Approximating Arcs Using Cubic Bézier Curves

Joe Cridge
www.joecridge.me

June 2015

Abstract

Bézier curves can be used to approximate elliptical arcs in systems where there is no native arc support; this is useful in many graphics (and other computer aided design) applications owing to the extensive support for cubic Bézier curves across the multitude of vector-based formats and specifications. We propose a simple method to approximate an arbitrary elliptical arc using up to four cubic Bézier curves and find that it can be accurate to one pixel when rendered to displays as large as the emerging 4K standard. Means of extending the method should greater accuracy be required are also suggested.

Introduction

Bézier curves provide a convenient means of expressing arbitrary, smooth curves in terms of a relatively small number of waypoints. This has important uses in vector graphics, since it allows succinct representation and storage of indefinitely scalable curved paths.

A general Bézier curve is comprised of a set of $n + 1$ waypoints. The first and last waypoints, $\mathbf{r}_0$ and $\mathbf{r}_n$, form the endpoints of the curve and are commonly known as *anchor points*, while the intermediate points lie around the curve (connecting to form a convex hull which encloses the curve, defining its shape), and so are known as *control points*. The curve is then described at all points from start to finish by a weighted sum over the waypoints in the parameter $t \in [0, 1]$. Specifically, the weighting of the j^{th} waypoint is given by the j^{th} Bernstein basis polynomial[1], hence:

$$\mathbf{B}_n(t) = \sum_{j=0}^{n} {}^{n}\mathrm{C}_j \, t^j \, (1-t)^{n-j} \, \mathbf{r}_j. \tag{1}$$

[1]See https://en.wikipedia.org/wiki/Bernstein_polynomial.

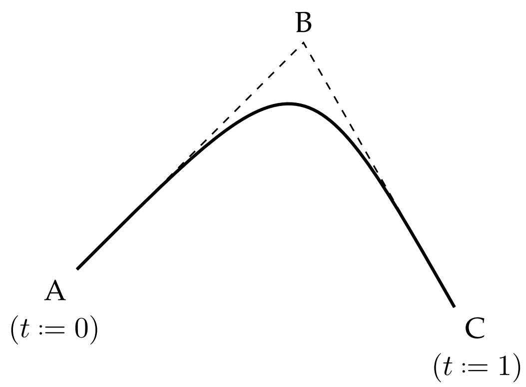

Figure 1: A quadratic Bézier curve.

Linear curve The first order Bézier curve is the simplest, and amounts to nothing more than linear interpolation. Suppose that we wished to describe the line from a point A to another point B; using a Bézier curve, we would write:

$$\mathbf{B}_1(t) = (1-t)\,\mathbf{a} + t\mathbf{b}. \tag{2}$$

Quadratic curve Second order Bézier curves provide a means of describing parabolic arcs. As in Figure 1, a parabolic arc between anchor points A and C with control point B is described by the equation

$$\mathbf{B}_2(t) = (1-t)^2\,\mathbf{a} + 2t\,(1-t)\,\mathbf{b} + t^2\mathbf{c}. \tag{3}$$

By differentiating Equation 3, it is easy to see that the arc is tangent to AB when $t = 0$ at A, and tangent to BC when $t = 1$ at C.

Cubic curve Cubic Béziers use two control points, which gives them enough degrees of freedom to start usefully approximating arbitrary curves. If we label the waypoints A through D (see Figure 2), we have:

$$\mathbf{B}_3(t) = (1-t)^3\,\mathbf{a} + 3t\,(1-t)^2\,\mathbf{b} + 3t^2\,(1-t)\,\mathbf{c} + t^3\mathbf{d}. \tag{4}$$

Similarly to quadratic Bézier curves, it is true that for cubic Béziers the curve is tangent to AB at A and tangent to CD at D.

Higher order Bézier curves exist, and provide greater accuracy when approximating an arbitrary curved form, but they are naturally more computationally expensive to evaluate.

It is our intention to devise a method that will allow the representation of elliptical arcs using Bézier curves to within a desired tolerance. Since cubic

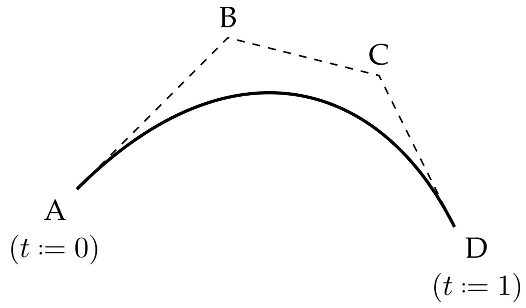

Figure 2: A cubic Bézier curve.

