

# SceneBeans: A Component-Based Animation Framework for Java

Nat Pryce and Jeff Magee  
Department of Computing,  
Imperial College.  
{np2,jnm}@doc.ic.ac.uk

*DRAFT VERSION*

## 1. Abstract

This paper presents SceneBeans, a framework for building two-dimensional, interactive animations from Java Beans components. A SceneBeans animation is defined as “scene graph”, a directed acyclic graph of Java Bean components that define a scene in terms of compositions and transformations of geometric shapes. Animation is achieved by modifying the Java Bean properties of the scene graph nodes between each frame. Animation behaviours are also packaged as Java Bean components, allowing the programmer to take a compositional approach to animating graphics.

Animations can be defined in XML. The SceneBeans parser interprets an XML document as a set of configuration commands defining instantiations of, and bindings between, dynamically loaded Java Bean components. Thus the file-format is open ended: new visual and behavioural components can be introduced on a per-application or per-document basis.

Finally, we present an example application of the framework where animations are used to visualise formal models of concurrent programs.

**Keywords:** animation, framework, component, Java, Java Beans, XML.

## 2. Introduction

Interactive, animated graphics are a powerful way of displaying complex information to users in a way that aids understanding. Current graphics toolkits and APIs do not provide support for animation, usually only providing functions to share display real-estate between programs, render primitive shapes to the display and collect user input events. Developers, therefore, are hindered from using animation in their applications by the need to implement most of the mechanisms required to define, compose and simulate and render animations.

SceneBeans is an object-oriented framework for building and controlling animated graphics. It aims to remove the drudgery of displaying animations, allowing programmers to concentrate on what is being animated, rather than on how that animation is played back to the user. SceneBeans is based upon industry and Internet standards, such as Java Beans [2] and XML [4]. Its component-based architecture allows application developers to easily extend the framework with domain-specific visual and behavioural components.

## 3. The Scene Beans Framework

### 3.1. Structured Graphics

The graphics drawn by SceneBeans are defined by a “scene graph”, a directed, acyclic graph (DAG), each node of which is a Java Bean that encapsulates a simple drawing command. Leaves of the graph draw graphical primitives such as ellipses, polygons, images or text. Intermediate nodes combine the primitive shapes into complex scenes by composing multiple scene graphs or modifying the way that a scene graph is rendered. For anything but the most trivial scene, the root of the graph is a composite node.

All nodes of the scene graph implement the `SceneGraph` interface, which provides operations for drawing the graph onto a graphics context, querying or setting a flag indicating if the node is animated that is used to optimise the rendering process, and passing a *Visitor* [3], of type `SceneGraphProcessor`, to the node. The `SceneGraphProcessor` uses double-dispatch to provide algorithms that walk over the scene graph with type information about the nodes that they are visiting.

Scene graphs are composed using `Composite` nodes. There are currently six types of composite node: `Layered`, that draws its sub-graphs one above the other; `Switch`, that draws one of its sub-graphs at a time selected by a bean property; and four node types that perform constructive area geometry operations on their sub-graphs: `Union`, `Intersection`, `Subtraction` and `Difference`.

There are two types of node that modify the way a graph is drawn: `Style` nodes that set the fill and line properties used to draw their subgraphs and `Transform` nodes that apply an affine transformation to their subgraphs. To facilitate the composition of animated transformations, there are different types of `Transform` node for each primitive transformation, `Translate`, `Rotate`, `Scale` and `Shear`, rather than a single type of transform node parameterised by a transformation matrix.

