SDCurve.js: A JavaScript Library for Interactive Subdivision Curves

Richard Pusch, Charles Perin, Sheelagh Carpendale

SDCurve.js: A JavaScript Library for Interactive Subdivision Curves

Richard Pusch* University of Calgary
Charles Perin† University of Calgary
Sheelagh Carpendale‡ University of Calgary

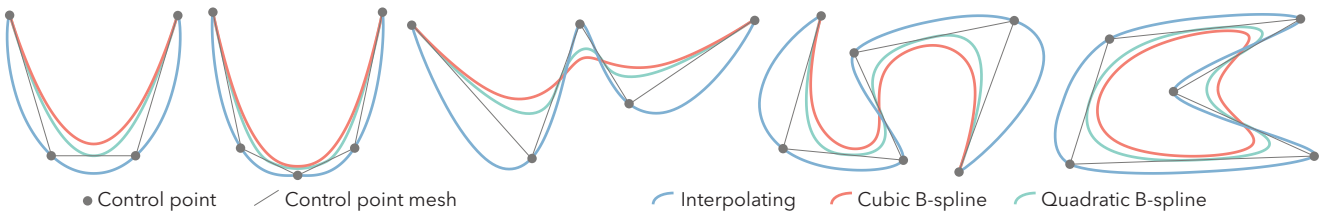

Figure 1: Sample curves created using SDCurve.js illustrating how using different schemes leads to different curves for the same set of control points. The first four images show open curves and the rightmost image shows closed curves.

ABSTRACT

We present SDCurve.js, a JavaScript library for creating interactive subdivision curves. Although subdivision curves are well-suited for use in interactive visualizations, the InfoVis community has used them sparingly. SDCurve.js is a lightweight, open-source, D3-compatible library that implements several common curve schemes, such as interpolating curves and B-Splines of any degree, in a subdivision framework that allows fast and easy curve interaction. We hope InfoVis researchers and designers will take advantage of it and contribute to its expansion.

1 INTRODUCTION

When implementing interactive visualizations, designers and developers often need to make use of curves. Curves can form the entire core of a visualization (e.g., [1]), or they may simply highlight certain aspects or link discrete elements together (e.g., [4, 7]). We focus on visualizations where it may be important to be able to interact directly with the curve, e.g., adjusting a lasso selection.

Bezier curves are a common choice for smooth, well-behaved curves. However, each control point affects all areas of the curve. This makes local control of the curve difficult, as adjusting one control point changes the entire shape of the curve. B-spline curves extend Bezier curves to provide local control, i.e. each point on the curve depends on a subset of the control points. However, they are ill-suited for some applications, as the control points are still unlikely to lie on the final curve, and interacting with the curve often must be done indirectly by manipulating these offset, often hidden, control points. *Interpolating schemes*, on the other hand, can be used to guarantee that a curve will pass through a set of control points. In practice, supporting multiple curve schemes through a unified framework focused on direct interaction is ideal. Designers can then work with the best scheme for their task.

A *subdivision curve* – a curve that is iteratively refined through linear operations on the control points – can create B-splines of any degree, but can also easily handle many interpolating schemes. This flexibility allows subdivision curves to be easily applied to many visualization needs. Despite these advantages, subdivision curves, popular in the graphics community, are rarely used in InfoVis. To make subdivision curves accessible to InfoVis researchers and designers, we contribute a fast, interactive JavaScript library that implements B-splines of any degree, including commonly used quadratic (Chaikin [3]) B-splines, and Dyn-Levin [5, 8] interpolating subdivision curves. Figure 1 shows how choosing different schemes leads to different curves. The curves can be created and animated in real time, are designed to be compatible with the popular D3 library [2], and alternate schemes can easily be added via open source development. This lightweight library, designed by InfoVis designers for InfoVis designers, will make it easier to create web-based visualizations where interactive curves are important.

*e-mail: rapusch@ucalgary.ca
†e-mail: charles.perin@ucalgary.ca
‡e-mail: sheelagh@ucalgary.ca

2 SUBDIVISION CURVES

Subdivision converts an initial set of control points to a fine set of curve points by applying repeated linear operations. *Linear masks* define the instructions for this process; a mask $[m_0, m_1, \ldots, m_t]$ operates on consecutive "coarse" points $P_n, P_{n+1}, \ldots, P_{n+t}$ and creates a new "fine" point $\sum_i m_i P_{n+i}$. With smartly chosen linear masks, the result of each step creates a refined curve that is continuous and smooth, and after only 4 or 5 steps, the curves are visually pleasing.

