SVG vs. Canvas on Trivial Drawing Application

Samuli Kaipiainen

University of Helsinki Department of Computer Science

Â Helsinki
Â GustafÂ HÃ¤llstrÃ¶minÂ katuÂ 2b
Â 00014Â UniversityÂ ofÂ Helsinki
Â Finland
Â +358-9-191Â 51157

Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â

Matti Paksula

University of Helsinki Department of Computer Science

Â Helsinki
Â GustafÂ HÃ¤llstrÃ¶minÂ katuÂ 2b
Â 00014Â UniversityÂ ofÂ Helsinki
Â Finland
Â +358-9-191Â 51157

Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â

Abstract

In this paper we will create a small vector drawing application with SVG, and a small pixel drawing application with Canvas. Then we will swap platforms: we'll create a pixel drawing application with SVG, and a vector drawing application with Canvas. This experiment gives useful info on the limits of both techniques.

On top of this, we also combine the best implementations of the both techniques as one complete web drawing application. This requires us to implement a bi-directional interface between SVG and Canvas. Final application and documented implementation details hopefully gives a useful starting point in choosing the right technique for web application development.

A hidden agenda here is to also demystify the SVG vs. Canvas situation, as it now seems that general public thinks Canvas is so sexy, not realizing that SVG might be better suited for their vector-based needs.

Table of Contents

Introduction

Recent popularity on Canvas development and support has made the need for SVG somewhat vague. As SVG is still not widely adopted and Canvas has recently acquired lot of support from developers and browsers, some might question the role of SVG in future web. However, this is not the case and there is a strong need for SVG. To illustrate this, we will create a small a) vector drawing application with SVG and b) pixel drawing application with Canvas. Both are easy and straightforward to develop. The assumption is, that these are the typical applications for both techniques.

What will be interesting, though, is when we swap the standard and the target application. Now the assumption is, that each takes considerably more work. We show this to be true with a c) pixel drawing application with SVG, and a d) vector drawing application with Canvas.

This experiment gives insight on the limits of each standard. If it would turn out, that either of these "wrong" approaches works better than the other one, that would suggest the concerning standard to be "better", or at least to be more limited in general.

SVG's native "file format" is naturally SVG (XML), while Canvas' is bitmap, such as PNG. It will also be interesting to experiment on transferring data between them. This would be required for the standards to be attached into one useful drawing application.

We'll then attempt to layer and combine the SVG vector drawing application with the Canvas pixel drawing application. This way we can bring the best of both standards into one small standards-compliant (and non-Flash) web drawing application. Here the assumption is, though, that it's easier to stay in one standard, as there is no useful interface between SVG and Canvas. This is partially proved wrong, as we will show possible workarounds to combine both standards.

First we'll create the trivial drawing applications, by hand-written JavaScript (with one exception) in effort to really find the limits of each standard. Then we'll see which standard is the "winner", and investigate on ways to exchange data between them.

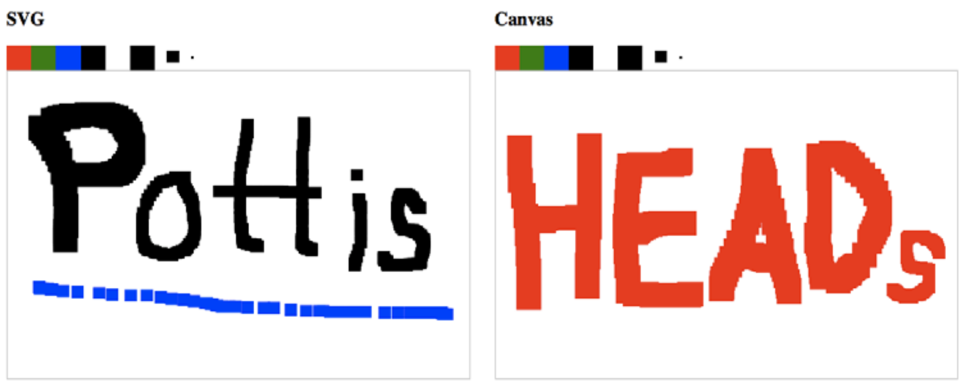
We conclude that there is no vs. situation between SVG and Canvas. Instead, both are needed and useful for future web, as both have their strong points and there's no reason to degeneralize those into just one standard.