Bézier curves are the *de facto* standard when it comes to vector-graphics, and are nearly universally available in modern computer graphics systems (including the PostScript and SVG formats), we will use tham exclusively in our method, favouring the use of a composite of multiple cubic Bézier curves over the use of a single curve of higher order.

Method

In order to determine the suitability of using cubic Béziers to model elliptical arcs, we will propose a method of fitting a Bézier to a small, circular arc, and then calculate by numerical means the maximum angle of arc that this method can be applied to whilst remaining within a given tolerable error in radius. To approximate arcs of angle greater than this practical maximum, we will use a composite of curves of smaller arcs, and to approximate non-circular arcs we will scale the waypoints of an approximated circular arc linearly[2] in the directions of the major and minor axes.[3]

Consider a circular arc of unit radius which subtends an angle 2α from the origin; it is convenient to centre this arc on the x-axis as in Figure 3. Labelling the Bézier waypoints A through D as before, and additionally denoting the midpoint of the circular arc M, we make the following observations:

[2]This is fine, so long as we are not concerned with the accuracy of the start and end angles of the arc. Specifically, the relationship between the angle ϑ' that should be specified and the desired angle ϑ to be rendered is $\tan\vartheta' = \frac{w}{h}\tan\vartheta$, where w and h are, respectively, the lengths of the horizontal and vertical axes (see below). In any case, the start and end angles will remain within the correct quadrant at least.

[3]For convenience, we will assume that the axes of the ellipse lie along the horizontal and the vertical; adjusting the method to allow for the axes to lie in an arbitrary direction is simply a case of rotating the set of calculated waypoints about the centre of the ellipse by the desired angle. Note that, again, the start and end angles of the arc would have to be adjusted to take this rotation into account.

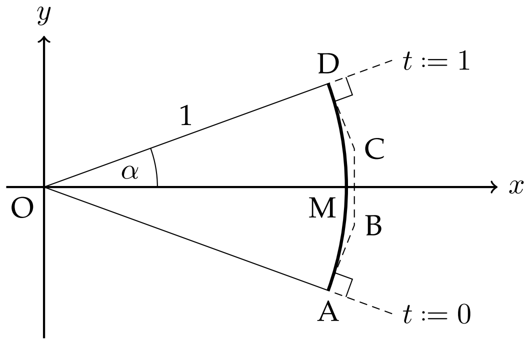

Figure 3: Set up for approximating a circular arc.

1. Points A, D, and M can be expressed simply as follows:

$$\mathbf{a} = \cos\alpha\,\hat{\mathbf{x}} - \sin\alpha\,\hat{\mathbf{y}}, \tag{5a}$$
$$\mathbf{d} = \cos\alpha\,\hat{\mathbf{x}} + \sin\alpha\,\hat{\mathbf{y}}, \tag{5b}$$
$$\mathbf{m} = \hat{\mathbf{x}}. \tag{5c}$$

   Here we have assumed that we wish for the endpoints of the Bézier arc to align exactly with the endpoints of the true circular arc: this is to ensure continuity when multiple arcs are joined together to form a composite.

2. We further require AB and CD to be tangents to the true arc at A and D, respectively, in order that the joins between Bézier curves be smooth. Hence,

$$(\mathbf{b} - \mathbf{a}) \cdot \mathbf{a} = (\mathbf{d} - \mathbf{c}) \cdot \mathbf{d} = 0. \tag{6}$$

3. Lastly, symmetry requires that BC be parallel to the y-axis and centered on the x-axis. This permits us to define the constants λ and μ that satisfy

$$\mathbf{b} = \lambda\,\hat{\mathbf{x}} - \mu\,\hat{\mathbf{y}}, \tag{7a}$$
$$\mathbf{c} = \lambda\,\hat{\mathbf{x}} + \mu\,\hat{\mathbf{y}}. \tag{7b}$$

Our method is then as follows: we will choose λ so that the midpoint of the Bézier arc coincides with M – the midpoint of the true arc – then adjust μ to ensure that Equation 6 is satisfied for this choice of λ. This should give us a good overall fit.