Figure 2 illustrates this process with the Chaikin "corner-cutting" scheme [3], whose linear masks are $[\frac{3}{4}, \frac{1}{4}]$ and $[\frac{1}{4}, \frac{3}{4}]$. That is, for any two consecutive non-endpoints p_n and p_{n+1}, the subdivision operation creates new points at $\frac{3}{4}p_n + \frac{1}{4}p_{n+1}$ and $\frac{1}{4}p_n + \frac{3}{4}p_{n+1}$. Special masks are designed to handle edgepoint cases; for Chaikin, each endpoint is carried forward to the next subdivision step, and the mask $[\frac{1}{2}, \frac{1}{2}]$ is used if p_n or p_{n+1} is an endpoint.

Applying different masks will result in different curves. The Chaikin scheme produces curves that are equivalent to quadratic B-splines with enough subdivisions. Quadratic B-splines have high local control and tend to get quite close to the control points; if a more gradual curve is needed, the degree of the B-spline can be increased. In our library, we implemented Stam's [9] algorithm, a faster version of the Lane-Riesenfeld algorithm [6], which indirectly applies masks through an iterative smoothing process. This can produce B-splines of any degree. For very high-degree B-splines, this algorithm requires many smoothing steps. Our library can be expanded to include the direct mask calculation for any arbitrary B-spline degree if certain high-degree curves are often used.

The Dyn-Levin subdivision scheme [5] directly interpolates the control points using a linear mask that is 4 points wide. In the non-endpoint case, each point is carried forward to the next subdivision level (mask $[0, 1, 0, 0]$), and a new point is created with the mask $[\frac{-1}{16}, \frac{9}{16}, \frac{9}{16}, \frac{-1}{16}]$. Endpoint conditions are used from Semmerud [8]. Unlike B-splines, this means that all control points lie on the curve.

Each curve type has different merits depending on the needs of the designer. Subdivision curves encapsulate the benefits of traditional B-splines while also easily supporting alternate schemes in

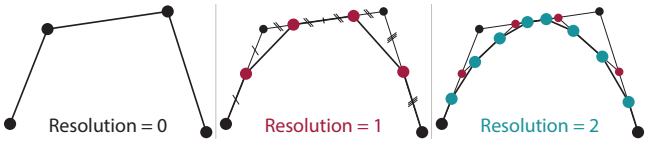

Figure 2: Several successive subdivisions for four control points using the Chaikin "corner-cutting" scheme [3].

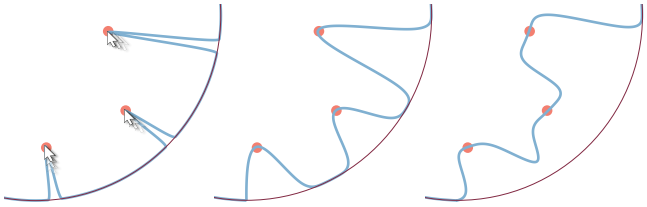

Figure 3: Different dragging behavior, from sharp to wide, can be achieved by changing the optional "width" parameter to `moveCurve()`. The same three dragging operations can result in different curves.

the same framework, and because they operate on a familiar control point paradigm, designers should find them easy to work with.

3 Concepts and Implementation

There are other JavaScript libraries that make it possible to create interactive curves, such as three.js[1], a graphics library dedicated to 3D rendering. D3 itself also provides some basic curve functionality, though it allows only a single B-spline degree and has no built-in support for curve interaction. To the best of our knowledge, there is no lightweight library dedicated to interactive curves that leverages the benefits of subdivision. In this section, we describe the basic, advanced, and interactive functionality of `SDCurve.js`.

3.1 Basic Functionality

Calling the `new SDCurve(arg)` constructor builds a subdivision curve object, where `arg` is a map with the following fields:

- `points`: required array of control points. All other parameters are optional and have a default value if not specified.
- `open`: boolean indicating whether the curve is open or closed (default: open).
- `resolution`: the number of subdivisions to perform (default: 5).
- `type`: the subdivision scheme to use. Can be Bezier, B-spline, or interpolating (default: B-spline).
- `degree`: the degree of the B-spline curve. Applies only when B-spline is the selected type (default: 2, i.e., quadratic).

