangle it is necessary to create a `rect` element:

```
...

<xsl:template match="Room">
<xsl:element name="svg">
  <xsl:attribute name="id"><xsl:value-of
        select="concat('Room10', @position)" /></xsl:attribute>
  <xsl:attribute name="width">
  <xsl:value-of select="(10 * @width)" />px</xsl:attribute>
  <xsl:attribute name="height">
  <xsl:value-of select="(10 * @depth)" />px</xsl:attribute>
  <xsl:attribute name="x"><xsl:value-of
        select="concat(((@position - 1) * 400), 'px')" />
  </xsl:attribute>
 <xsl:attribute name="y">0px</xsl:attribute>

 <xsl:element name="rect">
   <xsl:attribute name="id"><xsl:value-of
        select="concat('Room10', @position, 'Rect')" />
   </xsl:attribute>
   <xsl:attribute name="width">100%</xsl:attribute>
   <xsl:attribute name="height">100%</xsl:attribute>
   <xsl:choose>
     <xsl:when test="@position='1'">
       <xsl:attribute name="fill">#CCCCFF</xsl:attribute>
     </xsl:when>
     <xsl:when test="@position='2'">
       <xsl:attribute name="fill">#CCFFCC</xsl:attribute>
     </xsl:when>
     <xsl:when test="@position='3'">
       <xsl:attribute name="fill">#FFCC00</xsl:attribute>
     </xsl:when>
   </xsl:choose>
   <xsl:attribute name="stroke">black</xsl:attribute>
   <xsl:attribute name="stroke-width">5</xsl:attribute>
 </xsl:element> <!-- End of rect Element -->

</xsl:element> <!-- End of nested svg Element -->
</xsl:template>

</xsl:stylesheet>
```

The value of the `id` attribute of the `rect` element is constructed by concatenating the string literal `Room` with the value of the `number` attribute and the string literal

© Copyright IBM Corporation 2003. All rights reserved.

`Rect`; the stylesheet supplies the values of the `width` and `height` attributes literally.

The three rooms (101, 102, and 103) are to have different colored backgrounds, so the `xsl:choose` element specifies what happens depending on the value of the `position` attribute of the `Room` element in the source document. If the value of the `test` attribute of an `xsl:when` element returns the boolean value of `true`, then the content specified (in this case, the `fill` attribute) is added by the XSLT processor to the result tree.

Finally, the `stroke` and `stroke-width` attributes of the `rect` element are supplied literally. This leaves you with a current result of:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink"
      id="RoomsSVG" viewBox="0 0 400 250" height="250px" width="1200px">
  <g transform="translate(-400,0)" id="mover">
    <svg id="Room101" width="400px" height="250px" x="0px" y="0px">
      <rect id="Room 101Rect" width="100%" height="100%" fill="#CCCCFF"
          stroke="black" stroke-width="5"/>
    </svg>
    <svg id="Room102" width="400px" height="220px" x="400px" y="0px">
      <rect id="Room 102Rect" width="100%" height="100%" fill="#CCFFCC"
          stroke="black" stroke-width="5"/>
    </svg>
    <svg id="Room103" width="400px" height="250px" x="800px" y="0px">
      <rect id="Room 103Rect" width="100%" height="100%" fill="#FFCC00"
          stroke="black" stroke-width="5"/>
    </svg>
  </g>
</svg>
```

Now it's just a matter of adding the `text` elements.

Adding the text elements

Adding the text elements involves the same techniques used in Creating each room and Adding the background rectangle:

```
...
   <xsl:attribute name="stroke">black</xsl:attribute>
   <xsl:attribute name="stroke-width">5</xsl:attribute>
 </xsl:element> <!-- End of rect Element -->

 <xsl:element name="text">
   <xsl:attribute name="id"><xsl:value-of
        select="concat('Room10', @position, 'Label')" />
   </xsl:attribute>
   <xsl:attribute name="font-size">24pt</xsl:attribute>
   <xsl:attribute name="x">55px</xsl:attribute>
   <xsl:attribute name="y">100px</xsl:attribute>
   <xsl:attribute name="fill">black</xsl:attribute>
   Room 10<xsl:value-of select="@position" />
 </xsl:element> <!-- End of first text Element -->
```


© Copyright IBM Corporation 2003. All rights reserved.

