Advanced Mouse Event Model for SVG

Keywords: SVG, DOM, Mouse Events, HotDraw, UI Widgets

Mr Musbah Sagar
Research Assistant
Oxford Brookes University
Wheatley Campus
Oxford
Oxford
UK
msagar@brookes.ac.uk

Biography

Musbah Sagar is a Research Assistant at Oxford Brookes University. He has a BSc in Computer Engineering and an MSc in Web Technology from Oxford Brookes University, 2002. His work on his MSc dissertation, "An SVG Browser for XML Languages", was published in the Eurographics UK Chapter Conference, Birmingham, June 2003. He worked as a Web Programmer for Danfoss UK and then as a Research Assistant for Oxford Brookes University on a project called gViz; part of the e-science program. Currently, he is working on an EPSRC funded project, Open Overlays, which involves Grid Computing and Web Technologies. He is particularly interested in Computer Graphics, Games Programming and Web Technologies.

Prof David A. Duce
Professor
Oxford Brookes University
Wheatley Campus
Oxford
UK
daduce@brookes.ac.uk

Biography

David Duce is Professor in Computing at Oxford Brookes University. He has been involved in the development of standards for computer graphics for 20 years, starting with the Graphical Kernel System (GKS). Together with Bob Hopgood and Vincent Quint, he submitted a proposal to W3C entitled Web Schematics, which launched the SVG activity. He has participated in the development of SVG 1.0 and represents Oxford Brookes University on the Advisory Committee of W3C. His research interests include web graphics and mulitiservice systems.

Prof Christopher S. Cooper
Professor
Oxford Brookes University
Wheatley Campus
Oxford
UK

cscooper@brookes.ac.uk

Biography

Chris holds a degree in mathematics and physics from the University of Cambridge and a PhD from the University of London. For the past 25 years his main research interests have been in multiservice networks. From 1974 until 2001 he worked at the Rutherford Appleton Laboratory, England, before gaining a chair at Oxford Brookes University and becoming network strategist at UKERNA, the organization responsible for the UK academic network, JANET. Since 2004, Chris has been semi-retired and lives in the North West Highlands of Scotland, but continues to divide some of his time between teaching and research at Oxford Brookes and consulting for UKERNA on the continuing development of SuperJANET. Chris has worked on projects in the UK, Europe and the USA, and has consulted in the UK and USA. He has contributed to various aspects of development of JANET since its inception in the early 1980s, and to the whole of the SuperJANET programme, begun in the early 1990s and continuing to the present day. He helped to establish the MSc programme at Oxford Brookes in High Speed Networking and Distributed Systems in 1992, on which he has taught ever since. He is currently a co-investigator on the UK EPSRC funded project Open Overlays, in which Oxford Brookes University and Lancaster University are collaborating on the development of dynamically reconfigurable middleware in support of service oriented distributed systems architecture applicable to next generation grid and web services. Chris has served on JISC, EPSRC & NERC committees for research support and funding in the UK. He has reviewed for all of these, the US NSF, and the Royal Dutch Academy of Science, as well as reviewing articles for IEEE Journal on Selected Areas in Communications, Electronics Letters, Computer Communications, and IEE Communications.

Abstract

In this paper we will present our work on a new mouse event model built on top of the DOM Level 3 Event Model **[1]**, written in JavaScript for SVG **[2]**. The work is concerned with defining a higher level of abstraction particularly for handling mouse events. The work described in this paper is believed to ease the process of developing SVG applications in the future (i.e UI widgets) and may perhaps inspire changes to the current SVG mouse event model to make it easier to develop interactive applications for SVG.

Table of Contents

1. Introduction

Event processing code is at the heart of any graphical environment. Applications with Graphical User Interfaces are event-driven. Hence, applications written for SVG are no exception. Originally, SVG has not been intended to host graphical applications. Nevertheless, the SVG 1.2 working draft [3] suggests that SVG is heading towards being a rich environment in which to develop web applications.

SVG uses the DOM Level 3 Event Model to handle mouse events. The DOM Level 3 Event Model is also used by other XML applications (i.e. XHTML [4], MathML [5], XForms [6]). We anticipate that the DOM Level 3 Event Model is 'inadequate' to handle mouse events for SVG in order to develop stable GUI-based SVG applications. The work described in this paper depends fundamentally on a library called svgDraw2D developed in previous work at Oxford Brookes University. svgDraw2D was also used previously for the implementation of the SVG Browser for XML Languages [7] and the skML Visual Editor [8].

2. Mouse Input: Nature of the Problem

This section is intended to set the scene to present the argument for the new SVG mouse event model. We critically analyse three approaches for delivering mouse events to applications of different environments. This background will highlight the shortcomings of the current DOM Level 3 Mouse Event Model.

2.1 Microsoft Windows Operating System

Windowing operating systems make it possible to run several interactive applications simultaneously. Each application displays its content on a rectangular area; a window. Applications receive messages from the operating system for a variety of reasons. Messages could have originated from the operating system (i.e. window-create, window-size), an input device (i.e. mouse-down, mouse-click, key-press) or from the application itself. The operating system dispatches messages to the target window from a message queue. In case of the mouse input device, events occur when the user moves, presses or releases the mouse button. The operating system converts mouse events into messages; and then messages are delivered to the application whose window is positioned under the mouse pointer. Microsoft Windows operating system allows applications to 'capture the mouse'; in that case, the mouse events are delivered to the target window regardless of the position of the mouse pointer. Only one window at any particular time can 'capture the mouse' input. The mouse input can be discarded after the application has finished with it. Microsoft Windows supports four types of mouse messages: BUTTONUP, BUTTONDOWN, BUTTONDBLCLK (for the left, middle and the right mouse buttons) and MOUSEMOVE. Additional messages, MOUSEHOVER and MOUSELEAVE are sent to the specified application upon request (Windows does not voluntarily send those messages by default). Applications use mouse events generated by the operating system with the ability to 'capture the mouse' input to achieve any behaviour they desire.

MSDN library [9] has detailed information about handling mouse input in Microsoft's Windows operating system.

2.2 Java Environment

The Java 1.1 Event Model is based on the concept of "event listener" (similar to the DOM Event Model, see Section 2.3). Objects (or handlers) can register themselves or remove themselves as listeners of any mouse event type. The Java environment made the process of handling mouse events easy for Java developers by introducing a wider set of mouse events (*mousePressed*, *mouseReleased*, *mouseEntered*, *mouseExited*, *mouseClicked*, *mouseMoved* and most importantly *mouseDragged*). All Java mouse events are generated from windows simple mouse messages described earlier. Java hides the complexity of having to 'capture the mouse' once the mouse is out of the application window (to generate the *mouse drag event*). This extra layer of abstraction in handling mouse events has the great advantages of making the development of Java applications relatively straightforward.

2.3 DOM Level 3

In windows, mouse events are sent to a specified application, where in Java, mouse events are sent to a target component. In the DOM Level 3 mouse events are despatched to a specified element in the DOM tree.

The event model of DOM Level 3 is based on the 'event listener' model. The DOM Level 3 Event Model offers a generic event system that defines an Event Flow Architecture and an Event Handling Mechanism. When an event occurs, it propagates from the top-level element of the DOM tree (the root) down to the target element. Event listeners are notified when matching the event types are received. The event then bubbles up back to the root element of the DOM tree. It is allowed in DOM Level 3 to stop an event from propagating or bubbling at any stage of the event despatching process. The event model of DOM Level 3 supports a number of mouse event types; this includes: *click*, *mousedown*, *mouseup*, *mouseover*, *mousemove* and *mouseout*.

3. Critique of the SVG Mouse Event Model

The previous section shows that the DOM Level 3 Mouse Event Model provides low-level mouse event types similar to those provided by the Widows operating system (**Chapter 2**). However, a crucial feature of the Windows operating system was left out of the DOM Level 3 Mouse Event Model and that is the ability to 'capture the mouse'. This feature is vital to maintain the state of the mouse when the mouse pointer gets out of the painted area of an SVG element or the SVG canvas.

3.1 HTML and DOM Level 3

DOM was designed to work with HTML and XML. The DOM provides methods to manipulate HTML documents and HTML-specific events (load, unload, abort, error, select, change, submit, reset, focus, blur, resize, and scroll) to make HTML documents interactive.

Because HTML existed long before the arrival of the DOM, the DOM had to take HTML requirements into consideration; but what about XML applications? Does the DOM match the expectations of other XML languages (i.e. SVG, XForm, etc)? Or is there a room for improvement? The conclusion of this paper is that the DOM needs revision to adapt to the needs of other XML languages as it did with HTML.

3.2 Loss of synchronization of the Mouse State

Applications written for SVG suffer the long known problem of losing the mouse attention once the mouse pointer leaves the painted area of the SVG element - where a particular mouse event is being listened to (captured) - or goes completely out of the SVG canvas. So, what is wrong with the current DOM level 3 Event Model? And why it is not good enough for developing SVG applications?

First, consider this hypothetical scenario. Imagine that a user wants to drag the scrollbar of a TextList widget written for SVG. The user clicks the mouse button on the scrollbar moving box and starts dragging it. But before the mouse button is released the mouse pointer - accidentally - goes out of the boundary of the scrollbar (the reason can be a slow machine or intensive drawing) and the user looses the mouse focus (mouse events stop being delivered to the scrollbar mouse events handler function). The user releases the mouse button while the mouse pointer is out of the scrollbar boundary and then moves the mouse pointer back to be inside the scrollbar region. Because the mouse events handler of the scrollbar stops receiving mouse events once the mouse pointer is out of its boundary (or if the mouse events are being captured on the background it will stop receiving mouse events once the mouse pointer is out of the SVG canvas), the mouse state of the widget becomes out of synchronization with the real mouse state and that could cause all kinds of confusion.