For a created object, these values can be changed or queried by calling the `points()`, `open()`, `resolution()`, `degree()`, and `type()` functions on the object. The `curve()` function returns the array of points needed to draw the resulting curve. This array can be mapped to the `line()` function in D3 [2] and drawn to an SVG object.

3.2 Advanced Functionality

There are two ways of changing the control points on a curve. Calling the `points()` function with an array of points will change the entire curve to have a new set of control points. Calling the `adjustPoints()` function is useful if only a small part of the curve needs to be changed. `adjustPoints()` accepts a map of indices to new points. Only the control point at specified indices will change, and only the affected parts of the curve will be recomputed. This "lazy update" approach allows interactivity even on large curves. Our library also provides ways to measure distance along or near the curve. The `pointAt(u)` function returns the point of distance u ($0 \leq u \leq 1$) along the curve, where $u = 0$ is the beginning of the curve and $u = 1$ is the end of the curve. To accomplish this, we parameterize the curve by *arclength* in the background. We can also call the `getClosestPoint(point)` function to get the closest point on the curve to the provided point. This is helpful for snapping a moving mouse coordinate directly to the curve, for example.

[1] `https://github.com/mrdoob/three.js/`

3.3 Interactive Functionality

Manipulating Bezier or B-spline curves is typically done by moving control points. This can make fine control over the curve difficult, and control points are usually hidden in visualizations. Our library allows designers to specify any point on the curve and move it to a new location through the `moveCurve()` function. This is simply done by providing a point along the curve (for example, the location of a mouse click) and a movement vector to specify its new location. The curve will make the necessary changes to the nearest control points to achieve this effect. This allows the curve to be deformed by direct interaction, and no knowledge of the underlying control point structure is needed. The `moveCurve()` function optionally accepts a "width" argument, which specifies how many of the underlying control points are allowed to move. This supports influencing only a small area, causing sharp alterations to the curve, or a wide area for a more gradual, smooth transition. Figure 3 illustrates interacting with the curve with different "width" values.

4 Conclusion

Subdivision curves, popular in the graphics community, are rarely used in InfoVis. They offer great flexibility, can support standard curves, and are particularly well-suited to interacting directly with points on the curve instead of offset control points.

`SDCurve.js` is an open-source library available at `https://github.com/rpusch/sdcurve.js/`. It can be used "as is" for creating curves and is easily plugged into D3 visualizations. `SDCurve.js` will continue to evolve with novel algorithms, optimizations, and subdivision schemes. New schemes are easy to add and we hope the community of InfoVis developers and designers will contribute to expanding the library.

Acknowledgements

Thanks to support from NSERC, AITF, and SMART Technologies.

References

[1] B. A. Aseniero, A. Tang, and S. Carpendale. River: Using personalisation to support reflection on personal activities. In *Proceedings of the DIS'14 Workshop on A Personal Perspective on Visualization and Visual Analytic*, page 4, 2014.

[2] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE TVCG*, 17(12):2301–2309, Dec. 2011.

[3] G. M. Chaikin. An algorithm for high-speed curve generation. *Computer Graphics and Image Processing*, 3(4):346 – 349, 1974.

[4] C. Collins, G. Penn, and S. Carpendale. Bubble sets: Revealing set relations with isocontours over existing visualizations. *IEEE TVCG*, 15(6):1009–1016, Nov. 2009.

[5] N. Dyn, D. Levin, and J. A. Gregory. A 4-point interpolatory subdivision scheme for curve design. *Computer Aided Geometric Design*, 4(4):257 – 268, 1987.

[6] J. M. Lane and R. F. Riesenfeld. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2(1):35–46, Jan. 1980.

[7] C. Perin, R. Vuillemot, and J.-D. Fekete. Soccerstories: A kick-off for visual soccer analysis. *IEEE TVCG*, 19(12):2506–2515, 2013.

[8] H. Semmerud. Boundary conditions for interpolatory subdivision. Master's thesis, University of Oslo, 2013.

[9] J. Stam. On subdivision schemes generalizing uniform b-spline surfaces of arbitrary degree. *Comput. Aided Geom. Des.*, 18(5):383–396, June 2001.