Browser requirements for demos presented in the paper: Firefox 3.0+, Safari 4.0.

Pixel drawing application

Pixels are no doubt the easier setting, so we even implemented some extra features! The features of our pixel drawing applications are:

- Draw rectangle-shaped pixels
- Three selectable brush sizes
- Four selectable colors



Demo: http://db.cs.helsinki.fi/~paksula/svg_open_2009/1-pixels.xhtml

FigureÂ 1.Â Our pixel drawing applications

Pixels with Canvas

This is the natural application for Canvas. Drawing pixels is straightforward with Canvas' rectangle shape. Resulting image can also be exported as bitmap with Canvas' built-in `toDataURL()` function, as will be seen later in the section called â€œCanvas to SVGâ€.

```
function drawPixel(canvas, x, y) {
        var ctx = canvas.getContext("2d");
        ctx.fillRect(x, y, 1, 1);
}
```

ExampleÂ 1.Â Drawing pixels with Canvas

Implemention notes:

- We chose rectangles simply as there is no native circle function on Canvas.

Pixels with SVG

As there are no pixels in SVG, we need to emulate pixels by drawing SVG objects. This will look exactly the same as real pixels. However, each pixel is a new SVG shape added to the DOM, meaning that the DOM grows as drawing continues. Modifying DOM with raw JavaScript is quite verbose, too.

Strangely, it's not possible to get the rendered view out of web browser. SVG lacks the basic `toDataURL()` function. Thus, the resulting image is not exportable as bitmap without server-side support, exactly which will be done in the section called â€œSVG to Canvasâ€.

```
function drawPixel(SVGRoot, x, y) {

        var pixel = document.createElementNS("http://www.w3.org/2000/svg", "rect");

        pixel.setAttribute('style', "fill: black");
        pixel.setAttribute('x', x);
        pixel.setAttribute('y', y);
        pixel.setAttribute('width', 1);
        pixel.setAttribute('height', 1);

        SVGRoot.appendChild(pixel);
}
```

ExampleÂ 2.Â Drawing pixels with SVG

Implemention notes:

- One needs to be aware of the namespace situation, as if the shape is created without namespace, it simply won't render!

- Getting the mouse coordinate of the point clicked inside SVG element is somewhat buggier than in Canvas' case. It turns out, that for example jQuery's `offset()` function is 200 lines long. We used the new `getBoundingClientRect()` for getting the SVG element's offset, but it then turned out, that in Firefox 3.5 the bounding box of SVG element is defined by its children. This then makes `getBoundingClientRect()` useless for getting offset coordinates. We thus ended up using the deprecated `document.getBoxObjectFor()`, as it gives us consistent values.

- Using inline SVG element instead of an object element is a bit more risky, as the mime-type needs to be correct XHTML. In most situations, especially when working locally, using .xhtml filename extension takes care of this. We chose inline SVG, as it resembles the future of web, and goes hand in hand with the inline Canvas element.

Winner of pixels

Canvas. You guessed it.

As assumed, pixels are the natural environment for Canvas. Canvas has a basic API for drawing primitive shapes, which is all we need. Canvas is a low-level "framebuffer" for drawing pixels, and in that buffer, you can do anything imaginable, you just need to do it by hand. So in addition to drawing application, there are lot of traditional applications and games, like Wolfenstein 3D clones, that benefit from Canvas.