4. Overview of svgDraw2D

The svgDraw2D package is written for SVG in JavaScript to decouple the manipulation of DOM/SVG interfaces from writing graphics applications. The package provides a higher level of abstraction to JavaScript developers to manipulate graphics independently from the DOM API. It also provides capabilities for drawing sophisticated two-dimensional shapes, working with fonts, text and text layout, controlling colours; and it features layering management, styled tool tips and desktop canvas. The work on the svgDraw2D package was inspired by the Java AWT package.

4.1 Main Classes of svgDraw2D

Figure 1 illustrates the inheritance hierarchy diagram of the main svgDraw2D classes. The Foundational Classes (coloured blue) are described below in Section 4.2. Foundational Classes are also used with the svgSwing package (Section 6.1).

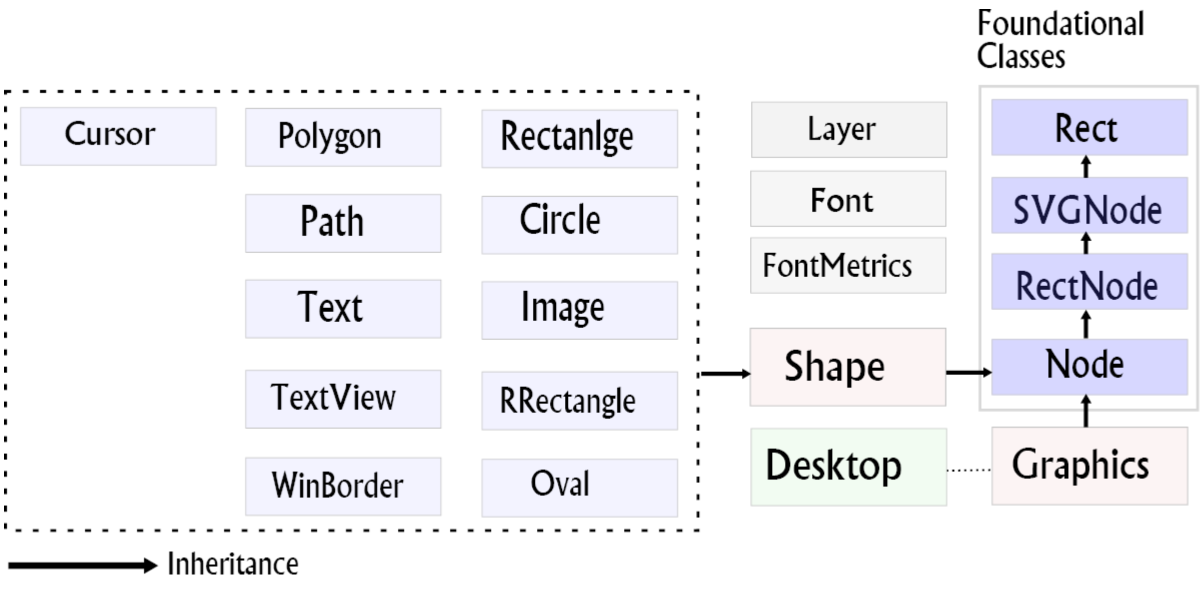


Figure 1: Main classes of svgDraw2D

4.1.1 Graphics Canvas

Graphics acts as a graphical container that can be used to generate SVG drawing primitives. It manages a graphics context by controlling how information is drawn; similar to the Java AWT Graphics class. svgDraw2D

Graphics contains methods for drawing, colouring and font manipulation. Additional methods are supported by Graphics such as flexible clipping, tool tip, cursor manipulation, DOM events handling, and coordinate transformation (scale, rotate and translate). Graphics permits external listeners to handle DOM events that originated from within the Graphics content; and for internal handling of events that originated elsewhere in the SVG document (global events, see **Section 4.1.7**).

4.1.2 Layer

The svgDraw2d package provides a layering feature. Graphics has to be associated to a layer to be valid. A default layer is used if the user has not specified one. Layers have a z-order property that is used for display order.

4.1.3 Shape

Graphics is used for drawing lines, images, rectangles, ovals and other drawing primitives. All drawing methods of Graphics return Shape objects (Rectangle, Text, TextView, Oval, Path, etc). Shape object provides an interface to manipulate the corresponding SVG primitive. Similarly to Graphics, Shape supports tool tip, cursor manipulation, DOM events handling and coordinate transformation.

4.1.4 Cursor

The Cursor object provides an interface to the system cursor. The cursor shape can be changed to any entry of the cursor default list supported by SVG/DOM (i.e. crosshair, move, e-resize, etc) or can be set to a Shape or a Graphics.

4.1.5 Tooltip

Graphics and Shape objects use the *setToolTipTest* method to create a tool tip. The tool tip is displayed automatically when the mouse pointer moves over the content of a Graphics or a Shape objects.

4.1.6 Font and Font Metrics

Font and FontMetrics provide methods and constants for font control.

4.1.7 Desktop

The Desktop is a key component in svgDraw2D. It continuously listens to all DOM events that occur on the SVG document (*mousedown*, *mouseover*, *mouseup*, *mouseout*, *mousemove* and *click*). It maintains a list of listeners that it notifies whenever an event is received. JavaScript objects can act as event listeners by registering themselves with the Desktop. Listeners should provide a callback method that the Desktop invokes to notify the object with an event. Desktop events are regarded as global events within the svgDraw2D package. This object also has a major role in implementing the Advanced Mouse Event handling code.

Section 5.2.3 provides more information about the Desktop.

4.2 Foundational Classes

4.2.1 Rect

Rect represents an axis-aligned rectangle. Graphical entities in svgDraw2D (i.e. Shapes, Graphics objects) are bounded within a rectangular area. The Rect class also provides an interface to change-size, rotate, scale and to translate a graphical entity.

4.2.2 SVGNode

The SVGNode class is used as a wrapper around an SVG node (or DOM tree element). The class ensures access to the SVG node corresponding to any Shape or Graphics in svgDraw2D. SVGNode provides convenient methods to return the actual SVG node, set/get/remove the SVG node attributes, set/get the Id, add/remove DOM event listeners, set visibility and opacity, set/get the cursor, set/get the tool tip text and finally to dispose the SVG node from the SVG document permanently.

4.2.3 RectNode

RectNode is responsible of applying all changes and transformations - made on the rectangular area defined by the Rect class - on the SVG content. For instance, when the user invokes any method on the Rect class, say obj.rotate(45), RectNode will change the properties of the SVG node referenced by the SVGNode class accordingly.

4.2.4 Node

This class is used to maintain a list of internal (local within the inheritance hierarchy of the class) and external listeners for DOM Level 3 mouse events. When a DOM Level 3 mouse event is received the class notifies all listeners of that particular event type. Subclasses of this class could have separate event handlers for any DOM event. For instance, if class A has registered to handle the mousedown event through the *processMouseDown* method and its subclass B wants to handle the same event but without stopping events being delivered to class A; in JavaScript this is not possible because if class B overrides the *processMouseDown* method it will never be able to pass control up to the superclass version of the method (e.g. use of super.processMouseDown ()). The Node class also provides an interface to its internal methods to register listeners for any global mouse events (see **Section 4.1.7**).

4.3 An example using the svgDraw2D Package

An example of a JavaScript code that uses svgDraw2D package is included to illustrate the capabilities of svgDraw2D. **Figure 2** shows the SVG document produced by running the JavaScript code listed in **Example 1**. The result contains four Graphics objects. From left to right, Graphics 1 contains an Oval, a Rectangle and a Round Rectangle shapes, also an Image with a Tooltip associated to it, a Text and a Path shapes. The mouse cursor is set to 'haircross' whenever the mouse pointer hovers over the Rectangle shape. Graphics 2 has the same content and with clipping enabled. The Path shape has been used as a text path. The Graphics object has been scaled down, rotated, translated to the left and the opacity has been set to 75%. Graphics 3 contains -from left to right- a Path, a Polygon, another Path (star-shaped) and a TextView objects. And Graphics 4 has the same content. The start shape was used as a cursor. The first Path shape was scaled down and rotated left 90 degree and the pyramid shape was scaled up. The TextView object was rotated to the right and its content text has been coloured.