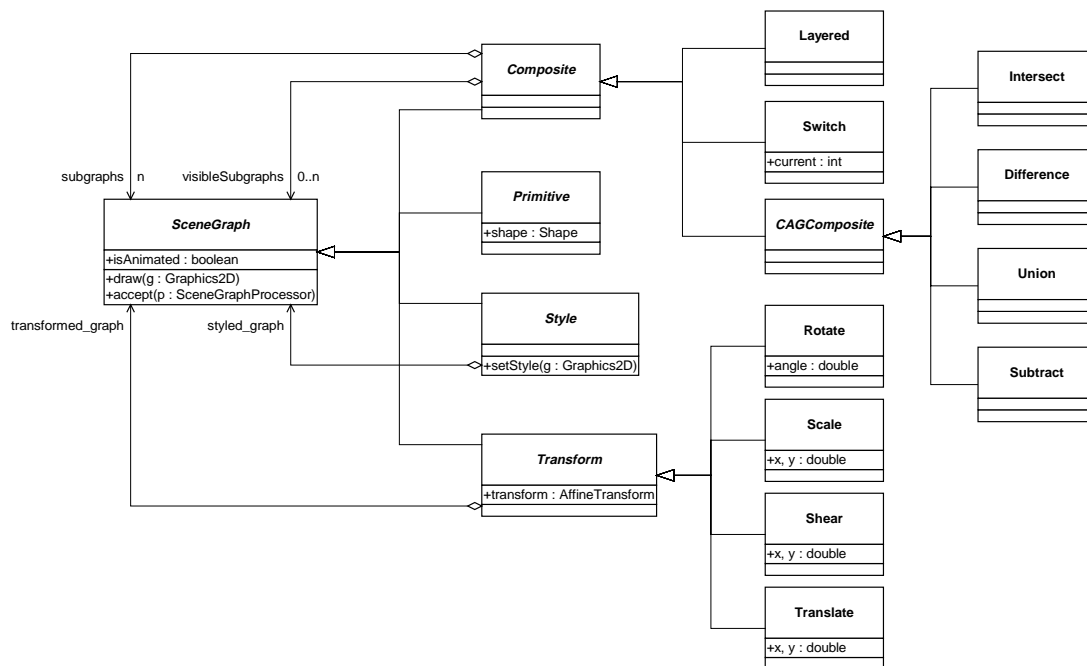


Figure 1. Taxonomy of scene graph nodes (subclasses of Primitive and Style are not shown)

The SceneBeans framework draws the distinction between `Transform` and `Style` nodes to support calculations that require knowledge of the geometry of the scene graph. Such calculations include “picking” – finding the path from the root of the graph to the primitive that contains a point – and dirty rectangle caching when updating the screen. A `Transform` node affects the geometry of its subgraphs and so its transformation must be taken into account when performing such calculations; a `Style` node, on the other hand, only changes the appearance of its subgraph and its function can be ignored by geometric calculations.

### 3.2. Animating Scene Graphs

Being Java Beans, scene graph nodes expose modifiable properties at their interface. Specifically to SceneBeans, these properties control their appearance when rendered. For example, a `Circle` bean exposes a `radius` property that specifies the radius of the circle drawn, an `HSVColor` bean exposes a `color` property that specifies the colour used to draw its subgraph and `hue`, `saturation`, `value` and `alpha` properties that specify the components of the colour, and a `Translate` bean exposes `x` and `y` properties that specify the translation to be applied to its subgraph.

A program can therefore display an animation by modifying properties of beans in the scene graph before drawing each frame. However, programming animations by explicitly locating nodes in the scene graph and modifying their properties is overly complex, and not really much of an improvement over existing methods of animation. Therefore SceneBeans provides components that automate the animation of scene graphs, termed *behaviours* and *activities*.

A “behaviour” is a bean that encapsulates a time-varying value and periodically announces an event containing the current value. Behaviours are typed: a `DoubleBehaviour` encapsulates a double-precision real number value, a `PointBehaviour` a point value, a `ColorBehaviour` a colour value and so on. A node in the scene graph is animated by registering an event listener with a behaviour that routes announced changes of the behaviour’s value to a property of the bean.

Behaviours can encapsulate any time varying value, such as the location of the mouse cursor or size of a window. However, most behaviours used within animations periodically calculate and announce the value of a time varying function. These behaviours are implemented as “activities”, objects that implement the `Activity` interface. This interface provides a method, `performActivity(t)`, that is invoked every frame of animation to perform an action based on the duration of that frame. SceneBeans treats the concepts of activity and behaviour as orthogonal: behaviours feed values into a scene graph and activities are simulated every frame, but a behaviour need not be an activity, and *vice versa*. In this paper we use the term “active behaviour” to refer to a behaviour that is also an `Activity`.

