PERFORMANCE OF BÉZIER CURVES RENDERING IN WEB BROWSERS

By

Ammar Hattab

APMA2821Q Project Report

Prof. Mark Ainsworth

Spring 2013

Brown University

Abstract

Bézier curves are used everywhere in graphics, enhancing their rendering performance will have an effect everywhere, in web browsers the case is more critical due to the need of successive and fast rendering in interactive and animated web pages especially in small devices with constrained resources where we could notice a choppy performance in rendering SVG graphics animations.

In this project we studied the usage of Bézier curves in modern web browsers, and different rendering methods used, and then we tested the performance of different web browsers in rendering Bézier curves, and compared the performance of rendering methods individually, then we focused on animation of Bézier curves; where we tested a new idea to enhance the performance of Bézier curves animations.

Introduction

Bézier curves are the most widely used curves in computer graphics, as they offer an efficient way to represent smooth curves [1]. For 30 years many rendering algorithms evolved to represent and render Bézier curves efficiently in different applications with a recent focus of hardware accelerated algorithms [2].

But at the same time the need for a new and more efficient ways to represent and render Bézier curves has increased with time for two reasons: the first reason is the increase usage of smaller devices with more constrained resources (ex: smart phones), the second reason is the evolution of web standards to add the ability for web developers to draw Bézier curves in several ways with strict conformance requirements for a high quality rendering and animation of draw Bézier [3].

To respond to these challenges web browsers have also evolved from only displaying text to providing the user with a rich interactive graphics experience and with complex applications and support for many platforms and device.

In web browsers (generally in all interactive and animated software) the main challenge is to have a smooth animation with a good, consistent frame rate, so the web browser implementer together

with the software developer should focus on preventing any choppy performance that could appear in animations and interactions “Jank Free” [4][5].

Bézier Curves in Web Standards

In HTML5 the standard way for web developers to draw graphics is using SVG and Canvas technologies, which are both standard-based on World Wide Web Consortium (W3C) [3][6], and because of that they do not require any plug-in or browser specific code, both of them provides standard commands to draw lines, arcs, and most importantly quadratic and cubic Bézier curves

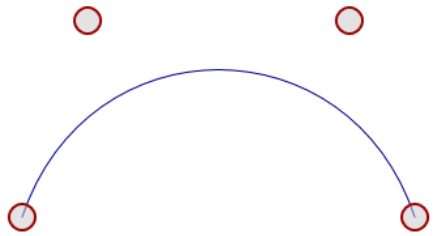

Figure 1 Bézier Curve Using SVG and Canvas

To draw this Bézier curve we can use the following commands:

a) Using SVG: <path d="M 100 250 C 150 100 350 100 400 250"/>
b) Using Canvas: moveTo(100, 250); BézierCurveTo(150, 100, 350, 100, 400, 250);

Browsers Rendering of a Bézier curve

A large portion of web browsers are open source, but normally their code is a huge forest of source trees with many libraries and many levels of abstractions (for example Chromium uses ~100 libraries and ~5000 build objects [7]).

Web browsers use their own rendering engines to display the formatted content on the screen. Each rendering engine uses its own way to render the content in order to offer same look and feel across different platforms (for example: WebKit, for Apple's Safari, Blink for Google Chrome, Gecko, for Firefox, Trident, for Internet Explorer, Presto, for Opera).

Now the browser rendering engine uses several graphics libraries to render Bézier curves (from both SVG and Canvas) depending on the platform, availability of Hardware acceleration, and many other criterions.

