The Australian National University **College of Engineering and Computer Science**

School of Computer Science

COMP2100 • COMP2500 • COMP6442

The Course Project 2011

GAGA

Introduction

Scalable **V**ector **G**raphics (SVG) is a standard to create and display vector graphics material based on the XML (e**X**tendable **M**ark-up **L**anguage). Vector graphics formats have a number advantages over bitmapped formats (like *JPEG*, *PNG*, *GIF*...) — much smaller file size, image quality is less affected by display resolution, ease of creation and editing, other technology can be incorporated to provide advanced features like animation and widget control (Synchronised Media Integration Language, *JavaScript*), SVG files can be generated programmatically (especially when a symmetry is involved, for examples see [5]).

SVG describes primitive shapes as leaf-type tag elements, whose style and geometrical properties specified by the attributes (dimensions, transformation directives, colours, stroke width and colour *etc*). The primitive elements can be grouped together as nested elements inside a container; this allows style, position and geometrical characteristics of several elements to be manipulated at once.

Almost all modern web browsers (except for *Internet Explorer*) can render SVG graphics (but the extent of the standard implementation varies — *Safari* and *Opera* can reproduce animation effects, Firefox — can't, at least in `3.x`, all can render gradient effects and translucency *etc*). Modern vector graphics editors (*Adobe Illustrator* and others) can export to SVG format. An open-source editor *Inkscape* uses SVG as native format for its files.

Our project in 2011 is to implement a rendering program which can display a document written in a *reduced* SVG format. We shall use the Java programming language and Java2D packages, in particular, to do this.

Scalable Vector Graphics format

We will work with SVG documents which only contains a *sub-set* all SVG-allowed elements: primitive graphics elements, their groups and most basic transformations.

Primitive Graphics Elements

The primitive elements include:

Line (interval)	`<line x1="0" y1="100" x2="100" y2="0"` `stroke-width="2" stroke="black" />`
Rectangle	`<rect x="25" y="70" height="30" width="50"` `fill="#ff8888" stroke="black" stroke-width="2"/>`

Circle	`<circle cx="140" cy="110" r="60" fill="none"`
	`stroke="#579" stroke-width="30"`
	`stroke-dasharray="3,5,8,13">`

Ellipse	`<ellipse cx="80" cy="170" rx="40" ry="30"` `fill="yellow" stroke="orange" stroke-width="25" />`

Polyline	`<polyline fill="none" stroke="blue" stroke-width="10"`
	`points="50,375 150,375 150,325 250,325 250,375`
	`350,375 350,250 450,250 450,375 550,375" />`

Polygon	`<polygon points="220,100 300,210 170,250"` `style="fill:#blue;stroke:red;stroke-width:2"/>`

Path	<path d="M 100 200 Q 200,400 300,200" fill="none" stroke="blue" /> <path d="M 100 200 L 200,400 300,200" fill="none" stroke="red"/>
Text	<text x="0" y="100" font-size="80" fill="red"> "Choosing a name for a software program is hard..."</text>
Image	<image xlink:href="nude.jpg" height="200" width="100" x="100" y="100"/>

The <image> element is used to insert a bitmapped image stored in a file. It also allows to insert an image described in the SVG format, as well. This last feature and the powerful <path> element, make possible the creation of sophisticated and visually rich graphics.

The tag element attributes include seen above geometrical and style declarations, and geometrical transformations which are applied to every pixel of the shape:

translate every point by a number of pixels in x- and y-directions	<tag transform="translate(-100,-100)" ... />
rotate every point on an angle around a specified point	<tag transform="rotate(120,219.5,241)".../>
scale an object by changing coordinates of **every point** by a common factor (can differ for x- and y-directions)	<tag transform="scale(1.2)".../> <tag transform="scale(1.2,0.8)".../>

Several transformations can be applied simultaneously: `transform="translate(0,100),scale(0.5)"`.

Container Elements

The container elements can be: Svg, Group, Use and Definition. Svg occurs only once in a document — it is the root of the content tree. Group allows to group several either primitive or container elements together, Use results in expansion in its place the element referenced by its `id` attribute. The element expanded by Use should be defined elsewhere in the document. Finally, Definition allows to introduce a primitive element or a group to use them elsewhere but *without displaying* them immediately.

Svg	<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" id="svg2" height="700" width="850"> <g ...> ....</g> <circle ...>...</circle> ... </svg>
Group	<g transform="translate(120,0)" fill="#bbb"> <rect x="100" y="100" width="100" height="20" fill="inherit" /> <ellipse id="e1" cx="150" cy="140" rx="30" ry="100" fill="#707" /> <rect x="100" y="130" width="100" height="20" fill="inherit" /> ¡/g¿
Use	<g id="G" > <rect x="100" y="100" width="100" height="20" fill="inherit" /> <ellipse id="e1" cx="150" cy="140" rx="30" ry="100" fill="#777" /> <rect x="100" y="130" width="100" height="20" fill="red" /> </g> <use xlink:href="#G" transform="translate(120,0)" fill="#bbb"/> <use xlink:href="#e1" transform="scale(1.2)" fill="#777"/>
Definition	<defs> ... </defs> (defines an element or a group, but does not display it; can be displayed with <use>)

The style attributes in elements inside a container, which are unset (defaulted), or set to `"inherit"` value, will be set to the corresponding values set in the parent container element (this works for all container elements — <svg>, <g> and <use>).