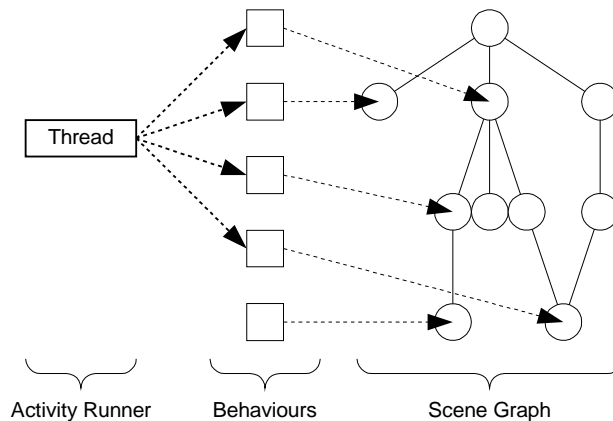


Figure 2. Behaviours modify properties of scene graph nodes

Each activity is managed by an `ActivityRunner` that is responsible for invoking its `performActivity` method. Activities and their runners can be organised as a hierarchy. The root activity runner is a thread that iteratively calculates the duration of each frame and passes the duration down to the activities that it manages. Intermediate nodes of the hierarchy act both as activities invoked by a higher `ActivityRunner` and as `ActivityRunners` for lower nodes.

In addition to behaviour events that are fired every frame to update time varying properties of scene graph nodes, activities can fire `ActivityEvents` to report that they have some significant state. A common form of activity is the “finite” activity that fires an event when it has completed its processing. Finite activities can be organised into hierarchical schedules of sequential and concurrent activities.

### 3.3. Creating Reusable Animations

A program can easily draw multiple copies of a structured graphic defined as a scene graph fragment by referencing that fragment from multiple points of the scene graph that each have different transformation or style nodes above them. Thus the same composite shape is drawn multiple times, with different positions, orientations and colours. Because scene graphs are made up of Java Beans, they can be serialised to files to build a library of “clip art”.

A scene graph fragment cannot be reused so easily when it is animated because serialising a scene graph will not save the behaviours that control it and connections between the application’s behaviours and the scene graph will be lost. `SceneBeans` therefore provides support for bundling a scene graph and the behaviours that control it into a reusable unit.

The basis for this support is the `Animation` class which acts as a *Facade* [3] for a scene graph and the active behaviours that modify that graph. An `Animation` is a `Composite` scene graph node that layers its subgraphs above one another. This allows an `Animation` to be easily embedded into a larger scene graph. An `Animation` is also an `ActivityRunner` that manages all the active behaviours that animate its subgraphs, and is itself an `Activity` that can be added to the `ActivityRunner` that manages the active behaviours controlling the graph in which it is embedded.



`AnimationEvents` when the user presses or releases mouse buttons on the visible portions of that sub-graph. The dispatch function of the `MouseClicked` class traverses the scene graph, transforming the point of the mouse click/release by the inverse transformation of each `Transform` node to map the point into the coordinate spaces of the transformed subgraphs. When it reaches a primitive node at the edge of the scene graph, it fires an event if the point is within the shape of the primitive and the primitive is a descendent of a `MouseClicked` node.

The `MouseMotion` class reacts to the user moving or dragging the mouse. Instances of `MouseMotion` act as a behaviours, feeding the location of the mouse pointer and the angle from the origin to the mouse pointer into other scene beans. Unlike `MouseClicked` nodes, which react only to events that occur on their subgraphs, all `MouseMotion` nodes react to the mouse all the time. The dispatch function of the `MouseMotion` class traverses the scene graph, transforming the current location of the mouse pointer when it reaches transform nodes, in the same way as that of the `MouseClicked` class. When it reaches a `MouseMotion` node, it announces the transformed point to the node's behaviour listeners. The `MouseMotion` node can be used to create interesting interactive effects, such as feeding input from one coordinate space to nodes in another coordinate space.

## 4. Defining Animations in XML

The SceneBeans framework provides programmers with a convenient programming model for creating and controlling animations, and a useful set of components that can be plugged together within that framework. However, it is not practical to expect end users to write Java programs in order to define animations for use in applications, and even for experienced programmers the edit/compile/debug cycle is slow and frustrating when fine-tuning animation parameters. Therefore, we have defined an XML-based file format for animations and implemented a parser that translates XML documents into Animation objects.

The XML document type definition (DTD) used by the SceneBeans parser is relatively minimal compared to DTDs for similar applications, such as the W3C's Scalable Vector Graphics (SVG) standard [14]. The DTD does not prescribe a limited number of component types and their options, but instead describes compositions of components that the parser loads dynamically and manipulates generically through the JavaBeans APIs.