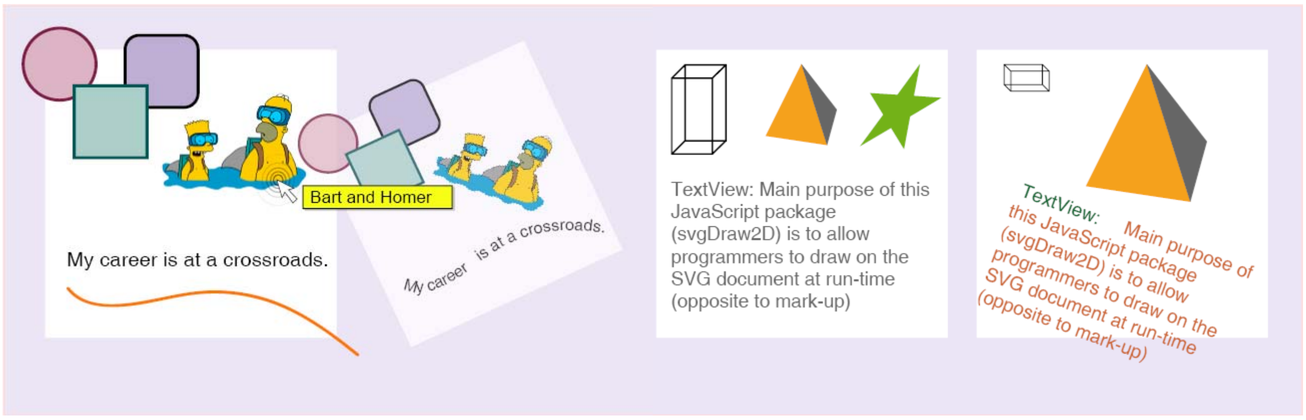

Bart and Homer
My career is at a crossroads.
My career is at a crossroads.
TextView: Main purpose of this JavaScript package (svgDraw2D) is to allow programmers to draw on the SVG document at run-time (opposite to mark-up)
TextView: Main purpose of this JavaScript package (svgDraw2D) is to allow programmers to draw on the SVG document at run-time (opposite to mark-up)

Figure 2: The result of the JavaScript example below (ASV version 6 beta required)

```
// Initialize all the packages [svgDraw2d and svgSwing]
initialise();
// Get the default Desktop
var desktop=ds_getDesktop();
desktop.setColor("#96BCFA");

var dx=20,dy=20,x=dx,y=dy,w=200,h=200, bgc="white";

// (1) Top left Graphics
var   g=new Graphics(x,y,w,h);
g.setBackground(bgc);
g.setColor("#DDB4C3");
g.setStrokeColor("#7F1C46");
g.setStrokeWidth(2);
// Oval shape
g.drawOval(10,10,25,25);
g.setColor("#BEAED4");
g.setStrokeColor("black");
// Round Rectangle shape
g.drawRoundRect(55,-10,50,50,7,7);
g.setColor("#ABCEC8");
g.setStrokeColor("#005758");
// Rectangle shape
var rect=g.drawRect(20,25,50,50);
// Set the cursor to 'crosshair'
rect.setCursor("crosshair");
// Image shape
var image=g.drawImage(80,30,100,68,"Pictures/barthomer.gif");
// Set the tool tip to something
image.setToolTipText("Bart and Simpson");
g.setColor("#FF7700");
g.setStrokeColor("none");
g.setStrokeWidth(2);
// Path shape
g.drawPath(15,165,"M0,1.265c9.208,11.949,25.065,13.006,39");
// Change the default font of Graphics
g.setFont(new Font("Helvetica","bold","14"));
g.setColor("black");
// Text shape
g.drawText(15,150,"My career is at a crossroads.");

// (2) Top right Graphics
g=new Graphics(x+(dx*2)+w,y,w,h);
g.setBackground(bgc);
g.setColor("#DDB4C3");
g.setStrokeColor("#7F1C46");
g.setStrokeWidth(2);
g.drawOval(10,10,25,25);
g.setColor("#BEAED4");
```

```
g.setStrokeColor("black");
g.drawRoundRect(55,-10,50,50,7,7);
g.setColor("#ABCEC8");
g.setStrokeColor("#005758");
g.drawRect(20,25,50,50);
var   image=g.drawImage(80,30,100,68,"Pictures/barthomer.gif");
// Rotate the Image by 45 degrees
image.rotate(45);
// And translate to 125,25
image.translate(125,25);
g.setColor("#FF7700");
g.setStrokeColor("none");
g.setStrokeWidth(2);
var   path=g.drawPath(0,0,"M0,1.265c9.208,11.949,25.065,13.006,39");
g.setFont(new Font("Helvetica","bold","14"));
g.setColor("black");
var   text=g.drawText(15,150,"My career is at a crossroads.");
// Set text path of Text to the Path object
text.setTextPath(path);
// Graphics clip turned on
g.setClipOn();
// Rotate Graphics by -25 degrees
g.rotate(-25);
g.translate(g.x-50,g.y);
// Scale down to by 20%
g.scale(0.8);
// Change opacity to 75%
g.setOpacity(0.75);

// (3) Bottom left Graphics
g=new Graphics(x,y+(dy*2)+h,w,h);
g.setBackground(bgc);
g.setColor("none");
g.setStrokeWidth(1);
g.setStrokeColor("black");
// Path shape (3D shape)
var  _3d=g.drawPath(10,10,"M28.4,62.375V9.725H0.5v52.65h27.9z");
_3d.setToolTipText("3D Object using path primitive");
g.setColor("#73B218");
g.setStrokeColor("none");
// Star shape
path = g.drawPath(140,10,"M22.21,9.353l-8.541,5.158l-1");
path.scale(2.5);
g.setColor("#F5A11C;");
g.setStrokeColor("none");
// Pyramid shape
g.drawPolygon(75,10,[24.589,0,42.152],[0,49.816,56.203]);
g.setColor("#666666;");
g.drawPolygon(75,10,[24.589,48.858   ,42.152],[0,31.614,56.203]);
// TextView shape
g.drawTextView(10,90,180,150,"TextView: Main purpose of");

// (4) Bottom right Graphics:
g=new Graphics(x+(dx*2)+w,y+(dy*2)+h,w,h);
g.setBackground(bgc);
g.setColor("none");
g.setStrokeWidth(1);
g.setStrokeColor("black");
_3d=g.drawPath(50,10,"M28.4,62.375V9.725H0.5v52.65h27");
_3d.scale(0.5);
_3d.rotate(90);
g.setColor("#73B218");
g.setStrokeColor("none");
star=g.drawPath(140,10,"M22.21,9.353l-8.541,5. -3");
g.setColor("#F5A11C;");
g.setStrokeColor("none");
var  p1=g.drawPolygon(75,10,[24.589,0,42.152],[0,49.816,56.203]);
p1.scale(1.7);
g.setColor("#666666;");
```

```
var p2=g.drawPolygon(75,10,[24.589,48.858   ,42.152],[0,31.614,56.203]);
p2.scale(1.7);
var tv=g.drawTextView(10,90,180,150,"");
// Colour text content of TextView
var para=tv.getParagraph();
para.addColoredFlowSpan("TextView:","#166635");
para.addColoredFlowSpan("Main purpose of this JavaScrip","#CC6635");
tv.rotate(20);
tv.translate(30,90);
g.setCursor(new Cursor(0,0,star));
```

Example 1: Source code of the svgDraw2D example

5. The new Mouse Event Model

We have made a decision to adapt Java AWT mouse model as the basis for our design and implementation. The model is able to process raw low-level DOM Level 3 mouse events and produces Java AWT-like events. The following sections will explain how our new mouse events can be captured using mouse listener and mouse motion listener interfaces. We will show how to prepare JavaScript objects to handle the new mouse events (*mousePressed*, *mouseReleased*, *mouseEntered* and *mouseExited*) and mouse motion events (*mouseMoved*, *mouseStartDragging*, *mouseEndDragging*, *mouseDragged*). The example below (**Section 5.3**) emphasizes the ability of the new mouse event model to capture all the above mouse events without the 'out of synchronous' problem explained earlier (**Section 3.2**) even when the mouse pointer moves out of the SVG canvas (goes out of the browser/viewer's window).

The new Mouse Event Model for SVG explained here makes use of the propagation flow phase to handle mouse events.

5.1 Architecture

Figure 3 shows a basic diagram of the architecture of the new mouse event model. EventManager must be the super class of any JavaScript object wishing to handle the new mouse events. With some help from the Desktop (**Section 4.1.7**), the EventManager converts raw low-level DOM Level 3 mouse events into AWT-like mouse events.

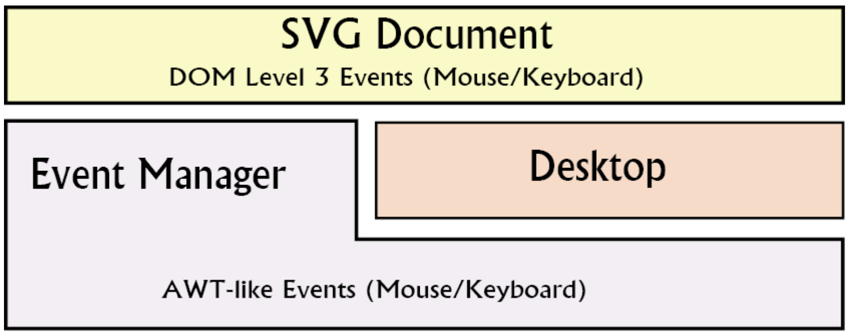

SVG Document
DOM Level 3 Events (Mouse/Keyboard)
Event Manager
Desktop
AWT-like Events (Mouse/Keyboard)

Figure 3: The architecture of the new mouse event model

EventManager has been designed and optimized to handle DOM Level 3 mouse events and generate AWT-like mouse events (mouse and mouse motion events). The Desktop is used to ensure delivery of the necessary mouse events to EventManager when the mouse cursor gets out of the painted area of its corresponding SVG content. In other words, we use the Desktop to simulate 'capture the mouse' mode in Windows operating system (see **Section 5.2.3**).

5.2 Implementation

The implementation of our new mouse event model makes extensive use of the svgDraw2D package. The use of svgDraw2D makes working with SVG easy for a JavaScript programmer. The programmer does not need to think in low-level terms of SVG nodes and attributes as he/she can achieve any desired drawings by using the svgDraw2D classes and APIs. However, making those drawings interactive was outside the capabilities of svgDraw2D. To make interactive documents in SVG, the programmer has to learn about DOM events and has to use DOM APIs. Hence, that was the initial motivation to implement the new mouse event model on top of the DOM Event Model so that it simplifies the process of producing interactive drawings and applications.