Calculating $\boldsymbol{\lambda}$ and $\boldsymbol{\mu}$ Evaluating Equation 4 at $t = 0.5$ gives:

$$\mathbf{B}(0.5) = \frac{1}{8}\mathbf{a} + \frac{3}{8}\mathbf{b} + \frac{3}{8}\mathbf{c} + \frac{1}{8}\mathbf{d} \stackrel{!}{=} \mathbf{m}. \tag{8}$$

Substituting in Equations 5 and 7 and rearranging for λ yields

$$\lambda = \frac{4 - \cos\alpha}{3}. \tag{9}$$

Equation 6 now gives:

$$\begin{aligned} \mu &= \sin\alpha + (\cos\alpha - \lambda)\cot\alpha & (10a) \\ &= \sin\alpha + \frac{4}{3}(\cos\alpha - 1)\cot\alpha. & (10b) \end{aligned}$$

Note that we are also able to Taylor expand these results if necessary:

$$\begin{aligned} \lambda &= \frac{1}{3}\alpha + \frac{1}{9}\alpha^3 + \frac{1}{360}\alpha^5 + O\big(\alpha^6\big)\,, & (11a) \\ \mu &= 1 + \frac{1}{6}\alpha^2 - \frac{1}{72}\alpha^4 + O\big(\alpha^6\big)\,. & (11b) \end{aligned}$$

Assessment of Accuracy

Equations 5, 7, 9, and 10 provide all of the information necessary to begin testing the model. As stated before, we will calculate the relative error in radius that occurs when the model is applied to an arc of a given total angle, and then use this data to determine the largest angle that should be approximated with a single Bézier curve. Such a calculation is most easily carried out using MATLAB, and source code for doing so is provided in Appendix A. The results have been plotted in Figure 4, and some key values are highlighted in Table 1.

Computer graphics For a graphics application it is reasonable to require that an approximated curve be accurate to within about one pixel when drawn to a target display. In 2015, the largest display size that an application is likely to be required to draw to is 4K, and so we require that the relative radial error does not exceed about 1 in 4000; the third column of Table 1 shows that this can be achieved by using one cubic Bézier per $\pi/2$ radians of arc.

Design and manufacture The accuracy requirements of computer aided design and manufacture are generally more stringent than those for computer graphics, but they can often still be met using Bézier curves. We see from Figure 4 that one curve per $\pi/4$ radians results in an error on the order of one part per million, although beyond this it may prove significantly more economical to work within a system that has native circular arc support.