### 4.1. Defining Scene Graphs

A SceneBeans document is contained within a top-level `<animation>` element that contains five types of sub-elements: a single `<draw>` element defines the scene graph to be rendered; `<define>` elements define named scene-graph fragments that can be linked into the visible scene graph; `<behaviour>` elements define behaviours that animate the scene graph; `<event>` elements define the actions that the animation performs in response to internal events; and `<command>` elements name a command that can be invoked upon the animation and define the actions taken in response to that command.

Figure 4 shows the definition of a simple scene: a red circle with a radius of 32 units centred in a square window of size 128 by 128 units (unless transformed, coordinates in an animation correspond to device coordinates, that is, pixels). The animation contains only the `<draw>` node and no behaviours, so the circle is not animated in any way. Both `<draw>` and `<define>` elements can contain the elements `<primitive>`, `<transform>`, `<style>` and `<compose>`, that correspond to the four basic classifications of scene graph nodes described in Section 3.1 and illustrated in Figure 1.

```

01 <animation width="128" height="128">
02   <draw>
03     <transform type="translate">
04       <param name="translation" value="(64,64)"/>
05       <style type="RGBAColor">
06         <param name="color" value="ff0000"/>
07         <primitive type="circle">
08           <param name="radius" value="32"/>
09         </primitive>
10       </style>
11     </transform>
12   </draw>
13 </animation>

```

Figure 4. An XML document defining a red circle centred in the display window

Examining the structure of the document in Figure 4, from the most nested compound element outwards, the `<primitive>` element instantiates a bean that implements the `Primitive` interface. The concrete type of the bean to be instantiated is determined by the `type` attribute; in this case, the type is `circle`. The SceneBeans parser maps the type name to a Java class by capitalising the first letter of the type name and then searching a list of packages for a class with that name. The `<param>` element, such as that contained within the `<primitive>` element, is used to set the value of bean properties. In this case, the `radius` property of the circle is set to 32. The visual appearance of all types of scene-graph node and properties controlling behaviours are all specified using the `<param>` element.

The primitive element is contained within a `<style>` element of type `RGBAColor` that sets the colour of the circle. Finally, the `<style>` element is contained within a `<transform>` element that translates the circle 64 units in the positive x and y directions. The `<param>` element of the style sets the style bean's `color` property to red; the value attribute is interpreted as six two-digit hexadecimal numbers indicating the red, green and blue components of the colour. The parser interprets the value attribute of `<param>` elements according to the type of the bean property being set, so convenient syntax can be used for real numbers, colours and points. Furthermore, expressions involving literal values and symbolic constants, such as "pi", can be used anywhere that a real value is expected.

Scene graphs are composed using the `<compose>` element; the `type` attribute of the element defines the type of composition – layer, switch, constructive area geometry operator and so on – applied to its sub-graphs. However, the `<transform>` and `<style>` elements, that correspond to beans that have only a single subgraph, can have multiple sub-elements, in which case the subgraphs are composed using a Layered bean. This reduces the need for compose elements significantly and is convenient when writing the XML by hand.

## 4.2. Defining Behaviour

Documents define animated graphics by creating and naming behaviour beans with the "behaviour" element and then animating the parameters of scene-graph nodes with the "animate" tag. Figure 5 shows an example document that defines a rotating rectangle. The behaviour tag is used to instantiate behaviour beans: the parser maps the algorithm of the behaviour to a Java class the same way as it does for scene-graph nodes, although it searches a different set of packages. Like scene graph nodes, param tags are used to configure behaviours by setting their JavaBean properties. Behaviours must be named by an "id" attribute so that they can be referred by an "animate" element within the scene graph. Animate elements create a binding between a behaviour and a property of a bean so that the behaviour modifies the value of the property over time, creating animation.

```

01 <animation>
02   <behaviour algorithm="loop" id="spin">
03     <param name="from" value="0"/>
04     <param name="to" value="2*pi"/>
05     <param name="duration" value="1"/>
06   </behaviour>
07
08   <draw>
09     <transform type="rotate">
10       <animate param="angle" behaviour="spin"/>
11       <primitive type="rectangle">
12         <param name="x" value="-16"/>
13         <param name="y" value="-16"/>
14         <param name="width" value="32"/>
15         <param name="height" value="32"/>
16       </primitive>
17     </transform>
18   </draw>
19 </animation>

```

Figure 5. XML definition of a rotating rectangle

Commands that can be invoked upon an animation are introduced by `<command>` elements, as shown in Figure 6. Command elements contain one or more action elements of types `<set>`, `<stop>`, `<start>`, `<reset>`, `<invoke>` or `<announce>`. The `<set>` tag sets the value of a parameter of a behaviour or scene-graph node in the animation. The `<stop>`, `<start>` and `<reset>` tags respectively pause, resume or reset the execution of an active behaviour. The `<invoke>` tag invokes another command defined by either the animation itself or by an included animation. The `<announce>` tag announces an event to the owner of the animation.