Many aspects of our mouse event model are similar to those of Java AWT event model; therefore, the implementation involved re-implementing some of the Java AWT classes and interfaces to JavaScript (i.e. MouseMotionListener, MouseListener, MouseEvent).

5.2.1 Main Classes of the New Mouse Event Model

Figure 4 shows the inheritance hierarchy of the model design which shows the place of the EventManager class among other classes from different packages (blue for svgDraw2D and grey for svgSwing).

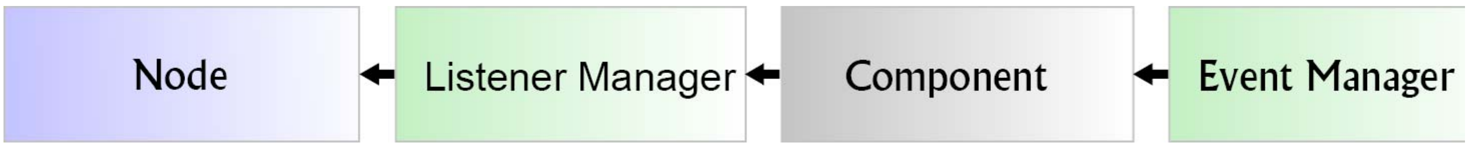


Figure 4: Main classes of the new mouse event model

The order of inheritance may look peculiar to the paper's readers. That is due to the prototype inheritance of JavaScript rather than the class inheritance of Java (more explanations will follow).

5.2.1.1 EventManager

Node class (**Section 4.2.4**) uses the *handleEvent* method to distribute any DOM Level 3 mouse event received to its internal and external event listeners. EventManager deals with DOM Level 3 events in a totally different way. Subclasses of EventManager will not be able to handle DOM events directly as EventManager overrides the *handleEvent* method. But instead, they will be able to handle the new events; AWT-like events. An object of type MouseEvent - that contains the event context information (i.e. x, y, number of mouse clicks, etc) - will be created and passed to the target component when a new AWT-like mouse event is generated.

5.2.1.2 Component

This class belongs to the svgSwing package. See **Section 6.1** for details.

5.2.1.3 ListenerManager

This class is used to maintain a list of internal and external listeners of the AWT-like mouse events. When a new event is received the class notifies all listeners of that particular event type. Listeners have to implement either *MouseMotionListener* interface or *MouseListener* interface or both.

5.2.2 Mouse Events Process Diagram

Figure 6 shows detailed information about the mechanism of the new mouse event model. The code of the new mouse event model runs in two separate modes; *Drag* mode and *Default* mode. **Figure 5** illustrates the state diagram of the two modes. It shows how the new mouse event model code switches between *Drag* mode and *Default* mode. **Figure 5** helps to explain and understand **Figure 6**.

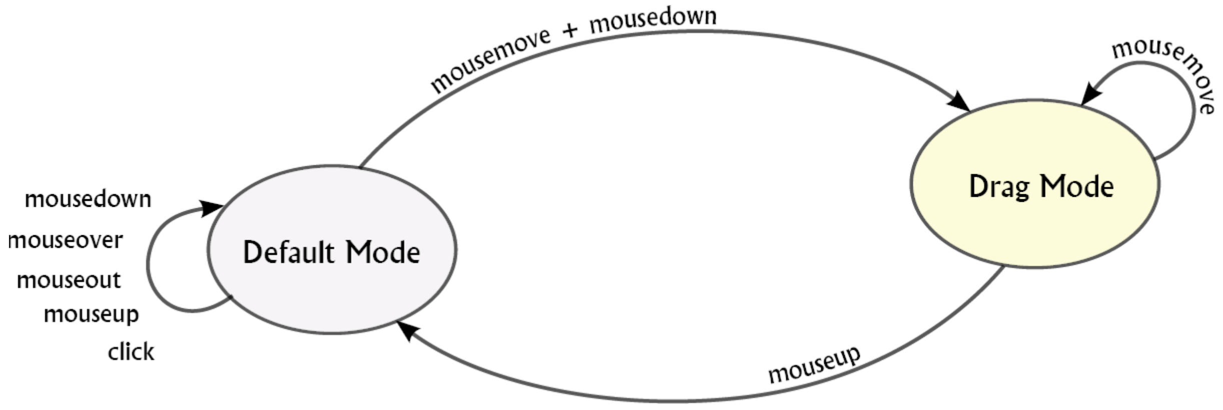


Figure 5: Drag mode state diagram

At the beginning, the mode is set to *Default* mode. For simplicity, assume that EventManager registers itself as a listener to all DOM mouse events available (refer to Node class, **Section 4.2.4**). If *mousedown*, *mouseover*, *mouseup*, *mouseout* or *click* events are received, EventManager generates *mousePressed*, *mouseEntered*, *mouseReleased*, *mouseExited* or *mouseClicked* respectively, and then sets **mouseState** to the type of the received event. If *mousemove* is received and **mouseState** was previously set to *mousedown* then, EventManager generates *mouseStartDragging* event and triggers the Desktop to enter the *Drag* mode.

The Desktop takes control from now onwards over DOM mouse events by stopping the propagation phase of all DOM mouse events. Subsequently, the EventManager will not receive any events. Desktop also stops delivering global mouse events to registered listeners (**Section 4.1.7**). In *Drag* mode, the Desktop listens only to *mouseup* and *mousemove* events. It then routes those events to EventManager. While the Desktop is still in *Drag* mode, EventManager generates *mouseDragged* event for each *mousemove* event received. And it generates *mouseEndDragging* event when it receives *mouseup* event. The EventManager will trigger the Desktop to switch to *Default* mode after it generates the *mouseEndDragging* event.

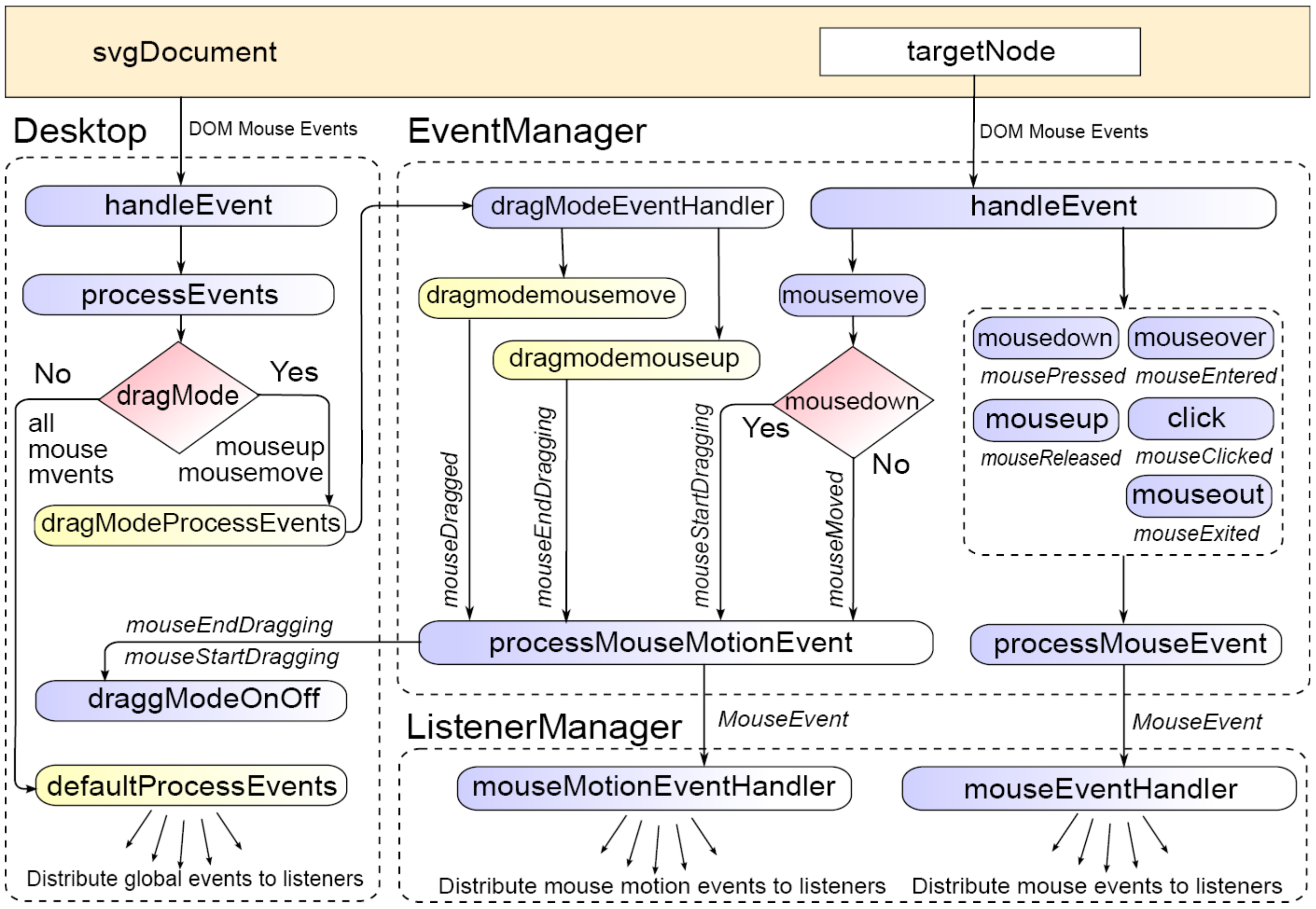


Figure 6: Mouse events process diagram

Events generated by EventManager are sent to the ListenerManager. The ListenerManager distributes all events received to its list of internal and external listeners (**Section 5.2.1.3**).