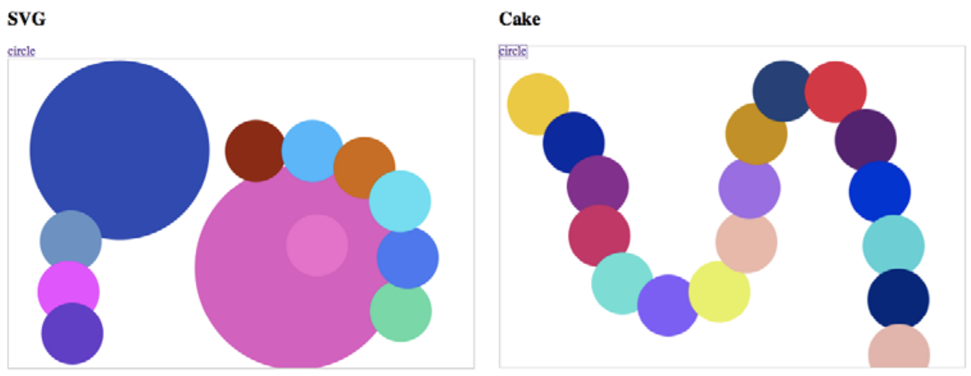
For SVG, each "pixel" is a separate SVG shape, bloating the SVG DOM. This slows down rendering, takes a lot of memory and results in a huge DOM tree. So for any real work, emulating pixels with SVG shapes is no good. Also, SVG doesn't work as a frame buffer, there's a big overhead in any implementation which tries to emulate this. SVG is for higher-level shapes, not for pixels.

Vector drawing application

Vectors are more interesting, especially for Canvas, since it doesn't support them natively. Also, being bitmap data, Canvas pixelates when scaling up. So we need to find a non-pixelating solution for Canvas.

For vector drawing, there are two parts: 1) rendering vector shapes on screen, and 2) allowing interaction with the shapes. So the features of our vector drawing applications are:

- Insert random-colored circles with 40 pixel radius
- Move the circles around (drag-n-drop)
- Scale the circles (with right-click-drag)
- Re-stack the circles on the top when clicked



Demo: http://db.cs.helsinki.fi/~paksula/svg_open_2009/1-vectors.xhtml

FigureÂ 2.Â Our vector drawing applications

Vectors with SVG

This is what SVG is all about. Vector shapes can be simply added to the SVG DOM, and they appear rendered on the browser window. It's just like adding the emulated pixels we already did.

SVG also handles high-level mouse events, telling us which element was clicked or dragged. The low-level core of user interaction needs to be done by hand with JavaScript. That's quite basic scripting, though. Using any SVG library might make it even easier, but we chose do it manually.

```
function addCircle(SVGRoot, radius, x, y) {

        var circle = document.createElementNS("http://www.w3.org/2000/svg", "circle");

        circle.setAttribute('style', black);
        circle.setAttribute('cx', x);
        circle.setAttribute('cy', y);
        circle.setAttribute('r', radius);

        SVGRoot.appendChild(circle);
}
```

ExampleÂ 3.Â Drawing vectors with SVG

Implementation notes:

- Again with the namespace situation and inline SVG, be careful.
- For drag-n-drop, getting exact mouse coordinates is not necessary, as we only need to get the changes in mouse movement and add them to those of the location of the shape being dragged.
- Also it's not necessary to add mouse event listeners to each shape, as it's enough to have them on the main SVG element, and the top-most shape clicked/dragged is relayed via event's target.
- Applying transformations can become tricky, as here we both *translate* and *scale*. We need to keep track of previous transformations, so we can re-apply them when a shape is being re-dragged. Also, the transformations need to be applied in correct order. Here it means that translation needs to be done last, otherwise scaling will be offset from the circle's centre. (We tried to simplify this with SVG's native `getCMT()` function, as done by SVG-Whiz, but found that just a new source of bugs and unexpected behavior. So we ended up keeping our own track of the transformations.)

Vectors with Canvas

As Canvas is just a bitmap and supports those functions well, we need to implement a scene graph on top of Canvas with JavaScript/DOM, in order to keep track of the state of all the vector shapes, and implementing tests on which shapes are under mouse cursor. That's a **huge** overhead (which SVG natively handles), and we won't be going down that road.