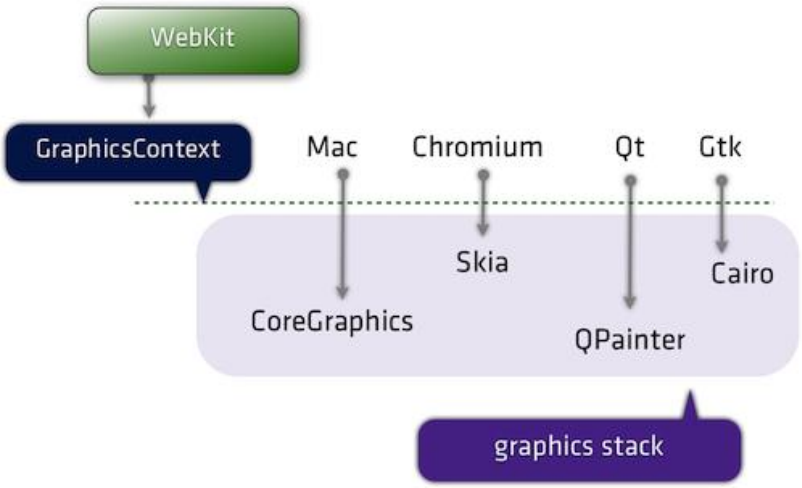

Figure 2 Browser uses different graphics libraries based on the platform

As an example Skia is a 2D graphic C++ library that has ~ 80,000 lines of code used by Google Chrome for many graphics operations, including text rendering [8], Skia uses a form of recursive subdivision in order to render Bézier curves (which will be discussed in the following section).

So in practice it will totally depend on the platform and the application, and the performance will vary accordingly.

Bézier Curves Rendering Methods

Typically in most rendering methods we start with control points, and find a piece wise linear approximation, then render those lines to pixels. The process of approximating the curve by line-segments is called the “flattening the curve.” [1]

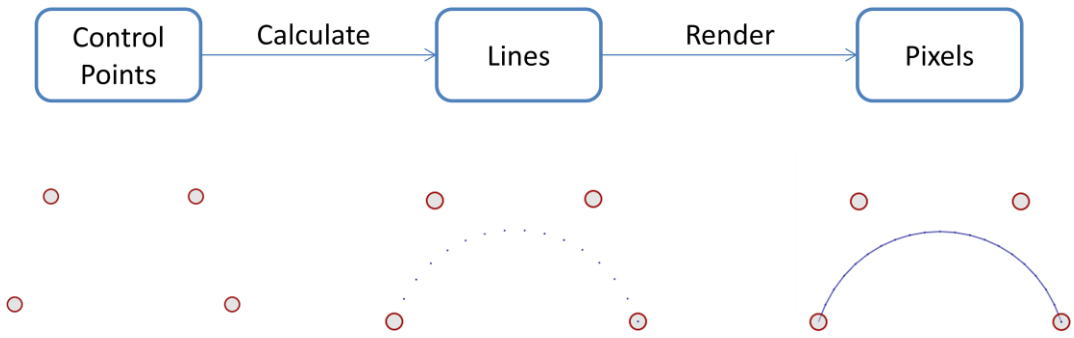

Figure 3 Flattening and rendering Bézier curve

Incremental Methods

In incremental methods we divide the curve into n equally spaced parameter intervals then starting from the first point we find the function value at the endpoints of each interval.

- Brute-Force:
  Evaluate the curve function value using direct Bézier curve cubic equation (it will cost 9 multiplies and 10 additions)
- Forward Difference [9]:
  Since cubic Bézier curve is third degree polynomial and its third derivative is always a constant; we can compute the point at next step by finding up to third forward differences. (it will cost only 6 Additions)
  f += d1
  d1 += d2
  d2 += d3

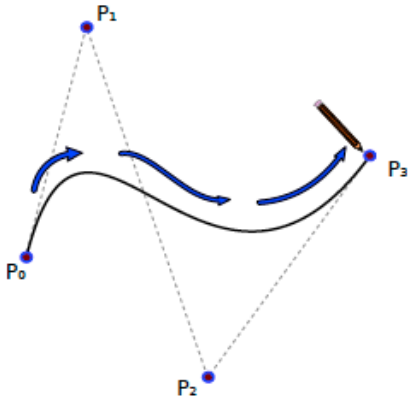

Figure4 Incremental Methods

Recursive De Casteljou Subdivision

This is one of the most common techniques used to generate an acceptable polyline for a given Bézier curve. [1]

In this algorithm the Bézier curve is recursively subdivided at t =1/2 until the individual Bézier sub-segment's approximating error is less than or equal to the specified flatness

So If the Bézier curve can be approximated to within tolerance by the straight line joining its first and last control points, then this line segment will be used, otherwise the subdivision of the curve continues at (at r = 1/ 2) recursively.