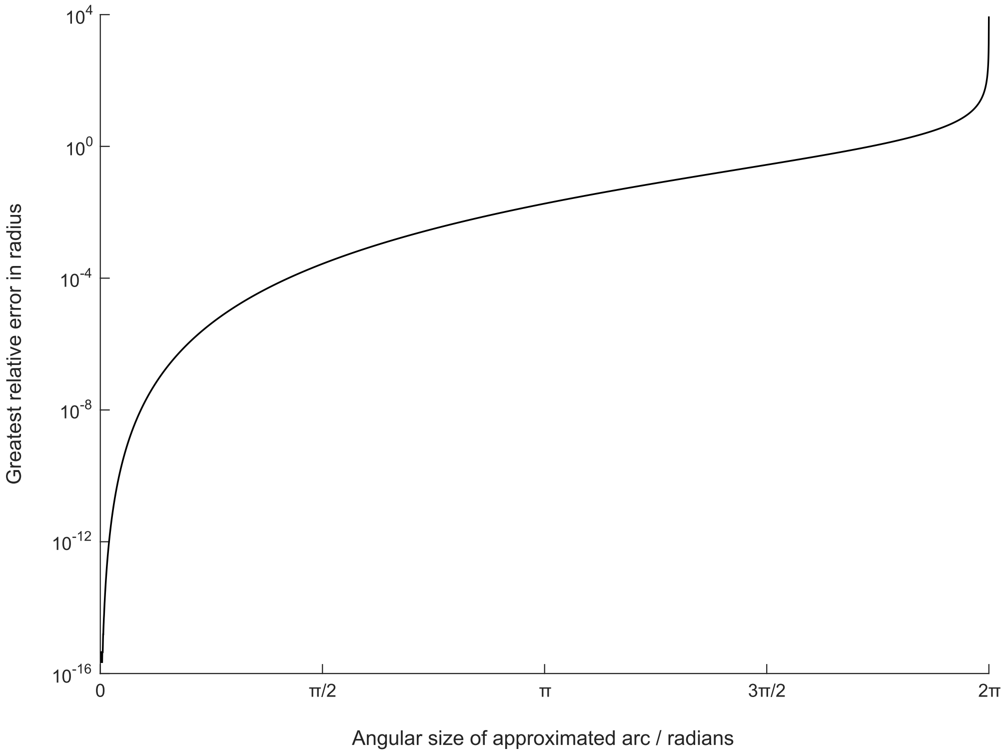

Figure 4: Relative error in radius versus total arc angle for a circular arc approximated by a single cubic Bézier curve.

Angle	Relative Error	Potential Misalignment
$\pi/4$	4.2×10^{-6}	None
$\pi/2$	2.7×10^{-4}	1 px
π	1.8×10^{-2}	75 px
$3\pi/2$	2.8×10^{-1}	1130 px

Table 1: Error in radius that occurs when using a single cubic Bézier curve to approximate a circular arc. The third column shows the potential misalignment that would occur when drawing the arc to fill a 4K display.

Proposed Implementation

Based on the preceeding analysis we propose the following method as a means of approximating elliptical arcs using cubic Béziers in a graphics application:

1. Adjust the start and end angles of the arc to take into account the linear scaling that will be performed in Step 4.
2. Split the required arc up into sectors of $\pi/2$ radians, with a final 're-mainder' sector if necessary.
3. Approximate every full sector as a cubic Bézier curve on the unit circle, as in Equations 5 through 10. The Bézier waypoints of each curve will need to be rotated around the origin according to the required start and end angle.
4. If the remainder sector exists, and has angular size greater than a practical minimum, use a cubic Bézier to approximate it in the same way as the other sectors. For displays no bigger than 4K, a minimum angle on the order of 10^{-4} radians ought to be sufficient.[4]
5. Scale the waypoints of the resulting composite Bézier linearly in the horizontal and vertical directions according to the specified elliptical width and height. This assumes that the end user is willing to rotate the returned arc themselves if they desire the axes to lie in a different direction, and that they will adjust the start and end angles to account for this.

If, in the future, greater accuracy is required, the implementation can be updated simply by replacing $\pi/2$ with a smaller angle.

An example implementation using the *p5.js* JavaScript library[5] can be found in Appendix B.

Comparison with Other Methods

Bézier curves are by no means the only way to approximate elliptical curves, and it would be worthwhile to briefly compare our method to its alternatives. A chord-based approximation is possibly the most popular alternative due to the ease of its implementation, and so it will serve as a useful comparison.

[4] This will mean that the arc starts and stops within one pixel of the specified angles at the radius.

[5] http://p5js.org/

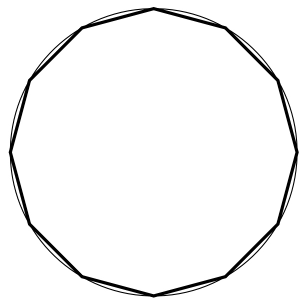

Figure 5: Use of chords to approximate a circle.

A simple chord method involves dividing the circle on which the arc is to be drawn into n equal sectors, and using an n-sided polygon to connect their vertices; when n is suitably large, the polygon serves as a good approximation to the circle. This is illustrated in Figure 5.

The greatest radial error here occurs at the midpoint of each sector: the true radius exceeds the effective radius at the chord by a fraction $1 - \cos\varepsilon$, where $2\varepsilon = 2\pi/n$ is the angle subtended by the chord. Achieving the same relative error as is afforded by the method described above requires $\varepsilon < 2.3 \times 10^{-2}$ and hence $n > 135$.