So we decided to take a shortcut by using the CakeJS library. This library provides scene graph implementation as required to support vector shapes that can be drawn and dragged on Canvas. Adding shapes becomes almost as easy as drawing pixels, and mouse interaction with shapes is as easy as with SVG. However, CakeJS is a huge 200+ KB library, adding a lot of extra code to the implementation.

```
function addCircle(cakeCanvas, radius, x, y) {

        var circle = new Circle(radius, { x: x, y: y, fill: black })

        cakeCanvas.append(circle);
}
```

ExampleÂ 4.Â Drawing vectors with Canvas and CakeJS

Implemention notes:

- CakeJS library is not very well documented, and one needs to search source and example code for basic API information. For example, the Cake functionality cannot be neatly added to an existing Canvas: `new Canvas(HTMLCanvasElement)`. When done that way, the mouse events do not work. Instead, Cake needs to create the Canvas element by itself, and append it to a placeholder element: `new Canvas(HTMLPlaceholderElement, width, height)`.
- Now the mouse event listeners need to be added to each shape, and in contrast to SVG, Cake doesn't relay the top-most shape clicked/dragged via the events target.
- As we are only drawing circles here, we ended up doing transformations (translate and scale) as "raw". That is, we simply change the `x`, `y` and `radius` attributes of the circle shapes.

Winner of Vectors

That's SVG.

As assumed, vectors make SVG happy. SVG handles both the drawing of vector shapes, and the hard parts of user interaction with the shapes. In addition, SVG can be easily exported from a vector editing application, such as Inkscape or Illustrator. This retains all of SVG structure, allowing easy scripting and animating with the SVG DOM.

For Canvas, there are no vectors and no user interaction with the drawn shapes. There are is just framebuffer, and that's how it was meant to be. Yet, there are libraries which implement all this on Canvas. It's questionable as to why should Canvas have these features, but we'll come back to this in the section called â€œNeed for Scene Graphâ€.

Combining the best techniques

As can be seen from the previous experiments, SVG is strong and robust with built-in shape handling, while Canvas is fast and efficient for pixel manipulation. For a complete drawing application, we need layered bitmaps on top of each other, something that would resemble Adobe Photoshop with layer ordering and simple shapes. For this, common interfaces are required between Canvas and SVG.

Both standards allow accessing their data through JavaScript, but there are no shared interfaces to utilize co-operation. SVG gives its data as (serialized) XML, while Canvas as base64-coded PNG.

Canvas to SVG

We came up with two reasonable solutions: Including Canvas as foreignObject into SVG, or to import Canvas bitmap data as image element into SVG. foreignObject allows SVG to be extended with anything the browser can render.

```
<?xml version="1.0">
<svg xmlns = "http://www.w3.org/2000/svg">
  <g>
    <foreignObject x="50" y="100" width="600" height="400">
      <body xmlns="http://www.w3.org/1999/xhtml">
                <canvas id="canvas" width="600" height="400"></canvas>
      </body>
    </foreignObject>
  </g>
</svg>
```

ExampleÂ 5.Â Inclusion of Canvas to SVG using foreignObject

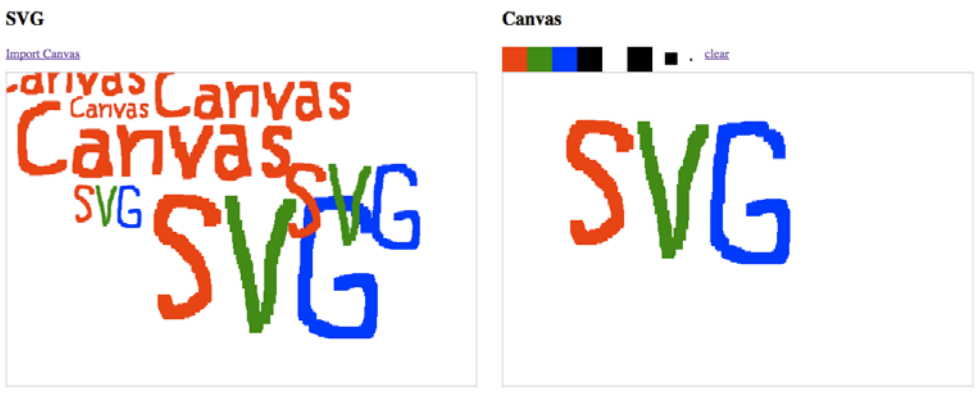
However, SVG doesn't know that included element is basically a bitmap. This makes rasterization of the composed vector image hard, as the component that makes the rasterization should also be able to render HTML Canvas element (or in fact any HTML element). As our purpose is to rasterize all image data into one single image, that is not an option. foreignObject is a good solution for example when including interactive features into SVG, but it is not good for our purposes right here.