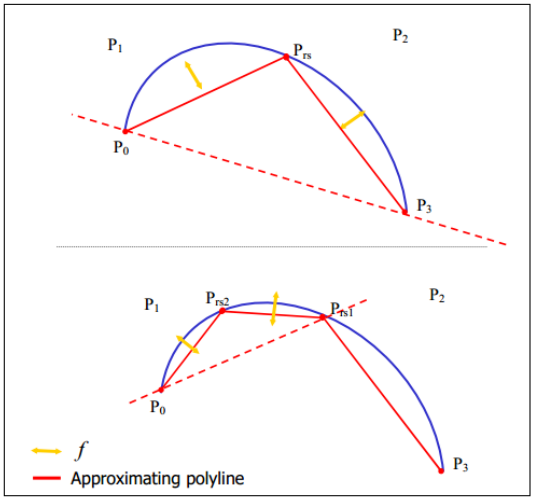

Figure 5 Recursive Subdivision

In practice, the approximation error which is the stopping condition (called: maximum transverse deviation) is not actually calculated, but estimated by another lower cost measure [10]

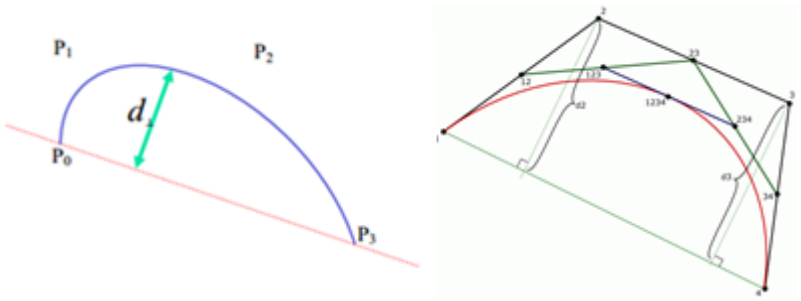

Figure 6 Maximum Transverse Deviation and Approximate Measure

Hybrid Approaches

Some other papers used features from both incremental methods and recursive subdivision method, for example Adaptive Forward Differencing is an incremental method but with dynamic step size, where at each step, we may double the step size or divide it by half depending on the distance between two successive points [11].

Method

jsPerf Performance Tests

In this project we used the online jsPerf tool to evaluate performance, which is an easy way to create and share test cases, comparing the performance of different JavaScript snippets by running benchmarks. [12]

Bézier Curve Browser Rendering Performance

In this project we want to evaluate the performance of Bézier curve rendering in a web browser, some previous studies evaluated the performance of rendering an SVG file in different web browsers [13] (Safari was the fast as in the figure below)

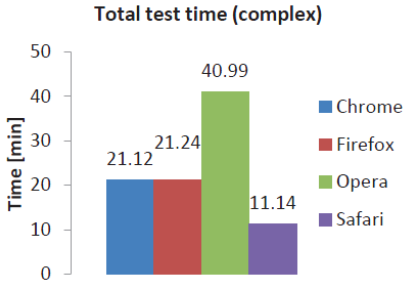

Figure 7 SVG Drawing Performance

We focused on evaluating the direct call of Bézier curves drawing functions, to avoid the overhead of SVG parsing and to focus only on Bézier curves rather than all the drawing methods and elements of SVG, so here we created a jsPerf with only a direct call to Bézier curve drawing function, and run that jsPerf on different browsers.