Having switched to *Default* mode, the Desktop carries on its normal operation (previous to *Drag* mode) and the EventManager starts to handle DOM mouse events of its own again.

5.2.3 Using Desktop to simulate 'Capture the Mouse'

Before we start, assume that the 'pointer-events' attribute of the Desktop is set to *visiblePainted* for simplicity.

From the previous section (**Section 5.2.2**), we have seen how the Desktop was used to simulate the 'capture the mouse' feature. Mouse events were being delivered to the EventManager even after the mouse cursor leaves the painted area of the target element.

In order for the Desktop to be able to capture mouse events, the Desktop listens to all mouse events occuring in the SVG document by registering itself as an event listener of the DOM tree root element. In addition, for mouse events to occur continuously regardless of whether the mouse cursor is on an SVG element or on an empty space, the Desktop has to create an invisible (or visible) SVG content that encompasses the whole client area of the host window (i.e. Rect element). Hence the Desktop will be able to intercept all mouse events that occur inside the client area of the browser window (see **Figure 7** below).

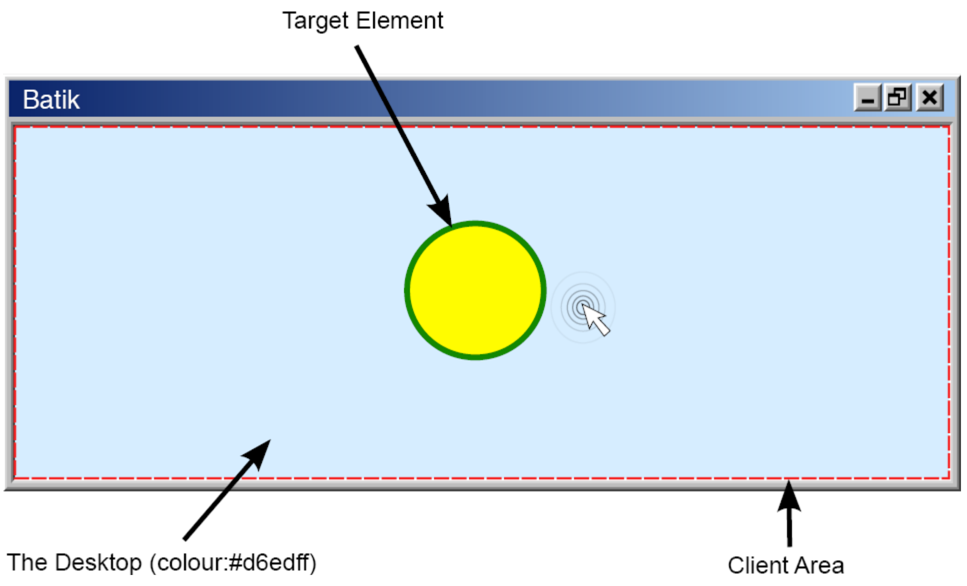


Figure 7: The application window client area filled with the Desktop content

But, what happens if the mouse cursor leaves SVG canvas completely and goes out of the browser window client area?

The answer should be simple; the mouse focus will be lost because SVG or DOM or both do not provide us with the means to 'capture the mouse'.

But fortunately, there is a way to overcome this problem. Our experiments show that SVG implementations, Adobe SVG plug in version 3/6 beta and Batik do 'capture the mouse' if the content of an SVG element -where mouse events are being captured- is larger than the client area. Hence, the SVG implementation ensures that event listeners of that element continue to receive events so long as the mouse is over the element painted area even if that area is not visible from within the client area view (clipped out).

Therefore, we have taken advantage of this feature and expanded the painted area of the Desktop to spread in all directions and grow as large as it possibly can (see **Figure 8**).

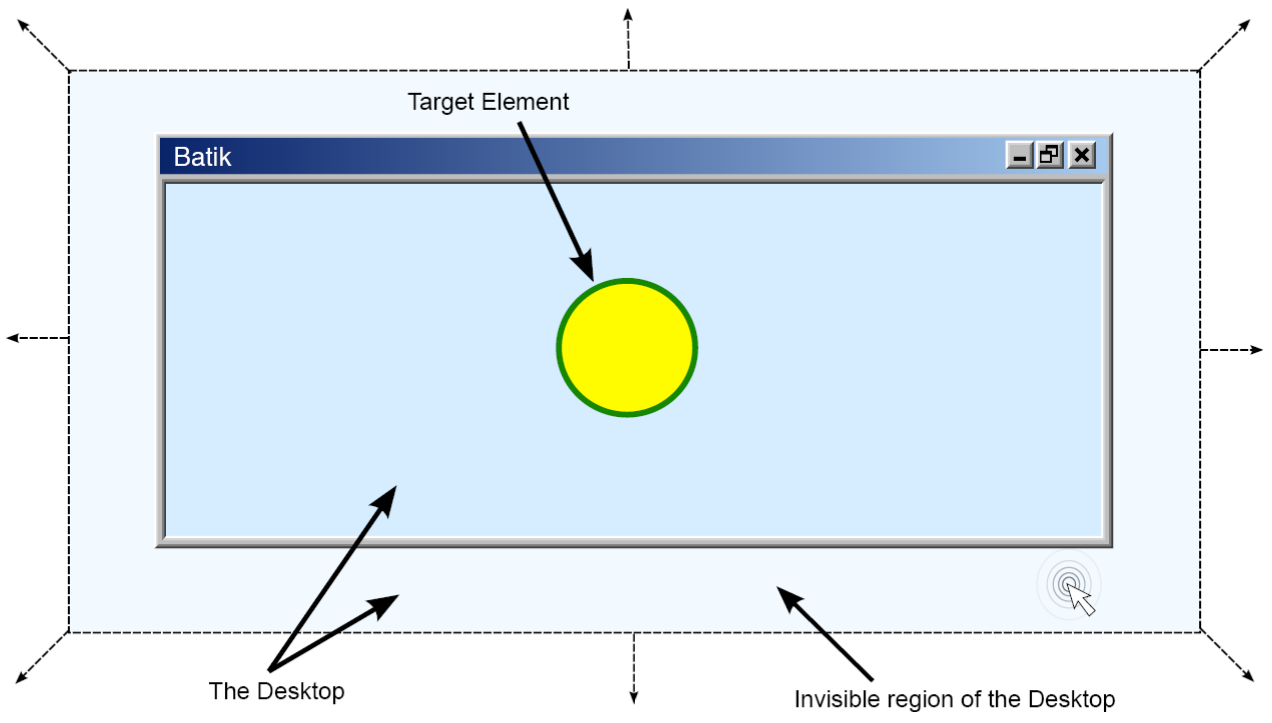


Figure 8: The Desktop content spans beyond the application window client area

This approach guarantees that the mouse focus can never be lost. Hence, our implementation was based on these findings.

5.3 Dragable Shapes

In this section, we will present an example to illustrate the minimal JavaScript code required to handle the new AWT-like events. As **Figure 9** shows, the code below creates four types of shapes, Rectangle, Round Rectangle, Oval and Image. After that, it creates instances of the class *Dragable*. The shapes are passed as an argument of the *Dragable* class constructor.

Class *Dragable* enables mouse motion events and registers itself as a listener to those events. It then overrides the default methods to handle *mouseStartDragging*, *mouseDragged* and *mouseEndDragging* events.

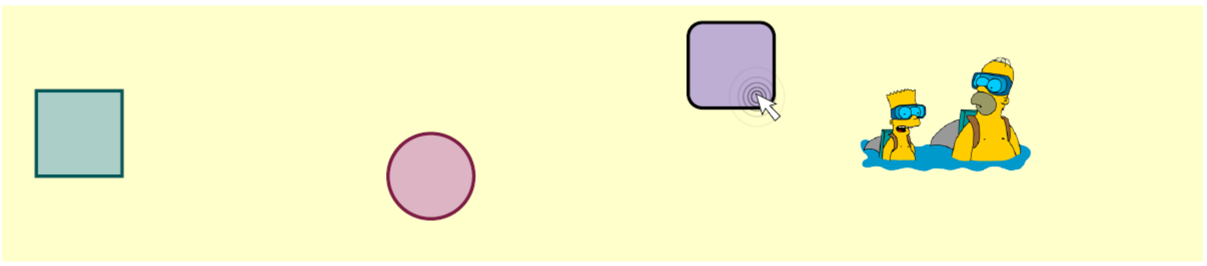


Figure 9: Four dragable shapes: Rectangle, Oval, Round Rectangle and an Image

Notice how the coordinate x and y in the code below (**Example 2**) are obtained from the MouseEvent object. Coordinates x and y are local to the SVG content (x, y range from 0, 0 to width and height).