Canvas supports toDataUrl("image/png") function that returns a data URI representation of current Canvas bitmap encoded in base64. This image data can be set as source for HTML img element or image element in SVG. We use this method to import Canvas to SVG with JavaScript.

```
function importCanvas(sourceCanvas, targetSVG) {

        // get base64 encoded png from Canvas
        var image = sourceCanvas.toDataURL("image/png");

        // Create new SVG Image element.  Must also be careful with the namespaces.
        var svgimg = document.createElementNS("http://www.w3.org/2000/svg", "image");
        svgimg.setAttributeNS("http://www.w3.org/1999/xlink", 'xlink:href', image);

        // Append image to SVG
        targetSVG.appendChild(svgimg);
}
```

ExampleÂ 6.Â Simplified import of Canvas to SVG.

Now we have successfully imported Canvas bitmap into SVG as a separate element. This element can be easily transformed like any SVG element. Transparency is preserved as well.



Demo: http://db.cs.helsinki.fi/~paksula/svg_open_2009/3-canvas2svg.xhtml

FigureÂ 3.Â Canvas to SVG using toDataUrl("image/png")

SVG to Canvas

If SVG would also support toDataUrl() function like Canvas, this would be trivial. However, that is not the case, and even stranger is that the Internet gives us no clues as to why is SVG missing this basic functionality. After all, the browser internally renders SVG as a raster bitmap, so why not provide that data for JavaScript?

So we need to implement full SVG parsing and rendering in Canvas or do SVG conversion to bitmap server-side. Some libraries, like CakeJS, have implemented SVG parsing. This method seems to be a bit experimental, and is once again re-implementing browser's native functionality with JavaScript.

Server-side conversion is a stable way rasterize SVG to bitmap. This can be done for example with ImageMagick, Batik or Inkscape. If we would have selected foreignObject as our "Canvas to SVG" -approach, we would not be able to fully rasterize SVG with these utilities.

```
<?php

$svg_xml = $_POST["svg_xml"];                   // SVG is sent as POST parameter in the request

file_put_contents("input.svg",$svg_xml);        // Write SVG to disk
system("convert input.svg output.png");         // Run conversion program (ImageMagick)
echo "output.png"                               // Return the name of rasterization for client

?>
```

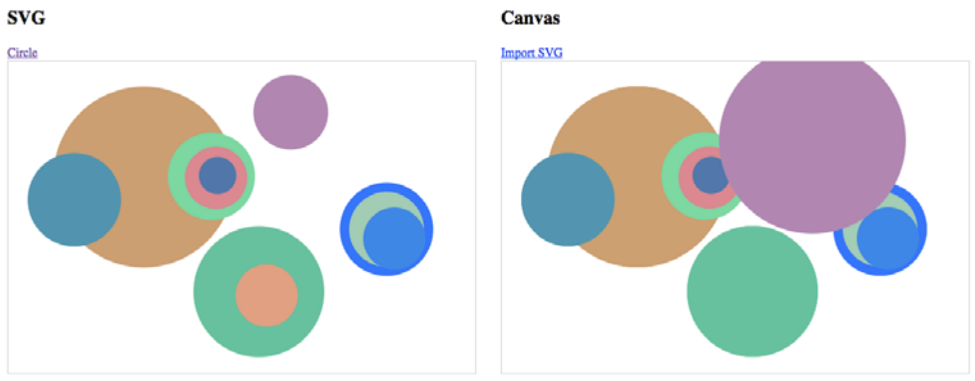
ExampleÂ 7.Â Simplified server-side conversion with PHP for Ajax implementation.