Bézier Curve Rendering Methods Performance

To evaluate the behavior of different rendering methods we have implemented the previous techniques (Brute Force, Forward Difference, Adaptive Forward Difference, and two versions of Recursive Subdivision) and created a tool that will render the curve according to the chosen technique and the specified parameters, the tool also returns the number of line segments approximated. [14]

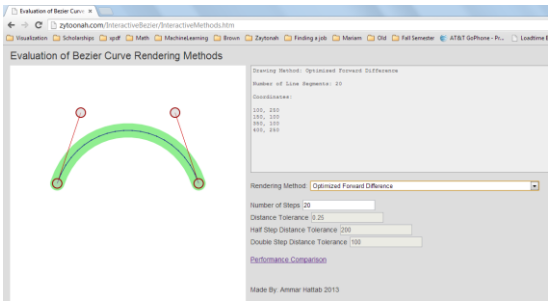

Figure 8 Bézier Curve Rendering Methods Tool

Then to evaluate the performance of rendering methods we created a jsPerf that compares the code of the methods in the previous tool using low and high parameter values (to control the approximation error) and run it on different browsers.

Bézier Curves Animation

Normally web developers do enhance animation (or interaction) speed by cashing the rendered pixels in memory and reusing them in next frames.

Using pixel cashing is more than ten-times faster than normally drawing the path [15]

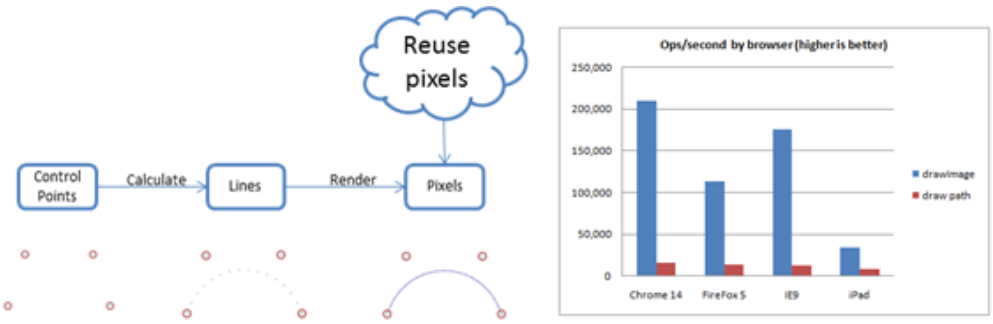

Figure 9 Cashing in Animation

Reusing Line Segments

Cashing the pixels is not feasible every time and requires more memory. So a new idea for enhancing Bézier Curves animation that we have tested in this project is to cash the generated line segments (instead of cashing pixels themselves) and reuse them in next frames.

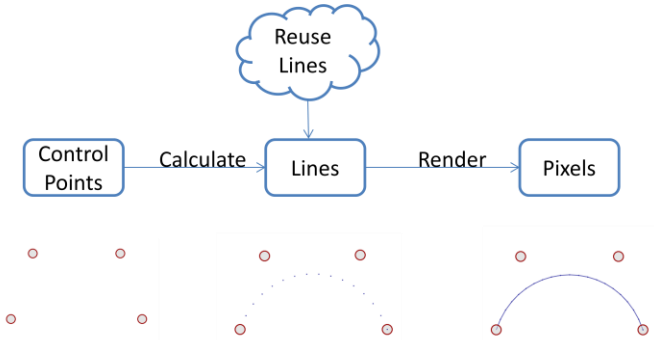

Figure 10 Reusing Line Segments Idea

In order to test this idea we have created a jsPerf that compares applying geometric transformations (Translation, Scaling, and Rotation) to the cashed lines (pre-rendered) as compared to applying them to the control points of the Bézier curve and rendering it again (generating line segments)