The full SVG specification contains advanced features like gradients, colour filters (blurring, distortion, *etc*), object morphologies, patterns, masks, region clipping and so on. Not all of them are currently implemented by applications which were deemed to be the primary users of the SVG format (namely, web browsers). Among the advanced features are animations and dynamic effects (like interactive SVG images which contain embedded control widgets). Some of them are partially implements. In the beginning, we will not use the above advanced features of SVG, and will only try to implement primitive graphics objects and containers listed above.

GAGA — a Java SVG Renderer

The beta-version of GAGA is capable of parsing an SVG file which contains a description of a scene graph using the reduced subset of SVG tags; it builds a document tree, performs its (incomplete) validations, and renders the scene graph using *Java2D* graphics package. GAGA implementation is incomplete. The salient lacking features include:

1. The program is yet to be able to process Polyline, Polygon and Path and Image graphics elements.
2. Colours (for `fill` and `stroke` attributes) can only be described in the `octal` or `hexadecimal` formats (*eg*, `fill="#ff0000"` can be processed, but `fill="red"` can't!)
3. In its dealing with the element attributes, GAGA currently can only understand style directives given in the form of `key="value"`, but unable to process style information provided in CSS style (see below).
4. It cannot process style and transformation defined in group elements and apply them to the nested container and graphics elements.
5. It cannot apply Use expansions because it must first collect all elements marked by the `id` attribute, but this feature is not yet implemented.

The GAGA Architecture

GAGA has a pipe-line architecture: the input character stream is broken into stream of tokens, which are then parsed to build the document tree (called here *a scene graph*), which then can be processed (validated, modified), and finally rendered in an Java graphical application window, Figure 1.

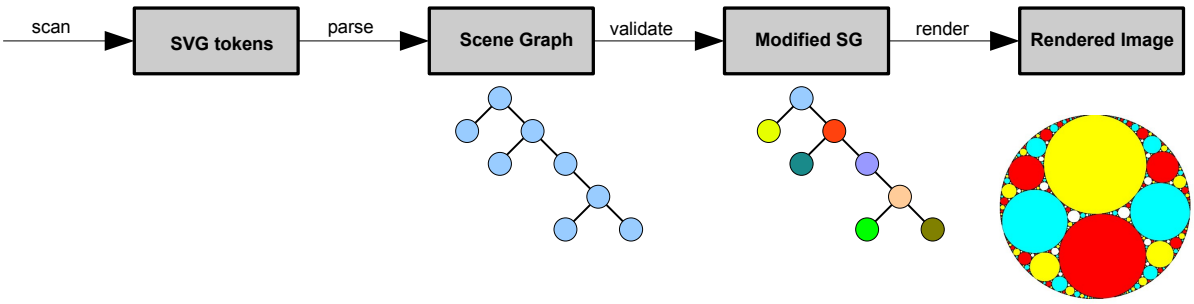

Figure 1: Pipe-line architecture of an XML processing system like GAGA

The stages in more detail:

1. *Scanning* is done using the standard *SAX* (**S**imple **A**PI for **X**ML) API from the Java standard library (the `org.xml.sax` package). The SAX class *DefaultHandler* is extended to implement the event handlers which are called when the scanner processes XML tags:

   - `startDocument()` and `endDocument()` to signal the beginning and the end of the entire document
   - `startElement()` and `endElement()` to signal the opening and closing element tags (name and attributes are retrieved as the method parameters)
   - `characters()` to collect and process characters inside a tag element

   The SAX API also provide event handlers to process comments, processing instructions, DTDs, resolve entities and check for errors (for example, whether the document is well formed *etc*). We only implement the begin/end event handlers for the tags (and the document) and characters assuming that we only deal with well formed SVG documents. We do not make use of DTD or PI event handlers since our program itself defines how the tags need to be processed and rendered.

2. We build the parse tree (aka *scene graph*) ourselves without using DTD/Schema provided by the W3C SVG specification. The tag parsing and tree building are done by the *SvgScanner* class (the one which extends *DefaultHandler*), which maps tag name and attributes into objects of the *SvgElement* hierarchy and creates a parse tree with the structure of the parsed SVG document.

3. The scene graph is processed by *Visitor* objects, which validate, modify and render the scene graph. Rendering is done with *Java2D* graphics library (packages `java.awt`, `java.awt.geom`, `java.awt.image` and others).