This takes into account the effect of Pan and Zoom transformation applied to the browser. (for example ctrl/alt keys in Adobe plug in). The application programmer is not aware of any transformation of this kind.

```
//========================
// Class: Dragable
//=======================
Dragable.prototype= new EventManager(); // [ superclass: EventManager]
function Dragable(/* Shape */ shape){
this.shape=shape;
this.initSVGNode(shape.getNode());
this.enableMouseMotionListener(false);
this.addMouseMotionListener(this);
}
//********************
// mouseStartDragging
//********************
Dragable.prototype.mouseStartDragging = function(/* MouseEvent */ event){
this.setCursor("move");
this.tempX=event.getX();
this.tempY=event.getY();
}
//********************
// mouseDragged
//********************
```

```
Dragable.prototype.mouseDragged = function(/* MouseEvent */ event){
var x=event.getX();
var y=event.getY();
this.shape.moveBy(x-this.tempX,y-this.tempY)
}
//********************
// mouseEndDragging
//********************
Dragable.prototype.mouseEndDragging = function(/* MouseEvent */ event){
this.setCursor("default");
}

//=======================
// Main:
//======================
function main(){
// Initilized all the packages [SvgDraw2d and SvgSwing and set viewerMode to either ASV or
initialise();
var         desktop = ds_getDesktop(); // Get the default Desktop
desktop.setColor("#96BCFA");

var   g=new Graphics(50,50,600,150);
g.setBackground("white");
g.setClipOn();
g.setColor("#DDB4C3");
g.setStrokeColor("#7F1C46");
g.setStrokeWidth(2);
var oval  =g.drawOval(130,100,25,25);
g.setColor("#BEAED4");
g.setStrokeColor("black");
var rrect = g.drawRoundRect(150,10,50,50,7,7);
g.setColor("#ABCEC8");
g.setStrokeColor("#005758");
var rect  = g.drawRect(20,50,50,50);
var image = g.drawImage(200,30,100,68,"Pictures/barthomer.gif");

new Dragable(oval);
new Dragable(rrect);
new Dragable(rect);
new Dragable(image);
}
```

Example 2: Dragable Shapes Source Code

6. Case Studies

In this section we will present two projects that we have developed on top of svgDraw2D and our new mouse event model. The first application is a GUI toolkit features a collection of classical GUI widgets. The second application is a JavaScript port of HotDraw [10][11].

6.1 svgSwing

Swing was the unofficial code name of the project that developed Java Foundational Classes (JFC) for Java. This inspired us to use 'svgSwing' as the name of this package; and the prefix 'svg' refers to that it is written for an SVG environment. This package is still in its early stages of development, although - to this point - the approach is satisfactory.

6.1.1 Implementation

svgSwing offers classes to help in developing GUI components. There are two main sets of components in svgSwing: Top Level Containers (e.g. Panel and Window) and Basic Components (e.g. Button and ScrollBox).

Figure 10 shows the main classes of svgSwing package to date. Classes of svgSwing are coloured grey. All classes inherit directly or indirectly from the Component class.

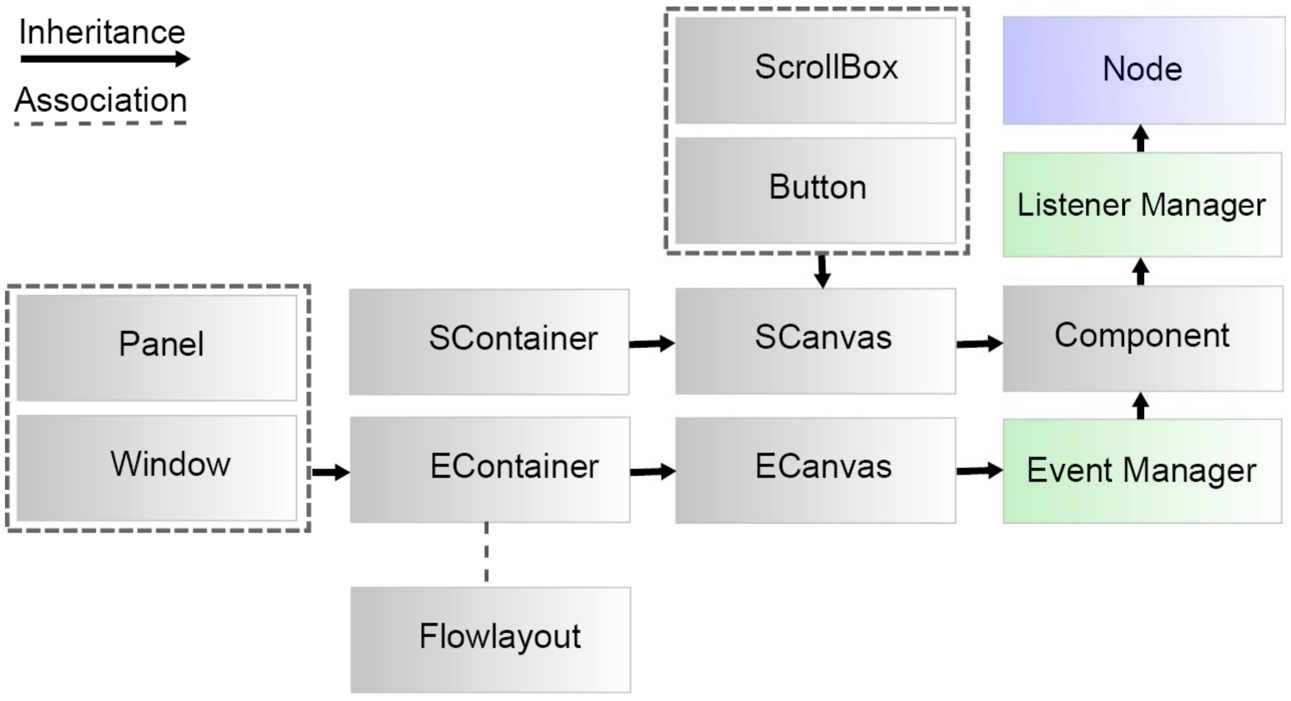


Figure 10: Inheritance hierarchy of svgSwing main classes

Components may register themselves with the ListenerManager to handle AWT-like events. Figure 11 shows that there are two sources of AWT-events for svgSwing components, direct and indirect sources.

Top Level Components inherit indirectly from the Component class through EventManager, hence they support direct AWT-like events. The Window class for instance may enable AWT-events and choose to listen to them through *Mouse Motion Interface* or *Mouse Interface*. The AWT-events received by the Window class have been generated by the EventManager and delivered by ListenerManager.

Button class handles AWT-like events generated from an indirect source; EContainer. EContainer is the root of a tree of Basic Components. AWT-like events received by EContainer are processed and then routed down the containment hierarchy to a target component (similar to DOM).