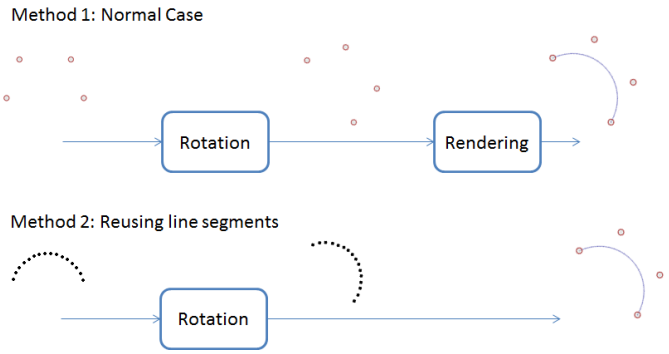

Figure11 Reusing Line Segments In Rotation

Results

Browsers Performance [16]

Running the jsPerf with a direct call to Bézier curve drawing function on different browsers, we get the following performance diagram (Figure 12), the first note is that in general drawing lines is faster than drawing Bézier curves;

In this test case (on a specific platform) Google Chrome was the fastest Browser in rendering Bézier curves which could be (but not necessarily) due to usage of a better algorithm in rendering (other reasons might be a better internal structure that handled the web request and responded to it in a faster way, or better utilization on available hardware acceleration).

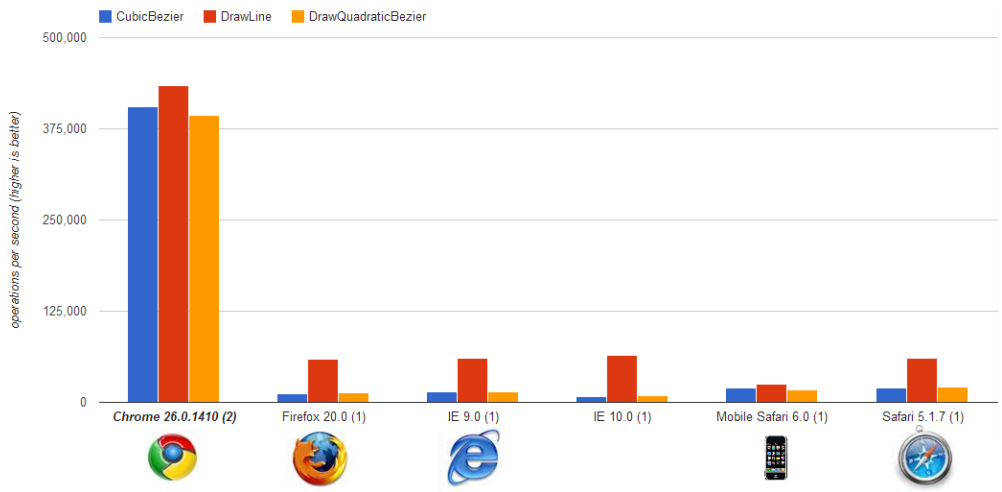

Figure 12 Browsers Performance

Rendering Methods Performance Results [17]

Running the jsPerf that compares the code of different Bézier curves rendering methods (discussed before) on different browsers, we get the following diagram (Figure 13)