4. The structure of the scene graph is defined as the *Composite* pattern, and it is shown on the Figure 2.

5. The *SvgElement* specialisation into different container and graphics elements is done via a composition of an element and its *Type* (this solution employs the *Strategy* pattern, when the specialisation of types is achieved not through subclassing but through composition with an appropriate object from the *Type* hierarchy).

6. The tree processing and rendering is done by visitor objects which are defined in a separate (from the *SvgElement*) hierarchy, formed by the *Visitor* interface. The *Visitor* interface matches the concrete classes in the `Element--Type` hierarchy. The concrete *Visitor* objects perform element specific operations (like rendering) when called by the corresponding element object. The *Visitor* pattern is a well known solution to remove operations performed by an object hierarchy outside the hierarchy, and by this to make addition of new operations (or change existing ones) independent from the hierarchy itself. In greater details the *Visitor* pattern is discussed in the lectures.

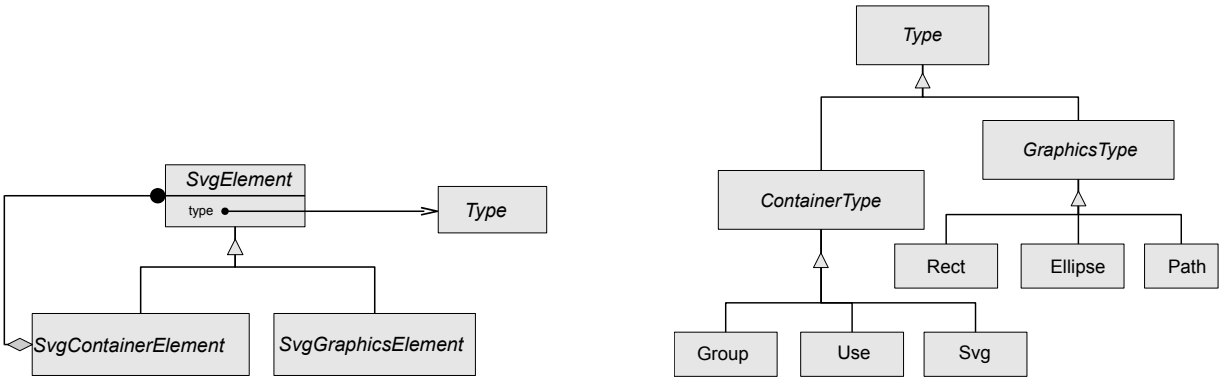

Figure 2: **Left**: Composite structure of the *SvgElement* hierarchy. **Right**: *Type*'s hierarchy.

Notes on literature and the name "GAGA"

The project will require additional studies of the XML, SVG and SAX (and the design patterns used in the original code). The two tutorials, [1] and [2], can be helpful, alongside with the official SVG standard, [3]. If you are not afraid of looking into a large code base, some ideas about how a rendering system like GAGA can be designed and implemented (at a much higher level of functionality), check out the Apache's *Batik* project, [4]. But know this — the *Batik* design and implementation did not influence the GAGA project in a slightest degree (we wanted to keep things at a more elementary level [1]).

Why the name "GAGA"? Honestly, it's not after Lady GAGA, or the song "Radio Ga-Ga" (one author is not Lady GAGA's fan, to say the least, and he has long overgrown *Queen*'s music). "GAGA" simply means "Give Assistance to Graphics Authors", which is the testimony to our lame imagination. Full stop.

References

[1] David Dailey, "An SVG Primer for Today's Browsers" An official tutorial with links to appropriate standard specification details.// http://www.w3.org/Graphics/SVG/IG/resources/svgprimer.html

[2] Gerald Bauer, "Scalable Vector Graphics (SVG). Creating High-End 2D Graphics Using XML" An old but useful SVG tutorial.
http://luxor-xul.sourceforge.net/talk/jug-nov-2002/slides.html

[3] The W3C SVG Standard Draft, "Scalable Vector Graphics (SVG) 1.1 (Second Edition) W3C Working Draft 22 June 2010" http://www.w3.org/TR/SVG11/

[4] "*Batik*, An SVG Java Toolkit",http://xmlgraphics.apache.org/batik/
An *Apache* toolkit for applications or applets that want to use images in the Scalable Vector Graphics. An open source Java framework which implements a large part (but not all!) of the SVG specification. The framework is used in an SVG rendered/editor, *Squiggle* (for Mac OS X only).http://apache.mirror.aussiehq.net.au/xmlgraphics/batik/Squiggle-1.7.dmg

[5] XML - Managing Data Exchange/SVG, A Wiki book about SVG.

School of Computer Science, ANU
Chris Johnson, Alexei Khorev
March 9, 2011

[1] And we also prefer to write our own code ©.