Dynamically modified SVG can be serialized to string with JavaScript and sent to conversion with Ajax.

```
function importSVG(sourceSVG, targetCanvas) {

        var svg_xml = (new XMLSerializer()).serializeToString(sourceSVG);

        $.post("convert.php", { svg_xml: svg_xml },                     // send xml as "svg_xml" POST parame
                        function(data) {                                // when conversion is ready
                        var ctx = targetCanvas.getContext('2d');
                                var img = new Image();

                        img.onload = function()
                                ctx.drawImage(img,0,0);                 // draw rasterized image to the Canv

                        img.src = data;
                        }
        );
}
```

ExampleÂ 8.Â Simplified SVG serialization and Ajax request with jQuery.

When we combine SVG vector drawing application with server-side conversion, we have a working method of transferring dynamically modified SVG to our Canvas bitmap.

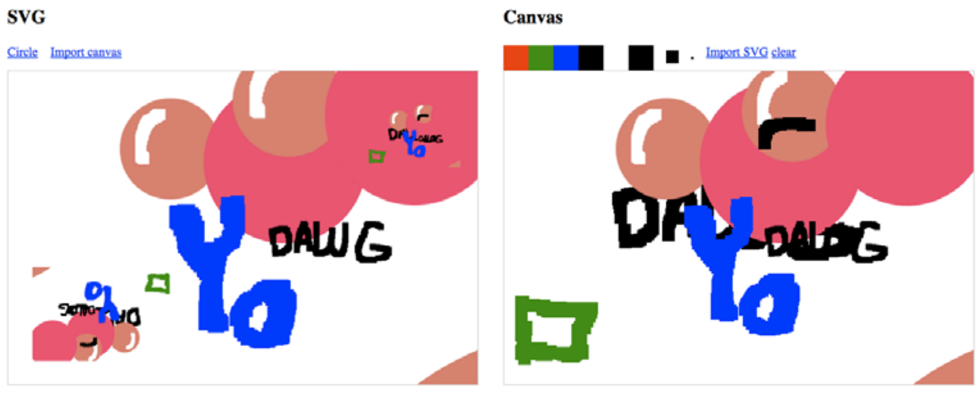


Demo: http://db.cs.helsinki.fi/~paksula/svg_open_2009/4-svg2canvas.xhtml

FigureÂ 4.Â SVG to Canvas using server-side conversion and Ajax

Putting it all together

So, we combined everything we learned from our experiments: Canvas based pixel drawing application, SVG based Vector drawing application and linked them together with canvas.toDataUrl() and server-side conversion. We are now consistently transferring bitmap data between Canvas and SVG.

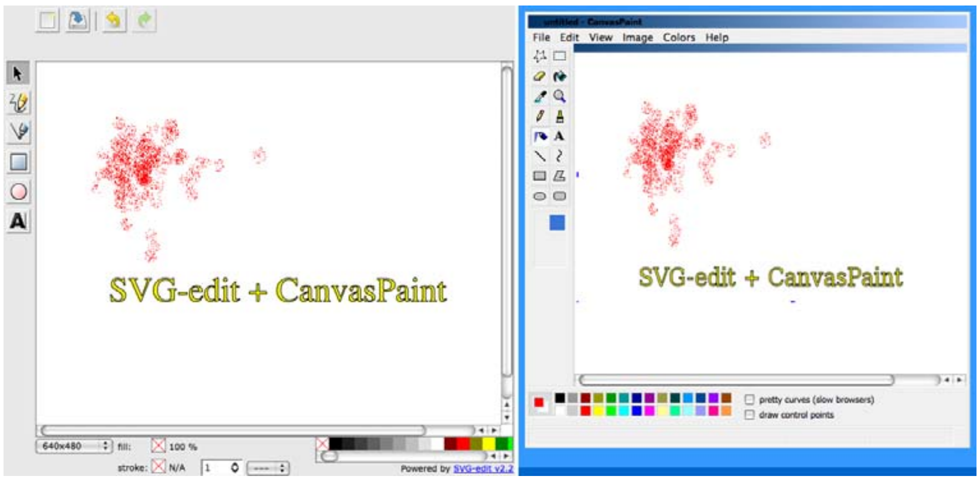


Demo: http://db.cs.helsinki.fi/~paksula/svg_open_2009/5-yodawg.xhtml

FigureÂ 5.Â SVG to Canvas and Canvas to SVG

In this example SVG and Canvas areas are separated to follow previous layouts. For a complete and more advanced drawing application a better layout would be needed. However, this demo is interesting as both of the drawing components can be replaced with any existing components and bind together with the interfaces described in our solution.