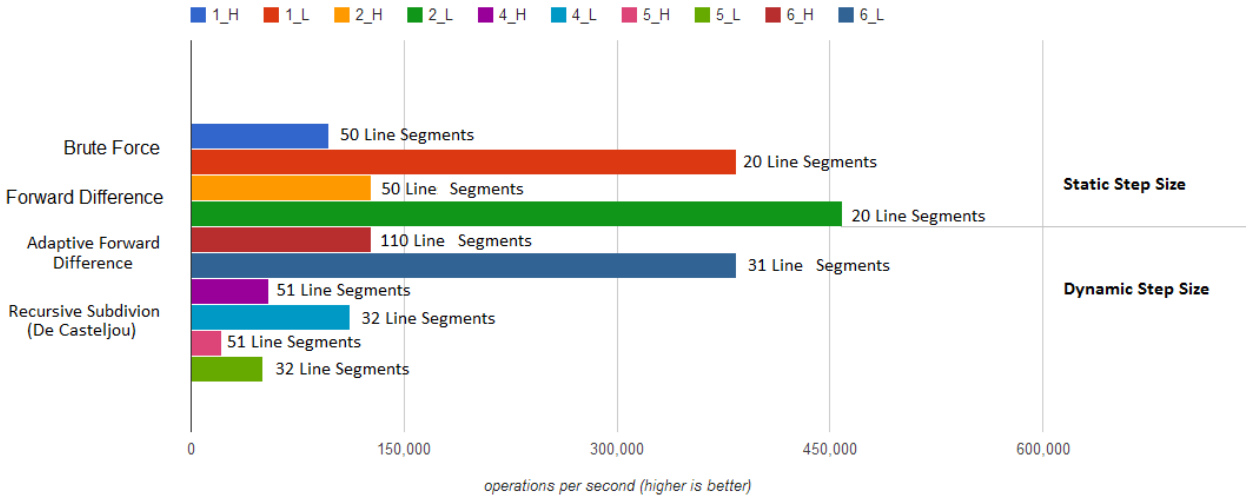

Figure13 Bezier Rendering Methods Performance

We found that our implementation of recursive methods were slower here, and we have a better performance in forward difference methods (both in lower and higher parameter values) but recursive methods in our case gave better precision than the forward difference ones, in practice they might use more efficient stopping condition estimate to have better results.

Transformations Performance Test [18]

Running the jsPerf that compares applying geometric transformations (Translation, Scaling, and Rotation) to the cashed lines (pre-rendered) as compared to applying them to the control points of the Bézier curve and rendering it again gave us the following figure.

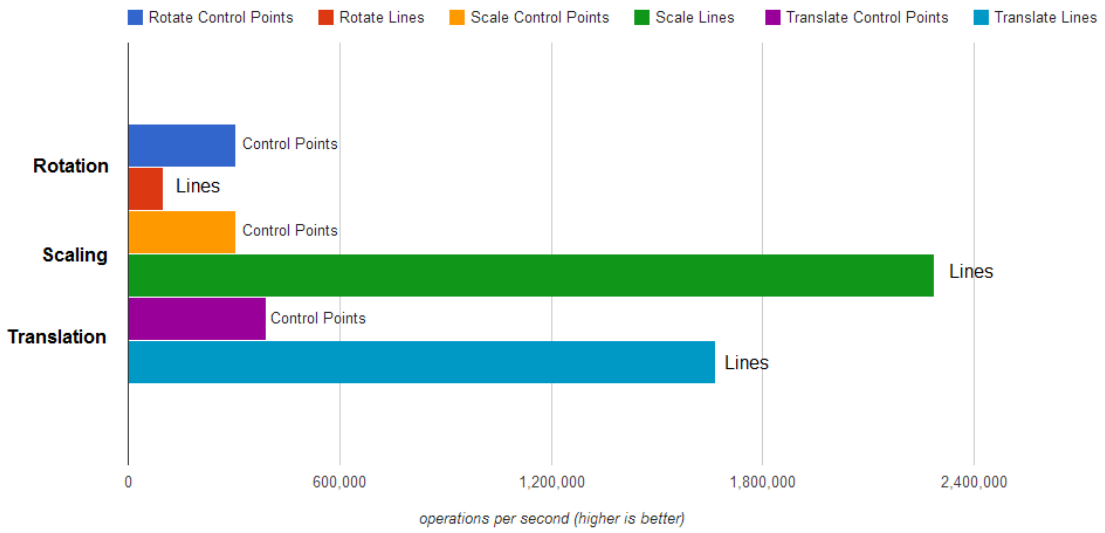

Figure 14 Transformations Performance Test

We could notice that Translation and Scaling is much faster to be done on the pre-rendered lines rather as compared to applying them to control points and re-rendering the curve, and that result is expected due as re-rendering the curves would require more computations (in order to generate the lines), so for example in fastest case of Forward Difference it would require 6 additions for each point, while in the case of scaling and translation we would need two operations only (addition or multiplications).

The unexpected result from the previous diagram was that rotation found to be slower when done on the pre-rendered lines as compared to applying it to the control points of the Bézier curve and re-rendering it, one reason might be due to the need to calculate the Sine and Cosine of the rotation angle and multiplying that value with each line points (while rotating the control points and re-rendering would not require that calculation), so we would expect this problem to be solved by using a better way to perform the rotation.