Whereas the Bézier method uses, on average, 1.6 coordinates in memory and 0.4 graphics calls per radian of arc, the chord method requires 43 and 21, respectively. It is clear that the Bézier method is significantly more efficient.[6]

Conclusion

We have proposed a simple method for approximating arbitrary elliptical arcs using cubic Bézier curves, and have found that this approach can offer a vast improvement in efficiency over other, chord-based methods, whilst being only mildly more difficult to implement, if at all. Our method is certainly not the best way to use Bézier curves to approximate arcs – the fact that all radial errors were positive is an indication of the room for improvement – but it is capable of achieving the accuracy required by most graphics applications using a modest amount of computation. An improved method would focus on minimising the total error over the entire path of each curve (rather than simply fixing the midpoint), but whether the improvement would justify the analytical effort involved in its derivation (and likely additional computation) remains to be seen. ■

[6] We are assuming that the system in question has native support for interpolating cubic Béziers, for example when this is provided by specialised functions on the graphics card. When this is not the case, the Béziers will themselves be approximated by chords further down the graphics pipeline, resulting in an overall efficiency that is similar to (but generally no worse than) the simple chord method.

A MATLAB Test Code

```
% Generates approximations of circular arcs of different sizes using a single
% cubic Bezier curve and then calculates and plots the maximum error in radius
% that occurs in each approximation.

EPSILON = 0.001;  % Approximately 1/20th of a degree.

% Iterate over arcs of increasing angle, using EPSILON as the step.
errors = zeros(floor(2 * pi / EPSILON), 2);
nDivisions = 1;
for angle = EPSILON : EPSILON : 2 * pi
  bezier = circular_arc_to_bezier(-angle / 2, angle / 2);
  points = evaluate_cubic_bezier(bezier, nDivisions);

  % Calculate the radial error on each interpolated Bezier point.
  curveErrors = zeros(nDivisions, 1);
  for pointNo = 1 : nDivisions
    radius = norm(points(pointNo, : ));
    curveErrors(pointNo) = radius - 1;
  end

  % Record the greatest error.
  maxError = max(curveErrors);
  minError = min(curveErrors);
  if (abs(minError) > abs(maxError))
    errors(nDivisions, 2) = minError;
  else
    errors(nDivisions, 2) = maxError;
  end
  errors(nDivisions, 1) = angle;

  nDivisions = nDivisions + 1;
end

disp('Created a list of (angle, error) pairs.');

clear angle bezier curveErrors EPSILON maxError minError nDivisions ...
      pointNo points radius;
```

Listing 1: bezierArcAccuracyTest.m

```
function bezier = circular_arc_to_bezier(startAngle, stopAngle)
% circular_arc_to_bezier Generates a cubic Bezier representing a circular arc.
%   bezier = circular_arc_to_bezier(startAngle, stopAngle) returns Bezier
%   waypoints representing a unit radius circular arc between `startAngle` and
%   `stopAngle` radians.

  TWO_PI = 2 * pi;

  % Make all angles positive...
  while (startAngle < 0)
    startAngle = startAngle + TWO_PI;
  end
  while (stopAngle < 0)
    stopAngle = stopAngle + TWO_PI;
  end

  % ...and confine them to the interval [0,TWO_PI).
  startAngle = mod(startAngle, TWO_PI);
  stopAngle = mod(stopAngle, TWO_PI);

  % Exceed the interval if necessary in order to preserve the size and
  % orientation of the arc.
  if (startAngle > stopAngle)
    stopAngle = stopAngle + TWO_PI;
  end

  % Evaluate constants.
  ALPHA = (stopAngle - startAngle) ./ 2;
  COS_ALPHA = cos(ALPHA);
  SIN_ALPHA = sin(ALPHA);
  COT_ALPHA = 1 ./ tan(ALPHA);
  PHI = startAngle + ALPHA;
  COS_PHI = cos(PHI);
  SIN_PHI = sin(PHI);
  LAMBDA = (4 - COS_ALPHA) ./ 3;
  MU = SIN_ALPHA + (COS_ALPHA - LAMBDA) .* COT_ALPHA;

  % Return Bezier.
  bezier = zeros(4,2);
  bezier(1,1) = cos(startAngle);
  bezier(1,2) = sin(startAngle);
  bezier(2,1) = LAMBDA .* COS_PHI + MU .* SIN_PHI;
  bezier(2,2) = LAMBDA .* SIN_PHI - MU .* COS_PHI;
  bezier(3,1) = LAMBDA .* COS_PHI - MU .* SIN_PHI;
  bezier(3,2) = LAMBDA .* SIN_PHI + MU .* COS_PHI;
  bezier(4,1) = cos(stopAngle);
  bezier(4,2) = sin(stopAngle);
end
```