```
  <xsl:element name="text">
    <xsl:attribute name="id"><xsl:value-of
         select="concat('Room10', @position, 'Type')" />
    </xsl:attribute>
    <xsl:attribute name="font-size">24pt</xsl:attribute>
    <xsl:attribute name="x">55px</xsl:attribute>
    <xsl:attribute name="y">150px</xsl:attribute>
    <xsl:attribute name="fill">black</xsl:attribute>
    <xsl:value-of select="@type" /> office
  </xsl:element> <!-- End of second text Element -->

  <xsl:element name="a">
    <xsl:attribute name="xlink:href">Rooms2ndFloor.html</xsl:attribute>

      <xsl:element name="text">
        <xsl:attribute name="id"><xsl:value-of
         select="concat('Room10', @position, 'Link')" />
        </xsl:attribute>
        <xsl:attribute name="font-size">15pt</xsl:attribute>
        <xsl:attribute name="x">55px</xsl:attribute>
        <xsl:attribute name="y">200px</xsl:attribute>
        <xsl:attribute name="fill">black</xsl:attribute>
        Click to view 2nd Floor
      </xsl:element> <!-- End of third text Element -->

  </xsl:element> <!-- End of the <a> Element -->

 </xsl:element> <!-- End of nested svg Element -->
 </xsl:template>

 </xsl:stylesheet>
```

As before, the `concat()` function helps to create the `id` attribute of the `text` element, incorporating the value of the `position` attribute, while the values of the `font-size`, `x`, `y`, and `fill` attributes are supplied literally.

Once the attributes of the `text` element are in place, the value of the `xsl:value-of` element helps to specify the actual text of the element by outputting the `position` attribute of the `Room` element from the source document. The construction of the second `text` element is similar.

Finally, the stylesheet specifies the link to the second floor. Notice the namespace information on the attribute name.

The final document

With all of these changes in place, the stylesheet can generate a complete document. In fact, if you wanted to, you could add a fourth room to the source document:

```
<?xml version='1.0'?>
<RoomData>
  <Room position="1" type="Standard" width="40" depth="25" />
  <Room position="2" type="Standard" width="40" depth="22" />
  <Room position="3" type="Standard" width="40" depth="25" />
  <Room position="4" type="Standard" width="40" depth="22" />
</RoomData>
```


© Copyright IBM Corporation 2003. All rights reserved.

This room would automatically be added to the SVG document:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     id="RoomsSVG" viewBox="0 0 400 250" height="250px" width="1200px">
  <g transform="translate(-400,0)" id="mover">
    <svg id="Room101" width="400px" height="250px" x="0px" y="0px">
      <rect id="Room 101Rect" width="100%" height="100%" fill="#CCCCFF"
            stroke="black" stroke-width="5"/>
      <text id="Room 101Label" font-size="24pt" x="55px" y="100px"
            fill="black"> Room 101</text>
      <text id="Room 101Type" font-size="24pt" x="55px" y="150px"
            fill="black">Standard office</text>
      <a xlink:href="Rooms2ndFloor.html">
        <text id="Room 101Link" font-size="15pt" x="55px" y="200px"
            fill="black">Click to view 2nd Floor
        </text>
      </a>
    </svg>
    <svg id="Room102" width="400px" height="220px" x="400px" y="0px">
      <rect id="Room 102Rect" width="100%" height="100%"
            fill="#CCFFCC" stroke="black" stroke-width="5"/>
      <text id="Room 102Label" font-size="24pt" x="55px" y="100px"
            fill="black"> Room 102</text>
      <text id="Room 102Type" font-size="24pt" x="55px" y="150px"
            fill="black">Standard office</text>
      <a xlink:href="Rooms2ndFloor.html">
        <text id="Room 102Link" font-size="15pt" x="55px" y="200px"
            fill="black">Click to view 2nd Floor
        </text>
      </a>
    </svg>
    <svg id="Room103" width="400px" height="250px" x="800px" y="0px">
      <rect id="Room 103Rect" width="100%" height="100%"
            fill="#FFCC00" stroke="black" stroke-width="5"/>
      <text id="Room 103Label" font-size="24pt" x="55px" y="100px"
            fill="black">Room 103</text>
      <text id="Room 103Type" font-size="24pt" x="55px" y="150px"
            fill="black">Standard office</text>
      <a xlink:href="Rooms2ndFloor.html">
        <text id="Room 103Link" font-size="15pt" x="55px" y="200px"
            fill="black">Click to view 2nd Floor
        </text>
      </a>
    </svg>
    <svg id="Room104" width="400px" height="220px" x="1200px" y="0px">
      <rect id="Room 104Rect" width="100%" height="100%"
            stroke="black" stroke-width="5"/>
      <text id="Room 104Label" font-size="24pt" x="55px" y="100px"
            fill="black">Room 104</text>
      <text id="Room 104Type" font-size="24pt" x="55px" y="150px"
            fill="black">Standard office</text>
      <a xlink:href="Rooms2ndFloor.html">
        <text id="Room 104Link" font-size="15pt" x="55px" y="200px"
            fill="black">Click to view 2nd Floor
        </text>
      </a>
    </svg>
  </g>
</svg>
```

Of course, to truly make use of the new room, you'd have to make some changes to both the ECMAScript and the stylesheet. The script won't allow the user to scroll far

© Copyright IBM Corporation 2003. All rights reserved.

enough to the right to see the new room, and the stylesheet only allows for three rooms, so the rectangle turns up with a default fill color of black, obscuring its contents. Still, these are minor issues that would be resolved in a full-fledged application.