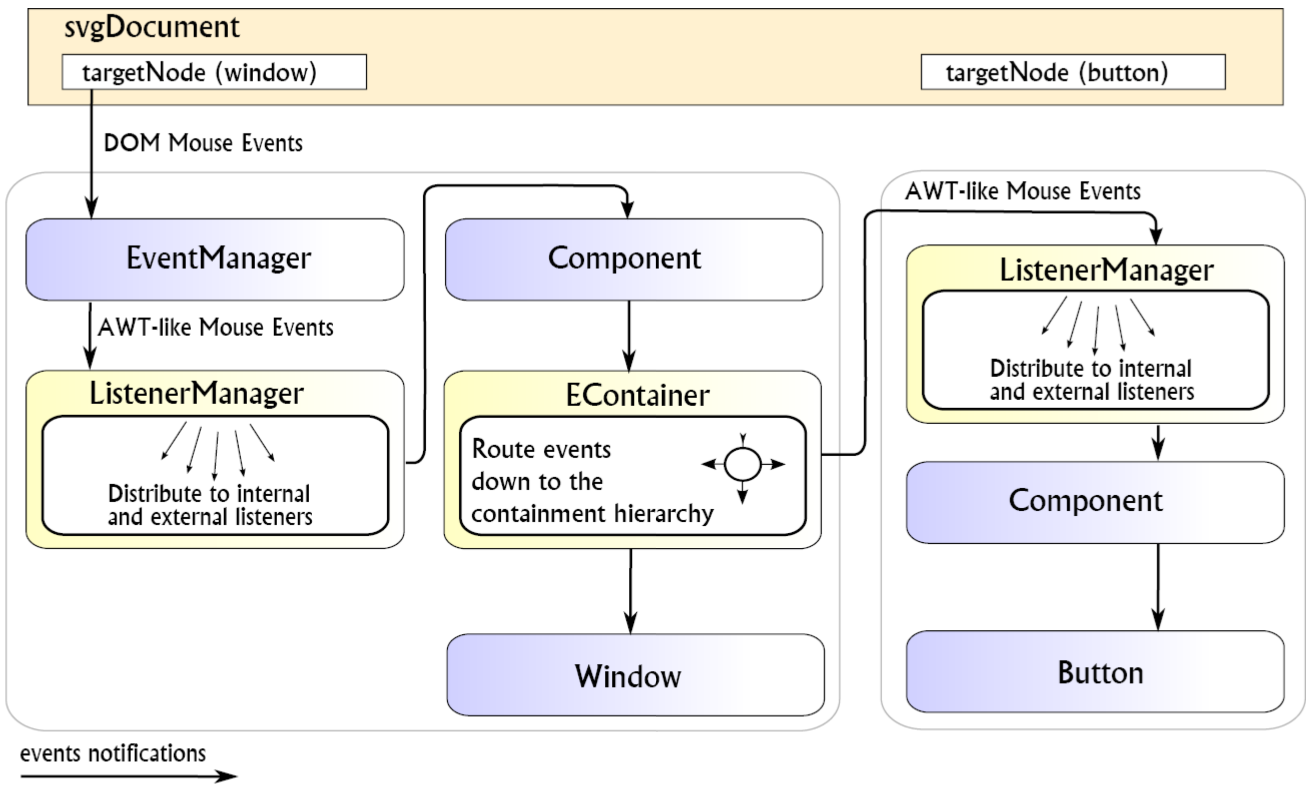


Figure 11: Mouse event flow within svgSwing components

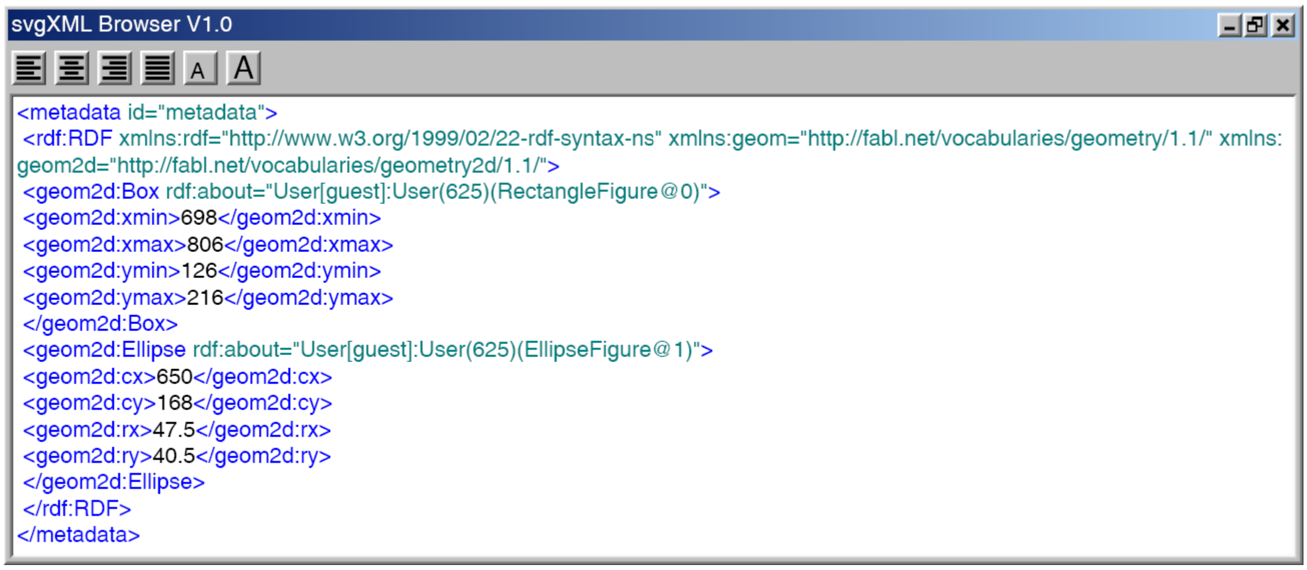
The svgSwing Package has adapted a look-and-feel similar to that of MS-Windows. This is temporary. We hope to improve the package to support a pluggable look-and-feel model in the future.

6.1.2 svgSwing Events

The svgSwing package divides events into two categories: *low-level* events (AWT-like mouse and keyboard events) and *semantic* events (e.g. buttonClicked, windowClosed). The Component class maintains a list of listeners to the component semantic events.

6.1.3 svgXML Browser

Figure 12 shows an application written with svgSwing. The application is an svgXML Browser that simply lays out an XML fragment into the client area of a Window object.



```
<metadata id="metadata">
 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns" xmlns:geom="http://fabl.net/vocabularies/geometry/1.1/" xmlns:
geom2d="http://fabl.net/vocabularies/geometry2d/1.1/">
 <geom2d:Box rdf:about="User[guest]:User(625)(RectangleFigure@0)">
 <geom2d:xmin>698</geom2d:xmin>
 <geom2d:xmax>806</geom2d:xmax>
 <geom2d:ymin>126</geom2d:ymin>
 <geom2d:ymax>216</geom2d:ymax>
 </geom2d:Box>
 <geom2d:Ellipse rdf:about="User[guest]:User(625)(EllipseFigure@1)">
 <geom2d:cx>650</geom2d:cx>
 <geom2d:cy>168</geom2d:cy>
 <geom2d:rx>47.5</geom2d:rx>
 <geom2d:ry>40.5</geom2d:ry>
 </geom2d:Ellipse>
 </rdf:RDF>
</metadata>
```

Figure 12: An svgSwing application: svgXML Browser (ASV version 6 beta required)

6.1.4 svgSwing Calculator

This is another example with the source code to illustrate how svgSwing GUI components are used.

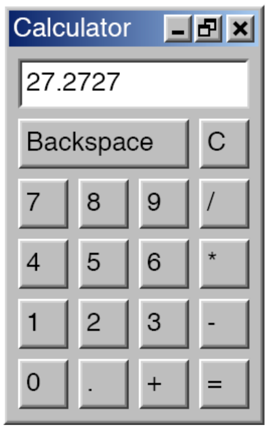


Figure 13: svgSwing calculator

```
//===============================================================
//
// Main : Main program
//
//===============================================================
function main(){
initialise();
var cal=new Calculator();
}
//==============================
// Class: Calculator
//==============================
function Calculator(){

this.result=0;
this.temp=0;
this.operation=null;

var win=new Window(250,200,130,210,"Calculator",null,true);
```

```
win.setLayout(new FlowLayout(CENTER));

var but1=new Button(0,0,85,25,"Key_backspace","Backspace");
var but2=new Button(0,0,25,25,"Key_clear","C");
var but3=new Button(0,0,25,25,"Key_7","7");
var but4=new Button(0,0,25,25,"Key_8","8");
var but5=new Button(0,0,25,25,"Key_9","9");
var but6=new Button(0,0,25,25,"Key_div","/");
var but7=new Button(0,0,25,25,"Key_4","4");
var but8=new Button(0,0,25,25,"Key_5","5");
var but9=new Button(0,0,25,25,"Key_6","6");
var but10=new Button(0,0,25,25,"Key_mult","*");
var but11=new Button(0,0,25,25,"Key_1","1");
var but12=new Button(0,0,25,25,"Key_2","2");
var but13=new Button(0,0,25,25,"Key_3","3");
var but14=new Button(0,0,25,25,"Key_minus","-");
var but15=new Button(0,0,25,25,"Key_0","0");
var but16=new Button(0,0,25,25,"Key_dot",".");
var but17=new Button(0,0,25,25,"Key_plus","+");
var but18=new Button(0,0,25,25,"Key_equal","=");

this.screen = new TextBox(0,0,115,25,"Sum","0");
win.add(this.screen);
win.add(but1); win.add(but2); win.add(but3); win.add(but4);  win.add(but5); win.add(but6);
win.add(but8);  win.add(but9); win.add(but10); win.add(but11); win.add(but12);win.add(but13
win.add(but15); win.add(but16); win.add(but17); win.add(but18);

but1.addActionListener(this);  but2.addActionListener(this); but3.addActionListener(this);
but5.addActionListener(this);  but6.addActionListener(this); but7.addActionListener(this);
but9.addActionListener(this); but10.addActionListener(this); but11.addActionListener(this);
but13.addActionListener(this); but14.addActionListener(this); but15.addActionListener(this)
but17.addActionListener(this); but18.addActionListener(this);
win.draw();
}
//*************
// performOperation
//*************
Calculator.prototype.performOperation = function(nextOperation){
if(this.operation!=null)
 switch(this.operation){
        case "+":this.result=this.temp+this.result; break;
        case "-":this.result=this.temp-this.result; break;
        case "*":this.result=this.temp*this.result; break;
        case "/":this.result=this.temp/this.result; break;
  }
if(nextOperation!=undefined){
 this.temp=this.result;
 this.operation=nextOperation;
 this.clear();
 } else
 this.operation=null;

}
//*************
// reset
//*************
Calculator.prototype.reset = function(){
  this.temp=0;
        this.operation=null;
  this.clear();
}
//*************
// clear
//*************
Calculator.prototype.clear = function(){
  this.result=0;
  this.screen.setText(this.result);
}
//*************
```

```
// actionPerformed
//*************
Calculator.prototype.actionPerformed = function(/* ActionEvent */ e){

var name=e.source.name;
var comm=e.getActionCommand();

 if(comm=="buttonClicked"){
 if(name=="Key_equal"){
   this.performOperation();
         this.screen.setText(this.result);
 } else
 if(name=="Key_clear")this.reset();
 else
 if(name=="Key_backspace"){
 var numStr = this.screen.getText();
  if(numStr.length>1){
   var number = parseInt(numStr.substring(0,numStr.length-1));
   if(number.toString().length<10){
     this.result=number;
     this.screen.setText(number);
    }
  } else this.clear();
 } else
 if(name=="Key_plus") this.performOperation("+");
 else
 if(name=="Key_minus") this.performOperation("-");
 else
 if(name=="Key_mult") this.performOperation("*");
 else
 if(name=="Key_div") this.performOperation("/");
 else
 if(name=="Key_dot");
 else {
   var numStr= this.screen.getText()+e.source.caption;
   var number = parseInt(numStr,10);
   if(number.toString().length<10){
    this.result=number;
    this.screen.setText(number);
   }
 }
}
}
```

Example 3: svgSwing calculator Source Code

6.2 Hotdraw

Hotdraw is a framework for developing 2-dimensional structured drawing editors. It is used to develop editors for various 2D drawings such as UML tools, schematic diagrams, blueprints and program design. Elements of drawings can be treated interdentally but they can also have constraints between them. Editing drawings is achieved using the mouse. The user selects the element of interest with the mouse and makes changes to it (resize, translate, etc). More figures and tools can be easily added to the framework.

HotDraw is being used to develop The Open Overlays Collaborative Workspace (svgCWE) [12], part of the Open Overlays project [13].