The `<event>` tag defines the performed by the animation in response to an event fired by one of its constituent beans. Attributes of the `<event>` tag identify the source and name of the event. The body of the tag defines the actions performed in exactly the same way as the `<command>` tag.

```

01 <command name="start-moving">
02   <invoke command="another-command"/>
03   <invoke object="included-animation" command="a-command"/>
04   <set object="object" param="x" value="0"/>
05   <stop behaviour="spin-sprite"/>
06   <reset behaviour="move-sprite"/>
07   <start behaviour="move-sprite"/>
08 </command>
09
10 <event object="move-sprite" event="stopped">
11   <announce event="sprite-stopped"/>
12 </event>

```

Figure 6. XML definitions of commands and events

### 4.3. Extending the Bean Namespace

Although the SceneBeans package includes a wide variety of useful visual and behaviour beans, specific animations or programs may need to use application-specific beans. The parser provides two mechanisms for making application specific beans available to an animation. Firstly, the parser provides methods to register additional packages that it searches for beans that implement behaviours or scene graph beans. This allows a program using the SceneBeans library to register its own packages with the parser. Secondly, the XML processor interprets the `<?scenebeans?>` XML processing instruction by registering additional packages of beans and the URL, relative to the document, of the codebase containing the bean classes. This allows additional beans to be specified for individual animations.

```

01 <?scenebeans category="behaviour"
02         codebase="spline_move.jar"
03         package="uk.ac.ic.doc.nat.spline_move"?>
04 <?scenebeans category="scene"
05         codebase="http://www-dse.doc.ic.ac.uk/~np2/beans.jar"
06         package="uk.ac.ic.doc.nat.morebeans"?>

```

Figure 7. XML processing instructions make additional beans available in an animation.

## 5. Animating Formal Models with SceneBeans

*TO BE WRITTEN.*

## 6. Related Work

Scene graphs have been widely used in toolkits for three-dimensional graphics and animation. For example, SGI's Inventor [4], VRML [6] and Java3D [7] all define a graphical scene as a scene graph, the nodes of which nodes are animated by behaviours. However, two-dimensional graphics toolkits have not usually provided a scene graph API. Notable exceptions are the Gnome Canvas and the Jazz toolkit.

The Gnome Canvas [8] is a widget for the Gtk user interface toolkit [9], provided as part of Gnome [10], a desktop environment for Linux and other Unix-like operating systems. The Gnome Canvas provides support for displaying flicker-free structured graphics. The application defines a canvas display as a DAG of canvas items. A canvas item is either a primitive graphical object or a group that lets an application treat multiple objects as if they were one. All canvas items have an associated affine transformation that is applied before they are drawn. The transformation is defined as a matrix, making it hard to compose animated transformations.

Jazz [11] is a user interface toolkit for building zooming interfaces in which user interface elements are placed in a logically infinite plane, around which the user navigates by panning and zooming. Jazz provides an API based on a DAG-structured scene graph. Like the Gnome Canvas, it associates an affine transformation matrix with each node, with the same disadvantages. Jazz is aimed at implementing user interfaces and supporting zooming, rather than building animations. The zoom level is passed to the nodes of the scene graph when they are rendered, allowing them to select different representations at different levels of zoom. Swing user interface components can be embedded in the scene graph, and therefore zoomed and transformed like any other node.

XML has previously been used to represent the structure of Java Bean aggregates. The Koala Bean Markup Language (KBML) [12] saves and loads Java Beans to and from XML documents, although it only supports a subset of possible bean implementations that they term "clean beans". IBM's Bean Markup Language (BML) [13] loads bean configurations from XML but cannot save configurations to XML. Thus, XML is used as an architecture description language, albeit one that is not particularly human-friendly. Both of these libraries are more general than SceneBeans' use of XML, which can only be used to compose beans that implement interfaces defined by the SceneBeans framework. However, the DTD used by SceneBeans makes the resulting documents much easier for humans to write by hand.