Limitations of XSLT 1.0

One limitation of XSLT 1.0 is lack of standard support for creation of multiple result documents. Individual XSLT processors may provide extension elements to achieve the creation of multiple output documents, so to create SVG images for the first and second floors (and any other floors that the building might have) it's necessary to create multiple XSLT stylesheets or use proprietary extension elements.

Alternatively, an `xsl:parameter` element could be added to the stylesheet to accept a parameter from the command line or the relevant application to specify a floor to be created. Appropriate use of XPath functions, such as the `concat()` and `substring()`, would enable appropriate elements in an extended XML source document to be appropriately processed.

However, an easier approach is desirable. XSLT 2.0 is on the horizon and will soon provide standard techniques to produce multiple result documents. XSLT 2.0 is currently at Working Draft stage at the W3C. It has many new features such as improved grouping functionality, and many new functions available from XPath 2.0, as well as the use of regular expressions. However, in creating SVG in the way you have done here, the facility to create multiple documents is of great interest.

In XSLT 2.0 it is possible to create multiple result documents. However, the transformation is not limited to creating documents of one specific type (SVG for example), but can create multiple documents -- for example, some HTML and some SVG.

Thus with a full data store containing information on each room in a building and an XSLT 2.0 stylesheet, it is possible to create multiple HTML documents, each with an appropriate SVG image.

Section 7. Summary

Tutorial summary

Because of its structure as an XML document, an SVG image can be both created

© Copyright IBM Corporation 2003. All rights reserved.

dynamically and manipulated interactively by a user. This tutorial has demonstrated the use of ECMAScript/JavaScript to enable interaction with an SVG image. It has also discussed dynamically producing an SVG image.

Topics included:

- Producing a simplified floor plan in SVG
- Embedding the SVG in an HTML document and communicating events in the HTML document to the SVG image
- Discussion of the SVG `viewBox` attribute and how to use it to control scrolling of SVG
- Scrolling between rooms using ECMAScript scripting
- Deleting parts of the floor plan under script control
- Adding new parts of the document under script control
- Dynamically generating SVG using XSLT

Using ECMAScript and XSLT with SVG is a powerful and flexible approach to creating interactive SVG dynamically. This tutorial has introduced you to techniques that you can apply in larger projects.

© Copyright IBM Corporation 2003. All rights reserved.

Resources

Learn

- Read the "Introduction to Scalable Vector Graphics" *developerWorks* tutorial (http://www.ibm.com/developerworks/edu/x-dw-xsvg-i.html) by Nicholas Chase (February, 2002).
- Read the Scalable Vector Graphics 1.0 Recommendation from the World Wide Web Consortium (http://www.w3.org/TR/SVG/).
- Read "An Introduction to Scalable Vector Graphics," by J. David Eisenberg (http://www.xml.com/pub/a/2001/03/21/svg.html).
- Read SVG Unleashed, by Andrew Watt, Chris Lilley et al.
- Check out the tutorials and samples in Adobe's SVG Zone (http://www.adobe.com/svg/basics/intro.html).
- Read "Scalable Vector Graphics: The Art is in the Code," by Eddy Traversa (http://www.webreference.com/authoring/languages/svg/intro/).
- For more information on XML in general and SVG in particular, turn to the *developerWorks* XML zone (http://www.ibm.com/developerworks/xml/).
- Find out how you can become an IBM Certified Developer in XML and related technologies (http://www-1.ibm.com/certify/certs/adcdxmlrt.shtml).
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Get your hands on two of the most widely used open-source XSLT processors -- Saxon (http://saxon.sourceforge.net/) and Xalan (http://xml.apache.org/).
- Download the Adobe SVG Viewer (version 3.0) from http://www.adobe.com/svg/viewer/install/main.html.
- Download the Batik SVG viewer and toolkit from the Apache Project at http://xml.apache.org/batik/index.html.
- Build your next development project with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in the discussion forum for this content.

About the author

© Copyright IBM Corporation 2003. All rights reserved.

Andrew Watt

Andrew Watt is an independent consultant with an interest and expertise in XML technologies. He wrote the world's first book on SVG, *Designing SVG Web Graphics* (New Riders), and was lead author for *SVG Unleashed* (Sams Publishing) as well as being an author of several books on XML and Web technologies. He has two all-SVG Web sites at http://www.xmml.com/ and http://www.edititwrite.com/ which demonstrate how SVG can be used to create useful sites. You contact Andrew at SVGDeveloper@aol.com.

© Copyright IBM Corporation 2003. All rights reserved.