Listing 2: circular_arc_to_bezier.m

```
function points = evaluate_cubic_bezier(bezier, nDivisions)
% evaluate_cubic_bezier Generates coordinates that represent a cubic Bezier.
%   points = evaluate_cubic_bezier(bezier, nDivisions) returns coordinates for
%   a series of `nDivisions` chords that represent the curve `bezier`.

  % Get waypoint vectors.
  A = bezier(1, : );
  B = bezier(2, : );
  C = bezier(3, : );
  D = bezier(4, : );


  % Return coordinates.
  points = zeros(nDivisions + 1, 2);
  pointNo = 1;
  for t = 0 : 1 ./ nDivisions : 1;
    points(pointNo, : ) = (1 - t).^3 .* A + 3 .* t .* (1 - t).^2 .* B + ...
                          3 .* t.^2 .* (1 - t) .* C + t.^3 .* D;
    pointNo = pointNo + 1;
  end
end
```

Listing 3: evaluate_cubic_bezier.m

B Example Implementation

```
/**
 * Approximate a general elliptical arc using [up to four] cubic Beziers.
 */
function exampleArc(x, y, w, h, start, stop) {

  // Make all angles positive...
  while (start < 0) {
    start += TWO_PI;
  }
  while (stop < 0) {
    stop += TWO_PI;
  }

  // ...and confine them to the interval [0,TWO_PI).
  start %= TWO_PI;
  stop %= TWO_PI;

  // Adjust angles to counter linear scaling.
  if (start <= HALF_PI) {
    start = atan(w / h * tan(start));
  } else  if (start > HALF_PI && start <= 3 * HALF_PI) {
    start = atan(w / h * tan(start)) + PI;
  } else {
    start = atan(w / h * tan(start)) + TWO_PI;
  }
  if (stop <= HALF_PI) {
    stop = atan(w / h *tan(stop));
  } else  if (stop > HALF_PI && stop <= 3 * HALF_PI) {
    stop = atan(w / h * tan(stop)) + PI;
  } else {
    stop = atan(w / h * tan(stop)) + TWO_PI;
  }

  // Exceed the interval if necessary in order to preserve the size and
  // orientation of the arc.
  if (start > stop) {
    stop += TWO_PI;
  }

  // Create curves
  var epsilon = 0.00001;  // Smallest visible angle on displays up to 4K.
  var arcToDraw = 0;
  var curves = [];
  while(stop - start > epsilon) {
    arcToDraw = min(stop - start, HALF_PI);
    curves.push(acuteArcToBezier(start, arcToDraw));
    start += arcToDraw;
  }
```

```
  // Draw curves
  var rx = w / 2.0;
  var ry = h / 2.0;
  curves.forEach(function (curve, index) {
    bezier(x + rx * curve.ax, y + ry * curve.ay,
           x + rx * curve.bx, y + ry * curve.by,
           x + rx * curve.cx, y + ry * curve.cy,
           x + rx * curve.dx, y + ry * curve.dy);
  });
}
```

Listing 4: exampleArc.js

```
/**
 * Generate a cubic Bezier representing an arc on the unit circle of total
 * angle `size` radians, beginning `start` radians above the x-axis.
 */
function acuteArcToBezier(start, size) {

  // Evaluate constants.
  var alpha = size / 2.0,
    cos_alpha = cos(alpha),
    sin_alpha = sin(alpha),
    cot_alpha = 1.0 / tan(alpha),
    phi = start + alpha;  // This is how far the arc needs to be rotated.
    cos_phi = cos(phi),
    sin_phi = sin(phi),
    lambda = (4.0 - cos_alpha) / 3.0,
    mu = sin_alpha + (cos_alpha - lambda) * cot_alpha;

  // Return rotated waypoints.
  return {
    ax: cos(start),
    ay: sin(start),
    bx: lambda * cos_phi + mu * sin_phi,
    by: lambda * sin_phi - mu * cos_phi,
    cx: lambda * cos_phi - mu * sin_phi,
    cy: lambda * sin_phi + mu * cos_phi,
    dx: cos(start + size),
    dy: sin(start + size)
  };
}
```

Listing 5: acuteArcToBezier.js