Conclusion

In this project we evaluated the performance of web browsers rendering Bézier curves and found that it depends very much on the platform and the application, in one test platform and using direct calls of Bézier curves drawing functions we found that Google Chrome was much faster than other browsers in drawing Bézier curves.

Then we compared the performance of rendering methods individually, and found that our implementation of recursive methods were slower generally but gave better precision.

Then we focused on animation of Bézier curves; where we tested a new idea of reusing generated lines to enhance the performance of animations and that is by applying geometric transformations (Translation, Scaling, and Rotation) to the cashed lines (pre-rendered) instead of to applying them to the control points of the Bézier curve and re-rendering it in each frame of the animation, we found that generally reusing the generated lines will give faster rendering results.

The importance of this result is that it could be applied by the web browser implementer to enhance the animation performance regardless of whether the web developer will use bitmap cashing or not.

We could build on this result in testing the performance of reusing lines for other types of transformations and may be modifications of the Bézier curves.

References

1. Ahmad, Athar Luqman. *Approximation of a Bézier Curve with a Minimal Number of Line Segments*. Diss. University of South Alabama, 2001.
2. Kilgard, Mark J., and Jeff Bolz. "GPU-accelerated path rendering." *ACM Transactions on Graphics* (TOG) 31.6 (2012): 172.
3. Ferraiolo, Jon. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*, W3C Recommendation 16 August 2011.
4. Tom Wiltzius. "*Jank Busting for Better Rendering Performance*". Nov 2012 Retrieved May 2013 from http://www.html5rocks.com/en/tutorials/speed/rendering/
5. Chet Haase and Romain Guy. "*For Butter or Worse: Smoothing Out Performance in Android UIs*", 28 June 2012, Retrieved May 2013 from https://developers.google.com/events/io/2012/sessions/gooio2012/109/
6. Canvas http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html#drawing-paths-to-the-canvas
7. Google Chromium Project website, *Life of a Chromium Developer*. Retrieved May 2013 https://docs.google.com/presentation/d/1abnqM9j6zFodPHA38JG1061rG2iGj_GABxEDgZsdbJg/
8. *Graphics and Skia*, The Chromium Project, Retrieved May 2013 http://www.chromium.org/developers/design-documents/graphics-and-skia
9. Maxim Shemanarev. *Interpolation with Bézier Curves*. Retrieved May 2013 http://antigrain.com/research/Bézier_interpolation/index.html
10. Hain, Thomas F. "Rapid termination evaluation for recursive subdivision of Bézier curves." *Proceedings of the international conference on imaging science, systems, and technology*. 2002.
11. S. Lien, M. Shantz, and V. Pratt. "Adaptive Forward Differencing for Rendering Curves and Surfaces", *Computer Graphics* (SIGGRAPH), 1987.
12. Mathias Bynens, Javascript Performance Playground jsPerf.com.

13. Avramović, Darko, et al. "Evaluating Web browser graphics rendering system performance by using dynamically generated SVG." *Journal of Graphic Engineering and Design* 3 (2012): 1.
14. Ammar Hattab. *Bézier Curves Rendering Methods Tool.* Retrieved May 2013. http://zytoonah.com/InteractiveBézier/InteractiveMethods.htm
15. Simon Sarris. *Increasing Performance by Caching Paths on HTML5 Canvas*. Retrieved May 2013. http://simonsarris.com/blog/427-increasing-performance-by-caching-paths-on-canvas
16. Ammar Hattab. *Bézier Curves Browsers Performance Test*. http://jsperf.com/curve-vs-lines
17. Ammar Hattab. *Bézier Curves Rendering Methods Performance Test*. http://jsperf.com/Bézier-rendering-methods-precision
18. Ammar Hattab. *Bézier Curves Transformations Performance Test*. http://jsperf.com/Bézier-rendering-transformations/2