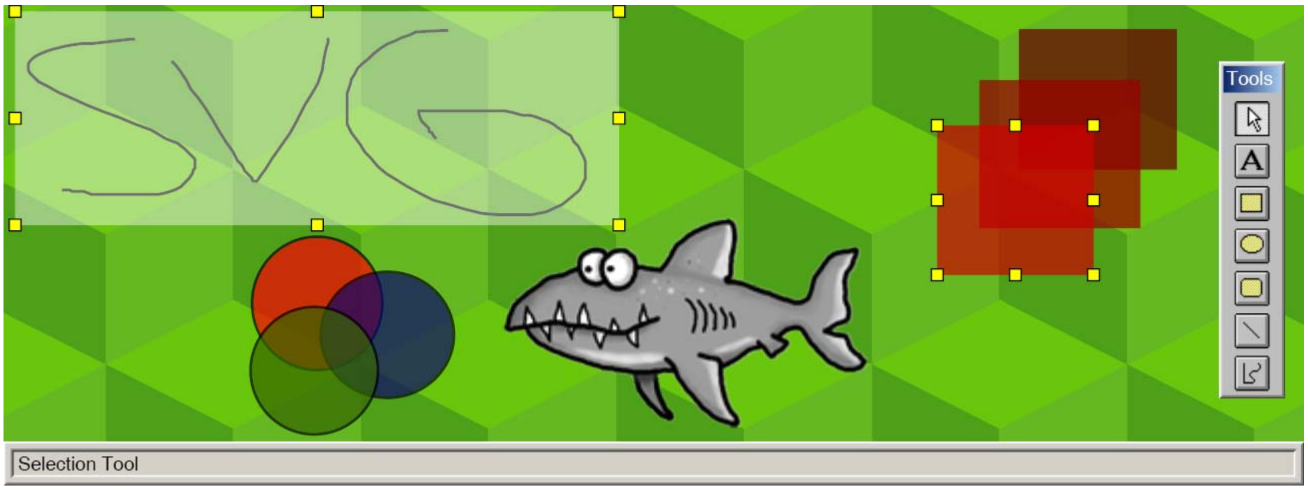


Figure 14: Screen shot of the JavaScript HotDraw in action

JavaScript HotDraw is a port of JHotDraw version 5.1 [11]. The implementation of JavaScript HotDraw has showed us the possibilities of porting Java applications to SVG. We anticipate that this will open the doors to recruit ideas and port applications from Java to SVG. The impact of doing that might change the way we conceive SVG altogether.

6.2.1 Overcoming Method Overloading Shortcomings of JavaScript

Most OO programming languages allow classes to have several methods with the same name but different lists of arguments; this is called method overloading. One of the shortcomings of JavaScript has been that this feature is not supported. Method overloading is obviously a normal practice for Java developers and it is rare that programmers do not use this technique. The developers of HotDraw, for example, have used method overloading quite heavily and there was no way to avoid this problem in the process of porting JHotDraw to JavaScript.

We have come up with a simple solution to this (see the source code below). We included this section in the paper because we think that this is a common problem that will need to be resolved when porting Java applications.

```
//*************
// Finds a top level Figure. Use this call for hit detection that should not descend into t
//
// Forms:
// ======
// (1) findFigure(Rectangle r)
// (2) findFigure(int x,int y)
//*************
/* Figure */ CompositeFigure.prototype.findFigure = function(/* int or Rectangle /* x, /*
/* FigureEnumeration */ var k = this.figuresReverse();
 // Form (1)
 if( x instanceof gRectangle){
 var r=x;
   while (k.hasMoreElements()) {
  /* Figure */ var figure = k.nextFigure();
  /* Rectangle */ var fr = figure.displayBox();
       if (r.intersects(fr)) return figure;
   }
  return null;
 }
 // Form (2)
  while (k.hasMoreElements()) {
```

```
  /* Figure */ var figure = k.nextFigure();
  if (figure.containsPoint(x, y)) return figure;
    }
return null;
}
```

Example 4: Method Overloading: This is a method of the CompositeFigure class of JHotDraw v5.1.

7. Related Work

The subject of handling mouse events is an area that almost everyone in the SVG community has experienced. SVG developers have been involved heavily in developing SVG interactive applications. One thing that everyone recognises and agrees upon is the lack of mouse drag event support in SVG. As we have explained earlier, SVG and DOM do not address this issue directly instead both ignore it completely. Nevertheless implementations of SVG demonstrate understanding of the importance of such a feature in SVG environments. ASV version 6 **[14]** for instance provides two new public methods for it plug in to handle *mouse drag event*; those methods are: *startSimpleDnD* (start simple drag and drop) and *captureMouse*. Documentation for these two new methods was not available to us.

Also, many from within the SVG community have used different approaches to develop methods for handing *mouse drag event*. All have succeeded in doing so but most of them introduced even more problems. For instance, the mouse loses focus, complex code structure, lack of flexibility **[15]**,**[16]**,**[17]**. This paper will not describe previous attempts to handle the *mouse drag event* in detail

The following points explain why the approach we have taken to resolve this issue is regarded as comprehensive and flexible to use compared to other approaches that we have seen:

1. We do not treat *mouse drag event* in isolation form other mouse events; we consider it to be one of them.
2. We approach the problem from a different prospective, equipped with an understanding of how mouse input is being handled in different environments (Windows OS, Java and C++) and reflect that understanding on SVG as a potential environment to develop applications.
3. Our approach can be used as an advanced alternative to the current DOM Level 3 Mouse Event support for SVG featuring the following advantages:
   a. Extended mouse events (*mouseDragged*, *mouseStartDragging*, *mouseEndDragging*, etc).
   b. Mouse Event Listeners for the new model.
   c. Support of multiple internal handlers.
4. We have provided case studies to show that our new mouse event model (with svgDraw2D and svgSwing) can form the foundation for porting Java applications to SVG which could have an impact on future SVG Recommendations (see **Chapter 8**).
5. We also provide use cases of the new mouse event model to illustrate its merits. For instance we explained svgSwing, an implementation for a real SVG-based GUI toolkit that looks, feels and behaves as one would expect.

8. Future Directions

We will continue to improve and enhance the new Mouse Event Model described in this paper. Also, more work is needed on svgSwing package and JavaScript HotDraw.

8.1 Recommendations

1. We strongly recommend introducing a mechanism to 'capture the mouse' for SVG environments or to introduce *mouse drag event* to the current mouse event support.
2. We suggest that mouse drag event should be delivered to the target element even if the mouse cursor goes out of the SVG canvas.
3. Mouse event listeners attached to the SVG document (DOM root) should deliver events whether the background of the SVG canvas is painted or not.

Acknowledgements

This work has been carried out as part of the Open Overlays project. Open Overlays project is funded by EPSRC Fundamental Computer Science for e-Science Program.

Bibliography

[1]
Document Object Model (DOM). Available at: http://www.w3.org/DOM/

[2]
Scalable Vector Graphics (SVG) 1.1. Available at: http://www.w3.org/Graphics/SVG/

[3]
Scalable Vector Graphics (SVG) 1.2 Working Draft. Available at: http://www.w3.org/TR/2004/WD-SVG12-20041027/

[4]
Extensible HyperText Markup Language (XHTML 1.0). Available at: http://www.w3.org/TR/xhtml1

[5]
Mathematical Markup Language (MathML) Version 2.0 (Second Edition). Available at: http://www.w3.org/TR/2003/REC-MathML2-20031021

[6]
XForms 1.0. Available at: http://www.w3.org/TR/2003/REC-xforms-20031014

[7]
An SVG Browser for XML Languages,Musbah Sagar, Proceedings of the Eurographics UK Chapter Conference, Birmingham, June 3-5, 2003. Published by IEEE Computer Science Press.

[8]
skML a Markup Language for Distributed Collaborative Visualization, David Duce, Musbah Sagar, to appear in EGUK Theory and Practice of Computer Graphics Conference 2005.

[9]
MSN Library, Mouse Input. Available at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/userinput/mouseinput.asp

[10]

Original Hotdraw for VisualWorks Smalltalk. Avaialbe at: http://st-www.cs.uiuc.edu/users/brant/HotDraw/HotDraw.html

[11]
Java Graphical Editing Framework (JHotDraw). Available at: http://www.jhotdraw.org/

[12]
The Open Overlays Collaborative Workspace, Duce, D., Cooper, C., Li, W., Sagar, M., Blair, G.S., Coulson, G., Grace, P., 4th Annual Conference on Scalable Vector Graphics, Enschede, the Netherlands, August 2005.

[13]
Open Overlays Project. Available at: http://www.comp.lancs.ac.uk/computing/research/mpg/projects/openoverlays/

[14]
Implementation White Paper, Adobe SVG Viewer version 6.0 pre-alpha (Build 38363). Available with the SVG plug-in at: http://www.adobe.com/svg/viewer/install/beta.html

[15]
Doing That Drag Thang, Antoine Quint. Available at: http://www.xml.com/pub/a/2002/02/27/drag.html (works with Adobe SVG Plug in version 3. Note: Loss of mouse focus).

[16]
Adobe SVG Demos, Draggable Element. Available at: http://www.adobe.com/svg/demos/drag.html (Node: loss of mouse focus if the element being dragged has lower z-order from other elements on the SVG document).

[17]
Drag-n-Drop Icons in SVG. Available at: http://www.universalmap.com/Work/smartmaps/howitworks/Examples/drag_n_drop/drag_n_drop.html (Note: loss of mouse focus if the mouse movements are distant)