The SVG [14] standard from the World Wide Web Consortium (W3C) is an XML document type for describing two-dimensional graphics and animations. SVG defines a scene using the Document Object Model (DOM), a tree structured representation of an XML document. DOM nodes are used to represent primitive shapes, styles, paths and groups. DAG structures are defined by referencing one part of the document from another with a URI. Like the Gnome Canvas and Jazz, any node can be given an affine transformation. SVG defines a number of basic animation

algorithms that can be declaratively applied to the properties of DOM nodes. More complex animations can be defined by embedding scripts within the SVG document.

SVG is significantly more complex than SceneBeans, mainly because it specifies all properties of the scene within the XML document. SceneBeans, in comparison, interprets the document as commands to load and compose external components that define and animate the scene. Therefore SceneBeans is more open to extension than SVG, at the expense of potential incompatibilities between applications using the framework. This is to be expected: SVG is designed as a standard, platform-neutral graphics format, while SceneBeans is designed for building application-specific animations. It is intended that applications using SceneBeans provide the additional animation components that they need, and that different, incompatible applications will have no need to share SceneBeans documents containing non-standard components.

## 7. Conclusions and Further Work

We have found that the SceneBeans framework provides elegant abstractions for defining animations. The geometric model upon which the scene graph is based provides a powerful mechanism for composition and reuse of animations. A particular benefit of the system is its use of dynamically loaded components; this made it easy for different developers to extend the framework with the components that they needed without requiring changes to the underlying framework or the XML DTD used to define animations.

We hope to encourage others to use the framework by demonstrating it in a wider variety of applications. For example, it has been useful in the construction of educational software for teaching junior science: compelling, computer-based physics simulations can be created by writing a simple behaviour bean encapsulating various physical laws and using it to control an animation defined in XML. We are also working on graphical tools to help non-programmers build animations without needing to edit Java or XML code, and libraries of reusable animations for various application domains.

We intend to use the framework as a test-bed in various research projects. In particular we are extending it with networking support as part of our research into dynamic software architectures for ad-hoc networking and collaborative applications. This will allow multiple users to share a space in which they can create, view and interact with animated graphics.

## 8. References

- [1] K. Arnold and J. Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, 1998. ISBN 0-201-31006-6.
- [2] R. Englander. *Developing JavaBeans*. O'Reilly and Associates, 1997. ISBN 1-565-92289-1.
- [3] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0-201-63361-2.
- [4] T. Bray, J. Paoli and C. Sperberg-McQueen (Editors). *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium Recommendation 10-February 1998. Published on the web at <http://www.w3.org/TR/REC-xml>.
- [5] J. Wernecke. *The Inventor Mentor : Programming Object-Oriented 3d Graphics With Open Inventor, Release 2*. Addison Wesley, 1994. ISBN 0-201-62495-8.
- [6] R. Carey and G. Bell. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley, 1997. ISBN 0-201-41974-2.

- [7] H. Sowizral, K. Rushforth and M. Deering. *The Java 3D API Specification*. Addison-Wesley, 1997. ISBN 0-201-32576-4.
- [8] Havoc Pennington. *GTK+/Gnome Application Development*. New Riders Publishing 1999. ISBN 0-735-70078-8.
- [9] *The Gtk Project*. Published on the web at <http://www.gtk.org>.
- [10] *The Gnome Project*. Published on the web at <http://www.gnome.org>.
- [11] B. Bederson and B. McAlister. *Jazz: An Extensible 2D+Zooming Graphics Toolkit in Java*. University of Maryland technical report CS-TR-4015, UMIACS-TR-99-24, May 1999.
- [12] P. Kaplan and T. Kormann. *Koala Bean Markup Language*. INRIA. Published on the web at <http://www.inria.fr/koala/kbml/>.
- [13] S. Weerawarana and M. Duftler. *Bean Markup Language*. IBM. Published on the web at <http://www.alphaworks.ibm.com/tech/bml>.
- [14] J. Ferraiolo (Editor). *Scalable Vector Graphics (SVG) Specification*. World Wide Web Consortium Working Draft, 12 August 1999. Published on the web at <http://www.w3.org/Graphics/SVG/>.