Combining existing components: SVG-Edit and CanvasPaint



Demo: http://db.cs.helsinki.fi/~paksula/svg_open_2009/6-svgedit_canvaspaint.html

FigureÂ 6.Â SVG-Edit and CanvasPaint interaction

By embedding two pre-made non-trivial components and using toDataUrl() and server-side conversion a complete drawing application could be achieved. We quickly tried this with SVG-Edit and CanvasPaint. Implementation is not stable as we did not modify the codebases of the components. This demonstration is just to illustrate the potential of SVG-Canvas data exchange. It also stresses how Canvas is more suitable for "MSPaint" and SVG is more suitable for "Inkscape".

Observations

During all the implementation and research work for this paper, we have seen many relating trends on the web. Here are some observations on those that we'd like to point out.

Definition of "Web Graphics"

If we take a look at graphics on the web, there is a clear need for vector graphics. W3C suggests that SVG would be used where needed. Generally, web graphics are naively defined as "72 dpi raster image, preferably JPEG". Even when source image is from vector origin, the image is rasterized for web, because of the lacking support / belief for vector graphics in web.

Situation might be even worse if graphics from vector origin, natively fit for SVG, should be done in Canvas. SVG provides nice addressable representation of vector graphics and is supported with various tools and editors. In XHTML specification the trend seems to be that structure is being generalized, like <h1>..<h6> into nested <h> elements, and even <img> into <object>. So why not <bitmap> and <vector> elements instead of <img>, <svg> and <canvas>?

To be truly able to make a transition from print media to web, various shapes and scalability of graphics has to be provided by the browsers in an addressible manner. This is where consistent and good SVG support is required.

Browser Compatibility

Browser compatibility is always a problem, especially when introducing new elements to HTML. Even though SVG is a much older standard than Canvas, it's less supported in browsers. That's partly because SVG is such a huge standard, covering everything from foreignObjects to animation.

As discussed in the section called â€œPixels with SVGâ€, getting mouse coordinates relative to an element's top-left corner is still really complicated. JavaScript libraries try to handle all the quirks involved, but it's a mess. The new getBoundingClientRect() brought by IE looks promising, but for SVG, there are still issues interpreting W3C's specs.

There are many libraries which try to make SVG easier to use with JavaScript, but we thought the library situation is somewhat messy, and for this paper it was more beneficial to do everything by hand. Most libraries handle adding SVG shapes with Canvas-like drawing commands, as this seems to be what web developers prefer.

The Speed Situation

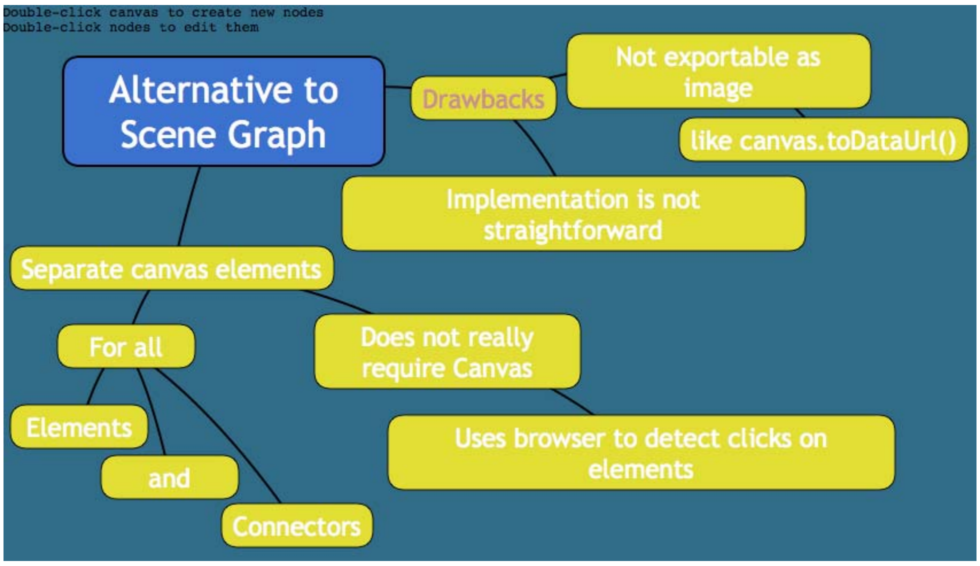
As Canvas will inevitably get speeded up by passionate JavaScript development, what will happen to SVG? SVG doesn't "automatically" get faster alongside any web technology, it would have to be sped up separately. Yet, SVG isn't nearly as fast as it could be, as when compared to native OpenGL vector engine implementations.

One conclusion from current research on SVG vs. Canvas speed situation: â€œSVG doesnâ€™t scale well to large numbers of objects, but Canvas doesnâ€™t scale well to large screens.â€ [http://unterbahn.com/2009/08/svg-vs/].

Need for Scene Graph

It seems like for interactive applications a scene graph support is needed.

Browser already provides a "scene graph". It keeps information of elements and provides required events for interaction. Layered elements are possible to implement with HTML and JavaScript. This method does not provide easy export of the resulting image. A demo of this approach is illustrated below.


Double-click canvas to create new nodes
Double-click nodes to edit them
Alternative to Scene Graph
Drawbacks
Not exportable as image
like canvas.toDataUrl()
Implementation is not straightforward
Separate canvas elements
For all
Does not really require Canvas
Elements
and
Connectors
Uses browser to detect clicks on elements

http://www.livedo.org/camima

FigureÂ 7.Â Using browsers scene graph

For interactive Canvas applications, a popular way seems to be implementing SVG-like scene graph on Canvas. Why not just use SVG? It's a huge overhead to implement SVG with Canvas, an overhead that the browsers already handle quite efficiently. There's no way of ever implementing SVG as fast with JavaScript than the browser does natively.

However, if Canvas would have a native scene graph implementation, then things could change. Interesting quote from W3C specification about canvas.toDataUrl(): â€œThe possible values are MIME types with no parameters, for example `image/png`, `image/jpeg`, or **even maybe `image/svg+xml`** if the implementation actually keeps enough information to reliably render an SVG image from the Canvas.â€ This would actually mean, that Canvas sort of implements SVG, lending even more confusion for their co-existence.

Conclusions

No vs. Situation!

Based on these findings, there really is no vs. situation. The right technique needs to be selected for the job, and we hope this paper gives a good starting point for that. When the right choices are made, SVG and Canvas can actually benefit from each other. Currently there are some things to look out for until the support is stable.

Yes, there's a limit on each standard's capabilitiness. For pixel flare and other "demo effects", go with Canvas. For intrinsic shapes and user interaction, go wth SVG. Graphical user interfaces on the web are an especially delicious application for SVG.

The Political vs. Situation

As Apple introduced Canvas, there was some controversy for Apple's decision to create a new proprietary element instead of supporting SVG, which still hadn't achieved broad web developer acceptance. [Wikipedia: Canvas] However, as stated its our believe that there is need for both standars, and all that's needed is stable support for both of them.

Sadly, Internet Explorer never started supporting SVG because of their VML effort, and Adobe stopped supporting their IE SVG plugin (maybe because of acquiring Macromedia and Flash?). This leaves SVG stranded, as it doesn't seem probable for SVG to ever appear on IE, which would be required for widespread support. There are, however, libraries which bring both SVG (SVG Web) and Canvas (Explorer Canvas) support for IE, using IE's native VML capability or Adobe Flash plugin.

Bibliography

We chose not to use the Docbook bibliography, as it does not render correctly. Instead, we are using inline links, which do not render in typical print output.